PhUSE 2010

P002

An Animated Guide:
Coding standards for SAS® Production Programs

Russ Lavery, Contractor K&L Consulting Services
Fort Washington, PA. , U.S.A.

ABSTRACT

Coding standards are not a matter of style or professionalism. Coding standards, if enforced by management, are a great help in a company's goal to have programmers produce code that is *inexpensive to write and to maintain*. This paper is one programmer's collection of coding standards. It is intended to be a starting point for discussion within companies that lack standards and a resource for people starting a career as a SAS programmer.

Implementing standards will cause the creation of

1) programs that are, in large measure, self-documenting and
2) a useful "outside of the program" document that describes the program logic.

The above two points will create "well documented programs" and will:

Simplify and enhance communication — Allow faster program validation
Provide "backup" for programmers — Reduce frustration and delays
REDUCE THE COST, AND INCREASE THE ACCURACY, OF PROGRAM MAINTENANCE

The 2010 additions to this paper are in suggestion 55 and the appendix.

INTRODUCTION

Production programs have several characteristics that are, at the heart, COST related issues.

1) Production programs are the core of the value that a programmer delivers to his/her company/client.
2) Production programs are run, periodically, for long periods of time.
3) During their years of life, most production programs need to be maintained.
4) During the life of a production program, responsibility for production program maintenance tends to be transferred from the person who wrote the program to another programmer - or through a *series* of other programmers. Maintenance is often not done by the person who wrote the program.
5) Code should be INEXPENSIVE to maintain.

As a definition, in self-documenting code, the purpose of the code or code section should be clear from the evidence *presented within the code* without referencing back to the design specifications. However production code should also have external documentation.

This paper, is a collection of suggestions for coding standards compiled a many years of reading SAS papers, lurking on SAS-L, listening to SAS programmers and suffering through bad code. It is hoped that this list sparks discussion and that it will be used as *one source* for the creation of coding standards.

This paper includes the thoughts of far too many people to allow me to cite and thank them all. However two should be cited for their overarching (and recent enough, so that I can remember) contributions.

Joe Perry wrote about the two-degree of separation coding philosophy. The name of the rule came out of the aerospace industry where, when a drawing was created by hand, the printing had to be clear enough, that when a third generation copy was viewed, the printing and all other details would be completely legible-so that the part could be built correctly. We should code so that when the ownership of the code is transferred TWO times, the code is still inexpensive to maintain.

Gregg Neilson repeatedly says: "BETTER, FASTER, CHEAPER" in talks. If we do not adopt this philosophy to drive costs out of the system we expose our employers/clients and ourselves to attack by more efficient competitors.

COLLECTED CODING SUGGESTIONS

1) Management must create, publicize and check the coding standards. This task can not be delegated to a low level person (like a contractor).

Reason: Coding standards will not be followed unless management gets involved, institutes code review and enforces the standards. If management lacks the will to enforce standards people will code for job security.

2) Production programs should:

A) **Have external documentation** of the business rules, business owner and caveats.
Reason: Know what environmental changes will require program changes and who should approve the change. It takes time and costs money to re-discover this information.
B) **Be self documenting** (a well maintained header block, good structure and comments).
Reason: The code is faster (cheaper) to transfer to a new programmer

3) Production code should not throw any errors or warnings. If you know enough to say the error/warning does not hurt you, you should know enough to be able to make the errors/warnings go away.

Reason: Easier QC of the log. Programmers picking up your work must check the errors. "Normal" errors make searching the log for "new" errors more difficult. "Normal" errors" are detected by log checking programs, making the log checking program ineffective. I have seen "production" programs that threw thousands of errors. Needless to say the program had not comments or documentation and was part of a six program series that ran to thousands of lines of code.

4) Emphasizing "CODING CLARITY" before "CODING CLEVERNESS" produces code that is cheaper to maintain.

Reason: Clear, simple code is faster (cheaper) to transfer to a new programmer. This idea will be repeated and restated several times in this paper in different ways and with varying levels of specificity.

5) When the "current owner" is available, the "current owner" of the program should create documentation and conduct a "code walk through" for the person taking over the code.

Reason: Creating/updating documentation is the proper responsibility of the "current owner" because the "current owner" can create documentation faster/cheaper than a person who is assuming responsibility for the program (and who has never seen the program before). When information must be re-discovered, costs go up. "Code walk throughs" are cost effective, especially if the "current owner" did not document the program well. Sometimes "code owners" make things difficult for the people taking over. Ego, power, raises and job security are involved.

6) External documentation should be structured, organized, created by the program owner/creator and in one document.

Reason: Bad documentation does not help us be "better, faster, cheaper". Bad documentation does not drive cost out of the system.Several pages of paper covered with scribbles and then layered over with post-it notes are not good documentation (or good specifications, if you are starting a project).

It is hard to create documentation when one is discussing step 15, and someone says "Oh I forgot to tell you something about step 3". The new person is typically taking notes on paper and there might not be space on the page, in the general area around step 3, to write the new information about step 3.

This "out of order" phenomenon is one reason why it is better to have the "last owner" create documentation, rather than dictate documentation the "the new person". If the last owner dictates the documentation, the last owner can (and often does) dictate poorly. Then the burden for finding all mistakes is on the new person. If the "last owner" has to create the documentation, it is obvious who "owns" any mistakes or lack of clarity – and the person who understands the process is the one who is creating the documentation.

Logically, the "last owner" has the understanding and is the one who can quickly create documentation. Code owners can make things difficult for people taking over. Ego, power, raises and job security are involved.

Management might want to consider the idea that "senior employees", or "departing employees", might *not really want to help* new people become productive and might have a career, or personal, interest in creating bad documentation - so long as they can not be blamed for the bad documentation.

7) Programs should have a maintained "Doc Block". This should include a list of files used and created.

```
/*******************************************************************************
Program:          _____
By:               _____
Infiles:          _____
Outfiles:         _____
Standard Macros Called:  _____
Modifications:
Usage / Instructions:
Business User:
Purpose:
*******************************************************************************
```

Reason: Provides an overview of the program without having to read the whole program. This reduces cost.

8) Agree with the client on a common project name and a directory: When you get a project ask "Should I save this?" and if the answer is "yes" ask "What is your directory name"?

Reason: Helps find old work quickly

Often managers/clients ask for the product XXXXXX project that you did "a while back" – not remembering that you have done 10 product XXXXXX projects – for that client alone.

Often a client/manager does not share enough for you to organize your work well. A manager that gives out projects as a series of small tasks, without an overview meeting or instructions, ends up with support people having directories named XXXXXJan and then XXXXXXjan21. In one place, I had directories with names like "EmergencyDataPullJan21" and "EmergencyDayaPullFeb02" with no idea how that data was to be used – but the manager wanted me to be able to re-create any file in minutes.

Without a common project name or an "overview meeting", programmers have no idea as to the relationship between the bits of work they have done. Also, the manager/client assigning tasks has no idea of the directory structure that the programmer created. The manager, since there is no manager-programmer organization, is forced to ask about *"the XXXXXX project that we did awhile ago. The one where we did work on the weekend"* or worse *"the XXXX project"*- as if there were only one such project.

9) Ask client if there are any reports to which *your* report must "foot". If yes, ask for the business rules that were used to create those reports and who can run the reports. QC your output against those reports before you show your numbers to the client.

Reason: First; knowing the business rules is essential for your coding. Secondly; knowing the answer before you start allows you to QC the **initial** data read (they might not have told you the name of the correct file). Having the report in your hands when you are doing your QC helps **you** find mistakes. It is better that you do the first QC check, not your boss or the client.

10) Divide programs into logical sections- see 51 and 55. Emphasize the structure of the program with section dividers and blank spaces. Use comments to record the business rules. This will be repeated below.

POOR CODE – NO SPACES & SPAGHETTI CODE ☹

```
data w;
set rsw. RTLWP503;

data m;
set rsm.FF00028QW;

data W1;
set W;
where wk_end_dt GT
'2007-08-17';

data M1;
set M;
where sales_lvl_ind
="D";

data W2;
set W1;
Age=age*12;

data M2;
set M1;
Age=age*12;
Run
```

Good – has structure- macros at top- uses indenting- has comments of business rules & program logic – collect logic (Where and transforms in one data step). ☺

```
/************************************
Section: Set Macros
************************************/
%let Maxdate=2008-08-17;
/************************************
Section: Weeks
************************************/
data Weekly; /*We Need weekly data for some odd reason*/
   set rsw. RTLWP503;
   /*Program started Aug 17*/
   where wk_end_dt GT '2007-08-17'd;
   AgeMo=age*12;
run;
/************************************
Section: Months -We need monthly district level data 4 QC
************************************/
data montlyChk;
  set rsm. RTLMZ_FF00028QW;
  where sales_lvl_ind ="D";
  AgeMo=age*12;
run;
```

Reason: This makes the program cheaper to understand and modify. Clients change their mind, and then change it back again. Coding in sections lets you more easily comment out a section that you think will be added in again. – If you comment out code that you think will be added back, plan a code cleaning step Please see 8).

11) Linear flow: Program flow should be: ProgA, ProgB, ProgC and NOT ProgB, Prog1, Prog_revised

Reason: Easier to follow.

12) Create a Logical Program Flow- see 51: If there are several programs in a system, consider creating a main program to call each program rather than having each program call the next. Macros and libraries should be defined in the main program. If not done, new programmers 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.
Reason: Others can read the main program and see the big picture.

13) Perform Unambiguous Merging
A) **Always use a BY** statement. (set the system option so not having a merge by statement is an error)
B) **Avoid having the same variables on more than one dataset** (except the BY variables).
C) **Some people say, do not merge more than 2 datasets at a time**.
D) **Do not allow the message:** NOTE: MERGE statement has more than one data set with repeats of BY values.
Reason: reduces errors.

14) Use SAS functions if you can, rather than coding your version of them.
Reason: Faster run times. Often easier to understand and more accurate as well.

15) Know and use procedures Let the SAS procedure do the work. If possible, use PROCs to debug data steps, rather than writing data step code. Learn PROC Tabulate and PROC Report.
Reason: Faster run times, easier to understand and more accurate

16) End every data step and PROC with a run. End SQL code with a quit;
Reason: this helps the compiler.

17) Group non-executable statements (length, attrib, retain, format, informat etc.) at the top of a data step- before executable statements.
Reason: Ease of reading.

18) Strive to minimize the number of data steps and PROCs. A read of the data costs money.
Reason: Faster run times are cheaper run times.

19) Only read and keep variables and rows you need. Small files process faster/cheaper.
Reason: Faster run times are cheaper run times.

20) Use PROC SQL or PROC datasets to delete working files that are no longer needed.
Reason: Reduce use of SasWork and disk space in general.

21) Remove junk code before you hand over a program. If you create a macro that is not called, or a data set that is not used, remove it before making the program production. In multi-program reports, people can spend hours trying to figure out if/where/when a macro, or file, is used. Dead code increases the difficulty for the next programmer and makes jobs run slower. I have seen programs that were 60% dead code.
Reason: When a program is transferred with lots of macros that are not called, and data sets that are not used, the new programmer must spend extra time to figure out if he/she missed the use of a macro (or data set) or if it is *really* not used.

22) Naming Conventions (also see 10, 51 and 55) .
For variables and data sets:

A) Names should not be re-used.
It is difficult to understand programs when a variable, or data set, means one thing in one place in the program but has a different meaning in another part of the program.
B) Names should be meaningful descriptions of datasets and variables.
Programs with variable and tables named D1, d2, d3 … are hard to read.
C) If a meaningful name is too long, use a shorter name but explain it with a comment.
Explain names whose brevity obscures their meaning and remember a 1 and a lower case l look very similar.
Reason: Programs that are easier to understand are easier, and cheaper, to maintain.

23) When the amount of typing is about the same, use a "keep" as a data set option rather than a "drop".
Imagine a data set containing the variables: name age sex height weight and class.
If we code: Data New(Keep=name height class); we document what is in the new data set.
If we code: Data New(Drop=age sex weight); we do NOT doTument what is in the data set.
On the same issue,
The use of the - - (double dash) as a way of abbreviating a variable list requires that the user know the order of variables in the Program Data Vector and this might not be so. Use sparingly.
Reason: Not using the - - makes it easier to see what variables are in a data set.

24) Variables with names like region01, region02 ….. region11 etc. sort better than variables named region1, region2, …region11.
Reason: It is a bit easier to read information if it sorts in a logical/orderly manner.

25) Code to check for missing and unexpected data. In If then, code an Else. In formats, code an Other.
Reason: the data is never as clean as it looks – or as clean as it was "in the last delivery".

26) Insert a blank line between SAS program steps (before each data step or PROC).
Reason: White space makes code easier to read.

27) Do not place more than one programming statement on a line.
Reason: Ease of reading.

28) Do not extend your code past the readable area (column 71-80)
Reason: If code goes out to column 150, a reader must scroll back and forth. On mainframes this text may be "lost".

29) Use parenthesis to clarify the sequence of operations
e.g. AdjustedChange= ((Q2Tot – (Q1Tot*&MacroVariable))*100)/(Q1Tot*&MacroVariable)
Reason: Ease of reading.

30) Break really complicated statements into a number of simpler statements. Being clever can cost money.
e.g. AdjLastQtr=(Q1Tot*&MacroVariable);
AdjustedChange=((Q2Tot- AdjLastQtr)*100) / AdjLastQtr;
Reason: Ease of reading.

31) Avoid unconventional, obscure and convoluted logic, unless you can not think of a simpler way.
If you must do cool (or weird or *elegant*) things, use lots of comments.
Reason: Ease of reading.

32) Keyword (named) macro parameters are preferable to positional parameters
(Author opinion: There is **NO** reason for mixing keyword and positional parameters).

Use named parameters:	**Avoid using positional parameters and mixing types:**
`%macro(`**`DSN`** `=Sashelp.Class,` `  `**`Where=%`**`str(sex="M";)`	`%macro(Sashelp.Class, %str(sex="M";)`
Code above is a bit easier to understand than the code to the right.	**There is no reason to mix parameter types.** `%macro(DSN=Sashelp.Class, %str(sex="M";)`

Reason: Ease of reading.

33) If speed is a consideration, use a sub-setting WHERE rather than an IF.
A sub-setting IF, however, does give more information in the log.
Reason: WHERE subsets the data before entering it into the Program Data Vector.
IF subsets the data after loading the record into the Program Data Vector.

34) When stepping through an IF condition and execution speed is an issue, consider checking 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;

Reason: Faster run times, though might not be worth the effort unless you are creating production code.

35) Use IF/ELSE for mutually exclusive conditions.

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

Reason: Code runs faster and cheaper. The ELSE/IF (above) will check only those observations that fail the first IF condition. In the code in the right part of the box (above), all observations will be checked twice.

36) Sort only the variables needed and use the options noequals where possible.
e.g. PROC SORT DATA=X(KEEP=A B C) NOEQUALS SORTSIZE=Max; If you want to dedup using PROC sort read (The Mystery of the PROC SORT Options NODUPRECS and NODUPKEY Revealed by Britta Kelsey,)
Reason: NoEquals makes Proc Sort faster and therefore cheaper. Sort has an "issue" with removing duplicates.

37) Check for violations of correct conditions to minimize messages in the log.
e.g.. Division by zero will throw a message to the log. Code to prevent the message.

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

Reason: Avoids messages in the log.

38) Define constants within a %LET SECTION or a driver file – see 51 and 56. Do not hard-code values in the program.

Use:

```
/*********************************
Section: Set Macros Section
*********************************/
%LET STARTYR = 2000; /*Start of do loop*/
%LET ENDYR = 2020;   /*END of do loop*/

/*********************************
Section: SAS code
*********************************/
Data new;
set old;
    DO I=&STARTYR TO &ENDYR;
    more sas code;
    end;
run;
```

Instead of:

```
Data new;
set old;
    DO I = 2000 TO 2020;
    more sas code;
    end;
run;
```

If you use driver files, and macros, to control execution of your programs DOCUMENT HOW THE DRIVER FILES ARE TO BE MAINTAINED!!!!

Reason: Code is easier to maintain if values are not hard coded and can be changed in one place.

39) Use macros if: - see 40
A) The routine is used more than once.
B) The routine depends on a value of a variable.
C) The routine requires programming logic that cannot be included in a DATA step.
Use macro libraries (with the AUTOCALL facility) if:
A) The routine is used by more than one program or B) The routine changes often.
Put the name of the macro on the %mend (especially if the macro is long or nested).
Reason: Using macros, generally, makes programs harder to read.

40) Use no more than 2 layers of nested macro calls – see 39
Reason: It is too easy to get lost after 2 calls.

41) Create permanent datasets at the end of the program - See 51.
Do not create permanent datasets at points scattered through-out the program. Create perm datasets 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.

42) If you are writing a SAS data set to an XLS tab, make the name of the SAS data set be the name of the tab- rather than some nonsensical name like A1. Put a note in the program saying that this data set will be sent to an Excel tab.

Reason: Easier to follow.
Note: Creation of XLS tabs is often macro driven and real names do not appear in the program. Comments help.

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

A) Avoid un-initialized variables.
e.g. Avoid the message: NOTE: Variable XX is un-initialized.
Reason: This might mean that a variable you think is in the data set is not there, or you have spelled the variable name wrong. Either way, this is really an error.

B)Avoid automatic numeric/character conversions; Use PUT/INPUT so your program does the converting.
e.g. Avoid the message:
NOTE: Character values have been converted to numeric values at the places given by.
Reason: the notes clutter up the log.

C) Avoid automatic formatting; Fix the program so that it uses the correct format.
e.g. Avoid: "NOTE: At least one W.D format was too small for the number to be printed."
When you see the above note, the decimal may be shifted by the "BEST" format.
Reason: This can sometimes cause loss of data.

D) Avoid excessive repetition of error messages. e.g. Use OPTIONS ERRORS=2;
Reason: Printing many errors clutters up the log.

44) Avoid getting data in Excel.
Excel allows a column (a variable in SAS) to be a mixture of numbers, dates and characters. This mixing of variable types can confuse SAS. When a file is several thousand rows deep, is difficult to QC, through visual examination. Because Excel is allows mixed type columns, a data feed in Excel can be fine one month and junk the next. Access, because of a rigidly defined data structure, is a better data transfer mechanism than Excel. In any case, communicate and agree on specs for file transfer.
Reason: Data from Excel files often has quirks that take time to fix.

45) Even if they are intended to contain "bad data", do not name data sets or variables "ERROR" or "warning". This makes it difficult for people to use an editor to search the log for errors or to use a log checking program.

If you want to print an error message, use the trick from Christopher Edel
His article is at: //www.nesug.org/proceedings/nesug03/cc/cc014.pdf
For custom control messages in a program, Christopher uses the following trick:

```
data _null_ ;
   put "WARN" "ING: - bad thing! Check!" _all_;
   put "E" "RROR: - a very bad thing! Check!" _all_;
run ;
```

By splitting the WARNING or ERROR between two literal strings, these tokens do not appear in the SAS Log unless the trigger condition emerges. No concatenation operator is needed between the two literal strings.

46) Consider using Select instead of IF:
See http://analytics.ncsu.edu/sesug/2001/P-614.pdf , where Andrew Ratcliffe wrote the following.
To help me with trapping unexpected values, I generally use select statements instead of if statements when I am testing for specific values.
For instance, if I have a variable that contains gender as 'M' or 'F', I might be tempted to code:

```
if gender eq 'M' then ...do male things...
   else ...do female things...
```

But a safer option is to use the select statement:

```
select (gender);
when ('M') ...do male things...
when ('F') ...do female things...
end
```

The advantage of using the select statement is two-fold. Firstly, you are making the valid values of the gender variable very clear. Secondly, the program will bomb if gender contains an invalid value. Thus, you get the earliest warning that something is wrong. If I had used an if statement, my program would have continued with an incorrect belief that we were dealing with a female observation.
You can use the select statement's otherwise clause to trap unexpected values, but that presupposes that you can do

something in the event that an unexpected value arises.

47) COMPLETELY document any use of regular expressions

Reason: Regular expressions can be very hard to understand.

48) Do NOT password protect programs!

Reason: I have seen the passwords lost as production programs were transferred to new people. Embarrassing!

49) Save everything and organize everything – ask for project closure

Save all the versions of your programs as you go along. Managers often get to version 9 of a project and decide they have to go back to version 3. Put "stuff" (even using secret codes you create) in the output that lets you, when a client waves a paper in front of you, know what program produced that piece of output/paper (footnotes and titles are good for holding secret codes). Add program name as an extra variable, in data sets that you hand off to clients.

Ask if projects are "over" and then take some time to clean up the directories. This is especially important is you are in a consulting environment. If you are a consultant, see if your company has a "clean & archive" policy.

Reason: Clients often are so harried, and so clueless about what you do, that they do not take any time to organize a project. They will come back 9 months later and ask for something to be re-run.

50) Use of white space, aligning logical sections and indenting makes program logic easier to grasp.

Examples of Good and Bad are below. The bad examples are taken from real "production" code.

Bad code appears as it was in the SAS program. It was not word wrapped by MS Word®.

Bad – No structure or spaces- Looks like the red formula created the variable named Optout

```
PROC SQL;
Create table odd as
sum(prev_abcnew12) as prev_abcnew12,
(sum(curr_abcnew12) -
sum(prev_abcnew12)) as NEWFile_12_chg,sum(optout) as optout
from perm.Dataset;
```

Bad-

No structure and not even a single space between variables.

```
  PROC sql; create table db.new_Holding_YYY_profile as select
  mnth,newid,vendor_id,pcidnew as pcid,field_force,region,district,
  territory,first_name,last_name,address,city,
  state,zip,zip_code,quintile,total_active_enrollees,baseline_abc_enrollees
  ,abc_new_enroll, p_new_enroll,
  cummul_new_enroll,location_id,tot_reff,m,legend from  new_bic_presc_profile
  union all select
  mnth, compress(input(put(vendor_id,12.),$12.))||"||
  input(put(location_id,12.),$12.),vendor_id,pcid,field_force,region,district,
  territory,first_name,last_name,address,city,
  state,zip_code,zip_code,quintile,Total_Active_Enrollees,Baseline_ABC_Enrollees,
  MTD_ABC_new_Enrollees as abc_new_enroll, MTD_P_new_Enrollees as p_new_enroll,
  YTD_ABC_new_Enrollees as cummul_new_enroll,location_id,MTD_Referrals as
tot_reff,m,legend from XXXX.YYY_YYYY_&d;
```

Note: The code above is not word-wrapped to fit in this box.
The word wrapping you see was in the original code.

Three Different layouts but all are **Good.**

```
Proc Sql;
Create table males
as
      select name
     ,age
     From
sashelp.class
     where sex="M";
quit;
```

```
Proc Sql;
Create table males
  as select name
           ,age
  From sashelp.class
      where sex="M";
quit;
```

```
Proc Sql;
 Create table males
 as
  select
       name
      ,age
 From
  sashelp.class
  where sex="M";
 quit;
```

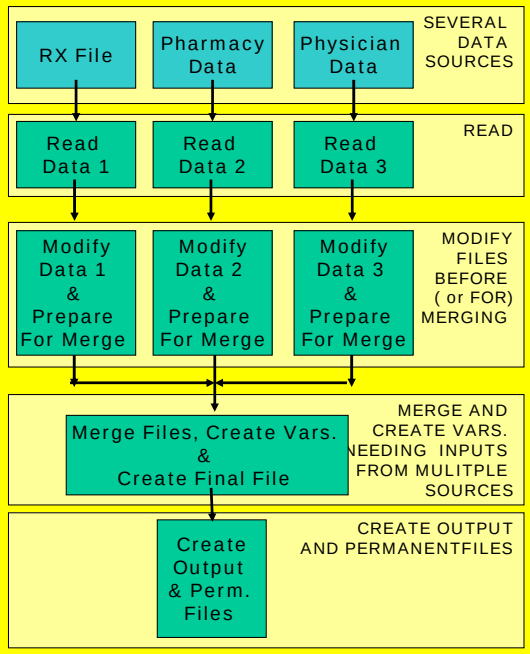
Good:

Align variables for easy reading

```
   Format     Agecat  Agecatf.
            Quest1   YesNOf.
            Quest5   AgreeDis.;
```

51) Consider the following program structures. Mark Lyons, a friend, says that since SAS programs often read several files and summarize them to a level where the several files can be merged into one report file, consider this:
Section 1: Read all files in one section (the section should have identified subsections)
Section 2: Modify Files and perform all transforms, summarizations and create merging variables (see above note)
Section 3: Merge all files into one in a section (the section should have identified subsections)
Section 4: Write reports and create perm files in a final section (the section should have identified subsections)

Since many projects have the structure shown below:



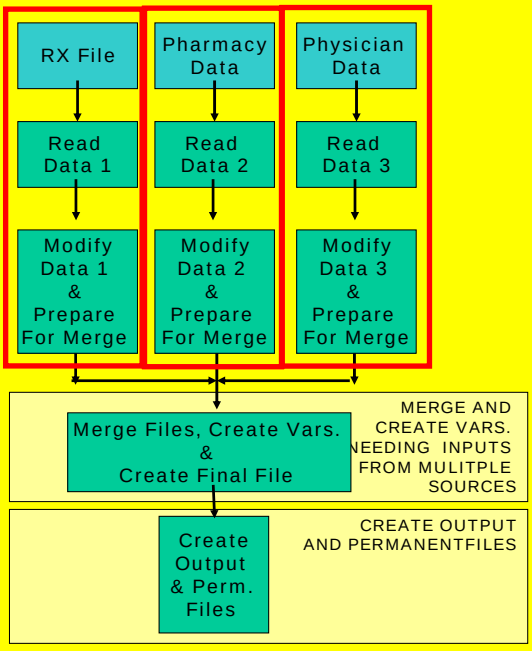
Consider the program structure below:

```
/**********
DocBlock  -  Details of DocBlock not shown
**********/
/***LIBRARY AND MACRO VARIABLE SECTION***/
  LIbname PermLib = "C:\ClientFiles" ;
/**  READ DATA SECTION***/
/*SECTION Read_1_1: Read Rx*/
Data Rx_zip; /*Rx data*/
  Lines of code; run;

/** SECTION Read_1_2: Read Pharmacy**/
Data Pharm; /*Rx data*/
  Lines of code; run;

/* SECTION Read_1_ 3:  Read Physician*/
Data doc; /*Rx data*/
  Lines of code; run;

/*** MODIFY DATA SECTION***;
** SECTION MODIFY_2_ 1:  Modify RX;
data Rx_Monntly_transpose;
set Rx;
  lines of sas code;

** SECTION MODIFY_2_ 2: Pharm;
data Pharm_w_Parsed address;
set pharm;
  lines of sas code;

** SECTION MODIFY_2_3: DOC;
data doc_w_Parsed address;
set doc;
  lines of sas code;

/***MERGE & MAKEPERM SECTION***/
data permlib.RX_Pharm_doc;
merge         Rx_Monntly_transpose
              Pharm_w_Parsed address
doc_w_Parsed address;
By Zip; Run;
```

Reason: Thinking about code in this way helps the programmer make the code modular. IF COMMENTED (especially commenting the business rules) and then structured within sections, the structure to left of the code is easy to describe and to understand.
If not commented, this can turn into spaghetti code.

Use comments to help show which type of structure you used in your code.

Since many projects have the structure shown below:



Consider the program structure below:

```
/**********
DocBlock  -  Details of DocBlock not shown
**********/
/***LIBRARY AND MACRO VARIABLE SECTION***/
  LIbname PermLib = "C:\ClientFiles" ;
/**  READ DATA SECTION***/
/*RX SECTION*/
/*SECTION Read_1_1: Read Rx*/
Data Rx_zip; /*Rx data*/
  Lines of code; run;
** SECTION MODIFY_1 2:  Modify RX;
data Rx_Monntly_transpose;
set Rx;
  lines of sas code;
/*Pharmacy SECTION*/
/** SECTION Read_2_1: Read Pharmacy**/
Data Pharm; /*Rx data*/
  Lines of code; run;
** SECTION MODIFY_2_ 2: Pharm;
data Pharm_w_Parsed address;
set pharm;
  lines of sas code;
/*Phaysician SECTION*/
/* SECTION Read_3_ 1:  Read Physician*/
Data doc; /*Rx data*/
  Lines of code; run;
** SECTION MODIFY_3_2: DOC;
data doc_w_Parsed address;
set doc;
  lines of sas code;

/***MERGE & MAKEPERM SECTION***/
data permlib.RX_Pharm_doc;
merge         Rx_Monntly_transpose
              Pharm_w_Parsed address
doc_w_Parsed address;
By Zip; Run;
```

Alternatively, if the code is thought of a vertical columns (see the red boxes), it can easily be made to run in parallel
A) by spawning child processes
or
B) using a macro.

In this structure, each red box is thought of as a section.

See 55 for a suggestion on naming files

52) Use White space and alignment of logical code units to help make code readable

Good

Align Do and End as well as %Do and %End…
Especially if the do is nested (nesting not shown).

Put the name of the macro on the %mend (especially if the macro is long or nested).

If the do extends over a few pages, and/or is nested consider putting a comment at the end;

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.

Good- align do & end

```
data MetricF;
set sashelp.class;
    If sex="F" then
    Do;
      wtKg=wt/2.2;
      height=height*2.25;
    end; /*If sex="F" then*/

run
```

Good-align %do-%end -Put the macroname on the %mend line

```
%macro Examp;
  %if  flag=Y %then
    %do;
          …sas code
          …sas code
    %end;/* %if flag=Y
%then*/
%mend Examp;
```

Good align do & end–label the end

```
If logic=True then
DO ;
  IF X=1 then
     DO;
       Statements;
       Statements;
     END; /*End of X=1*/
  ELSE IF X NE 1 then
      DO;
        Statements;
        Statements;
        Statements;
      END;/*End of X NE 1*/
END;  /*End of Logic=True*/
```

Poor- hard to read

```
If logic=True then DO;
IF X=1 then DO;
statements;
END; ELSE IF X NE 1 then DO;
statements;
END;END;END;
```

A SAS-L comment on Indenting and layout:
XXX makes the point that other coding standards are important, including indentation and double spacing.
As a team leader, I would find the example code he submitted unacceptable, even if you put RUN statements in.
I would give that programmer a copy of our coding standards and send them back to their desk.
I think correct indentation and spacing are more valuable in *understanding the intentions of the coder* than adding run statements. And, I always put titles inside of the PROC that they go with, it is just a habit.
I agree strongly that indentation is probably the most important factor in coding SAS.
I've seen programs that have statement after statement strung together, separated only by the semicolon – almost impossible to decipher. Those who don't indent *should* be flogged.
Easily-readable code obviously makes life so much easier and quicker for another person to decipher and modify.
As you've noted, it's important to set a standard and have everyone stick to it.

Author comment/rant on use of white space: I get arguments about indenting, white space and layout and I am always astounded to hear people, with many years of experience, argue against this practice.

Having to explain layout to an experienced programmer is, to me, like having to tell someone "Put your socks on first and then put on your shoes".

Left justify the "outermost logical unit (for both data step or macro coding).

Be consistent with indentation.

```
Data Personal;
  IF BEGPER < 1000 THEN
    DO;
          STARTQTR = BEGPER;
          BQ = BEGPER - (10*BYY);
          IF BQ > 0 THEN BMM = (3*BQ) - 2;
             ELSE IF BQ = 0 THEN BMM = 1;
    END;
    ELSE DO;
          BYY = INT(BEGPER/100);
          IF BMM IN (1,4,7,10) THEN STARTMO = 1;
            ELSE IF BMM IN (2,5,8,11) THEN STARTMO = 2;
            ELSE IF BMM IN (3,6,9,12) THEN STARTMO = 3;
    END;
   IF ENDPER < 1000 THEN
    DO;
        ENDQTR = ENDPER;
        IF EQ > 0 THEN EMM = 3*EQ;
           ELSE IF EQ = 0 THEN EMM = 12;
    END;
   ELSE
    DO;
       EYY = INT(ENDPER / 100);
       ENDQTR = INT(((EMM + 2)/3) + (EYY*10));
           IF EMM IN (1,4,7,10) THEN ENDMO = 1;
          ELSE IF EMM IN (2,5,8,11) THEN ENDMO = 2;
          ELSE                           ENDMO = 3;
    END;
run;
```

```
%macro Outer;
   ….SASCODE…..
   ….SASCODE…..
    %macro Inner;
        ….SASCODE…..
           %macro nested;
                  ….SASCODE….
           %mend nested;
           %nested;
    %mend Inner;
    %Inner;
        ….SASCODE…..
        ….SASCODE…..
%mend Outer;
%outer;
```

SAS-L Comment

A source of coding standards and generally good programming practices is Steve McConnell's "Code Complete" from Microsoft Press,(c) 1994. The coding samples are C or pascal, but most of the issues are directly applicable to SAS. There is a sequence on code layout, comments, and self-documenting code which may be insightful. McConnell (and the SAS-L person I took this from) highly recommend code walkthroughs or reviews for both quality assurance and the only realistic means of enforcing or developing standards.

53) Remove "dead" (non-functional) code from programs.
Below is an example of the need to clean out dead code.

**Original variable in the driver files should have had a meaningful name.
As a coding suggestion, minimize renaming fo data sets and variables to make logic easier to follow.
If original var. is ProgStartDate, then make macro variables have names like ProgStartDate1, ProgStartDate2, ProgStartDate3 etc.**

This was Actual Production Code! →

This program started by reading a cell in a .XLS driver table that held a variable called RP.

The name Rp is not very helpful and there was no note in the XLS driver sheet explaining how the variable was to be used or maintained.

The programmer brought in the variable and immediately renamed it three times.

Then s/he DID NOT USE any of the variables s/he created.

Instead s/he went back to a different xls driver table and got the same information. S/he put this new information into a macro variable and *did* use that macro variable later in the program

```
PROC sql;  create table info1 as
select distinct a.*, b.rp
from local.info as a left join local.rpt_dates as b on
compress(a.rpt)=compress(b.rpt);  quit;

%macro report;*Get product names from info table;
PROC sql;create table files as  select distinct compress(rpt) as
report
,rp as rpdate /*RENAMED & var is Never used */ from info1;
run;quit;

PROC sql; select count(*) into:nc from files;/*Get # of product*/
    %let nc=&nc;run;quit;

PROC sql;
select report,rpdate into:c1-:c&nc,:d1-:d&nc /*RENAMED AGAIN-Macro Never
used*/ from files; quit;
%do j=1 %to &nc;%let c&j=&&c&j; %let
Macro Never used !!!!!  :-0*/
%end;
run;
```

This code is HARD TO FOLLOW

The "dead code" in the program above should have been removed.

```
%do j=1 %to &nc; %let ct=&&C&j; %let DT=&&D&j;

PROC sql; select compress(ref,"-"),ref  into:rctdate
,:rctdate1
    from local.refdata_&ct; /*new data source RENAMED and used*/
    %let rctdate=&rctdate;
    %let rctdate1=&rctdate1;   run;quit;
More sas code
```

There were TWO Driver tables for this one program. One might ask if this makes sense.

54: Use named macro parameters to "tell" nested macros (and readers of your code) what variables an "inner"/nested macro takes from a more global referencing environment;

When inner is called below, it is obvious, from the macro call, what variables it is taking from the outer macro's referencing environment. This code is easier to understand and therefore cheaper to maintain.

```
/*Section __:     Easier to read */
%macro inner(lib= ,dsn=,
NumVar=&Numvar);
%do i=1 %to &sqlobs;
PROC print data=&lib..&dsn;
where &Numvar=&&MVar&i ;
run;
%end;
%mend inner;

%macro outer(lib=
           ,dsn=
           ,NumVar=);
PROC SQL ;
    SELECT DISTINCT &numvar
    INTO :Mvar1-:Mvar999
    FROM &lib..&dsn;quit;
%if  &SqlObs GT 999 %then
           %do;
           data _null_;
           put "ER""ROR";
           run;
           %end;
   %inner(lib=&lib
       ,dsn=&dsn
    , NumVar=&numVAr);
%mend outer;

options mlogic mprint symbolgen ;
%outer(lib=sashelp
           ,dsn=class
           ,NumVar=age);
```

←Easy read
Harder read→

When inner is called it makes its information from the outer macro's reference environment, but what information it takes is not obvious. This style of coding is harder to understand and more costly to maintain.

```
/*Section __:     Harder to read*/
%macro inner;
%do i=1 %to &sqlobs;
PROC print data=&lib..&dsn;
where &Numvar=&&MVar&i ;
run;
%end;
%mend inner;

%macro outer(lib=
           ,dsn=
           ,NumVar=);
PROC SQL ;
    SELECT DISTINCT &numvar
    INTO :Mvar1-:Mvar999
    FROM &lib..&dsn;quit;
%if  &SqlObs GT 999 %then
           %do;
           data _null_;
           put "ER""ROR";
           run;
           %end;


%inner;

%mend outer;
options mlogic mprint symbolgen ;
%outer(lib=sashelp
           ,dsn=class
           ,NumVar=age);
```

55: Avoid complex nested IF statements. Consider specifying all the IF criteria on every statement.

The logic of nested if statements can be hard to follow, especially when combined with poor layout.

Hard to read this nesting of IF statements.

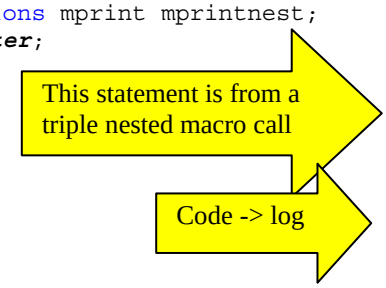
```
Data Complex;
Length Class $ 14;
Set sashelp.class;
If Age Lt 13 then
Do;
If Sex="M" then
Do;
If height GT 55 then Class="YoungMTall"
Else Class="YoungMTall"; end;
Else
Do;
If height GT 55 then Class="YoungFTall"
Else Class="YoungFTall";
End; end;
Else
Do; Class="Enough Already"; end;
   run;
```

Easy to read. Each if states all the info for the logic.

```
Data Simple;
length Class $ 14;
set sashelp.class;
If       age LT 13 and Sex="M"
    then Class="YoungMTall";
else if age GE 13 and Sex="M"
     then Class="YoungMTall";
Else if age LT 13 and Sex="F"
      then Class="YoungFTall";
else if age GE 13 and Sex="M"
      then Class="YoungFTall";
else       Class="Enough Already";;
run;

/* ;-(  THIS DOES REQUIRE EXTRA TYPING
BUT IS OFTEN WORTH THE EFFORT */
```

56: Learn to use the mprintnest option when nesting macros (and, by the way, avoid nesting macros).
This new feature gives additional information.

```
%macro outer;
     data _null_;
       %inner ;
       run;
%mend outer;


%macro inner;
    %inrmost;
%mend inner;

%macro inrmost;
  put 'Test the PUT';
%mend inrmost;

options mprint mprintnest;
%outer;
```




```
3015  %macro outer;
3016       data _null_;
3017          %inner ;
3018      run;
3019
3020  %mend outer;
3021  %macro inner;
3022      %inrmost;
3023  %mend inner;
3024  %macro inrmost;
3025      put 'This is the text of the PUT statement';
3026  %mend inrmost;
3027
3028  options mprint mprintnest;
3029  %outer;
MLOGIC(OUTER):  Beginning execution.
MPRINT(OUTER):   data _null_;
MLOGIC(INNER):  Beginning execution.
MLOGIC(INRMOST):  Beginning execution.
MPRINT(OUTER.INNER.INRMOST):   put 'Test the put';
MLOGIC(INRMOST):  Ending execution.
MPRINT(OUTER.INNER):   ;
MLOGIC(INNER):  Ending execution.
MPRINT(OUTER):   ;
MPRINT(OUTER):   run;

This is the text of the PUT statement
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      user cpu time       0.00 seconds
      system cpu time     0.00 seconds
      Memory                              135k

MLOGIC(OUTER):  ENDING EXECUTION.
```

57) Order your files in the SAS explorer in the same order that they are created. (also see 10, 22 and 51).

Please see appendix for a longer explanation of the naming convention.

If programs are divided into sections (Suggestion 51), and I suggest using the SECBLOCK abbreviation, consider naming files in using the logic below. Naming files this way adds a bit of typing, but I suggest that it makes it easy to find the files in the SAS explorer, and makes the program easier and faster to debug.

Preliminary step: Ordering files in the SAS work directory.

This appendix suggests two alpha ordered naming conventions (two levels of complexity). Both naming conventions require that that windows explorer order files in the SAS work directory in alphabetical order. This can be done in a few simple steps.

To have the SAS work directory sort into alphabetical order:

Open the windows explorer and navigate to any dir that has files in it.
Sort the files by name in ascending order
Select tools-> Folder options ---.> click on the view tab
Select "Apply to All Folders".

Now, files will be sorted alphabetically in all folders
– including the SAS work folder.

Note that SAS will not display file names as you create them.

```
data classOfGirls;
set sashelp.class(where=(Sex="F"));
run;
```

Is displayed as Classofgirls. The capitalization is deliberately changed.

It is suggested that the reader also make the files display alphabetically in the SAS explorer by selecting View-details and then clicking on the header of the name column to make the files appear alphabetically.

Second step: the file naming convention.

In this convention, it is suggested that program sections should be identified by lower case letters and not numbers I suggest that, in an ASCII system, original file names have the following structure:

The lower case section letter – a primary sorting letter – four underscores - a human readable name.

An example of a filename might be: aa____Sales42010.

The underscores allow the programmer to add code that creates files, in the middle of a program, and still have the files appear in the SAS viewer in the order in which they are created.

For a more complete example, if the section is Section a:

The first file created in section a, would be:	aa___SalesAtZipLevel	← naming logic: a is section a – a means first file
The second file created in section a would be:	ab___Zip2TerrMapping	← naming logic: a is section a – b means second file
The third file created in section a, would be:	ac___SalesWTerr	← naming logic: a is section a – c means third file
A later file, created in section c, might be:	cf___SalesWTerr	← naming logic: c is section f – f means fourth file

In the naming standard, provision must be made for easily adding code and files in the middle of an existing program. In the old days, cards (and I do remember cards) cards were numbered so they could be easily re-ordered if dropped (and they were dropped). The cards were originally numbered in multiples of 10 to allow the programmer to add new cards. If cards needed to be added between the cards 20 and 30 the new cards might be numbered 22 & 24. The goal was to be able to add new logic without re-typing lots of program code.

If we needed to modify a program that used the standard suggested by this paper, and if the change required inserting a file between aa___MeaningFulName and ab___ MeaningFulName2, the naming convention would be to replace one of the underscores with a letter. We would assign letters in a way that continues to keep the ASCII sorted in the order of creation.

The first file was:	aa___SalesAtZipLevel
We add a file	aaa__SalesRep4Terr
	▲ replace a dash with a letter–this sorts after aa___
We add a file	aab__Customers4Terr
	▲ replace a dash with a letter – this sorts after aaa__
The old second file was:	ab___Zip2TerrMapping
The old third file was:	ac___Sales_Terr_Rep_NoOfCust

A lot of modifications to the program might result in the following file names, where the dashes were replaced with a letter.

The first file was: aa___SalesAtZipLevel
We added aaa__SalesRep4Terr
We now add aaaa_AreadableName
▲replace a dash with a letter – this sorts after aaa__
We now add aaab_ AreadableName
▲ replace a dash with a letter – this sorts after aaaa
We added aab__Customers4Terr
We now add aaba_ AreadableName
▲ replace a dash with a letter – this sorts after aab__
We now add aabb_ AReadableName
▲ replace a dash with a letter – this sorts after aaba
The second file was: ab___Zip2TerrMapping
The third file was: ac___Sales_Terr_Rep_NoOfCust

If files are named in this manner, and if the programmer is working on an ASCII system that displays files in alpha order, the files will sort in the order in which they are created in the program. This naming convention involves a little more typing than naming files a1, a2 a3, but it will save the programmer time when tracking errors or taking over responsibility for production code.

Reason: If files are named in this manner, and if the programmer is working on an ASCII system, the files will sort in the order in which they are created. It is a little more typing but it will save the programmer time when debugging and when working through new code.

58) CREATE A MANAGEMENT DOCUMENT FOR PRODUCTION ENVIRONMENTS-SEE 38

The head of the department needs to be able to represent the department in meetings. It is the duty of the department to give the manager the information s/he needs to do her job.
One of the issues that a manager encounters is the approval of changes to files being accessed by the production programs in the department s/he manages. When this happens, the manager needs to have easy access to high level information about "production jobs" and it is suggested that the programmers create a document with the following layout. Since this document should be in an electronic format, it can be searched to find all the occurrences of a file.
Most of the information recorded in this document should also be in the "DocBlock", that is at the top of the program, so it should be should be easy to obtain when this document needs to be created.

Report name and responsible programmer (s)	Run date time	Source files	Source Program and Files created and customer(s)	Driver sheet or source file must be changed when:	note
Regional Sales Dashboard Pat Murdock Vijay Bhaskar	Sunday Evening: 1 AM Available Monday at 8 AM	Sales.NE, Sales.NW, Sales,SE,, Sales.SW ref.calendar ref.ProdCD2Name	S:\reports\RegSlsDash\prod\Progs\RegSlsDash.sas S:\reports\RegSlsDash\prod\reports\RegSlsDash_YYYYMMDD.xls Customers are: Regional Directors	1) Driver: new product is added or removed 2) Driver: when Regi. managers change 3) Calendar: changed every Jan 1st	
Monthly Sales Summary Chi-Yuan Lee Mark Lyons	First Wed of month: 1 AM Available First Thursday of month at 8 AM	Sales.NE, Sales.NW, Sales,SE, Sales.SW ref.ProdCD2Name	S:\reports\MoSlsSumm\prod\Progs\MoSlsSumm.sas S:\reports\MoSlsSumm\prod\reports\MoSlsSumm_YYYYMMDD.xls Customers are: Regional Directors, J. Blum, E. Davis	1) Driver: new product is added or removed 2) Driver: when Reg. managers change	

Reason: A document like this would allow a manager to be in a meeting, hear mention of a discussion of a change, and be able to tell the meeting "that change might affect the …. report that we create for J. Blum, E. Davis and the regional directors". The manager can then probe for details like implementation time for the change. A document like this would allow a high level manager to bring back, from meetings, warnings that files are about to be changed or that driver files will need to be changed – and a time frame for the change.

CONCLUSION

It is hoped that this paper will help us all get “BETTER, FASTER and CHEAPER”.

Why does management need to be involved in, and reward, good coding practices? Please consider these stories as answers.

In my first job out of grad school I worked for a consulting firm. While the president was doing a team building exercise, and exhorting us to be co-operative, the senior SAS programmer asked for the president’s opinion.

The senior programmer said: “If I help new people, it slows me down and decreases my performance – making me look bad. I help them build skills, it increases their output and makes them look good. It is my understanding, that you give the department head a fixed amount of money to be divided among the programmers as raises. It seems that, if I help new programmers, you will take money from me and give it to them. Why should I do that?”

The president had no answer.

One of my students came back from his summer job as a programmer with a funny story. He had been a programming intern for the accounting department for a major city. He said that three of the programmers had managed to write a section of the city payroll program in (undocumented) assembler language. After that the three programmers could not be fired or disciplined.

Suggested reading: Kerr, S., 1975, "On the Folly of Rewarding A, While Hoping for B,"
Academy of Management Journal, Vol. 18, pp. 769-783.

ACKNOWLEDGMENT

I have collected these ideas into a word document over the space of several years. Often the same idea would be mentioned in several papers, and by several different authors. I am sorry to say that most of the original sources have been lost. I offer my apologies, and my thanks, to all those who have written on this topic and have been unattributed sources for this collection of suggested “dos and don’ts”.

I suggest reading a paper that Louise Hadden presented at SUGI31

Hadeen, Louise, 2006. “SAS® programming Guidelines” *Proceedings of the Thirty-First Annual SAS Users Group International Conference* paper 121-31

CONTACT INFORMATION

Your comments and questions are valued and encouraged.
Contact the author at:
Russ Lavery Russ@Russ-Lavery.com
Ardmore PA

APPENDIX 1: A FURTHER EXPLANATION OF SUGGESTION 55 - A SUGGESTED FILE NAMING CONVENTION:

ABSTRACT:

Programming standards are not a matter of style but are means to reduce the cost and time of writing and maintaining programs. This file naming convention will reduce the cost of maintaining programs. Being a low-cost producer of programs is a competitive advantage for an employing company and for programmers. Leading a department of "fast program maintainers" allows a manager to meet goals.

This file naming convention is especially useful for "production programs". "Production", as opposed to ad-hoc, programs are run on a periodic basis. Production programs are complex and need periodic modifications as client needs change. Importantly, responsibility for production program maintenance is often transferred someone who did not write the program. Actually, maintenance responsibility is often transferred multiple times before the program's usefulness ends and it is discontinued. It is cost effective to invest some extra time, when creating production programs, to make them easier to maintain.

INTRODUCTION:

Long production programs can generate hundreds of working files. Stepping through a program, viewing the contents of the working files, and following the logic of the program can be slowed by the time it takes to find, in the SAS explorer, the data set that was created by the code just executed. The file naming convention presented in this paper will cause the files to appear in the SAS explorer in the order in which the files were created and make the program easier to understand and cheaper to maintain. It is the responsibility of management to decide on, teach and then reward the use of coding standards.

STRUCTURING THE PROGRAM FOR EASE OF MAINTENANCE:

SAS programs often have a tree-like structure that can be seen in Figure 1. Data is read from several sources and prepared, in several steps, for a merge. Some post-merge processing is often required to put the data in the form desired by the client. Often there is a final output stage.

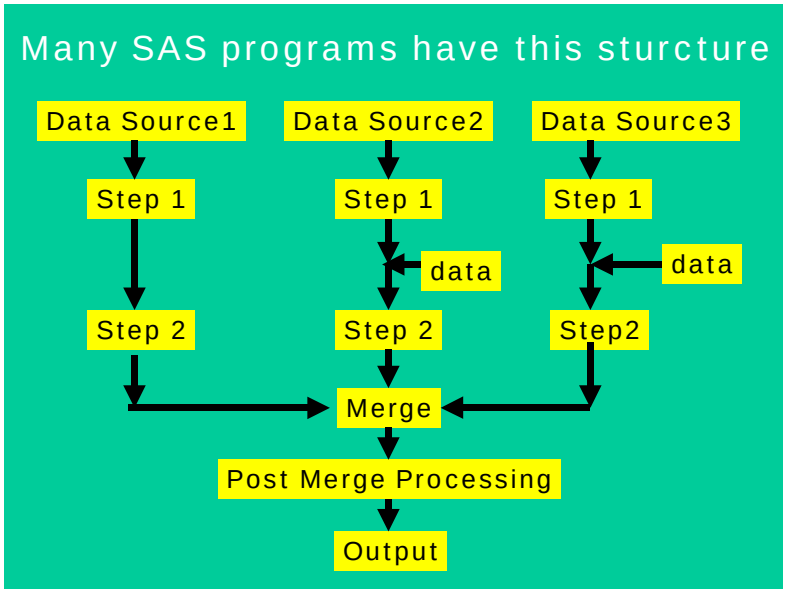


FIGURE 1

Regardless of the exact structure of the program, it can be broken down into sections. The sections can be defined in several ways (see FIGURE2 and FIGURE3) as the logic of the program "appears" to the programmer.

In FIGURE 2, the programmer has thought of the problem as a problem of preparing data sets for merging. In FIGURE 3, the programmer has thought of the problem as on of performing (fairly) similar steps on a number of input files. Both approaches are valid.

The author has found that breaking a program into sections, drawing a flowchart in an external document and annotating the sections in the code (so that the sections in the code can be "mentally linked" to the flowchart) has greatly reduced the cost/time to transfer a program to a new programmer.

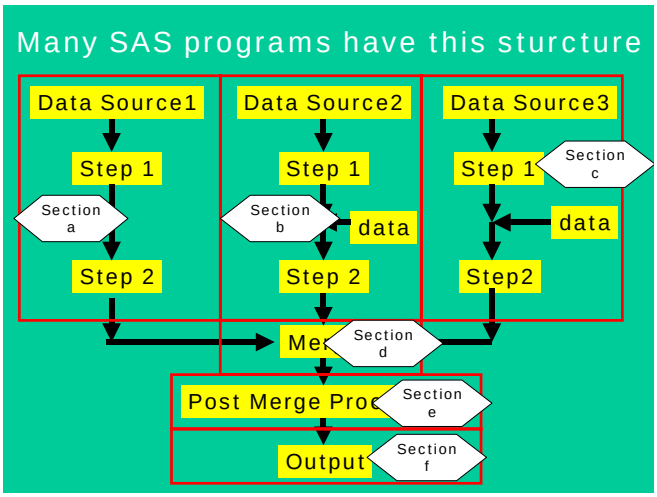


FIGURE 2

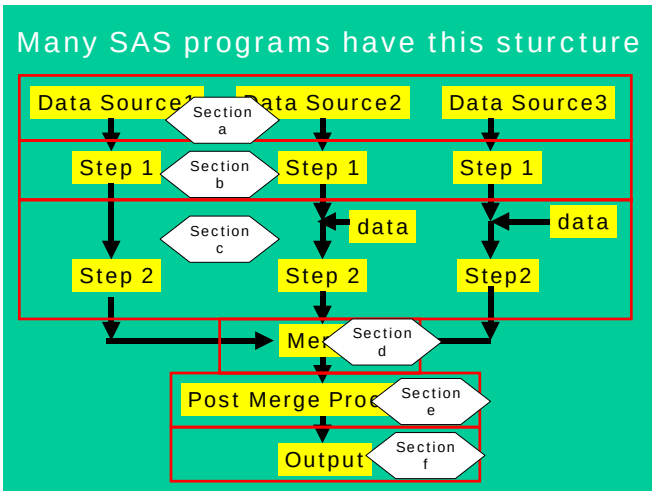


FIGURE 3

TWO ALPHA BASED NAMING CONVENTIONS (SIMPLE AND MORE COMPLEX) FOR FILES

It is generally conceded that file names should give information about the contents of the file. The fact that file names like a1, a2 and a3 are poor programming practice is rarely debated any more. The position set forth in this paper is there is a positive payback (time expended in extra typing is much less than time saved in transferring a program) for the effort of creating longer and more meaningful data set names. The author feels that anyone writing one-time ad-hoc programs can use file names like a1 and a2, if s/he desires, but that a department policy should prevent files like that being transferred to another programmer. For production programs, the author feels that some extra effort in structuring programs and naming work files has great payback.

PRELIMINARY STEP: ORDERING FILES IN THE SAS WORK DIRECTORY.

This appendix suggests two alpha ordered naming conventions (two levels of complexity) and mentions a third convention. The first two naming conventions require that that windows explorer order files in the SAS work directory in alphabetical order. This can be done in a few simple steps.

To have the SAS work directory sort into alphabetical order:

Open the Windows explorer and navigate to any directory that contains files.

Sort the files by name in ascending order

Select tools-> Folder options ---.> click on the view tab

Select “Apply to All Folders”.

Now, files will be sorted alphabetically in all folders – including the SAS work folder.

It is suggested that the reader also make the files display alphabetically in the SAS explorer by selecting View-details and then clicking on the header of the name column to make the files appear alphabetically.

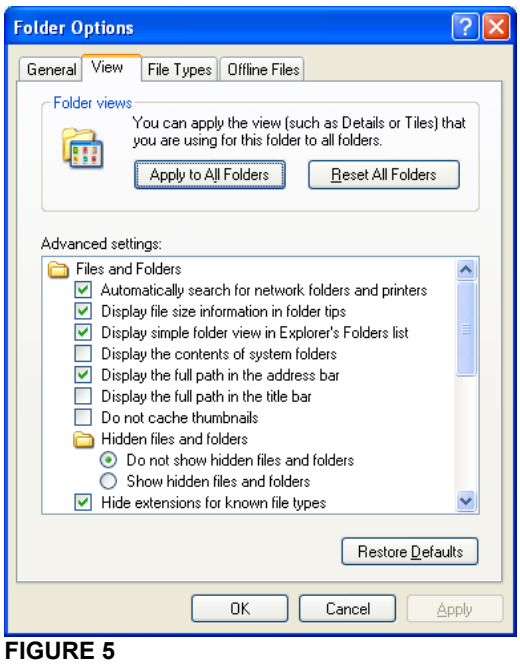


FIGURE 5

1 OF 3) A SIMPLE PROGRAMMING/NAMING CONVENTION

In the first naming convention, it is suggested that as the programmer creates the program in sections and s/he adds a simple two character prefix (a letter and an underscore) to the file name. The prefix **will simply identify the section in which the file was created**. I suggest that letters be used to identify sections because one letter (one keystroke) can have 26 values and one number (one keystroke) can only have ten values. Using letters to identify sections will allow a programmer to have 26 sections without having to use two keystrokes. As will be seen below, in ASCII, numbers sort before the dash and using letters makes for flexibility in the creation of the naming of files. In this simple programming convention the leading letter can be uppercase and the underscore is simply a visual separator between the section letter and the “informative file name”

Examples of naming files with the simple convention might be:

For a file created in section A:

```
Data A_RawDataNational;    /*  "A" is for for section A"  */
```

For a file created in section C:

```
Data C_NationalTransposed; /*  "C" is for for section C"  */
```

This simple, and fast, naming prefix will group all the files, in the SAS viewer, by the program section that created it and sort files within a section sorted in alpha order. This is not the convention recommended by this appendix but, if the program has a large number of logical sections, this simple "gross" grouping of files by section, will make it much easier for a new person to function independently when doing program maintenance.

In the situation where the program maintenance is being transferred to another programmer, a program flowchart, and this naming convention, will make it easier for a senior programmer to delegate a one-time maintenance task to a junior programmer. The task can be delegated by saying things like:

"We need to make a change to section A of the program. In section A, bring in the variable "x" from the file …. and do ….. before you do the merging in section F. We want to have the output look like this example, so you will need to do ….. in section H." This is a big help to a new programmer because s/he knows that s/he does not need to change anything in the program except sections A,F and H.

In order to make changes to a program, a programmer must understand the program and this naming convention, combined with a flowchart, greatly reduces the degree of understanding/training that is required to delegate maintenance tasks. Having programs that are easier to understand, and files that are faster to find, reduces programming costs.

2 of 3) The suggested program structure and file naming convention:
In this second, and higher level, naming convention, it is suggested that the programmer structures the program as above and then adds a simple five-character prefix to file names.

The prefix:

1) Identifies the section in which the file was created.
2) Causes files to appear in the SAS viewer in the order in which the files were created
3) Allows for new code to be inserted and still have the files sort in the order in which they were created. The benefit of this is that files are very fast to find in the SAS explorer.

Even programmers, that the author has seen habitually use file name like a and a2, have commented that this file naming convention, and a program flowchart, make for easy-to-understand programs. However, adoption of the convention, by those people, seems to be another matter. One can only wait and hope.

The key to understanding the suggested programming structure, and file naming convention, is the sorting order of ASCII characters. Characters are sorted, in ASCII in the order shown below. Note that not all ASCII characters are allowed in SAS file names. Please focus on the characters in red.

```
blank ! " # $ % & ' ( ) * + , - . /0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z[ \] ^
_
a b c d e f g h i j k l m n o p q r s t u v w x y z
{ } ~
```

Note that all upper case letters sort before any lower case characters (**and before the underscore**) and that the _ sorts before any lower case letter. The underscore is a commonly used to provide "visual separation" in file names. Since the underscore sorts before any lower case letter the proposed convention requires the use of lower case letters in the prefix of the fine name. In ASCII,

a_ sorts before aa or ab
and
a___ sorts before aa_ or ab_.

We will use these facts in our naming convention and is required that programmers use lower case in creating the prefix. The _ will sort before any lower case letter and, therefore , can be used as a filler/spacer character and to control sorting of the file names. The use of _ will allow us to do create new files and still have the files appear in order of creation.

In this convention it is suggested that program sections should be identified by letters and not numbers

I suggest that, in an ASCII system, original file names have the following structure:

a lower case section identifying letter
– a lower case primary sorting letter
– four underscores
- a human readable name.

An example of a filename might be: aa____Sales2010.

Note that SAS will not display file names as you create them.

```
data classOfGirls;
set sashelp.class(where=(Sex="F"));
run;
```

Is displayed as Classofgirls. The developer likes this change of capitalization.

The underscores allow the programmer to add code that creates files, in the middle of a program, and still have the files appear in the SAS viewer in the order in which they are created.

For a more complete example, if the section is Section a:

The first file created in section a, would be:	aa___SalesAtZipLevel	← naming logic: a is section a – a means first file
The second file created in section a would be:	ab___Zip2TerrMapping	← naming logic: a is section a – b means second file
The third file created in section a, would be:	ac___SalesWTerr	← naming logic: a is section a – c means third file
A later file, created in section c, might be:	cf___SalesWTerr	← naming logic: c is section c – f means fourth file

In the file naming standard, provision must be made for easily adding code and files in the middle of an existing program. A convention from the days of cards is modified and used here. In the old days, cards (and I do remember cards) cards were numbered so they could be easily re-ordered if dropped (and they were dropped). The cards were originally numbered in multiples of 10 to allow the programmer to add new cards. If cards needed to be added between the cards 20 and 30 the new cards might be numbered 22 & 24. The goal was to be able to add new logic without re-typing lots of program code.

If we needed to modify a program, that uses the standard suggested by this paper, and if the change required inserting a file between aa___MeaningFulName and ab___ MeaningFulName2, the naming convention would be to replace one of the underscores with a letter. We replace underscores with letters in a way that continues to keep the files sorted in the order of creation.

The first file was: aa___SalesAtZipLevel
We add a file aaa__SalesRep4Terr
▲ naming logic above: replace a dash with a letter – this sorts after aa___
We add a file aab__Customers4Terr
▲ naming logic above: replace a dash with a letter – this sorts after aaa__
The old second file was: ab___Zip2TerrMapping
The old file was: ac___Sales_Terr_Rep_NoOfCust

A lot of modifications to the program might result in the following file names, where the dashes were replaced with a letter.

The first file was: aa___SalesAtZipLevel
We had added aaa__SalesRep4Terr
We now add aaaa_AreadableName
▲ naming logic above: replace a dash with a letter – this sorts after aaa__
We now add aaab_ AreadableName
▲ naming logic above: replace a dash with a letter – this sorts after aaaa
We had added aab__Customers4Terr
We now add aaba_ AreadableName
▲ naming logic above: replace a dash with a letter – this sorts after aab__
We now add aabb_ AReadableName
▲ naming logic above: replace a dash with a letter – this sorts after aaba
The second file was: ab___Zip2TerrMapping
The third file was: ac___Sales_Terr_Rep_NoOfCust

If files are named in this manner, and if the programmer is working on an ASCII system that displays files in alpha order, the files will sort in the order in which they are created in the program. The naming convention is a little more typing than naming files a, a2 a3, but it will save the programmer time when tracking errors or taking over responsibility for production code.

AN ALTERNATIVE THAT DOES NOT INVOLVE CHANGING THE SORT ORDER OF FOLDERS:

An alternative does exist if the programmer does not want to change the sort order of folders.

If the programmer wants to avoid the issue of ASCII alphabetical sorting, a numeric prefix naming convention can be adopted.

The logic for the prefix could be:

S (file names must start with a letter or underscore, so why not "S" for section)

a two digit number (to identify the section)

followed by an underscore (as a visual separator)

followed by a string of four zeros(the zeros get replaced by numbers as files are added to the program)

followed by an underscore (as a visual separator)

followed by an informative file name.

```
/*Section 1*/
Data S01_1000_ReadRaw;
X=1;
Run;
/*Section 2*/
Data S02_1000_SomeStep2;
x=1;
run;

Data S02_2000_SomeStepaddedLater;
x=1;
run;
Data S02_3000_SomeStepaddedStillLater;
x=1;
run;
/*Section 3*/
Data S03_1000_SomeStep3;
x=1;
run;
```