Paper 060-2011

SAS® WOW! How to Streamline Your SAS Programs by Shedding Lines and Adding Substance

Lisa Mendez, SRA International Inc., San Antonio , TX.

Lizette Alonzo, SRA International Inc., San Antonio , TX.

ABSTRACT

Would you like to slim down your Base SAS programs by eliminating needless lines of code while adding substance? Join us for some real life examples that we came across when we were tasked to update our SAS programs. We'll show you how to "transpose" data when PROC Transpose doesn't seem to work, condense your input when inputting multiple files with the same file structure, and how to use macro variables wisely, not carelessly. All this can be yours by adding some sample code to your SAS toolbox. But wait, there's more! See the value of program headers and comment boxes, for no additional cost! That's right! All this can be yours just by reading this paper! Act now!

INTRODUCTION

The purpose of this paper is to provide you with concepts and shortcuts to writing programs or code more efficiently. This paper guides you through a process of rearranging data using arrays, thus incorporating macros, if needed, to condense the program. This capability is handy if, for instance you have data by months and you would like to view your months as columns instead of rows. You can switch the view of your data using an array. Another technique covered in this paper is the functionality to read in multiple files in one DATA step. Although you can not always use this technique, learn how and where to use it. Additionally, this paper covers the use of program headers and comment boxes to describe the purpose of the program and to make notes within the code. This is particularly helpful when the code is shared amongst several programmers, or if you don't use the program very often and may forget the rationale for each step used. Even seasoned programmers may forget the purpose of every piece of the code, so don't waste time trying to solve the code and use comment boxes. Once you have mastered the concepts presented in the paper you will be able to perform these strategies with more confidence and achieve a more effective and efficient code.

USING AN ARRAY TO TRANSPOSE DATA

Even if you have never used arrays, you can master this technique to help transpose your data. Many times our data is not formatted the way we prefer. For this example we will look at a data set that has one record for each fiscal month for multiple variables: ASFTE, AVFTE, and Salary. Figure 1 is a sample of data in its original structure. Notice we have one record for each fiscal month (FM) except for FM06.

Figure 1. Sample original dataset

	fy	fm	pdmisid	fcc4	svc	skill	asfte	avfte	salary
1	2008	01	0117	BAAA	F	1	69	19.91	204808
2	2008	02	0117	BAAA	F	1	68.15	21.72	211775
3	2008	03	0117	BAAA	F	1	68.05	21.35	218794
4	2008	04	0117	BAAA	F	1	68	20.25	204598
5	2008	05	0117	BAAA	F	1	64.7	20.1	195425
7	2008	07	0117	BAAA	F	1	66	20.81	201259
8	2008	08	0117	BAAA	F	1	65.5	20.89	183412
9	2008	09	0117	BAAA	F	1	91.55	47.99	494271
10	2008	10	0117	BAAA	F	1	83.3	48.31	510506
11	2008	11	0117	BAAA	F	1	84.05	37.01	337282
12	2008	12	0117	BAAA	F	1	80.2	34.52	309793

We prefer to have one row for ASFTE, AVFTE, and Salary with each value for each fiscal month for each variable. Figure 2 illustrates one row (observation) for ASFTE. Notice the value for FM06 is missing.

Figure 2. Sample of Preferred Format

fy	pdmisi	svc	fcc4	result	skill	FM01	FM02	FM03	FM04	FM05	FM06	FM07	FM08	FM09	FM10	FM11	FM12
2008	0117	F	BAAA	asfte	1	69	68.15	68.05	68	64.7	.	66	65.5	91.6	83	84	80

To get the data into the format we prefer, we used an array in a DATA step (in conjunction with a do loop), and a macro. We began by setting up the DATA step using one of the expense variables, and then create the macro. Below is the entire code, but we suggest setting up the DATA step first and ensuring it works for one variable before creating the macro. The example shows only one call of the macro for the variable ASFTE.

Sample SAS Code

```
%macro results (variable);

Data ❶ rotate_&variable;
       ❷ set all_fte_salary;
       ❸ length result $6.;

       ❹ salary = round(salary, .01);

       ❺ array months {12} FM01-FM12;

       ❻ do i = 1 to 12;
              result = "&variable";
              if fm = i then months {i} = asfte;
       end;
run;

❼ PROC MEANS data = rotate_&variable nway noprint;
       ❽ class  fy pdmisid cdmisid svc fcc4 result status skill;
       ❾ output out = means_&variable (drop = _type_ _freq_) sum=;
run;

%mend results;

❿%results (asfte);
```

Explanation of the Sample SAS code

The DATA step:

❶ The output data set is called *rotate_asfte.* The macro variable resolves to “asfte”.

❷ The incoming data set is called *all_fte_salary*

❸ The LENGTH statement is used to set the length of a new variable *result* to six characters.

❹ Round the *salary* variable

❺ Define the array

In order to use an array, you must define it in the DATA step prior to it being referenced. Arrays only exist for the duration of the DATA step in which it is defined.

ARRAY Array-name {n} <$> <length> array-elements;

Where

ARRAY	SAS Keyword to define the array
Array-name	a valid SAS name that is not the same as a variable in the dataset
{n}	the index to provide the number of elements in the array (optional)
<$>	used if the elements in the array are characters
<length>	used to define the length of new variables created in the array (optional)
array-elements	a list of variables of the same type (all numeric or all character) in the array

Our array will have 12 elements – an ASFTE expense value for each fiscal month

❻ The iteration Do Loop will use the array and assign the expense value to the variable *result (asfte)* for each fiscal month (FM)

The standard syntax for iteration Do Loop is

DO <*index variable*> **=** <*start value*> **TO** <*stop value*> <**BY** <*increment value*>> ;

SAS commands;

END;

Where

DO	is a keyword that causes the SAS commands to be repeated
<*index variable*>	must be a SAS variable
=	sets the index variable to the start value
<*start value*>	can be numeric constant or a numerical variable
TO	compares the index variable to the stop value. If the index variable does not exceed the stop value, SAS performs the statements in the loop. If the index variable exceeds the stop value, SAS exits the Do Loop by going to the next statement after END;
<*stop value*>	can be a numeric constant or a numerical variable
<BY <*increment value*>	changes the index variable by the amount of the increment value (or by 1 if no increment value is specified). The increment value can be negative.

After SAS performs the *SAS commands* it returns to the top of the loop, the *index variable* is changed by the amount of the *increment value* (by 1 is the default), and the loop continues until the *index variable* is past the *stop value*.

Figure 3 shows the dataset after the DATA step. Notice the ASFTE variable values. At this point we have still have 12 records (one for each FM), but we now have columns for each FM. Now we can condense the data using a PROC MEANS by specific variables to achieve our end result. We will also drop some variables that will no longer be needed.

Figure 3. SAS Dataset after the DATA step (Rotate_asfte.sas7bdat)

	fm	pdmisid	fcc4	svc	skill	result	FM01	FM02	FM03	FM04	FM05	FM06	FM07	FM08	FM09	FM10	FM11	FM12	i
1	01	0117	BAAA	F	1	asfte	69	.	.	.	.	.	.	.	.	.	.	.	13
2	02	0117	BAAA	F	1	asfte	.	68.2	.	.	.	.	.	.	.	.	.	.	13
3	03	0117	BAAA	F	1	asfte	.	.	68.05	.	.	.	.	.	.	.	.	.	13
4	04	0117	BAAA	F	1	asfte	.	.	.	68	.	.	.	.	.	.	.	.	13
5	05	0117	BAAA	F	1	asfte	.	.	.	.	64.7	.	.	.	.	.	.	.	13
6	06	0117	BAAA	F	1	asfte	.	.	.	.	.	.	.	.	.	.	.	.	13
7	07	0117	BAAA	F	1	asfte	.	.	.	.	.	.	66	.	.	.	.	.	13
8	08	0117	BAAA	F	1	asfte	.	.	.	.	.	.	.	65.5	.	.	.	.	13
9	09	0117	BAAA	F	1	asfte	.	.	.	.	.	.	.	.	91.6	.	.	.	13
10	10	0117	BAAA	F	1	asfte	.	.	.	.	.	.	.	.	.	83.3	.	.	13
11	11	0117	BAAA	F	1	asfte	.	.	.	.	.	.	.	.	.	.	84	.	13
12	12	0117	BAAA	F	1	asfte	.	.	.	.	.	.	.	.	.	.	.	80	13

❼ PROC MEANS is used to condense the data. The input dataset is rotate_asfte.

❽ Variables in a CLASS statement of PROC MEANS specifies one or more variables that the procedure uses to group the data. For each of the 12 records in Figure 3, the values are the same for fy, pdmisid, svc, fcc4, result, and skill.

❾ The output data set is means_asfte. Figure 4 illustrates the output dataset.

Figure 4. Dataset After PROC MEANS.

fy	pdmisi	svc	fcc4	result	skill	FM01	FM02	FM03	FM04	FM05	FM06	FM07	FM08	FM09	FM10	FM11	FM12
2008	0117	F	BAAA	asfte	1	69	68.15	68.05	68	64.7	.	66	65.5	91.6	83	84	80

READING IN MULTIPLE FILES – UTILIZING THE ASTERISK (*)

Programmer efficiency techniques include combining steps into one where possible. In addition, programs that run in a production environment can be written with flexibility as to limit the editing of the program code when the program is submitted.

Let's say that we need to use nine raw data files to create one SAS data set. The raw data files have the same file structure and similar names. They are all vertical bar (|) separated files with the same fields in the same order.

When we began to re-write our MEWACS process, we encountered multiple DATA steps to read in the text files into nine SAS datasets into the Work library, and another DATA step to set the files together to create one final SAS dataset into a permanent library (see Figure 3). Although this method worked it used more I/O and it is bulky. Instead, we decided to shed those lines of code and consolidate into one DATA step. Since the filenames are similar, we can use the asterisk (*), much like a wildcard, in the INFILE statement.

Figure 5. Reading in multiple data sets and setting them together.

Sample Old SAS Code Input Files.sas *

```
Data salary_2009_A (keep = fy fm svc pdmisid cdmisid fcc4 skill perdir exp_ind);
     infile "&Path:\11970.163\Products\MEWACS\MEWACS4.0\&ExtractDate\Raw Data\m4.0 salary 2009 - A.txt"
            delimiter = "|" missover;
     input  fy$ fm$ pdmisid$ cdmisid$ fcc4$ skill$ svc$ perdir?? exp_ind$;

  Run;

Data salary_2009_F (keep = fy fm svc pdmisid cdmisid fcc4 skill perdir exp_ind);
     infile "&Path:\11970.163\Products\MEWACS\MEWACS4.0\&ExtractDate\Raw Data\m4.0 salary 2009 - F.txt"
            delimiter = "|" missover;
     input  fy$ fm$ pdmisid$ cdmisid$ fcc4$ skill$ svc$ perdir?? exp_ind$;

  Run;

Data salary_2009_N (keep = fy fm svc pdmisid cdmisid fcc4 skill perdir exp_ind);
     infile "&Path:\11970.163\Products\MEWACS\MEWACS4.0\&ExtractDate\Raw Data\m4.0 salary 2009 - N.txt"
            delimiter = "|" missover;
     input  fy$ fm$ pdmisid$ cdmisid$ fcc4$ skill$ svc$ perdir?? exp_ind$;

  Run;

Data M4RawDS.raw_Salary;
     Retain fm perdir fy pdmisid cdmisid fcc4 skill svc exp_ind;
     Set salary_2009_A
         salary_2009_F
         salary_2009_N;

  run;
```

For this example, let's say we have three Salary text files; one for each service for fiscal year 2009. Figure 4 shows the filenames of the three text files.

Figure 6. Sample Salary text files.

Name	Size	Type
m4.0 salary 2009 - A.txt	11,742 KB	Text Document
m4.0 salary 2009 - F.txt	10,608 KB	Text Document
m4.0 salary 2009 - N.txt	7,554 KB	Text Document

The Sample SAS code below, will illustrate how to utilize the asterisk (*).

Sample SAS Code

```
Data ❶ M4RawDS.raw_salary ❷(keep = fy fm svc pdmisid cdmisid fcc4 skill perdir
exp_ind);
      ❸ Retain fm perdir fy pdmisid cdmisid fcc4 skill svc exp_ind;
      ❹ infile "&Path:\11970.163\Products\MEWACS\&ExtractDate\m4.0 salary *.txt"
           delimiter = "|" missover;
      ❺ input  fy$ fm$ pdmisid$ cdmisid$ fcc4$ skill$ svc$ perdir?? exp_ind$;
      ❻ If fy ne 'Fiscal Y';
Run;
```

Explanation of the Sample SAS code

❶ The DATA step will create a dataset named raw_salary in a permanent library named M4RawDS. Figure 6 illustrates a partial listing of the output dataset.

❷ The KEEP statement keeps only the variables that are needed

❸ The RETAIN statement is used to order the variables in the output SAS dataset

❹ The INFILE statement uses the pathname of the text file. The entire pathname is in double quotes because of the macro variables (Path and ExtractDate). In the text file name the asterisk is placed after the word “salary” so that all files that begin with “m4.0 salary” and end in the file extension “.txt” are included.

❺ The INPUT statement declares the variables and their characteristics. $ signify character variables, ?? (which is a modifier) signifies that invalid data may occur in the perdir field and is used to reset the automatic error variable to 0, eliminating the error condition flagged because of the invalid data. The offending variable is then set to missing but the log will be clear.

❻ If your data files have header rows, header rows will appear in the data set when using the asterisk (*). In order to delete the header rows, you can use an IF statement after the input statement. If you do not do this, you will end up with header rows for each text file in your dataset. You should know your data in order to create the test the condition to delete the header rows.

GLOBAL MACRO VARIABLES

A macro variable is a great tool to use when you need to change text in a program by substituting a value. It allows you to update your code almost effortlessly, thus reducing the risk of errors. Macro variables may be used anywhere in SAS programs, except within data lines. When a macro variable is defined, the macro processor adds it to one of the program's macro variable symbol tables. If the macro variable is defined in a statement that is outside a macro definition, the variable is held in the global symbol table, which the macro processor creates at the beginning of a SAS session. This macro variable, a global macro variable, exists for the remainder of the current SAS session.

Let’s say you have a SAS program that is run monthly, but the data used in the program comes from different folders and subfolders located on the C: drive on your computer. Wait a minute… sometimes those folders are not located on your C: drive but perhaps on a shared drive at work, for instance, P: drive. Do you have to change your code every month? With a global macro variable, you can speed up the process ensuring you make the change in only one place throughout the program. You can use the %LET statement to create the macro variable and assign it a value.

Syntax for macro statement %LET:

%LET macro-variable =<value>;

Where,

macro-variable	is either the name of a macro variable or a text expression that produces a macro variable name. The name can refer to a new or existing macro variable.

value	is a character string or a text expression. Omitting value produces a null value (0 characters). Leading and trailing blanks in value are ignored. To make them significant, enclose value with the %STR function.

Now let's look at an example using the %LET macro statement.

Sample SAS Code 1 (partial code shown)

```
/*************************************************************************/
/* CREATE MACRO VARIABLES AND UPDATE THE VALUES EVERY MONTH              */
/*************************************************************************/
❶ %Let Path = P;  /* Specify the drive where the data is located*/
❷ %Let ExtractDate = 201011_October; /* Specify the current folder name */
❸ %Let FY = 2010; /* Current Fiscal Year */
❹ %Let PFY = 2009;      /* Previous Fiscal Year */
```

Explanation of Sample Code 1

Note that the **%Let** creates the macro variables.

❶ Create macro variable **Path** and assign it a value of **P**, the drive where the data is located.
❷ Create macro variable **ExtractDate,** and assign it a value of **201011_October,** the name of the folder it will reference.
❸ Create macro variable **FY** and assign it a value of **2010,** the current fiscal year.
❹ Create macro variable **PFY** and assign it a value of **2009,** the previous fiscal year.

Sample SAS Code 2 (partial code shown)

```
/*************************************************************************/
/* ASSIGN GLOBAL LIBRARY REFERENCES                                      */
/*************************************************************************/
LIBNAME M4RawDS ❶"&Path:\11970.163\MEWACS\❷&ExtractDate\Raw Data";
LIBNAME Files   ❶"&Path:\11970.163\MEWACS\Files for Comp Memo & Exec Sum";

/*************************************************************************/
/* Input the text files                                                  */
/*************************************************************************/
Data M4RawDS.raw_mtfs;
   infile ❶"&Path:\11970.163\MEWACS\❷&ExtractDate\Raw Data\m4.0 MTFs.txt"
   dlm="|" firstobs=2;
   length pdmisname$ 85. cdmisname$ 85.;
   input fy$ pdmisid$ cdmisid$ pdmisname$ cdmisname$ svc$ ;
   if fy in (❸"&FY" , ❹"&PFY");
Run;

Data M4RawDS.raw_fcc4;
   length fcc4desc$ 75. ;
   infile ❶"&Path:\11970.163\MEWACS\❷&ExtractDate\Raw Data\m4.0 fcc4.txt"
   dlm = '|' firstobs = 2;
   input fy$ pdmisid$ fcc4$ fcc4desc$ cdmisid$ ;
   if fy in (❸"&FY" , ❹"&PFY");
Run;
```

Explanation of Sample SAS Code 2

Note that the (&) symbol calls the macro variables created in Sample Code 1, above.

❶ Resolves to "**P**:\11970.163\MEWACS\...." This is useful if your files are not always located in the same drive of your computer. For instance, one month they are on your C drive while other months they are in a shared drive, P.

❷ Resolves to "…**201011_October** \Raw Data\...".
❸ Resolves to **2010**. The quotes indicate character data. This may be useful if your raw data includes several years of data, but you are only interested in a particular fiscal year.
❹ Resolves to **2009**. The quotes indicate character data. This may be useful if your raw data includes several years of data, but you are only interested in a particular fiscal year.

PROGRAM HEADERS

Program headers are necessary to ensure that programs are used correctly. Many "down and dirty" programs are written without program headers, and then end up being used more than intended. Every program that is written should have a program header; even temporary programs. How many times does a temporary program turn into a production program? Adding a program header to EVERY program you write is good programming practice and will help more than you know later down the road.

Program headers can be set up in a "skeleton" or "shell" format to ensure consistency and ease of use.

Sample Header Block Shell

```
/*************************************************************************************/
/*  Program Name:                                                                    */
/*  Date Created:                                                                    */
/*  Author:                                                                          */
/*  Purpose:                                                                         */
/*                                                                                   */
/*                                                                                   */
/*  Inputs:                                                                          */
/*  Outputs:                                                                         */
/* --------------------------------------------------------------------------------- */
/*  Macros:                                                                          */
/*                                                                                   */
/* --------------------------------------------------------------------------------- */
/*  Date Modified:                                                                   */
/*  Modified by:                                                                     */
/*  Reason for Modification:                                                         */
/*  Modification Made:                                                               */
/* --------------------------------------------------------------------------------- */
/*  Date Modified:                                                                   */
/*  Modified by:                                                                     */
/*  Reason for Modification:                                                         */
/*  Modification Made:                                                               */
/*************************************************************************************/
```

Explanation of the Header Block

- Program name: Should match the SAS program name
- Date created: General date the program was written (ie. June 2010)
- Author: Names of all programmers who helped create the SAS Program
- Purpose: The reason the program was written. Justification for the program should also be provided.
- Inputs: List all input files
- Outputs: List all permanent output files.
- Macros: List any macros that are called by the program. Internal macros need not be listed.
- Modification Information
  - Date Modified: Specific date the program was modified (ie. July 10, 2010)
  - Modified by: Name of the person who modified the program
  - Reason for Modification: Provide the detailed reason why the program was modified. (ie. policy change, per a specific individual, change in algorithm, etc.)
  - Modification Made: Provide the detail of the modification. Be as specific as possible.
  - Each time the program is modified, a modification block should be added.

Sample Header Block

```
/*************************************************************************************/
/*  Program Name: M4.0 P2 04 Personnel Profiles.sas                                  */
/*  Date Created: July 2009                                                          */
/*  Author: Lizette Alonzo & Lisa Mendez                                             */
/*  Purpose: Generates file/dataset for Personnel Profiles in MEWACS 4.0             */
/*  Inputs:  (extract date)\Raw Data\raw_fte.sas7bdat                                */
/*           (extract date)\Raw Data\raw_salary.sas7bdat                             */
/*  Outputs: (extract date)\Online Data\MEWACSOL4_PersonnelProf_(extract date).txt */
/* ---------------------------------------------------------------------------------- */
/*  Macros: None                                                                     */
/*                                                                                   */
/* ---------------------------------------------------------------------------------- */
/*  Notes: FTE data comes from Class 7                                               */
/*         Salary data comes from Class 9 and 8                                      */
/* ---------------------------------------------------------------------------------- */
/*  Date Modified: 2/26/2010                                                         */
/*  Modified by: Lisa Mendez                                                         */
/*  Reason for Modification: Change to the Post Purification Salary BO queries       */
/*  Description of Modification:  Deleted the criteria "if skill = N then delete"    */
/*     as the BO query does not pull skill type = N.  Changed code to deal with      */
/*     pre-purification salary data from classes 9 & 8 and post purification         */
/*     salary.                                                                       */
/* ---------------------------------------------------------------------------------- */
/*  Date Modified: 3/08/2010                                                         */
/*  Modified by: Lisa Mendez                                                         */
/*  Reason for Modification: Web developer requested "PRE-purification" and          */
/*  "POST-purification" flags be changed to "0" and "1" in the final file            */
/*  Description of Modification:  Changed the PRE-purification and                   */
/*     Post-purification flags to 0 and 1.                                           */
/* ---------------------------------------------------------------------------------- */
```

COMMENT BOXES

Call us obsessive, but we believe that a comment block should precede almost all DATA steps and procedures. We do this primarily because of Eagleson's Law. Eagleson's law states that any code of your own that you haven't looked at for six or more months might as well have been written by someone else. Comment blocks help out other programmers as well as yourself. You might know what you are coding and why you are coding it at the time you code it, but chances are the next programmer to touch your program will have to read through it to understand what is going on. Give the guy or gal a break and use comments to help him or her out.

Comment boxes can also be standardized to help understand the program at a quick glance. Here are some examples of standardized comment boxes.

Sample Comment Boxes

A comment box preceding a DATA step or Procedure (single line between)

```
/*-------------------------------------------------------------------------*/
/*     Input the commands data set into the work library                   */
/*-------------------------------------------------------------------------*/
```

A comment block within a DATA step or procedure (no line)

```
/* Change missing FM values to 00 to ensure proper processing */
if fm = '' then fm = '00';
```

A comment box at the beginning of a macro (double line)

```
/*=========================================================================*/
/*     Macro to Rotate the data                                            */
/*=========================================================================*/
```

Additional comment box (smaller asterisked box)

```
/************************/
/*   PRE PURIFICATION   */
/************************/
/*-------------------------------------------------------------------------*/
/*  Create temp data set with PRE-purification fte & salary data.          */
/*-------------------------------------------------------------------------*/
Data pre_fte_salary (keep = fy fm svc pdmisid cdmisid fcc4 skill asfte avfte
salary status);
       set pre_fte
             pre_salary;

run;

/************************/
/*   POST PURIFICATION  */
/************************/
/*-------------------------------------------------------------------------*/
/*  Create temp data set with POST-purification fte & salary data.         */
/*  POST fte data comes from Class 7 & POST salary comes from class 9 & 8. */
/*-------------------------------------------------------------------------*/
Data post_fte_salary (keep = fy fm svc pdmisid cdmisid fcc4 skill asfte avfte
salary status);
       set post_fte
             post_salary_abcfg
             post_salary_de_nonx
             post_salary_de_costpool;

run;
```

CONCLUSION

SAS has so many tools and procedures that can be manipulated to achieve the same results, but, if efficiency and simplicity are important to you, try using some methods discussed in this paper. Next time you need to transpose your data, use an array and note comments accordingly. Also, don't forget to include program headers to save time trying to recall what you updated in your program since the last run. Use global macro variables to substitute text in several programs and if you need to read in multiple files, make use of the asterisk if possible. Remember, as with most SAS procedures, there is so much more to learn and discover with every tip outlined in this paper so enjoy the tools and have fun with it!

REFERENCES

First, Steve & Schudrowitz, Teresa. *Arrays made easy: an introduction to arrays and array processing*. SUGI 30. Paper 242-30

SAS Documentation for 9.2.3.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Lisa Mendez
SRA International
1777 NE Loop 410, Ste 510
San Antonio, TX 78217
Phone: 210-832-5213
Fax: 210-824-9578
E-mail: lisa_mendez@sra.com

Lizette Alonzo
SRA International
1777 NE Loop 410, Ste 510
San Antonio, TX 78217
Phone: 210-832-5263
Fax: 210-824-9578
E-mail: lizette_alonzo@sra.com