PharmaSUG 2012 - Paper TF17

The Use and Abuse of the Program Data Vector

Jim Johnson, Ephicacy Corporation, Jersey City, NJ, USA

ABSTRACT

Have you ever wondered why SAS® does the things it does, or why your programs ñget away withò the things that they do, or why SAS would not do what you wanted it to? A key operational component of SAS is the program data vector. Without it SAS would not function as we know it. With knowledge of how the program data vector functions, programmers can better understand how SAS works. This paper will help you understand how the program data vector works, how data steps use it, and how you can exploit, manipulate, and trick it. Many examples are included; the magic behind of the DOW-loop and other mysteries will be discussed. There is something in this paper for all levels of programmers from the very beginner to the most advanced.

INTRODUCTION

Imagine a pile of fifty cinder blocks, each weighing about forty pounds. If you were asked to move the pile of cinder blocks from one room to another room fifty feet down the hall, you can imagine the amount of effort and the number of trips that will be necessary to accomplish the task. Now imagine a pile of fifty bricks, each weighing about five pounds. The amount of effort needed to move those bricks down the hall to another room is less, because you can take more units per trip, and therefore fewer trips. Finally, imagine a pile of fifty pebbles. Obviously the effort required to move them to the end of the hall would be almost nil. The same is true for the function of the SAS System. The more you make the system do, the harder it will be and the longer it will take. If, however, you can reduce your cinder blocks to bricks, you can help SAS do its job better and faster. Knowing how the program data vector works, and how to use it efficiently can go a long way toward reducing your cinder blocks to bricks. If every time you write SAS code you imagine cinder blocks and bricks and pebbles, you can become a better programmer.

This paper is deals only with the *data step*. Procedures are compiled code and each has unique methods of using the program data vector. Another paper intended for SAS procedures is listed in the recommended reading section of this paper

PART 1:
WHAT THE PROGRAM DATA VECTOR IS AND HOW IT WORKS

In this section the program data vector will be defined and described. Examples of how the program data vector is designed to work in the data step and some of the tools that can be used to change the contents of the program data vector will be discussed.

THE PROGRAM DATA VECTOR

The SAS Language Reference defines the program data vector as:

> ñA logical area in memory where SAS builds a data set, one observation at a time. When a program executes, SAS reads values from the input buffer or creates them by executing SAS language statements. The data values are assigned to the appropriate variables in the program data vector. From here, SAS writes the values to a SAS data set as a single observation.ò

Stated another way: The program data vector is a storage place in memory that contains all of the variables encountered by the data step. The order of the variables in the program data vector is determined by when they are encountered. These variables may have indicators flagging them to be kept or dropped or renamed. When the program runs, the program data vector contains the observation currently being processed. At the end of the step, the data are output according to the drop, keep, or rename instructions encountered in the program.

At compilation time SAS creates the program data vector. The program data vector contains all the variables in the input data set and the variables created in the DATA step statements.

Automatic variables are created automatically by the DATA step. Automatic variables are:

- Retained from one iteration of the DATA step to the next
- Not written to the output data set

- Cannot be kept, dropped, or renamed

Some of the common automatic variables include:

- _N_: the number of times the DATA step has iterated
- _ERROR_: indicates if an error has occurred in the data step. The value will be 0 if no errors occurred. It will be 1 if one or more errors occurred.
- first.*BY-variable* and last.*BY-variable*: These temporary variables always appear in pairs, one pair for each BY-variable on a BY statement. They have the value of 1 or 0 if the condition is true or false respectively.

Temporary variables are created through the use of certain SAS options and statements, and like automatic variables, are not written to the output data set. Some common temporary variables include:

- IN=*variable*: IN= is a data set option that indicates whether a particular data set has contributed to the current observation. The user specifies a variable name with the option that will have the value of 1 if the data set contributed to the observation or 0 if it did not.
- END=*variable*: END= is an option to the SET statement that indicates the end of the input data has been reached. The user defines a variable name that will have the value of 1 if the end of the data has been reached, 0 if it has not. Only one END= can be specified on the SET statement. If multiple data sets are given in the SET statement, the END= variable will be set to 1 only when the last observation of the last data set has been read.

User defined variables can be assigned the value of automatic and temporary variables if it is necessary to have their values in the output data set.

The program data vector holds all variables, user defined, automatic, and temporary, from the data sets, input sources, or created during the execution of the data step. The program data vector is used to populate the output data sets. All variables in the output data sets are in the program data vector, but not all variables on the program data vector are written to the output data sets.

During execution SAS will set the non-retained and non-input variables to missing in the program data vector at the beginning of each iteration of the DATA step. SAS reads data from an input file or from a SAS data set directly into the program data vector, replacing the previously existing values.

DROP / KEEP/ WHERE / RENAME

Use of DROP, KEEP, WHERE, and RENAME will affect the program data vector and can have a profound effect on the operation of SAS. It is important to understand when and where to use these operations. Use them in the wrong place and they will have little effect on the program data vector and efficiency.

Efficient use of the program data vector requires knowing the difference between a ***SAS statement*** and a ***SAS data set option*** and when each takes effect in the data step.

SAS STATEMENTS appear in the body of the data step code. The action of SAS statements occur more globally within the data step, at different times, and may have different effects than their SAS data set option counterpart. Some examples of SAS statements are:

```
keep pt age sex;
rename dob=birthday;
where complete=1;
```

SAS DATA SET OPTIONS allow you to specify actions on specific SAS data sets as the data are read or written and therefore appear in statements like DATA and SET. These options include keeping, dropping, and renaming variables and subsetting data. Data set options are specified in parentheses following a SAS data set name, and the option name is followed by an equal sign and option details. Below are examples of SAS data set options in bold.

```
data demog(keep=pt age sex);
set source(rename=(dob=birthday));
proc print data=final(where=(complete=1));
```

Many of the differences between SAS statements and SAS data set options are based around when the data are written or read. This paper may use õon the way inò to refer to reading data, and õon the way outò to refer to writing data.

Some of the differences between SAS statements and SAS data set options are:

1. Where they are specified
   - SAS statements appear anywhere in the data step code.
   - Data set options appear only in the DATA statement or input statements such as SET, MERGE, or UPDATE.
2. How globally they apply
   - SAS statements apply to all data sets created or read in the data step.
   - Data set options apply only to the data set to which they are attached.
3. When they take effect
   - The SAS statements DROP, KEEP, and RENAME take effect on the way out as the observation is written to the output data set. The WHERE statement takes effect on the way in as the data are read from the input data sets.
   - Data set options can take effect on the way out if specified on an output data set, or on the way in if specified on an input data set.

DROP Statement

- The DROP statement indicates which variables in the program data vector should not be written to the output SAS data sets.
- The DROP statement takes effect as the observation is written to the output data sets (on the way out).
- One DROP statement applies to all output data sets in the data step and can be used anywhere in the data step.
- Multiple DROP statements can be used in a data step. All will apply to the output data sets.
- All variables listed on the DROP statement are on the program data vector and available for use in the current data step until the observation is output.

DROP= data set option

- The DROP= data set option used on an input data set specifies the variables not to be read from the data set to the program data vector (on the way in).
- The DROP= data set option used with an output data set it lists the variables on the program data vector that are not to be written to the output data set (on the way out).
- All data set options must be specified for each data set to which they apply.
- Multiple DROP= data set options can be used with any one data set.

KEEP statement

- The KEEP statement indicates which variables in the program data vector should be written to the output SAS data sets.
- The KEEP statement takes effect as the observation is written to the output data sets (on the way out).
- One KEEP statement applies to all output data sets in the data step and can be used anywhere in the data step.
- Multiple KEEP statements can be used in a data step. All will apply to the output data sets.
- Variables **not** listed in the KEEP statement are on the program data vector and available for use in the current data step until the observation is output.

KEEP= data set option

- The KEEP= data set option used on an input data set lists those variables to be read from the data set to the program data vector (on the way in).
- The KEEP= data set options used with an output data set it specifies variables to be written from the program data vector to the output data set (on the way out).
- All variables are on the program data vector and available for use in the current data step until the observation is output.
- All data set options must be specified for each data set to which they apply.
- Multiple KEEP= data set options can be used with any one data set.

If both DROP and KEEP are used in the same data step, the DROP will take effect first. If no DROP or KEEP are specified, all variables except automatic and temporary variables in the program data vector will be written to the data sets. The same is true with DROP= and KEEP= data set options.

The order that the DROP and KEEP statements or data set options appear does not matter.

This example uses data set options on the way in.

```
data one;
  a = 1;  b = 2;
  output;
run;

data two;
  set one(drop=a keep=a);
run;

NOTE: There were 1 observations read from the data set WORK.ONE.
NOTE: The data set WORK.TWO has 1 observations and 0 variables.
```

The output data set above contains zero variables because on input the DROP= eliminated the variable A and the KEEP= effectively dropped everything except the variable A causing SAS to read zero variables from data set ONE.

This example uses statements instead of data set options.

```
data two;
  set one;
  drop a;
  keep a;
run;

WARNING: The variable a in the DROP, KEEP, or RENAME list has never been
referenced.
NOTE: There were 1 observations read from the data set WORK.ONE.
NOTE: The data set WORK.TWO has 1 observations and 0 variables.
```

The effects of the DROP and KEEP *statements* occur on the way out. The WARNING message is generated because the variable A was dropped by the DROP statement before it could be kept by the KEEP statement. The same message would occur if data set options are used on the output data set instead of the input data set in example 1.

This example uses data set options on the way out.

```
data one;
  a = 1;  b = 2;
  output;
run;

data two(drop=a keep=a);;
  set one;
run;

WARNING: The variable a in the DROP, KEEP, or RENAME list has never been
referenced.
NOTE: There were 1 observations read from the data set WORK.ONE.
NOTE: The data set WORK.TWO has 1 observations and 0 variables.
```

The output data set above contains zero variables because on input the DROP= eliminated the variable A and the KEEP= effectively dropped everything except the variable A causing SAS to read zero variables from data set ONE.

```
data one;
  a = 1; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output;  /** duplicate **/
  a = 3; b = 2; c = 3; output;
run;
```

```
data two;
  set one(drop=a);
  if a = 1;
run;


NOTE: Variable a is uninitialized.
NOTE: There were 4 observations read from the data set WORK.ONE.
NOTE: The data set WORK.TWO has 0 observations and 3 variables.
```

No observations are in data set TWO because variable A was dropped and is no longer available for use in the subsetting IF statement. Because it was dropped using a data set option on the input data set (on the way in), it was never available to the program data vector, thus the uninitialized note was produced. The variable A is one of the three variables in the data set TWO but is missing, i.e. uninitialized.

WHERE statement

- The WHERE statement provides a subset of observations in the data set.
- The WHERE statement takes effect on the way in. If an observation does not meet the condition in the WHERE statement, it will not be read. Therefore, the observation will never reach the program data vector.
- One WHERE statement applies to all input data sets in the step.
- Multiple WHERE statements can be specified in a data step. Only the last one will be used.
- The WHERE statement cannot be executed conditionally; that is, you cannot use it as part of an IF-THEN statement.
- The WHERE statement takes effect immediately after the input data set options are applied.

WHERE= data set option

- The WHERE= data set option used on an input data set subsets the data as on the way in. Observations that do not meet the where condition are never read, therefore never reach the program data vector.
- The WHERE= data set option used on an output data set subsets the data on the way out. The observations will be in the program data vector and will be written to the output data set only if the condition is met.
- All data set options must be specified for each data set to which they apply.
- Only one WHERE= data set option can be used with any one data set.

WHERE statement used with WHERE data set option

If the WHERE= data set option is used on an input data set (on the way in) and the WHERE statement is also used in the program code, the data set option will be applied to the data set, and the WHERE statement will be ignored. Input data sets not specifying a WHERE= data set option will be subject to the WHERE statement.

```
data one;
  a = 1; b = 3; c = 3; output;
  a = 2; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output;  /** duplicate **/
  a = 3; b = 2; c = 3; output;
  a = 3; b = 3; c = 3; output;
run;

data two;
  set one(where=(a=1));
  where b=2;
WARNING: The WHERE statement cannot be applied to a data set on the last
SET/MERGE/UPDATE/MODIFY statement.  Either the data sets listed failed with open
errors or they already specify a WHERE data set option.
run;

NOTE: There were 1 observations read from the data set WORK.ONE.
      WHERE a=1;
NOTE: The data set WORK.TWO has 1 observations and 3 variables.
```

In the next example the data set ONE is set twice, once with the WHERE= data set option, once without. SAS applies the WHERE= data set option to one input data set, but not to the other. The WHERE statement applies only

to the first input data set, the one without the data set option. As a result, the final output data set contains 4 observations, one of which has the value of 3 for the variable B, which appears to conflict with the WHERE statement.

```
data two;
  set one
      one(where=(a=1));
  where b=2;
run;

NOTE: There were 3 observations read from the data set WORK.ONE.
      WHERE b=2;
NOTE: There were 1 observations read from the data set WORK.ONE.
      WHERE a=1;
NOTE: The data set WORK.TWO has 4 observations and 3 variables.
```

Obs	a	b	c
1	1	3	3
2	2	2	3
3	2	2	3
4	3	2	3

Figure 1. Results of data step.

WHERE statement versus subsetting IF

The next several examples demonstrates that a WHERE statement is not always the same as a subsetting IF. The first two steps below produce exactly the same data sets. The step using the WHERE statement is more efficient because only 9 of the 1500 records were read. The subsetting IF is required to have the data in the program data vector before the condition can be evaluated.

```
data group1;
   set expose;
   if return > 15;
run;


NOTE: There were 1500 observations read from the data set EXPOSE.
NOTE: The data set WORK.GROUP1 has 9 observations and 14 variables.

data group1;
   set expose;
   where return > 15;
run;


NOTE: There were 9 observations read from the data set EXPOSE.
      WHERE return >15;
NOTE: The data set WORK.GROUP1 has 9 observations and 14 variables.
```

The step below successfully uses FIRST.ID as a subsetting IF.

```
data first_date;
   set expose;
   by id date;
   if first.id;
run;

NOTE: There were 1500 observations read from the data set WORK.EXPOSE.
NOTE: The data set WORK.FIRST_DATE has 243 observations and 2 variables.
```

When the subsetting IF is replaced with an equivalent WHERE statement, the step fails.

```
data first_date;
   set expose;
   by id date;
   where first.id;
         -------
         180
ERROR: Syntax error while parsing WHERE clause.
ERROR 180-322: Statement is not valid or it is used out of proper order.
```

The WHERE statement fails because data not meeting the WHERE condition never makes it into the program data vector, and since FIRST.ID cannot be determined until the data are in the program data vector, SAS cannot continue.

Attempting to use a WHERE statement with a variable that exists in the program data vector but not in the input data will cause a different error.

```
data first_date;
   set expose;
   by id date;
   where code=1;
ERROR: Variable code is not on file WORK.EXPOSE.
   code + 1;
run;

NOTE: The SAS System stopped processing this step because of errors.
```

Subsetting IF statement versus WHERE statement used with END= data set option

Using the WHERE statement below, only two records are read from the input data set. Since the data step only sees two records, the END= option recognizes the last observation has been read and the ôLAST OBS REACHEDô message is written to the log.

```
data a;
  a = 1; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output;  /** duplicate **/
  a = 3; b = 2; c = 3; output;
  a = 3; b = 3; c = 3; output;
run;

data b;
  set a end=last;
  where a=2;
  if last then put "LAST OBS REACHED";
run;

LAST OBS REACHED
NOTE: There were 2 observations read from the data set WORK.A.
      WHERE a=2;
NOTE: The data set WORK.B has 2 observations and 3 variables.
```

When a subsetting IF statement is used in place of the WHERE statement, the last observation never reaches the ñif last thenòstatement and the ôLAST OBS REACHEDômessage is not written to the log.

```
data b;
  set a end=last;
  if a = 2;
  if last then put "LAST OBS REACHED";
run;

NOTE: There were 5 observations read from the data set WORK.A.
```

```
NOTE: The data set WORK.B has 2 observations and 3 variables.
```

Different positioning of the subsetting IF statement would produce different results, but could also further reduce efficiency if the data step were to contain more program statements.

SUMMARY OF WHERE STATEMENT VERSUS SUBSETTING IF

- The WHERE statement is executed on the way in. Observations not meeting the WHERE condition are never added to the program data vector.
- The subsetting IF is executed after the observations reach the program data vector.
- When the WHERE statement is used with a BY statement, the WHERE statement is executed first.
- When a subsetting IF is used with a BY statement, the BY statement is processed first.
- When the WHERE statement is used with a MERGE statement, the WHERE statement is executed first.
- When a subsetting IF statement is used with a MERGE statement, the MERGE is executed first.

RENAME statement

- The RENAME statement specifies variables to be renamed and their new names.
- The RENAME statement takes effect on the way out.
- One RENAME statement applies to all output data sets in the data step and can be used anywhere in the data step.
- Multiple RENAME statements can be used in a data step. All will apply to the output data sets.
- The program data vector contains the variables with their old variable names, therefore, programs should continue to use the old variable names for processing in the current data step.

RENAME= data set option

- The RENAME= data set option used on an input data set will rename variables to the new name on the way in. The program should use the new variable names for the processing in the current data step.
- The RENAME= data set option used on an output data set will rename variables to the new name on the way out. The program should use the old variable names for the processing in the current data step.
- All data set options must be specified for each data set to which they apply.
- Multiple RENAME= data set options can be used with any one data set.

HIERARCHY

The order of execution of these operations is:

1. DROP
2. KEEP
3. RENAME

If statements are combined with data set options, the hierarchy is the same within the effective timing described above.

Below several SAS statements and data set options have been combined to demonstrate timing and hierarchy.

```
data one;
  a = 1; b = 3; c = 3; output;
  a = 2; b = 1; c = 3; output;
  a = 2; b = 2; c = 3; output;  /** duplicate **/
  a = 3; b = 2; c = 3; output;
  a = 3; b = 3; c = 3; output;
run;

1  data two(keep=new_code rank a b rename=(rank=index));
2  set one(rename=(c=code1)) end=last;
3  by b;
4  drop b;
5  rank = b;
6  where a = 2;
```

```
7  rename code1=new_code;
8  keep code1 rank a b;
9  put _all_;
10 run;

   WARNING: The variable b in the DROP, KEEP, or RENAME list has never been
   referenced.
   WARNING: The variable b in the DROP, KEEP, or RENAME list has never been
   referenced.
   last=0 a=2 b=1 code1=3 FIRST.b=1 LAST.b=1 rank=1 _ERROR_=0 _N_=1
   last=1 a=2 b=2 code1=3 FIRST.b=1 LAST.b=1 rank=2 _ERROR_=0 _N_=2
   NOTE: There were 2 observations read from the data set WORK.ONE.
         WHERE a=2;
   NOTE: The data set WORK.TWO has 2 observations and 3 variables.

   proc print data=two;
   run;

   NOTE: There were 2 observations read from the data set WORK.TWO.
```

Obs	a	new_code	index
1	2	3	1
2	2	3	2

Figure 2. Results of data step.

The RENAME= data set option on the input data set ONE in line number 2 is the first operation to take effect, renaming the variable C to CODE1. The variable C does not exist on the program data vector because the variable was changed on the way in.

The next operation is to read data from the input data set ONE. The WHERE statement in line number 6 will function at this time. Only records meeting the condition will be read into the program data vector.

Two executable statements exist in this data step.

- Line number 5: the variable RANK is populated with the contents of the variable B. The variable RANK will take the type and length attributes of variable B.
- Line number 9: writes the contents of the program data vector to the log.

The put statement in line 9 shows that the program data vector contains the following variables:

- LAST (from the END= option to the SET statement);
- A, B, and CODE1 (formerly named C) from the input data set ONE;
- FIRST.B and LAST.B (from the BY statement);
- RANK (from the assignment statement in line 5);
- _ERROR_ and _N_ (automatic variables)

Note the value of the temporary variable LAST is 0 (false) for the first observation output to the log, and 1 (true) for the second. The second output observation is the third observation in the data set ONE, not the last. This observation receives the value 1 indicating it is the last observation in the data set because of the WHERE statement on line 6. Only two of the original five observations in the data set ONE met the condition in the WHERE statement, thus the second of the two was the last in the data set and receives the indicator setting LAST to 1.

The DROP, RENAME, and KEEP *statements* in lines 4, 7, and 8 will be performed on the way out as the observation is written to the output data set in the following order:

- The DROP statement will prevent the variable B from being written to the output data set. The variable B is still on the program data vector and available for use and is being used in the assignment statement in line 5.
- The KEEP statement will limit which variables are written to the output data set to only the variables listed. The variable B is on the KEEP statement as well as the DROP statement. The DROP executes first. When the KEEP executes, the first WARNING message is generated because the variable B no longer exists to be written.
- The RENAME statement will change the variable name CODE1 to NEW_CODE. The RENAME occurs after the KEEP statement, therefore the KEEP statement uses the old name.

As the observation is written to the output data set, additional instructions are provided by the KEEP= and RENAME= data set options.

- The KEEP= data set option executes first keeping only the variables NEW_CODE, RANK, A, and B. Since B has been previously dropped a second WARNING message is generated.
- The RENAME= data set option then changes the variable name RANK to INDEX.

The PROC PRINT shows all variables in the output data set. The output data set contains three variables: A, NEW_CODE, and INDEX. The temporary variables LAST, FIRST.B and LAST.B and the automatic variables _ERROR_ and _N_ were not written to the output data set. The order of the variables in the output data set is determined by when they were encountered in the data step.

- A came in first from the input data set;
- B was dropped;
- C was renamed to CODE1, then renamed to NEW_CODE;
- RANK was created using an assignment statement, then renamed to INDEX.

Therefore, the final order of the variables is A, NEW_CODE, and RANK.

RETAIN

The RETAIN statement specifies variables whose values are **not set to missing** at the beginning of each iteration of the DATA step, causing SAS to hold the value from one iteration of the DATA step to the next. Only variables not in the input data sets can be retained. The next example illustrates a number of different scenarios that are explained in detail below.

```
data one;
  a = 1; b = 1; c=5;output;
  a = 1; b = .; c=5;output;
  a = 1; b = .; c=5;output;
run;

data two;
  d = 0; e = 0; output;
  d = 1; e = 0; output;
  d = 2; e = 0; output;
run;

data three;
  set one
      two;
  retain a 5 b;
  if _n_ = 1 then f = 0;
  a + 1;      /** in retain stmt **/
  b = b + 1;  /** in retain stmt **/
  e + 1;      /** sum stmt, auto retain **/
  f = f + 1;  /** NOT retained **/
  g + 1;      /** sum stmt, auto retain **/
run;
```

Obs	a	b	c	d	e	f	g
1	2	2	5	.	1	1	1
2	2	.	5	.	2	.	2
3	2	.	5	.	3	.	3
4	1	.	.	0	1	.	4
5	2	.	.	1	1	.	5
6	3	.	.	2	1	.	6

Figure 3. Results of data step.

Variable A – input from data set ONE; variable is retained (explicitly and implicitly); value is calculated. The RETAIN, both explicit and implicit have no obvious effect on the first three observations because the value comes from the input data set. When the second input data set is encountered, the value is set to missing and the effects of the RETAIN become evident. At observation 4, the value is set to missing due to its non-existence in the second input data set, the sum statement calculates the value on observation 4: missing (input value) + 1 = 1 and a new value is output.

Variable B – input from data set ONE; variable is retained; value is calculated. Though the variable is explicitly retained, the value is not retained from the first observation because the variable comes from one or more of the input data sets.

Variable C – input from data set ONE. At observation 4 the variable is reset to missing when the second input data set is encountered.

Variable D – input from data set TWO. The variable exists in the first three output observations, though it has a missing value. The variable values are introduced to the program data vector starting at observation 4.

Variable E – input from data set TWO; implicitly retained by the sum statement; value is calculated. The sum statement calculates the value on observation 1: missing (input value) + 1 = 1. At the second observation the value is incremented by one, and again in the third observation. At observation 4, the value is set to 0 by the second input data set. The value is then incremented and output. At observation 5 the value is again set to 0 by the second input data set, incremented, and output.

Variable F – value is initialized when _N_ is 1; value is calculated. The variable is not on an input data set, and it is not retained, therefore, at observation 2, it is reset to missing. The statement used to increment the variable F is not a sum statement, therefore, when the value is reset to missing at observation 2, the result of the addition is missing.

Variable G – value is calculated; implicitly retained by the sum statement. The value is not reset to missing at observation 4 because the value is retained and is not on an input data set.

SUMMARY OF SOME OF THE RULES OF RETAIN

- Retaining a variable on an input data set has no effect. Their values will be set by the input data set.
- At the start of each iteration of the data step, non-retained variables created in a data step are set to missing.
- Variables created with a sum statement are automatically retained.

If a variable name is specified in the RETAIN statement with no initial value and receives no value from processing, the variable is not written to the data set, and a note stating that the variable is uninitialized is written to the SAS log.

In the example below, variable A is retained with an initial value of missing. Variables B and C are retained with no initial values. Of the three variables in the data step, only two are written to the output data set. Variable C is not written because it has no value, nor have its attributes been assigned.

```
data one;
  retain a . b c;
  format b 8.3;
run;

NOTE: Variable b is uninitialized.
NOTE: Variable c is uninitialized.
NOTE: The data set WORK.ONE has 1 observations and 3 variables.
```

Obs	a	b
1	.	.

Figure 4. Results of data step.

PART 2:
HOW TO USE, ABUSE, MANIPULATE, AND EXLOIT THE PROGRAM DATA VECTOR

This section will show how to take advantage of the program data vector's strengths and weaknesses, better understand why SAS does what it does, and some ways to get it to do things you want it to do.

CHANGING LENGTH OF EXISTING VARIABLE

```
data one;
  var1 = 'abc';
run;

data two;
  set one;
  attrib var1 length=$5;
WARNING: Length of character variable has already been set.  Use the LENGTH
statement as the very first statement in the DATA STEP to declare the length of a
character variable.
run;

NOTE: There were 1 observations read from the data set WORK.ONE.
NOTE: The data set WORK.TWO has 1 observations and 1 variables.
```

The LENGTH (or in this case ATTRIB) statement should appear before the SET statement so that the variable with its new attributes enters the program data vector before the SET statement. The program data vector variable attributes will initially come from the first statement. Subsequent statements can augment the variable attributes.

```
data two;
  attrib var1 length=$5;
  set one;
run;

NOTE: There were 1 observations read from the data set WORK.ONE.
NOTE: The data set WORK.TWO has 1 observations and 1 variables.
```

MISSING AS A DATA STEP KEYWORD

Some programmers use MISSING as a data step keyword. No documentation exists describing the existence of MISSING as a data step keyword. MISSING is supported by PROC SQL, but not by the data step, at least not explicitly.

This program checks to see if the variable DATE is null.

```
data one;
  date = '04may2003'd; output;
  date = '05may2003'd; output;
  date = '06may2003'd; output;
  date = .           ; output;
  date = '07may2003'd; output;
run;

data two;
  set one;
  if date = missing;
run;

NOTE: Variable missing is uninitialized.
NOTE: There were 5 observations read from the data set WORK.ONE.
NOTE: The data set WORK.TWO has 1 observations and 2 variables.
```

The expected results are received in spite of the NOTE indicating that the variable MISSING is uninitialized. The variable MISSING is uninitialized which ultimately gives it a missing value. The uninitialized variable MISSING is then compared to data set variables to locate null values. This works nicely, but for all the wrong reasons.

When the variable MISSING is created in the program data vector it takes the type and length attributes of the variable to which it is being compared. In the example above, MISSING becomes a numeric variable of length 8 with a value of missing.

In the example below, the character variable TEXT is being checked for missing values. In this case the variable MISSING will take on the type and length attributes of the variable TEXT: character type, length of 3.

```
data one;
  text = 'abc'; output;
  text = 'def'; output;
  text = '';    output;
  text = 'ghi'; output;
run;

data two;
  set one;
  if text = missing;
run;
NOTE: Variable missing is uninitialized.
NOTE: There were 4 observations read from the data set WORK.ONE.
NOTE: The data set WORK.TWO has 1 observations and 2 variables.
```

There is nothing particular about the variable name MISSING, it could just as well have been any other user defined variable name such as NULLTEXT, ALPHA, or TESTER.

RETAINED OR NOT?

Given the data set below, VAR2 needs to have the value of 1 for every occurrence of X in VAR1 and 2 for every occurrence of Z.

var1	var2
X	.
Z	.
Z	.
X	.
X	.

Figure 5. Source data set

```
 proc sort data=one;
  by var1;
  run;

 data two;
  set one;
  by var1;
  if first.var1 then var2 + 1;
  put var1=  var2=;
run;
```

```
var1=X var2=1
var1=X var2=.
var1=X var2=.
var1=Z var2=1
var1=Z var2=.
```

Figure 6. Results of data step.

The first attempt looks like a RETAIN statement is necessary since VAR2 is null on all observations but where VAR1 changes é but, isn't the sum statement automatically retained? Well, let's try adding an explicit RETAIN statement:

```
data two;
  retain var2;
  set one;
  by var1;
  if first.var1 then var2 + 1;
  put var1=  var2=;
run;
```

```
var1=X var2=1
var1=X var2=.
var1=X var2=.
var1=Z var2=1
var1=Z var2=.
```

Figure 7. Results of data step.

There is no change after adding an explicit RETAIN. Why doesn't the RETAIN retain? The problem comes because the variable VAR2 is on the input data set. Retained or not, the variable VAR2 is reset by the SET statement. One solution is to use a DROP= data set option to remove the variable VAR2 on input.

```
data two;
  set one(drop=var2);
  by var1;
  if first.var1 then var2 + 1;
  put var1=  var2=;
run;
```

```
var1=X var2=1
var1=X var2=1
var1=X var2=1
var1=Z var2=2
var1=Z var2=2
```

Figure 8. Results of data step.

OVERWRITTEN VARIABLE ATTRIBUTES

The two data sets below both contain the variable NAME, but with different attributes. When they are SET together some attributes are changed.

```
data one;
  name = 'Mary';
run;

data two;
  label name="Subject name";
  name = 'Scott';
run;

data all;
  set one
        two;
  put name =;
run;
```

```
name=Mary
name=Scot
```

Figure 9. Results of data step.

The length attribute from data set ONE overrides the length attribute of the variable in data set TWO resulting in truncated values.

When the data set ALL is printed using the following PROC PRINT, the statement above seems to be contradicted because the variable label from data set TWO is used.

```
proc print data=all label;
run;
```

Obs	Subject name
1	Mary
2	Scot

Figure 10. Results of Proc Print.

Variable attributes INFORMAT, FORMAT, and LABEL can be set multiple times. Variable attributes LENGTH and TYPE can be set only once.

When data set ONE is read the program data vector sets up the variable NAME with a length of $4 and a null label. When data set TWO is encountered, the length of NAME is different, but the length has already been set and cannot be changed. The variable NAME from data set TWO has a label assigned which will replace the existing label, in this case the existing label was blank.

CHANGING A VARIABLE'S ATTRIBUTES WITHOUT CHANGING ITS NAME IN ONE DATA STEP

This technique is very simple given the features described in this paper. In the example below the input data set DEMOG contains the character variable AGE. Using a combination of the DROP= and RENAME= data set options, the variable AGE is changed from character to numeric in one data step.

```
data demog;
    age = '25';

data demog(rename=(num_age=age)
    drop=age);
  set demog;
  num_age = input(age,8.);
```

This works because the order the data set options are processed is DROP=, KEEP=, RENAME=. Thus the original variable AGE is dropped, then the numeric variable NUM_AGE is renamed to AGE as the data are written to the output data set.

MULTIPLE DATA SETS IN THE SET STATEMENT

Are the following programs equivalent? Obviously not, or they would not be in this paper! Take a moment to review each program. Be assured, there are no tricks in either program. On the surface, based on everything we all know about SAS, these programs are equivalent.

PROGRAM 1

```
1  data version2 failure;
2     set round1 round2 round3;
3     if version ne v2 then do;
4        output failure;
5        v2=version;
6     end;
7     output version2;
8  run;
```

PROGRAM 2

```
data combined;
   set round1 round2 round3;
run;

data version2 failure;
   set combined;
   if version ne v2 then do;
      output failure;
      v2=version;
   end;
   output version2;
run;
```

Figure 11. Two logically equivalent programs.

The difference between the two programs is that program 1 sets all the data sets and does the processing in one step and program 2 sets the data sets on one step and does the processing in a second step. The logic for both programs are exactly the same. The variables VERSION and V2 are only input variables, so no unexpected messages will be generated.

If these programs are exactly the same, then why are the outputs different?

Partial log from program 1:

```
NOTE: There were 3 observations read from the data set WORK.ROUND1.
NOTE: There were 3 observations read from the data set WORK.ROUND2.
NOTE: There were 3 observations read from the data set WORK.ROUND3.
NOTE: The data set WORK.VERSION2 has 9 observations and 2 variables.
NOTE: The data set WORK.FAILURE has 3 observations and 2 variables.
```

Partial log from program 2:

```
NOTE: There were 3 observations read from the data set WORK.ROUND1.
NOTE: There were 3 observations read from the data set WORK.ROUND2.
NOTE: There were 3 observations read from the data set WORK.ROUND3.
NOTE: The data set WORK.COMBINED has 9 observations and 2 variables.

NOTE: There were 9 observations read from the data set WORK.COMBINED.
NOTE: The data set WORK.VERSION2 has 9 observations and 2 variables.
NOTE: The data set WORK.FAILURE has 5 observations and 2 variables.
```

Program 1 produces 3 observations in the FAILURE data set while program 2 produces 5 observations in the FAILURE data set. Why are they different? More importantly, which is correct?

The program and log provide no hint what the problem is. The trouble lies in the data itself. Closer inspection of the data shows that data set ROUND2 does not contain the variable V2.

	version	v2
round1	1	1
	1	1
	1	4
round2	2	
	2	
	2	
round3	3	3
	3	.
	3	3

Figure 12. Input data.

When program 1 runs:

- SAS compiles the program and creates the program data vector, in this case with the variables VERSION and V2.
- When the program executes each record is read in and if VERSION is not equal V2, the faulty data are written to the FAILURE data set and V2 is set to the value of VERSION, and all records are written to the VERSION2 data set.
- In the last observation of ROUND1, VERSION is not equal to V2, so V2 is set to VERSION, in this case 4. The FAILURE data set now contains 1 observation.
- In the first observation of ROUND2 the variable V2 does not exist on the input data set ï there is a difference between being missing and non-existent ï in this case V2 is non-existent, however V2 is on one or more of the other input data sets, so the variable is not set to missing at the top of the DATA step. The value of VERSION (2) comes from the input data set ROUND2 and the value of V2 (4) remains from the previous data step iteration. The program execution sets V2 equal to VERSION because the values are not equal. The FAILURE data set now contains 2 observations.
- The second observation of ROUND2 the variable V2 is non-existent but is not set to missing because it exists on one or more of the input data sets. The value of VERSION (2) is provided by the input data set and the value of V2 (2) remains from the previous execution of the data step. For this observation the values are equal so a record is not written to the FAILURE data set.
- The third observation of ROUND2 processes exactly as the second observation.
- The first observation of ROUND3, the value of VERSION (3) is provided by the input data set and the value of V2 (3) also comes from the input data set.
- The second observation of ROUND3, the value of VERSION (3) is provided by the input data set and the value of V2 (.) also comes from the input data set. VERSION is not equal to V2, so V2 is set to VERSION and the FAILURE dataset now contains 3 observations.
- The last observation of ROUND3, the value of VERSION (3) is provided by the input data set and the value of V2 (3) also comes from the input data set.

PROGRAM 1 places the following observations in the FAILURE data set

```
          version    v2
round1      1         1
            1         1
            1         4  <-- failure

round2      2            <-- failure
            2
            2

round3      3         3
            3         .  <-- failure
            3         3
```

Figure 13. Program 1 failures.

When program 2 runs:

DATA STEP 1

- The first step combines the three data sets without doing any other processing.
- SAS compiles the program and creates the program data vector, in this case with the variables VERSION and V2.
- The data from ROUND1 is read directly into the COMBINED data set.
- As the data from ROUND2 are read the non-existent variable V2 is set to missing in the COMBINED data set.

- The data from ROUND3 is read directly into the COMBINED data set.

DATA STEP 2

- When the program executes each record is read in and if VERSION is not equal V2, the faulty data are written to the FAILURE data set and V2 is set to the value of VERSION, and all records are written to the VERSION2 data set.
- Observation 3 (originally the last observation of ROUND1), VERSION (1) is not equal to V2 (4), so V2 is set to VERSION. The FAILURE data set now contains 1 observation.
- Observation 4 (originally the first observation of ROUND2), VERSION (2) is not equal to V2 (.), so V2 is set to VERSION. The FAILURE data set now contains 2 observations.
- Observation 5 (originally the second observation of ROUND2), VERSION (2) is not equal to V2 (.), so V2 is set to VERSION. The FAILURE data set now contains 3 observations.
- Observation 6 (originally the last observation of ROUND2), VERSION (2) is not equal to V2 (.), so V2 is set to VERSION. The FAILURE data set now contains 4 observations.
- Observation 7 (originally the first observation of ROUND3), VERSION (3) is equal to V2 (3), so a record is not written to the FAILURE data set.
- Observation 8 (originally the second observation of ROUND3), VERSION (3) is not equal to V2 (.), so V2 is set to VERSION. The FAILURE data set now contains 5 observations.
- Observation 9 (originally the last observation of ROUND3), VERSION (3) is equal to V2 (3), so a record is not written to the FAILURE data set.

PROGRAM 2 places the following observations in the FAILURE data set

```
          version    v2
round1      1         1
            1         1
            1         4  <-- failure

round2      2            <-- failure
            2            <-- failure
            2            <-- failure

round3      3         3
            3         .  <-- failure
            3         3
```

Figure 14. Program 2 failures.

THE MAGIC OF THE DOW-LOOP

The Whitlock DO-loop, known as the DOW-loop, is a clever programming technique that is being accepted and used by more and more programmers. Many papers have been written about the DOW-loop, some of which are referenced in the *Recommended Reading* section below. This paper will not attempt to debate the uses and advantages of this programming technique, but rather will simply describe how it actually works.

Below is a sample DOW-loop.

```
1  data averages(drop=score: total);
2     do scores = 1 by 1 until (last.student);
3        set grades;
4        by student;
5        total = sum(total,score);
6     end;
7     average = total / scores;
8  run;
```

The DOW-loop puts the SET statement inside an explicit control structure, in this case a DO é UNTIL loop, thus manipulating the program data vector to the programmerôs advantage.

Instead of controlling the SET statement with the implicit loop between the DATA statement (1) and the step boundary (8) it is controlled with an explicit loop between the DO é UNTIL (2) and the loop end (6).

With control of the SET returning to the explicit DO loop instead of the DATA statement, it prevents the non-retained variables from being reset to missing.

During compilation the program data vector is built in the order the variables are encountered.

SCORES	LAST. STUDENT	STUDENT	SCORE	FIRST. STUDENT	TOTAL	AVERAGE	_ERROR_	_N_

Table 1. Program Data Vector as created by data step.

For convenience, the variables _ERROR_ and _N_ will be omitted, and the FIRST. and LAST.STUDENT variables will be shown together at the end of the program data vector.

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT

Table 2. Abbreviated Program Data Vector.

LINE 1: The program data vector is initialized and contains

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT
.	.	.	.	.	1	1

Table 3. Program Data Vector after line 1.

LINE 2: The DO loop begins with an initial value of 1

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT
1	.	.	.	.	1	1

Table 4. Program Data Vector after line 2.

LINE 3: The SET statement reads the input data. Notice the change in value for LAST.STUDENT.

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT
1	1	100	.	.	1	0

Table 5. Program Data Vector after line 3.

LINE 5: The value of TOTAL is calculated

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT
1	1	100	100	.	1	0

Table 6. Program Data Vector after line 5.

LINE 6: Control returns to the initiating DO statement in line 2 because LAST.STUDENT is not true. Because control did not go to the implicit loop DATA statement, the value the non-retained variables SCORES, TOTAL, and AVERAGE are *not* set to missing.

LINE 2: The DO loop increments the value of SCORES by 1

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT
2	1	100	100	.	1	0

Table 7. Program Data Vector after line 2.

LINE 3: The SET statement reads the input data. This observation is the LAST.STUDENT.

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT
2	1	80	100	.	0	1

Table 8. Program Data Vector after line 3.

LINE 5: The value of TOTAL is calculated

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT
2	1	80	180	.	0	1

Table 9. Program Data Vector after line 5.

LINE 6: Control exits the DO loop because LAST.STUDENT is now true.

LINE 7: The value of AVG is calculated

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT
2	1	80	180	90	0	1

Table 10. Program Data Vector after line 7.

LINE 8: The observation is written to the output data set AVERAGES and control returns to the DATA statement.

LINE 1: The program data vector is initialized and contains

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT
.	1	80	.	.	0	1

Table 11. Program Data Vector after line 1.

The variables SCORE, TOTAL, and AVERAGE have been reset to missing at the DATA statement because they were not retained and not on the input data set. The variables STUDENT and SCORE were not set to missing at the DATA statement because they are on the input data set and will not be changed until the SET statement. The FIRST. and LAST.STUDENT flags have not been reset because will not be changed until the SET statement.

Control moves to line 2, the DO loop, SCORES is set to 1, STUDENT, SCORE, FIRST. And LAST.STUDENT are unchanged.

SCORES	STUDENT	SCORE	TOTAL	AVERAGE	FIRST. STUDENT	LAST. STUDENT
1	1	80	.	.	0	1

Table 12. Program Data Vector after line 2.

Control moves to line 3, the SET statement, and since no data remains, the DATA step exits.

TEMPLATES

Templates can be used to get what you want from a data step. This is an efficient technique, but care needs to be used to manipulate the program data vector appropriately. As indicated earlier in this paper, some variable attributes can be changed multiple times. This technique works best if subsequent data sets do not have the attributes FORMAT, INFORMAT, and LABEL set. If they do contain attributes undefined in the template or are different from those in the template, those attributes will come through in the final data set. A trick to fix this is shown at the end of this section.

Create a template data set, with zero observations, that define the variables and attributes desired in the final data.

```
data demog;
  attrib pt    length=8  label='subject number'  format=z6.;
  attrib sex   length=$6 label='subject gender';
  attrib age   length=3  label='subject age';
  attrib race  length=8  label='ethnic origin'   format=race.;
  pt = .;  sex = '';  age = .;  race = .;
  if _n_ = 0;
run;

NOTE: The data set WORK.DEMOG has 0 observations and 4 variables.
```

The statement

```
If _n_ = 0;
```

allows the data set to be created with zero observations.

```
data garbage;
  pt     = 101;
  sex    = 'male';
  age    = 21;
  dob    = '21mar1985'd;
  adddte = '10jan2003'd;
  moddte = '15jan2003'd;
  race   = 2;
  output;
run;

NOTE: The data set WORK.GARBAGE has 1 observations and 7 variables.
```

The data set GARBAGE contains variables used in data processing. Many intermediate and unnecessary variables exist. The desired variables do not have the required attributes. Some of the lengths are wrong, there are no formats, informats, or labels assigned.

Using the template created earlier in the SET statement will set the desired attributes. The template data set must be referenced first, which will load its variable attributes into the program data vector. When the second data set is encountered, the values will be ñforcedòinto the attributes already defined in the program data vector. Using the template dataset again as the last in the SET statement protects against the possibility that one or more of the earlier data sets changed the format, informat, or label attributes. The first use of the template permanently sets the type and length of the variables. The second use of the template corrects any changes in LABEL, FORMAT, or INFORMAT imposed by other data sets.

Since the data set FINAL will have more variables than defined in the template DEMOG, adding a KEEP or DROP statement will limit the variables in the data set.

```
data final;
  set demog    /** TEMPLATE, zero observations     **/
      garbage  /** data set with all sorts of junk **/
      demog;   /** TEMPLATE, zero observations     **/
  keep pt sex age race;
run;
```

```
NOTE: There were 0 observations read from the data set WORK.DEMOG.
NOTE: There were 1 observations read from the data set WORK.GARBAGE.
NOTE: There were 0 observations read from the data set WORK.DEMOG.
NOTE: The data set WORK.FINAL has 1 observations and 7 variables.
```

_N_

This one I found while researching the DOW loops and just found it so interesting I had to include it here.

Paul Dorfman, in his paper ñThe DOW-Loop Unrolledò, concludes that ñthe data step has an internal counter of the implied loop iterations and simply moves its next value to _N_ once program control is once again at the top of the stepò

The following example shows that you cannot change the *internal* value of _N_.

```
data one;
put 'TOP ' _n_=;
   input _n_ student score;
   if _n_ = 3 then score=99;
put _all_;
cards;
1 1 100
2 1 100
3 1 100
1 2 100
2 2 75
3 2 75
run;


TOP _N_=1
student=1 score=100 _ERROR_=0 _N_=1
TOP _N_=2
student=1 score=100 _ERROR_=0 _N_=2
TOP _N_=3
student=1 score=99 _ERROR_=0 _N_=3
TOP _N_=4
student=2 score=100 _ERROR_=0 _N_=1
TOP _N_=5
student=2 score=75 _ERROR_=0 _N_=2
TOP _N_=6
student=2 score=99 _ERROR_=0 _N_=3
TOP _N_=7
NOTE: The data set WORK.ONE has 6 observations and 2 variables.
```

The program logic maintained and used its own value of _N_ because when _N_=3 for each student the score was set to 99. SAS maintained and used its own internal value of _N_.

CONCLUSION

Knowing that all the variables read are present, whether ultimately kept, dropped, or renamed, knowing when and how the program data vector processes the variables it encounters, knowing the difference between SAS statements and SAS data set options, and knowing how to exploit their timings provides the programmer with valuable

information to understand why SAS does what it does, and how it can be manipulated to enhance the operation of the program.

With this knowledge, and the simple tools provided by SAS, the programmer can change cinder block into bricks, and even bricks into pebbles. This goes a long way toward efficient programming.

What has been discussed in this paper is quite literally the tip of the iceberg. A good working knowledge of the program data vector can only be obtained through years of programming experience, experimentation, and trial and error. Readers are encouraged to research and experiment further on their own.

REFERENCES

SAS Institute Inc. (February 2000*) SAS OnlineDoc®, Version 9*, Cary, NC: SAS Institute Inc.

RECOMMEDED READING

- Brucken, Nancy. 2011. 2 PROC TRANSPOSEs = 1 DATA step DOW-Loop. *The Proceedings of the Pharmaceutical SAS Users' Group 2011 Conference.* Cary, NC: SAS Institute. Available at www.lexjansen.com/pharmasug/2011/cc/pharmasug-2011-cc26.pdf

- Brucken, Nancy. 2008. One-Step Change from Baseline Calculations, and Other DOW-Loop Tricks. *The Proceedings of the Pharmaceutical SAS Users' Group 2008 Conference.* Cary, NC: SAS Institute. Available at www.lexjansen.com/mwsug/2008/pharma/MWSUG-2008-P01.pdf

- Chakravarthy, Venky. 2003. The DOW (Not that DOW!!!) and the LOCF in Clinical Trials. *The Proceedings of the SAS Users' Group International 28 Conference*. Cary, NC: SAS Institute. Available at www2.sas.com/proceedings/sugi28/099-28.pdf

- Dorfman, Paul. 2009. The DOW-Loop Unrolled. *The Proceedings of the Northeast SAS Users' Group 2009 Conference.* Cary, NC: SAS Institute. Available at www.nesug.org/Proceedings/nesug09/bb/bb06.pdf

- Johnson, Jim. 2005. The Use and Abuse of the Program Data Vector (The Procs). *The Proceedings of the Pharmaceutical SAS Users' Group 2005 Conference.* Cary, NC: SAS Institute. Available at www.lexjansen.com/pharmasug/2005/technicaltechniques/tt03.pdf

ABOUT THE AUTHOR

Jim Johnson has been programming with SAS in the Pharmaceutical Industry since 1986. He has presented at many local, regional, and national conferences and has been teaching in the SAS Certificate Program at Philadelphia University since its inception in 1997. Jim has a reputation as a õproblem solverò and efficiency enthusiast. His recent work includes large SAS systems, writing programs that write programs, standard macro programming, and advanced validation and documentation skills.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at: stargazer2960@gmail.com.