SAS Coding Efficiency Notes

Author: Ou Zhang,

Last update: 8-15-2017

Paper 1: Getting more for less: A few SAS programming efficiency issues

Requirement:

1. Should the program be easy to maintain?
2. Should it execute as fast as possible?
3. Should the data base size be minimized?
4. Do system resources such as memory need to be conserved?

Take-home message:

1. Evaluate the structure of the program to minimize the number of individual steps.
2. Combine DATA steps and PROC steps when possible
3. Minimize the number of data set reads
4. Minimize the use of PROC SORT
5. Use table lookup techniques (i.e., use IF ELSE instead of multiple IF command) –conditional branching
6. Write the LOG to a file (ALTLOG=, or PROC PRINTTO)
7. Use the WHERE statement/option instead of IF
8. Data compression (whether use COMPRESS= or not)
9. Compiled stored macros (depends on macros)
10. Use the KEEP= and DROP= data set options
11. Control variable length

Paper 2: Writing Efficient SAS Codes

There are two aspects of efficiency:

- Efficient use of computing resources
- Efficient use of human resources

Take-home message:

1. Sometimes, if possible, overwriting same dataset and variable (to release SAS from holding too many data on disk
2. Using numeric variable names (Using numbers at the end of variable names rather than characters)

   It does make a big difference to human resources (typing and looking up field names)

```
Data one; input
        Q1 Q1b Q1c Q1d Q1other
        Time_SH Time_Wk
        Com_Ex Web_Ex Res_Ex
        Q4a Q4b Q5c Q5d;
cards;
```

```
Data one;
        input Q1-Q16;
cards;
```

   One may argue that the dataset might have a set of meaningful item ID and the programmer should not alter them arbitrarily. This issue can be easily solved by assigning a set of temp id for data manipulation and analysis, and then replacing the temp ID with the original ID in the final output.

3. Using a value list in a variable
4. PROC TRANSPOSE vs Array (Use PROC TRANSPOSE wisely)

****

From a web link: http://www.sascommunity.org/wiki/Good_Programming_Practice_for_Clinical_Trials

- Insert parentheses in meaningful places in order to clarify the sequence in which mathematical or logical operations are performed.

  Example:

```
data test02;
  set test01;
  if (visit=0  and vdate lt adate1)
  or (visit=99 and vdate gt adate2) then delete;
run;
```

- Avoid unnecessary sorting. CLASS statement can be used in some procedure to perform by-group processing without sorting the data.

  Example:

```
proc means data=osevit;
  var prmres;
  class treat;
run;
```

- If possible (i.e. not a sorting variable), use character values for categorical variables or flags instead of numeric values.

Reason: It saves space. A character “1” uses one byte (if length is set to one), whereas a numeric 1 uses eight bytes.

- Use the LENGTH statement to reduce variable size.

Reason: Storage space can be reduced significantly. Note: Keep in mind that a too limited variable length could reduce the robustness of the code (lead to truncation with different sets of data).

- Use simple macros for repeating code.
- The safest comment to use inside a macro is the PL/1 style comments in the form of /* comment */. Since this style of comment is read character by character, special characters and macro triggers such as % and & do not cause a problem and are truly seen as a comment.

  Example:

```
/* This comment's bytes are processed character by character. */
```

Paper 3: SAS Programming guidelines (Very useful)

Purpose of Programming Guidelines

- 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

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

Section 1: Naming convention

1. Programs can be named sequentially indicating their sequence

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

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

4. 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
   a. Truncated names
      If you wish to shorten names, truncate rather than abbreviate.
   b. IN=name
      Name each IN variable using IN plus the 1st letter of the dataset name
   c. Array names
      Begin array names with an underscores.

      Reason: It will distinguish them from regular variable names.

      Example: ARRAY _X (10) X1-X10;

   d. Format Names
      If a format applies to only one variable, then name the format with the variable name plus FMT or name the format with a description plus FMT.

Section 2: Compatibility

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

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

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

2. Options
   - Do not hard-code LINESIZE and PAGESIZE (They vary by printer)
   - Do not use COMPRESS as a global option (COMPRESS does not always save space)
3. Character sets
   - Do not use non-printing characters or special fonts (They vary by printer and platform).

```
Use:

        IF X  NOT=  Y;
        or
        IF X  NE  Y;

instead of:

        IF X  ^=  Y;
```

Section 3: Documentation

Clear and descriptive internal program documentation is essential.

1. Program Header
   Place a program header at the beginning of every program (This is the introduction to the program. It provides an overview of the purpose of the program and its position).

2. Dataset Labels
   Use a dataset label when saving a permanent SAS dataset. When others use the data, they will know what it is.

3. Variable Labels
   Use variable labels when saving a permanent SAS dataset. When others use the data, they will know what the variables are.

4. Comment
   - 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

- Do not over-comment

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

6. DATA=on PROC Statement
   Always use DATA= on PROC statement

7. Intermediate Output
   When creating intermediate output, always use a title statement. It keeps track of what you are looking at in the output file.

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

PROC PRINT DATA=TRANDATA;
TITLE 'Transposed Data';
RUN;
```

Section 4: Appearance

1. Single statement per line
2. Blank lines
   - Skip a line before every DATA and PROC statement
   - Separate blocks of code
     It makes the program readable. It clearly distinguishes different operations.
3. Alignment and Indentation
   - Left-justify DATA, PROC, OPTIONS statement, Indent all statement within.
   - Indent statement within a DO loop. Align END with DO.
   - Indent statement within an IF block. Align END with IF and ELSE.

```
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 statement.

```
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  */
```

4. Placement of Statement
   Use a standard sequence for placing statements and group like statement together
   - 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.

5. Date Formats
   Use date formats when reporting SAS dates. They are easier to interpret than SAS date values. They will ensure that you have the date that you think you have.

6. INPUT Statement
   Use @col VARNAME for input when possible. For all types of input, use a separate line for each variable.
   Reason:
   - It is easier to see.
   - It is easier to change.
   - It is easier to comment out lines for testing.
     Example:

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

INPUT  A1 $
       A2
       A3 $
       ;
```

7. 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.

Section 5: Efficiency

1. DROP and KEEP
   - When inputting a flat file, input only the variable 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*)
   - DROP intermediate variables used for calculations.
   - When outputting a dataset, KEEP only the variable needed.
2. 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.

3. 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.

4. 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.

5. IF/ELSE conditions
   Use ELSE statement 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.

6. Sort: Sort only the variable needed

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

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

   Use:

   ```
           AFLAG = '1'
   ```

   instead of:

   ```
           AFLAG = 1;
   ```

Section 6: Maintainability

The most important reason for making a program clear and understandable is to make it maintainable. A program is also easier to maintain if the log is easy to follow.

1. Use of constant
   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.

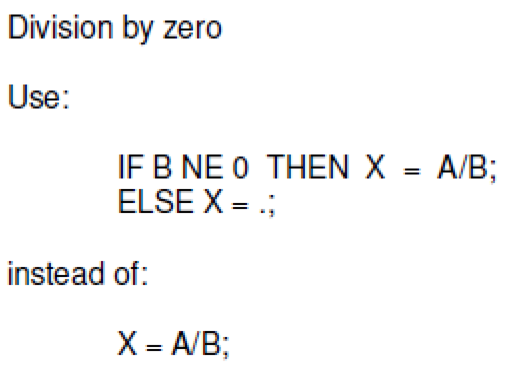
2. Nested calls
   Use no more than 2 layers of nested macro calls.
   Reason: it is too easy to get lost after 2 calls.

3. Output
   Do not create permanent datasets scattered within the program. Create them all at the end of the program.
   Reason: Placing output statement 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 tried to test it, he/she may write over data because it was not know 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.

4. Clarity
   - Avoid unnecessary notes or warning messages in the log.
   - Avoid uninitialized variables
   - 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.
   - Avoid automatic formatting (This can sometimes cause loss of data. Fix the program to use the correct format.)
   - Avoid excessive repetition of error messages.
     Example: OPTIONS ERRORS=2;

   - Use OPTIONS NOOVP;
     This will eliminate the triple (overprinted) error messages.

5. Exception Handling
   - Check for violation of correct conditions.
     Reason: it will avoid the error message if the conditions is violated. It will allow you to track the occurrences of violations.



Division by zero

Use:

```
IF B NE 0  THEN  X  =  A/B;
ELSE X = .;
```

instead of:

```
X = A/B;
```

6. Unambiguous Merging
    - Always use a BY statement
    - Never have the same variables on more than one datasets (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.

7. 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.

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

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

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

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

Section 7: Macros

1. 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 parameters.
   - The routine requires programming logic that cannot be included in a DATA step.
2. 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.

3. Macro Parameters
   Use general but self-documenting names for macro parameters.
   ```
   %MACRO CALCMAC(DSN=,STARTDATE=);
   ```
4. 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

5. Positional parameters vs named (keyword) parameters
   If you have one or two parameters, them it is ok to use positional parameters. But if there are more, then it is best to use named (keyword) parameters.
   Reason:
   - 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=);
     ```

6. Global vs local parameters
   Use local macro parameters rather than global parameters, when possible.
   Reason:
   - Avoid 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.

7. Macro comments
   - Use macro comment %*
     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.

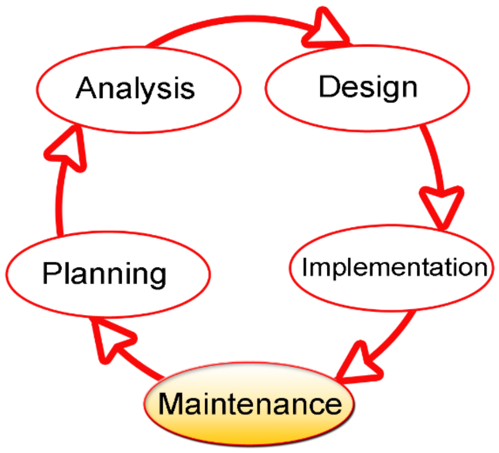
8. Macro options
   When you program with macros, use the options MPRINT, SYSMBOLGEN, and MLOGIC.

   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.

Paper 4: Guidelines for Coding of SAS Programs

SAS Coding needs to find the right balance between computer efficiency and maintainability.



Software development life cycle methodology

- Don't place more than one programming statement on a single line.
- Make sure that titles and footnotes used in printed reports accurately reflect the information being presented.
- Make sure to use date formats when reporting SAS dates.
- Do not extend any statement beyond column 72.

Simplify complicated code and complex expression:

- Insert parentheses in meaningful places in order to clarify the sequence in which mathematical or logical operations are performed,
- Break really complicated statements into a number of simpler statements,
- Don't be too clever,
- Even if you think that it may be more elegant to do so, refrain from using unconventional, obscure, and convoluted logic, except when you can't think of a simpler approach and, if you must, then insert lots of explanatory comments,
- Avoid implicit coding -- know your defaults, but don't overdo invoking them implicitly (for example, provide names on DATA steps and PROC steps that create data sets, and specify data set names as inputs to SAS procedures),

1. Group the executable statements.

   Place most of the non-executable statements in a DATA step before all of the executable statements. In particular, place variable attribute and other declarative statements near to the top of the DATA step, and ahead of the executable statements.

Paper 5: 50 Ways to Make Your SAS® Code Execute More Efficiently

Section 1: Processing more than one file in each DATA step

1. Read the data once and write it twice to two different output datasets:

```
Data temp_file_1
   temp_file_2;
        Set Perm.input_file;
If a = ‘OK’ then output temp_file_1;
           else output temp_file_2;
Run;
```

2. Build three data sets, each with different records and variables from the same file. But, read the input file only once.

```
Data
temp_file_1 (keep=var1 var2 var3)
temp_file_2 (keep=var4 var5 var6)
temp_file_3 (keep=var7 var8 var9);
     Set Perm.input_file;
     If a eq ‘OK’ then output temp_file_1;
     If b eq ‘OK’ then output temp_file_2;
     If c eq ‘OK’ then output temp_file_3;
Run;
```

3. When the input dataset is the same for all of the tables that are created the code can be reduced to the following. This also reduces processing time because only one pass over the input dataset is required.

```
        proc freq data = sashelp.shoes;
                table region / list out=region_freq1;
                table region*product/ list out=region_freq2;
                table region*stores / list out=region_freq3;
  run;
```

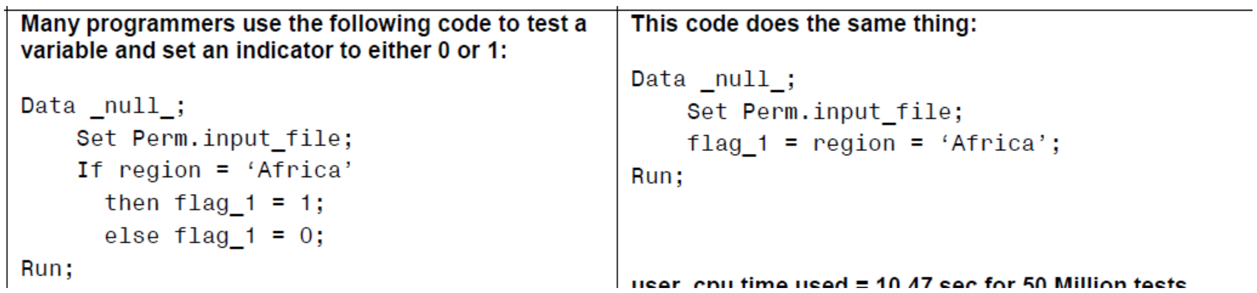
4. subset multiple data files and then merge the files with code

```
data file_all;
        set sample_data1 (where=(var1='Y') in=a)
            sample_data2 (where=(var2='Y') in=b)
            sample_data3 (where=(var3='Y') in=c);
        select;
                when (a) then file_numb = 1;
                when (b) then file_numb = 2;
                when (c) then file_numb = 3;
        otherwise;
        end;
```

```
run;
```

Section 2: Combining steps to make simple tasks take less code

1. Auto-define binary conditional variable



Many programmers use the following code to test a variable and set an indicator to either 0 or 1:

```
Data _null_;
    Set Perm.input_file;
    If region = ‘Africa’
      then flag_1 = 1;
      else flag_1 = 0;
Run;
```

This code does the same thing:

```
Data _null_;
    Set Perm.input_file;
    flag_1 = region = ‘Africa’;
Run;
```

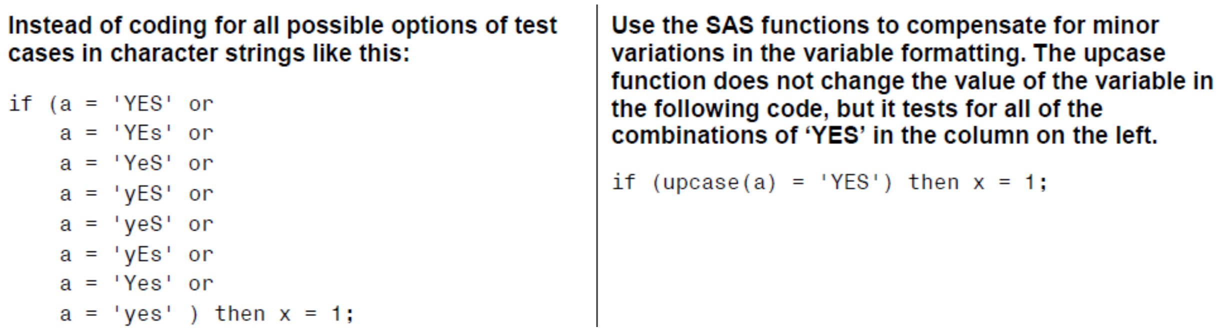
user cpu time used = 10.47 sec for 50 Million tests

Section 3: Using Macro variables to simplify maintenance

1. Note here the macro variable was used as part of a variable name and as a numeric constant, only one change is required to change the array size and usage.

```
%let max_size = 15;
Data array_test;
     Array counters {&max_size} var_01-var_&max_size;
     Do i = 1 to &max_size;
     Counters(i) = 0;
End;
```

Section 4: Using built in features rather than writing your own code

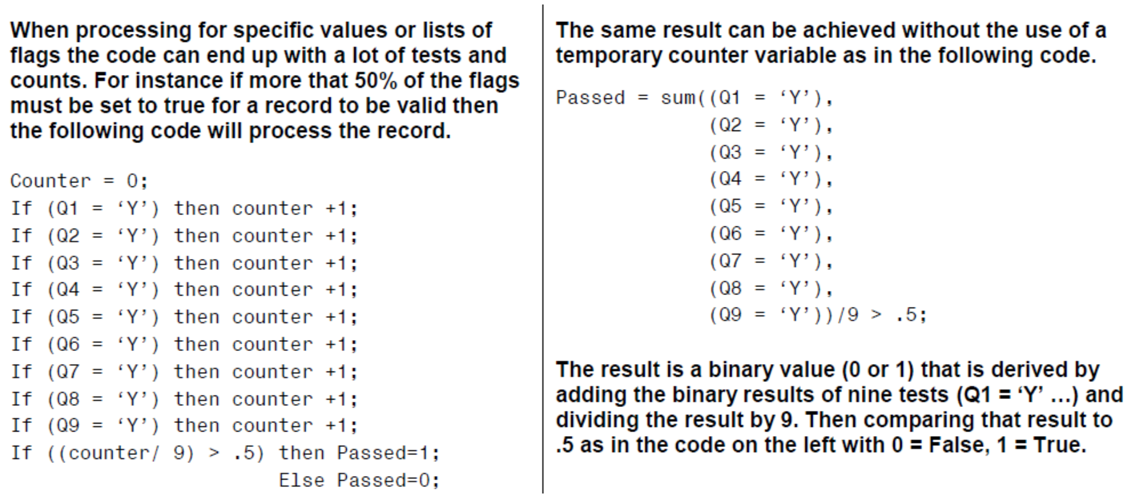


Instead of coding for all possible options of test cases in character strings like this:

```
if (a = 'YES' or
    a = 'YEs' or
    a = 'YeS' or
    a = 'yES' or
    a = 'yeS' or
    a = 'yEs' or
    a = 'Yes' or
    a = 'yes' ) then x = 1;
```

Use the SAS functions to compensate for minor variations in the variable formatting. The upcase function does not change the value of the variable in the following code, but it tests for all of the combinations of ‘YES’ in the column on the left.

```
if (upcase(a) = 'YES') then x = 1;
```


When processing for specific values or lists of flags the code can end up with a lot of tests and counts. For instance if more that 50% of the flags must be set to true for a record to be valid then the following code will process the record.

```
Counter = 0;
If (Q1 = 'Y') then counter +1;
If (Q2 = 'Y') then counter +1;
If (Q3 = 'Y') then counter +1;
If (Q4 = 'Y') then counter +1;
If (Q5 = 'Y') then counter +1;
If (Q6 = 'Y') then counter +1;
If (Q7 = 'Y') then counter +1;
If (Q8 = 'Y') then counter +1;
If (Q9 = 'Y') then counter +1;
If ((counter/ 9) > .5) then Passed=1;
                            Else Passed=0;
```

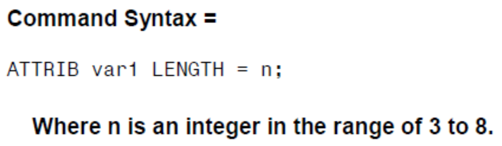
The same result can be achieved without the use of a temporary counter variable as in the following code.

```
Passed = sum((Q1 = 'Y'),
             (Q2 = 'Y'),
             (Q3 = 'Y'),
             (Q4 = 'Y'),
             (Q5 = 'Y'),
             (Q6 = 'Y'),
             (Q7 = 'Y'),
             (Q8 = 'Y'),
             (Q9 = 'Y'))/9 > .5;
```

The result is a binary value (0 or 1) that is derived by adding the binary results of nine tests (Q1 = 'Y' ...) and dividing the result by 9. Then comparing that result to .5 as in the code on the left with 0 = False, 1 = True.

Section 4: Ways to save disk space

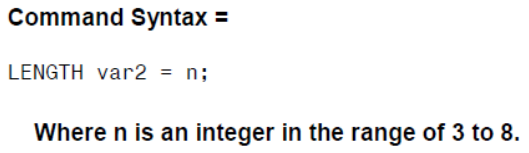
1. Use the SAS ATTRIBUTE command to change the size of numeric variables, but remember that the maximum size of the number is smaller if the size of the variable is smaller.


Command Syntax =

```
ATTRIB var1 LENGTH = n;
```

Where n is an integer in the range of 3 to 8.

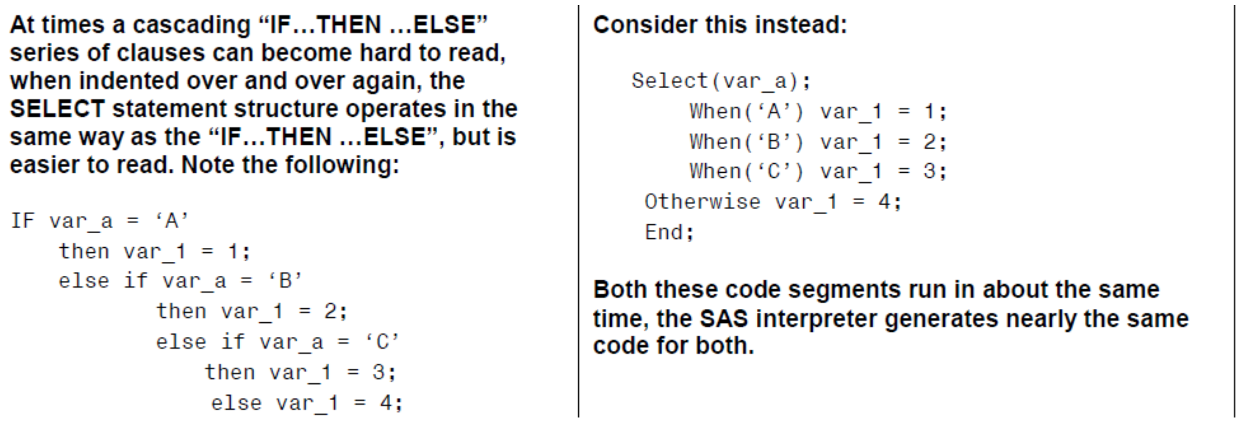
2. Use the SAS LENGTH command to change the size of numeric variables, but remember that the maximum size of the number is smaller if the size of the variable is smaller.


Command Syntax =

```
LENGTH var2 = n;
```

Where n is an integer in the range of 3 to 8.

Section 5: Ways to make the program code just read better


At times a cascading "IF…THEN …ELSE" series of clauses can become hard to read, when indented over and over again, the SELECT statement structure operates in the same way as the "IF…THEN …ELSE", but is easier to read. Note the following:

```
IF var_a = 'A'
    then var_1 = 1;
    else if var_a = 'B'
            then var_1 = 2;
            else if var_a = 'C'
                then var_1 = 3;
                 else var_1 = 4;
```

Consider this instead:

```
  Select(var_a);
       When('A') var_1 = 1;
       When('B') var_1 = 2;
       When('C') var_1 = 3;
   Otherwise var_1 = 4;
   End;
```

Both these code segments run in about the same time, the SAS interpreter generates nearly the same code for both.

3. MATCHING "ENDS"

Paper 6: Top Ten SAS® Performance Tuning Techniques

Section 1: CPU Time

1. Use KEEP= or DROP= data set options to retain desired variables.
2. Use WHERE statements, WHERE= data set option, or WHERE clauses to subset SAS datasets.
3. Create and access SAS datasets rather than ASCII or EBCDIC raw data files.
4. Use IF-THEN/ELSE or SELECT-WHEN/OTHERWISE in the data step, or a Case expression in PROC SQL to conditionally process data.
5. Turn off the macro facility when not needed
6. Avoid unnecessary sorting-plan its use.
7. Use procedures that support the CLASS statement to take advantage of group processing without sorting.
8. Use the Stored Program Facility for complex data steps.
9. CPU time and elapsed time can be reduced with the SASFILE statement.
10. Use DATA step hash programming techniques to merge (or join) SAS datasets.

Section 2: Data Storage

1. Use KEEP= or DROP= data set options to retain desired variables.
2. Process only the variables you need which removes unwanted variables from the program data vector (PDV).
3. Use LENGTH statements to reduce the size of a variable.
4. Use data compression strategies to reduce the amount of storage used to store datasets.
5. Create character variables for data that won't be used for analytical purposes.
6. Shorten data by using informats and formats.
7. Use a DATA _NULL_ when writing to external files.
8. When the default physical BLKSIZE of 6KB is used more DASD space is often needed to hold a specified amount of data.
9. When insufficient disk space is unavailable to perform a sort process, consider using the SORT procedure's TAGSORT option.
10. Remove unwanted SAS datasets with PROC DATASETS.

Section 3: I/O

1. Read only data that is needed from external data files.
2. Minimize the number of times a large dataset is read by subsetting in a single DATA step.
3. Use KEEP= or DROP= data set options to retain only desired variables.
4. Use WHERE statements to subset data.
5. Use data compression for large datasets.
6. Use the DATASETS procedure COPY statement to copy datasets with indexes.
7. Use the SQL procedure to consolidate steps.
8. Store data in SAS datasets, not external files to avoid excessive read processing.
9. Perform data subsets early to reduce the number of reads.
10. Use indexed datasets to improve access to data subsets.
11. Use the OUT= option with PROC SORT to reduce I/O operations.
12. Experiment with different values for the BUFNO= option to adjust the number of open page buffers when processing SAS datasets.

Section 4: Memory

1. Read only data that is needed.
2. Process only the variables you need which removes unwanted variables from the program data vector (PDV).
3. Use WHERE statements, WHERE data set options, or WHERE clauses to subset datasets when possible.
4. Avoid storing SAS catalogs in memory because they consume large quantities of memory.
5. If using arrays, create them as _TEMPORARY_ to reduce memory requirements.
6. Increase the REGION size when the amount of available memory is insufficient.
7. Use the SORTSIZE= system option to limit the amount of memory that is available to sorting.
8. Use the SUMSIZE= system option to limit the amount of memory that is available to summarization procedures.
9. Use the MEMSIZE= system option to control memory usage with the SUMMARY procedure.
10. Use the MVARSIZE= system option to specify the maximum size of in-memory macro variable values.

Section 5: Programming Time

1. Use the SQL procedure for code simplification.
2. Use procedures whenever possible.
3. Document programs and routines with comments.
4. Utilize macros for redundant code.
5. Code for unknown data values.
6. Assign descriptive and meaningful variable names.
7. Store formats and labels with the SAS data sets that use them.
8. Use the DATASETS procedure COPY statement to copy data sets with indexes.
9. Test program code using "complete" test data.
10. Assign redundant steps to function keys, particularly during debugging and tuning operations.

/***/

From Webpage:
http://support.sas.com/documentation/cdl/en/lrcon/69852/HTML/default/viewer.htm#n0a1u9b2buxl5yn1nv12rnoppiip.html

Techniques for Optimizing I/O

Using the OBS= and FIRSTOBS= Data Set Options

You can also use the OBS= and FIRSTOBS= data set options to reduce the number of observations processed, especially for the big data.

When you create a temporary data set and include only the necessary observations, you can reduce the number of I/O operations that are required to process the data. See FIRSTOBS= Data Set Option in SAS Data Set Options: Reference and OBS= Data Set Option in SAS Data Set Options: Reference for more information.

Paper 7: Tips and Techniques for the SAS® Programmer

These are the topics identified by the acrostic POWER.

- P – Procedures
- O – Output
- W – Work Habits
- E – Efficiency
- R – Resources

Functions

The IFC or IFN function may be more convenient than using an IF/ELSE statement. IFC returns a character value and IFN returns a numeric value. For example, instead of using

```
        if results > 70 then grade = 'pass';
        else grade = 'fail';
```

you can use this instead:

```
Grade = ifc(results > 70,'pass','fail'); /* store character value */
```

To check for number of values that are missing, use the NMISS function

```
if nmiss(of q1-q20)>7 then delete;
```

The %SYSFUNC, %QSYSFUNC macro functions can execute most SAS functions and return the results with an optional format. Here is an example to put the current date and time in a desired format by using the DATE and TIME functions in %SYSFUNC.

```
title "%sysfunc(date(),worddate.)";
title2 "at %sysfunc(time(),time.) ";
```

Results: April 7, 2011
at 08:00:00

The IFN and IFC functions can also be used in %SYSFUNC. The macro variables minAge and maxAge are created in the following PROC SQL code and then used to create the footnote with the &SYSFUNC and IFC function. The footnote will have the value of either 'Ages OK' or 'Out of Range' depending on the range of age values. In the SQL code, the macro variables minAge and MaxAge are proceed by a colon (:).

```
proc sql;
   select min(age), max(age)
   into :minAge, :maxAge
   from sashelp.class;
quit;

%put Min age is &minAge Max age is &maxAge;

"%sysfunc(ifc(&minAge>11 and &maxAge<17,'Ages OK','Out of Range'))";
```

Selection of concepts and organizing principles focus on these 4 areas:

- Documentation
- Processing
- Programming guidelines
- File storage

Modular Programming:
Modular programming is used to break up a large program into manageable units.

Keep an open mind:

- The "one way" was the program that was quickly written without forethought.
- The "simpler way" was the program that resulted after some thought and a desire to keep it simple.

EFFICIENCY:
What Matters Most? What are the tradeoffs?

- computer time
- memory
- disk storage
- computer memory
- time for coding
- time in maintaining the program

Macro:

We all know it is a good idea to document our SAS code with comments. To document your macro code, use the macro comment (**%* … ;**). Better yet, use delimited comments (/* … */). Neither macro comments nor delimited comments (**/* … */**) are passed out of the macro, when it is resolved.

At some point, you will start writing more complex macro code. When you start writing complex macro code with %do loops, %if's, multiple ampersands, and complicated quoted strings, you might want to see the generated SAS code without wading through all the log comments. You can do this by directing the code that is generated by the macro to a specified file by using the MFILE and MPRINT system options. The syntax is as follows:

```
filename mprint 'c:\hips\programs\testmac.sas'; *'pathname and name of file';
options mprint mfile;
%your_macro
```

The fileref must be MPRINT. The pathname must include the name of the external file where the code that is generated by the macro is to be stored. After this program finishes executing, you can take the code generated and run it to understand what is actually happening and fix any problems.

Topic: Macro inside a Macro?

- **Best practice ... do not nest macros**
- Consider nesting definitions only when:
    - The macros are always married and never useful as separate macros, and
    - The outer macro is used only once per program
- Consider that the inner macro could at some point have expanded functionality that might make it useful for other applications.
- Also consider that when you nest macros, each time you execute the outer macro it redefines the inner macro. That represents extra work (even if it might be a small amount of extra work).

Useful Printer System Options

To reset the page number, the code is:

```
options pageno=1;
```

To set orientation and paper size:

```
options orientation="landscape";
options orientation="portrait";
options papersize="legal";
```

To remove page breaks and put dashes (-) across the page:

```
options formdlim="-";
```

To reinstall page breaks (there is no space between the quotes):

```
options formdlim="";
```

To left justify output:

```
options nocenter;
```

EFFICIENCY

What Matters Most?

In programming, to be efficient you will want to make good and optimal use of your resources. First ask "**What matters most**?"

- Is it computer time, memory, disk storage, computer memory, your time now, or
- your time in maintaining the program?

Look at the tradeoffs. Decide which resources to optimize and which ones you can afford to use less efficiently. If you or someone else needs to maintain the code, write the code using good programming style and organization skills.

Think

Thinking and planning before programming is one of the best tips on efficiency. Marje Fecht in her paper *THINK Before You Type… Best Practices Learned the Hard Way* gives best practices to minimize effort and maximize results. I still have my IBM desk placard that says “Think."

Keep It Simple

Keep it simple, easy and efficient. Use procedures instead of data steps. Avoid unnecessary data steps. For example, permanently save only the final data set. Use temporary data sets to store intermediate results.

The Constant Stays The Same

Assign a constant to a variable in a RETAIN statement instead of an assignment statement.

```
data School;
     retain TestYear 2008
     District "Honolulu";
programming statements
run;
```

Null and Void

Use _NULL_when you do not need to create a SAS data set.

```
data _null_;
     set Company;
     Amount=SUM(OF Sold1-Sold31);
     if Amount<90 then
     put Product= Amount=;
run;
```

Avoid Infinity

Avoid division by zero by testing for it. This will save computer time.

```
if Number ne 0 then Avg=Total/Number;
else Avg=.;
```

Save Time Sorting

Use the NOEQUALS option on PROC SORT to reduce computer time and memory. By default SORT keeps the same order of the observations with identical BY variable values as they were in the input data set. The NOEQUAL option tells SAS to ignore the order of observations within BY groups.

```
proc sort
     data=Survey noequals;
     by dept;

run;
```

Paper 8: The 7 Habits of Highly Effective SAS-ers

- Know Your Problem
- Use the Right Tool
- Fewer Steps Get You Farther
- Stay Tall and Thin
- Too Much of a Good Thing Is Bad
- Skip the Expensive Stuff
- Sharpen the Saw

Know Your Problem

Know your data and know your problem. In the real world, we often have to start with one procedure before we can analyze our results and decide on a better procedure.

Use the Right Tool

Fewer Steps Get You Farther
Stay Tall and Thin
Too Much of a Good Thing Is Bad
Skip the Expensive Stuff
Sharpen the Saw

Paper 9: The Most Important Efficiency Techniques

Most approaches to efficiency focus on speeding up the program. However, the programmer should consider much more than that.

- Is the program easy to understand and maintain?
- Does it require vast amounts of other resources (disk space, memory, tape drives to name a few)?
- Does the analyst have to search through pages and pages of output to locate a few key numbers?
- How much time can you afford to learn and apply new techniques?

Reading raw data

```
data retired_males;
       infile rawdata;
       input Gender $ 81
Age 82-84 @;

/* The trailing @ holds each raw data line temporarily, in case the second input statement needs to read more variables from the same line.*/

if gender='M' and age >= 65;
       input LastName $ 1-20
       FirstName $ 21-35
       Street $ 36-55
       City $ 56-75
       ZipCode $ 76-80;
run;
```

Reading from SAS Data Sets

When working with SAS data sets, use keep and drop to control which variables get processed. The comment statement below is identical to the keep= data set option:

However, adding keep= or drop= to the set statement provides additional savings by limiting which variables get read in:

```
data bigwigs (keep=id nextyear);
       set employees (keep=id salary bonus);
       nextyear = salary * 1.05 + bonus;
       if nextyear > 100000;
run;
```

The same principal applies when sorting data. The first program sorts all the variables, **because keep= applies to the output data set**:

```
proc sort data=huge out=tiny (keep=var1 var2 var3);
       by var1;
run;
```

Testing on a Sample

The subset might contain anywhere from 0 to 5,000 observations. Instead of guessing, modify the data step to select a subset with exactly the right number of observations:

```
data first_50;
       infile rawdata;
       input name $10.;
       if name='Bob';
       found + 1;
       output;
       if found=50 then stop;
       drop found;
```

```
run;
```

Subsetting Considerations

When subsetting observations, where is usually faster than if:

```
data just_bob;
   set everybody;
   if name='Bob';
run;
```

```
data just_bob;
   set everybody;
   where name='Bob';
run;
```

Procedures can use where, eliminating the need for a data step to subset observations. The first program contains an extra data step:

```
data highrisk;
   set patients;
   where cholesterol > 240;
run;
proc means data=highrisk;
   var height weight pulse;
run;
```

```
proc means data=patients;
   where cholesterol > 240;
   var height weight pulse;
run;
```

However, if more than 3 procedures each need the same subset, create the subset. The same where statement in multiple procedures would force each procedure to subset the observations.

This principle applies even when the subset is based on variables rather than observations.

File Handling Considerations

- Eliminate extra passes through the data. Some of the examples are simple ones:
- Don't sort if you don't have to. When a procedure (such as proc freq) is not using a by statement, it does not require sorted data.
- Don't sort data sets that are already sorted.
- Process permanent SAS data sets. There is no need to copy a data set to the work area, just to run a procedure against it.
- Eliminate data steps by combining all data manipulation into a single step.
- Create multiple output data sets in a single data step.
- Append using proc append, not a data step.
- If a data step is needed to create analysis variables, perhaps that data step can also perform the analysis.
- Sort once, in the most detailed order.
- Store data in the most commonly needed sorted order. If sorting into a different order, use out= to avoid replacing the original data set.

Some Sorting Options

The **noequals** option is saying that the order within each by group does not matter:

```
proc sort data=sales noequals;
      by state;
run;
```

Tagsort:

```
proc sort data=sales tagsort;
       by state;
run;
```

The normal action for proc sort is to sort the entire observation, including all the variables. The tagsort option requests a different method. It takes the value of state, and appends a "tag" (a numeric value holding the observation number in the incoming data); it then sorts just state and the tag. Once all the sorting has been completed, it uses the tag to retrieve the remaining variables from the original data set. Speed will vary (and could even increase), depending on the operating system and release of the software. But the sorting process uses a lot less disk space.

Some Miscellaneous Considerations

These two statements are different:

```
data _null_;
data;
```

- The first statement creates no output data set, saving the CPU time needed to output as well as the storage space.
- The second statement says, "I'm too lazy to name the data set." The software will supply the name.

Data Manipulation

It's hard to slow down a program significantly, via data manipulation. But a couple of principles are worth noting.

Storage Space

Some methods to save on storage space are simple:

- Save just the necessary variables.
- Compress SAS data sets.
- Use the minimum length needed for each variable.
- Save a code, print a format.

Compressing a SAS data set is easy. Add any of these statements:

```
  options compress=yes;
  options compress=char;
  options compress=binary;
```

The first two methods are identical, compressing repetitions of the same character. Those methods are better at compressing character strings.

The third compresses patterns of bytes. It is better at compressing numeric variables, since a series of variables with a missing value all contain the same pattern of bytes.

Views are sets of instructions on how to extract data, rather than the data itself. Here is an example, where the bold section of the data statement instructs the software to save a view:

```
data golfers / view=golfers;
       infile rawdata;
       input golfer $char24.
       score 5.
       course $char20.;
run;
```

Views can slow the program down. In particular, do not use the same view multiple times.

Paper 10: Quick Hits - My Favorite SAS® Tricks

CODE REDUCTION

CONDITIONAL CONCATENATION

The SAS 9 family of `CAT` functions reduces complexity when concatenating strings!

- `CAT` concatenates multiple strings in one function call
- `CATT` - same as `CAT` but also `TRIM`s
- `CATS` - same as `CAT` but also `STRIP`s leading and trailing blanks
- `CATX` - same as `CATS` but you can specify a delimiter.

Old:
```
old = put(n,1.) || ' ' || trim(a) || ' ' || trim(b) || ' ' || c;
```

New:
```
new = CATX ( ' ', n , a, b, c);
```

CONDITIONAL ASSIGNMENT OF VALUES

IFC (**IFN**) returns a character (numeric) value based on whether an expression is

- true *result of expression NOT 0 and NOT . (missing)*
- false *result of expression is 0*
- missing *result of expression is . (missing)*

IFC (**IFN**) is coded as

```
IFC(expression-to-evaluate , true-result , false-result , missing-result);
```

Old:
```
    if ch_dm = 'Y' then dm = 'DM';
    else dm = ' ';
```

New:
```
    dm = IFC(ch_dm = 'Y' , 'DM' , ' ');
```

```
/*** NO NEED TO CREATE EXTRA TEXT VARIABLES ***/
 length channels_IFC_CATX $12;
 channels_IFC_CATX = CATX(
                          '_'      /* note delimeter of underscore */
                        , IFC( ch_dm = 'Y' , 'DM' , ' ')
                        , IFC( ch_on = 'Y' , 'ON' , ' ')
                        , IFC( ch_cc = 'Y' , 'CC' , ' ')
                        , IFC( ch_st = 'Y' , 'ST' , ' ')
                         ) ;
```

ch_dm	ch_on	ch_cc	ch_st	channels_IFC_CATX
Y	Y	Y	Y	DM_ON_CC_ST
N	Y	Y	N	ON_CC
Y	N	N	Y	DM_ST
N	Y	Y	Y	ON_CC_ST

It is helpful to note that when SAS evaluates an expression, a value of ZERO denotes FALSE and any other numeric non-missing values other than zero denotes TRUE.

MORE FUNCTIONS FOR YOUR TOOLKIT

A few more of my favorite (and more obscure) functions include:

- TRIM**N** ➔ removes trailing blanks – returns null for values that are all blank
- COUNT, COUNT**C** ➔ counts # of occurrences of a string or Character
- INDEX, INDEX**C**, INDEX**W** ➔ locates position of first occurrence of string, Character, or Word
- LENGTH (min=1) , LENGTH**N** (min=0) ➔ position of last non-blank.
  Note: LENGTHN returns 0 for values that are all blank
- LARGEST (*k , var1, var2, ...)* ➔ kth largest non-missing value
- SMALLEST (*k , var1, var2, ...)* ➔ kth smallest non-missing value
- PROPCASE ➔ handles upper / lower case to assist with Proper Names and Addresses, etc.

CODE GENERALIZATION

Before you begin writing a program, think about *what could change about this request in the future**?***

CODE MODULARIZATION: SOURCE CODE AND DRIVER PROGRAMS

My approach is to utilize ***driver*** programs that contain all of the parameters and other input, and that call the appropriate ***source*** modules.

INTNX – move in intervals

To dynamically generate the dates above, you need the ability to determine the current date and then move in increments of months back from today. Additionally, you need to be able to identify the first and last day of the month (without worrying about the nuances of the calendar). The INTNX function is one of the most versatile of the SAS date functions, enabling you to move forward and backward from a date and time using the interval of your choice, such as month, quarter, day, week, etc.

The **INTNX** function increments dates by intervals:

INTNX (interval, from, n < , alignment >);

- ***interval*** – interval name eg: 'MONTH', 'DAY', 'YEAR'
- ***from*** – a SAS date value (for date intervals) or datetime value (for datetime intervals)
- ***n*** – number of intervals to increment from the *from* value
- ***alignment*** – alignment of resulting SAS date, within the interval. Eg: **B**EGINNING, **M**IDDLE, **E**ND.

Example: Create 3 macro variables that contain the current and two previous months in the format: **yymm**

```
%let M0 = %sysfunc( today() , yymmN4.);

%let M1 = %sysfunc(intnx( MONTH ,
                    %sysfunc( today() ) ,
                    -1) , yymmN4.); /** go back one month from today **/

%let M2 = %sysfunc( intnx( MONTH ,
                    %sysfunc( today() ) ,
                    -2) , yymmN4.);  /** go back two months from today **/
```

Note: when INTNX is used in %sysfunc, do not use quotes for the arguments of INTNX.

SPACE MANAGEMENT - BEST PRACTICES

PROC DATASETS is handy to manage your SAS datasets including deleting files, modifying attributes, changing names, etc.

```
proc datasets lib = work
      memtype = data details;
      /** delete old data **/
      delete ChqReport_1a
             ChqReport_1b
             ChqReport_1c ;
quit;
```

Note: Before deleting intermediate data, you should confirm there are no error conditions, etc.

MINIMIZE THE AMOUNT OF DATA YOU READ

You do not have to read data to change many data set attributes. Many SAS programmers rely on the DATA step to handle changes to variable attributes (labels, formats, renaming, etc.). However, the DATA step reads every record which is problematic if your data include millions of records. Instead, the following PROC DATASETS example

- changes a data set name
- changes variables names
- assigns a variable format.

No data values are read!

```
proc datasets lib = project memtype = data details;
       /** rename dataset **/
       change ChqReport = ChqReport_&lastMonth ;
       /** change variable attributes in ChqExtract **/
       modify ChqExtract;
       rename  txns = Transactions
               Date = Txn_Date;
       format TotalAmt Dollar12.;
quit;
```

Paper 11: