Robust Programming Techniques in the SAS® System

Alice M. Cheng, Pfizer Inc., New York, New York

Abstract

Have you ever had a program that runs on some occasions and fails on others? Has a change in input data values affected the error-free execution of the program, or worse, produced incorrect results? Chances are that the program concerned is not robust enough. In this paper, the author provides robust techniques for documentation practices, data processing and macro-processing in the SAS system in order to avoid such problems.

Keywords: Robustness, Documentation, Data Processing, Macro-Processing.

Introduction

Robust programming is a defensive programming method to ensure accurate output and error-free execution, in spite of various changes. These changes include, but are not limited to, changes in personnel, data values, the existence/non-existence of certain observations and the system configuration.

Based on this definition, one can take numerous precautions to ensure robustness. However, one should be aware that robust programming techniques are often implemented at the expense of program efficiency and programmers’ time. Therefore, one should attempt a balance between robustness and efficiency.

A. Documentation Practices

Good documentation is critical not only to satisfy regulatory requirements, but also to ensure that latecomers to the project will have a clear understanding of the programs, be able to produce the accurate results and if necessary, be able to trace any errors.

A.1 Documentation in SAS Programs

Every SAS program should have a header --- comments at the beginning of a program. The following features are recommended in a header:

- The name of the program (including the path);
- The name of the project;
- The name and affiliation of the programmer;
- The date on which the program was first written;
- The name and affiliation of the reviewer;
- The date of review;
- Other project members such as project leaders, statisticians and data managers (optional);
- The purpose of the program;
- The version of SAS and the operating system used to execute the program;
- The input data set(s) used, if applicable;
- The macro(s) used, if applicable;
- The location of SAS log and output (including the path). Output can be a data file, a data set or a report;
- The location of the test plan for this program;
- Any special warnings or comments;
- For macros, the definition of any arguments and default values.

A change history section is critical to keep track of all the changes made to the program. This section should include:

- The name and affiliation of the programmer who made the changes;
- The date on which changes were made;
- The name and affiliation of the reviewer;
- The date of review on the revised program;
- The reasons for the changes;
- The location of SAS log and output of the revised program (including the path). Output can be a data file, a data set or a report;
- The location of test plan for the revised program;
- Any special warnings or comments.

In addition, programmers should provide clear and concise documentation of the purpose of each section of the SAS code. Such documentation can be embedded within the body of the program. Are you programming according to certain specification in a rulebook? If so, identify the name, date and version of the rulebook. Does someone define the rules or request for the changes? Document the name of that person and the date the request was made; summarize the reasons for the changes. Good documentation eliminates confusion and achieves robustness by ensuring the correct modification in any future edition of the program.

A.2. Documentation in SAS Log

SAS, by default, provides users with NOTE, WARNING and ERROR messages in the log. These messages are important and should be carefully examined. In the program development stage, however, additional warning and diagnostic messages are helpful. Users can define SAS options or write SAS code to generate additional messages in the SAS log.

A.2.1 Set SAS Options for More Diagnostic Messages

In addition to the regular NOTE, WARNING and ERROR messages,

```
OPTIONS MSGLEVEL = 1;
```

instructs SAS to print INFORMATORY messages about merging, sorting and optimization. (Reference [10], Page 92.) For instance,

```
INFO: The VARIABLE X ON DATA SET WORK.A WILL BE OVERWRITTEN BY DATA SET WORK.B.
```

is an informative message in the log that warns users of the consequence of a merge.

For macro developers, the statement

```
OPTIONS SYMBOGEN MPRINT MLOGIC;
```

is essential in development stage. It displays the macro variable reference values, the logic involved and the SAS code generated by a macro. As helpful as these statements are, programmers should turn off these options during a production execution to reduce the amount of output to the log.

A.2.2 Use of PUT and %PUT Statements to Display Messages

Often, even though a program runs error-free, there are certain values and observations that programmers should be aware of. PUT and %PUT statements empower programmers to display messages in the SAS log. As an example, consider the statements:

```
DATA _NULL_;
SET DEMOG;
IF _AGE >= 100 THEN
  PUT '(ALERT) PATIENT ’ PAT_ID ’ IS ’ _AGE ’ YEARS OLD.’;
RUN;
```

send warning messages to the log alerting the programmers of any centenarian in the DEMOG data set. It may be advisable to verify such an extreme age value.

Likewise, the %PUT statement can display macro variables and text in the log. This statement is more powerful in the sense that it is a global statement and can appear anywhere within a SAS program. There is no need to quote text when a %PUT statement is applied. For instance, the statements:

```sas
%PUT LIST OF MACRO VARIABLES DEFINED BY USERS;
%PUT _USER_;
```

display a list of user-defined macro variables and values.

In addition, %PUT statement can be used to inform programmers of the beginning and the ending of a macro. This is very helpful for debugging purposes. Moreover, %PUT statement can also be used to display warning messages to alert programmers of any missing or invalid values.

### A.2.3 Program Termination

In general, a robust program/macro should check the relevancy of input data values before it continues with the processing. In extreme cases of missing or invalid key variable values, the program execution should be aborted. Statements such as

```sas
ABORT ABEND N;
ABORT RETURN N;
```

enable the macro developer to terminate an execution with the specified return code of N, where N can be any integer ranging from 0 to 65,535. An error message will appear in the SAS log informing users of the termination. This differs from the statement:

```sas
OPTIONS ERRORABEND;
```

that causes the session to end abruptly for any SAS error, including syntax errors. When a program execution is terminated due to an error, a SAS log file with error messages will be produced if either of the followings is true:

- the –ALTLOG option has been defined in the configuration file or at invocation;
- the PRINTTO procedure with the LOG option has been specified at the beginning of the program.

### A.3 Documentation in SAS Output

In most work environments, the customer specifies the format of the program output. In highly regulated industries such as the pharmaceutical industry, it is critical that the output can be reproduced, if necessary. Displaying the date and the time of data extraction, the date and the time of report generation, the operating system and the location of the source code is useful to keep track of the production process.

The following automatic macro variables are useful for this purpose:

- `&SYSSCP` --- provides an abbreviated name of the operating system.
- `&SYSSCPL` --- provides a more specific name of the operating system.
- `&SYSVER` --- provides the release of SAS software used in execution.
- `&SYSDATE` --- provides the date at the beginning of the job session.
- `&SYSTIME` --- provides the time at the beginning of the job session.

Note that the values for `&SYSDATE` and `&SYSTIME` do not change during a job session. If you are really interested in the exact date and time a report is produced, consider using the `DATE()` and the `TIME()` functions, respectively. In addition, the date and the time of data extraction can be obtained from the variable MODATE in the output data set produced by the CONTENTS procedure. Occasionally, a report cannot be generated because of a lack of data. Hence, a robust program should check whether a standard report production is possible based on the current data. If a report cannot be produced, the program should produce a special report informing readers of the possible reasons for the absence of a standard report.

### B. Data Processing Techniques

A number of precautions can be taken to ensure robustness when processing SAS data. Below are some suggestions:

#### B.1. Guard Against Numeric Representation Errors

A numeric representation error is one of the common, subtle errors that most programmers will encounter. A minute discrepancy between two theoretically identical values can be detected in hexadecimal representation. It is advisable to implement tolerance when comparing numerical values.

Consider Example B.1 where P = 1 - 0.9 and Q = 0.1. Obviously, the values of P and Q are equal. Yet, under SAS Version 6.12 in the Windows environment, P and Q have different internal values; such a minute difference can be detected when the HEX16 format is applied. By employing a tolerance of .001, the minute difference due to a numeric representation error is disregarded; P and Q are correctly identified as equivalent in value.

**Example B.1 Example of Numeric Representation Error and the Use of Tolerance in Comparison**

(SAS Program)

```sas
DATA _NULL_;
P = 1 - 0.9;
Q = 0.1;
PUT 'WITHOUT FORMAT SPECIFICATION,'
   'P = ' P HEX16 . /
   'Q = ' Q HEX16. /;
IF P = Q THEN PUT 'P EQUALS Q.' /;
ELSE PUT 'P DOES NOT EQUAL Q.' /;
PUT 'WITH A TOLERANCE OF 0.001,'
   'IF ABS(P - Q) < .001 THEN'
   ELSE PUT 'P DOES NOT EQUAL Q.' /;
run;
```

(SAS Log)

```
WITHOUT FORMAT SPECIFICATION,
P = 0.1000000000000000
Q = 0.1000000000000000
WITH A TOLERANCE OF 0.001,
P = 0.1000000000000000
Q = 0.1000000000000000
```

The concept of tolerance can also be applied to data set comparison. When comparing two data sets using the COMPARE procedure, the METHOD option enables programmers to select the type of comparison applied to judge the equivalence of numeric values. The CRITERION option allows programmers to specify the magnitude of tolerance to be a value between 0 and 1. Interested

Example B.1 (continued)

```sas
IN HEXDECIMAL FORMAT,
P = 3FB9999999999999
Q = 3FB9999999999999
WITHOUT TOLERANCE,
P DOES NOT EQUAL Q.
WITH A TOLERANCE OF 0.001,
P EQUALS Q.
```

B.2 Handle Character Values With Care

B.2.1 Input of Character Values

Depending on the data file definition, inputting data can be a challenging process. In extreme cases, one has to write extensive SAS code in order to read in the data correctly. When inputting character values, one has to be aware of the following:

- the layout of the data file;
- the longest possible length of each character string and record;
- some spaces may be part of the character value, not the delimiters;
- the existence/non-existence of delimiters;
- the character that is used as a delimiter, if any;
- the existence/non-existence of semi-colons as part of the data.

Armed with this knowledge, let us consider Examples B.2.1a and B.2.1b.

Example B.2.1a Input of Character Strings by Assignment Statements

(SAS Program)
DATA COACH1;
  NAME='PHIL JACKSON';
  OUTPUT COACH1;
  NAME='JEFF VAN GUNDY';
  OUTPUT COACH1;
RUN;

(Output Data Set)
SAS DATA SET COACH1
  NAME
  PHIL JACKSON
  JEFF VAN GUNDY

(Comments)
Note that 'JEFF VAN GUNDY' has been truncated to 'JEFF VAN GUN' in output data set COACH1. NAME is of character type with length = 12, the number of characters in 'PHIL JACKSON'. SAS uses the length of the value of the variable in the first assignment statement as the length for that variable.

(Resolution)
Use either of the following LENGTH or ATTRIB statement directly after the DATA statement:

LENGTH NAME $ 14.;
ATRIB NAME LENGTH=$14;

Note: 14 is the length of the string 'JEFF VAN GUNDY', the longest record for the variable NAME.

(Further Comments)
Note that if an INPUT statement is used, by default, SAS uses a blank as a delimiter and assigns any character variable to be of length = $8. Therefore, in this example, with

INPUT NAME $;

PHIL and JEFF will be the values for NAME in the first and second observations, respectively.

Furthermore, if the leading blanks are to be inputted, the starting column has to be specified and the $CHARw. informat should be used. The w value specifies the number of characters for the variable with this informat, including the leading blanks.

Example B.2.1b Input of Character String Embedded With Blank and Semi-Colons

(SAS Program)
DATA COACH2;
  INPUT NAME : & $14. ;
  CITY : & $30. ;
CARD4;
PHIL JACKSON CHICAGO; LOS ANGELES
JEFF VAN GUNDY NEW YORK
RUN;

(Output Data Set)
SAS DATA SET COACH2
NAME
PHIL JACKSON CHICAGO; LOS ANGELES
JEFF VAN GUNDY NEW YORK

(Comments)
Because of the existence of a semi-colon within the data, it is critical that a CARD4 statement and 4 semi-colons (;;) are used to signify the beginning and the ending of the data. In addition, the colons (:) in the INPUT statement inform SAS to read from the next non-blank character. The ampersand (&) tells SAS that character values can be embedded with single blanks. Until two consecutive blanks have been reached, the characters would be considered to be part of the value of a character variable.

B.2.2 Comparison of Character Data

When comparing character values, one has to decide if case and blanks are important. SAS programmers should beware of the following:

- Leading and Trailing Blanks
  SAS ignores trailing blanks, but it considers leading blanks and embedded blanks in string comparisons and in the calculation of a macro variable length. Use the LEFT( ) function to get rid of the leading blanks; use the TRIM( ) function to get rid of trailing blanks. Trimming is necessary for the display of character values without trailing blanks, but it is not required for string comparison and length calculation. LENGTH( ) function provides the length of a string, including the leading and embedded blanks. Note that the minimum length of a blank string is always 1. In addition, programmers should be aware that all macro variables have character values. However, blanks and the calculation of length for macro variables are handled differently than for data step variables. In macro-processing, the %LENGTH( ) function includes the leading and trailing blanks in length calculation; the minimum length is zero while the maximum length is 32,768.

- Embedded Blanks
  Sometimes strings contain embedded blanks. Are these blanks important to your character comparison? If not, use the COMPRESS( ) function to get rid of all blanks before comparison. Are multiple embedded blanks considered to be equivalent to a single blank? If so, the COMPBL( ) function can convert multiple consecutive blanks into a single blank.

- Uppercase vs. Lowercase
  By default, the string comparison procedure is case-sensitive. If cases do not make a difference in a comparison, apply the UPCASE( ) or the LOWCASE( ) function to the string before making the comparison. The UPCASE( ) function converts all lowercase characters into the corresponding uppercase characters. The LOWCASE( ) function, on the other hand, converts all uppercase characters into the corresponding lowercase characters.

B.2.3 Special Characters

The BYTE(n) function generates the n\textsuperscript{th} character in the ASCII or EBCDIC collating sequence. Be sure to know which collating sequence your host system uses. Even though one can see these
characters on the screen, some characters may have special meanings. For example, `BYTE(12)` in the EBCDIC collating sequence generates the female symbol sign (♀) on the screen and it instructs some printers to start a new page. It is always advisable to generate these special characters by means of the `BYTE()` function and then print these characters to uncover their special meanings.

### B.3 Beware of Missing Values

Missing values should be handled with special care. Below are some features of missing values of which a programmer should be aware.

#### B.3.1 Input of Missing Values

Be extremely cautious when handling data with missing values. Ideally, a period (.) or a special missing value character (_. A ... Z) is used to indicate a missing value. The `MISSOVER` option in the `INFILE` statement is useful to prevent SAS from reading data from the next record. Other useful tools include the `LRECL=` and `PAD` options in the `INFILE` statement. The `LRECL=` option specifies the physical line length of the file while the `PAD` option pads a record with blanks.

#### B.3.2 Order of Missing Values

There are 28 missing values in the SAS System. This includes the period (.) and special missing values (_. A B C ...). The order of these missing values in relation to other non-missing values in ascending order is as follows:

_. A B C ... 1 10 100 1000

Note that special missing values are not case-sensitive. In other words, A is equivalent to a.

#### B.3.3 SAS Programming with Missing Values

- **Identification of Missing Values**
  
  Because of the potential existence of any of the 28 missing values, a statement such as
  
  ```
  IF X = . THEN PUT 'X IS MISSING.';
  ```
  
  may not capture all missing values. Instead, the following statement should be used:
  
  ```
  IF X <= .Z THEN PUT 'X IS MISSING.';
  ```
  
- **False Path in a IF Statement**

  In most programming languages, the value of 1 in the IF condition is treated as true and 0 is treated as false. However, in SAS, any values other than 0 and the 28 missing values are treated as true. Example B.3.3 demonstrates this important concept:

  **Example B.3.3 Identification of Values for the TRUE and FALSE Path in a IF Statement**

  (SAS Program)
  ```sas
  DATA _NULL_;
  ARRAY X(9) _TEMPORARY_ (_ _ . A .Z -1 -1E-12 0 1E-12 1);
  DO I = 1 TO 9;
    IF X(I) THEN
      PUT X(I) BEST12. ' IS TRUE.';
    ELSE PUT X(I) BEST12. ' IS FALSE.';
  END;
  RUN;
  ```

  (SAS Log)
  ```
  _ IS FALSE.
  . IS FALSE.
  A IS FALSE.
  Z IS FALSE.
  -1 IS TRUE.
  -1E-12 IS TRUE.
  0 IS FALSE.
  1E-12 IS TRUE.
  1 IS TRUE.
  ```

  To guard against the fact that the zero value and the 28 missing values will all lead to the same false path, a better programming practice is to use the following statement:
  
  ```
  IF X=0 THEN PUT 'X IS ZERO.';
  ELSE PUT 'X IS NOT ZERO.';
  ```

  This way one can clearly distinguish a zero value from any other values.

- **Effect on Functions and Assignment Statements**

  The `SUM()` and `MEAN()` functions ignore any of the 28 missing values. The `NMISS()` function provides the number of missing values in its argument, including the period and the special missing values. In addition, any non-functional operations in assignment statements will result in a missing value. SAS displays a `NOTE` message to alert the programmer that a missing value has been generated as a result of performing an operation on a missing value.

- **Effect on Updating**

  The `UPDATE` statement in a data step enables users to update data from the original data set with data from another data set as long as the data values from the latter are not regular missing values. In other words, data cannot be updated with a regular missing value (.), but can be updated with a non-missing value or a special missing value (_. A ... Z).

### B.4 Duplication in Key Variable Values

In most data sets, key variables are used to uniquely identify an observation. These key variables can be easily indexed in SAS by means of the data step, PROC DATASETS or PROC SQL. It is always a good idea to identify the observations with duplication in the key variables.

Consider Example B.4. A composite index SALE_IDX is generated based on EMP_ID and QTR. The `NMISS()` option in the INDEX CREATE statement excludes observations with all missing index variable values from the index table and the `UNIQUE` option instructs SAS to create indices only if the indices are unique. For data clarification purposes, it is always a good idea to identify observations that have duplicates in key values. In Example B.4, observations with duplicate key values are output into the data set `DUPLICATE` for further examination. For more information on the causes, impact and detection of duplicate observations, please read Reference [1].

**Example B.4 Duplication in Key Variable Values**

```sas
* CREATE COMPOSITE INDEX SALE_IDX.*;
PROC DATASETS LIBRARY=WORK;
MODIFY SALES;
INDEX CREATE SALE_IDX=(EMP_ID QTR) /UNIQUE NMISS;
QUIT;
* OUTPUT OBSERVATIONS WITH DUPLICATION *;
* IN KEY VARIABLE VALUES. *
DATA DUPLICATE;
SET SALES;
BY EMP_ID QTR;
IF NOT (FIRST.QTR AND LAST.QTR);
RUN;
```

### B.5 Transposing Data Sets

Transposing data sets is one of the most common data processing tasks. Often, for reporting and data analysis purposes, programmers have to transform data from a multiple observations per subject data structure to a data structure where each subject has only one observation.

Consider Example B.5.1. To transform a patient-visit based data set like `VITAL` to a patient based data set like `VITALT1`, one can use the `TRANSPOSE` procedure or take advantage of the use of an
array in a data step. Each method has its advantages and disadvantages.

**B.5.1 PROC TRANSPOSE**

PROC TRANSPOSE is easy to use. There is no need to inform SAS of the number of observations for each subject. However, there are certain precautions the programmer take.

- **Watch Out for Missing Observations**
  It is very important to use the ID statement in PROC TRANSPOSE. Consider Example B.5.1. There is no observation for the systolic blood pressure of Patient 102, Visit 2 in the input data set VITAL. Without the ID statement in PROC TRANSPOSE, an erroneous observation has been generated in the output data set ERR_VS. In this example, the variable SBP2 in the second observation should contain a missing value for the systolic blood pressure measured at the second visit. As a result, for Patient 102, SBP2 should be missing. However, without the ID statement in PROC TRANSPOSE, the systolic blood pressure for Visit 3 is incorrectly assigned to SBP2 while a missing value is given to SBP3. This is obviously an error!

- **Unique Identification of an Observation**
  Make sure that there is only one variable specified in the ID statement. If more variables are needed to uniquely identify an observation, the rest of the variables should be listed in the BY statement. The programmer should check to ensure that the BY variables and the ID variable form a set of unique identifiers of the source data set. If this is not true, SAS displays ERROR and WARNING messages in the log and the BY group concerned will not be transposed. Similarly, any missing value in the ID variable will result in the disqualification of the corresponding observation from the transposing procedure.

- **Missing Variables Due to Inadequate Data**
  Often programmers are asked to write code while there are still incoming data or the data have yet to be cleaned. Suppose a clinical trial study has 4 patient visits and data are collected at the end of each visit. It is very likely that at time of programming, there are no data for a later visit. In Example B.5.1, PROC TRANSPOSE does not create the variable SBP4 in the output data set VITALT1 because there are no observations for the fourth visit in the source data set VITAL. However, in anticipation of this variable, a subsequent part of the program may refer to SBP4, which results in an error! There are three ways one can resolve this problem. First, use an array to transpose the data. Second, create a dummy patient with all necessary observations in the source data set. Then after applying PROC TRANSPOSE, delete the record for this dummy patient from the output data set. Third, transpose the source data set. Then, create a dummy data set that contains only one dummy observation with all necessary variables. Next, use SET statement to append the dummy data set to the output data set from transposing. This forces SAS to create all the necessary variables. Finally, delete the dummy observation.

  Of the three aforementioned methods, the first method is preferred because at least in the pharmaceutical industry, deleting a record based on a patient ID is strongly discouraged.

Example B.5.1 (continued)

```
/* CORRECT TRANPOSING OF DATA. */;
PROC TRANSPOSE DATA=VITAL PREFIX=SBP 
OUT=VITALT1(DROP=_NAME_); 
   BY PATNO; 
   ID VISIT; 
   VAR SBP; 
RUN;
```

(SAS Data Sets)

<table>
<thead>
<tr>
<th>SAS DATA SET VITAL</th>
<th>SAS DATA SET ERR_VS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATNO VISIT SBP</td>
<td>PATNO SBP1 SBP2 SBP3</td>
</tr>
<tr>
<td>101 1 126</td>
<td>101 126 130 132</td>
</tr>
<tr>
<td>101 2 130</td>
<td>102 140 125</td>
</tr>
<tr>
<td>101 3 132</td>
<td></td>
</tr>
<tr>
<td>102 1 140</td>
<td></td>
</tr>
<tr>
<td>102 3 125</td>
<td></td>
</tr>
</tbody>
</table>

- **Effect of OBS=0 option**
  The OBS=0 option is useful for debugging syntax errors. However, with this option on, SAS is unable to generate any variables generated from PROC TRANSPOSE. Therefore, any subsequent reference to these variables will result in ERROR messages in the SAS log. As you shall see in Example B.5.2, transposing data, by means of arrays, avoids such problems.

**B.5.2 Use of Arrays to Transpose Data**

An array can be used to transpose a data set. This method is slightly more complicated than PROC TRANSPOSE, but it is more robust.

- **Dimension of an Array**
  To use an array for transposing a data set with multiple observations per patient to a patient-based data set, programmers need to know the dimension of the array. If the dimension is too small for the input data, an array subscript out-of-range error will result. On the other hand, if the dimension is too large, some variables will have missing values.

  In Example B.5.2, the dimension of the array is derived from the data and the protocol. This value is then put into a macro variable MAXVISID by means of PROC SQL. The last visit in this example is 4. Hence, had the data been cleaned and completed, MAXVISID should have a value no greater than 4. In data step VITALT2, the protocol violation is first checked before further execution. Values for systolic blood pressure for each visit are retained and output to the VITALT2 data set once when all observations for a patient have been read.

- **Beware of Duplicate Observations**
  By means of the array method specified in Example B.5.2, whenever there are duplications in the values of the BY variables, only the last observation in that BY group will be considered. This result may not always be ideal. Hence, programmers should check to ensure uniqueness in the BY variable values.

Example B.5.1 Using PROC TRANSPOSE to Transpose Data

(SAS Program)

```
* INCORRECT TRANPOSING OF DATA. *;
PROC TRANSPOSE DATA=VITAL PREFIX=SBP 
OUT=ERR_VS(DROP=_NAME_); 
   BY PATNO; 
   VAR SBP; 
RUN;
```
### Example B.5.2 Using An Array to Transpose A Data Set

**(SAS Program)**

```sas
* FIND THE DIMENSION OF THE ARRAY TO BE USED* ;
* FOR TRANSPONING. PUT THIS VALUE IN THE *;
* MACRO VARIABLE MAXVISIT. THE DIMENSION *;
* SHOULD BE THE LARGER OF THESE 2 VARIABLES: *;
* THE LATEST VISIT NUMBER ACCORDING TO DATA *;
* AND THE LATEST VISIT NUMBER ACCORDING TO *;
* THE PROTOCOL.(I.E., 4 IN THIS EXAMPLE.) *;
PROC SQL NOPRINT;
SELECT LEFT(PUT(MAX(MAX(VISIT), 4), 3.)) INTO :
MAXVISIT FROM VITAL ;
QUIT ;
* FIRST, CHECK FOR PROTOCOL VIOLATION. *;
* IF NO PROTOCOL VIOLATIONS, TRANSPOSE THE *;
* DATA. *;
DATA VITALT2;
IF _N_= 1 THEN DO;
VMAX = INPUT( SYMGET(' MAXVISIT'), 3.);
IF VMAX > 4 THEN DO;
PUT '(PROTOCOL VIOLATION):  ' ;
PUT 'HIGHEST VISIT NUMBER IS ' VMAX'.';
STOP ;
END ;
END ;
ARRAY TMP_ SBP (& MAXVISIT) SBP1-SBP&MAXVISIT;
DROP VISIT SBP VMAX ;
DO UNTIL ( LAST.PATNO) ;
SET VITAL;
BY PATNO VISIT;
TMP_ SBP(VISIT)= SBP ;
END ;
RUN ;
```

**(SAS Data Sets)**

<table>
<thead>
<tr>
<th>DATA SET VITAL</th>
<th>DATA SET VITALT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATNO VISIT SBP</td>
<td>PATNO SBP1 SBP2 SBP3 SBP4</td>
</tr>
<tr>
<td>101 1 126</td>
<td>101 126 130 132</td>
</tr>
<tr>
<td>101 2 130</td>
<td>102 140 . 125</td>
</tr>
<tr>
<td>101 3 132</td>
<td></td>
</tr>
<tr>
<td>102 1 140</td>
<td></td>
</tr>
<tr>
<td>102 3 125</td>
<td></td>
</tr>
</tbody>
</table>

### B.6 Merging and Joining Data Sets

SAS data sets can be combined by means of linking the common key variables. In the data step, this procedure is called merging; in PROC SQL such process is referred as joining. Although both merging and joining are common data processing tasks, there are pitfalls that can trap even the experienced programmers!

#### B.6.1 Always Merge with a BY Statement

Merging without a BY statement instructs SAS to perform one-to-one merging — a merging process in which SAS combines the first observation from each input data set specified in the MERGE statement to form the first observation of the resulting data set. The same process applies to subsequent observations until the last record from the input data set with the most observations has been read.

One-to-one merging is a legitimate, but not recommended operation in SAS. A single missing observation or a duplicate observation is enough to cause a linking error! So always use a BY statement to perform merging!

#### B.6.2 Watch Out for Automatic Retain in Match-Merging

Match-merging is based on the values of the common variables from two or more input data sets. The names of these common variables will be listed in the BY statement.

In a SAS data step without MERGE and BY statements, values for all the variables are missing values at the beginning of each data step iteration. A RETAIN statement has to be used to specify variables whose values users would like to retain for the next iteration. However, with MERGE and BY statements in a data step, all input variables will only be initialized to missing at the beginning of each BY group. Within the same BY group, the values of all input variables will automatically be retained for the next iteration. The automatic implicit retain feature in match-merging is an important concept that deserves special attention. It can cause unintentional errors in data processing. What makes it even more frightening is that one will not get any special messages from SAS!

**Example B.6.2 Data Errors Due to Automatic Retain**

**(SAS Program)**

```sas
DATA SALES2;
MERGE SALES1 CURRENCY ;
BY EMP_ID;
IF COUNTRY='UK' THEN UNIT='POUNDS';
RUN ;
```

**(Input Data Sets)**

<table>
<thead>
<tr>
<th>SAS DATA SET SALES1</th>
<th>SAS DATA SET CURRENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID COUNTRY SALES</td>
<td>EMP_ID UNIT</td>
</tr>
<tr>
<td>101 US 150</td>
<td>101 DOLLARS</td>
</tr>
<tr>
<td>101 UK 100</td>
<td>102 DOLLARS</td>
</tr>
<tr>
<td>101 US 155</td>
<td></td>
</tr>
<tr>
<td>101 US 120</td>
<td></td>
</tr>
<tr>
<td>102 US 130</td>
<td></td>
</tr>
</tbody>
</table>

**Output Data Set**

<table>
<thead>
<tr>
<th>SAS DATA SET SALES2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID COUNTRY SALES</td>
</tr>
<tr>
<td>101 US 150</td>
</tr>
<tr>
<td>101 UK 100</td>
</tr>
<tr>
<td>101 US 155</td>
</tr>
<tr>
<td>101 US 120</td>
</tr>
<tr>
<td>102 US 130</td>
</tr>
</tbody>
</table>

Note: The third and fourth observations are wrong! The country is US while the units of the currency are pounds!

Consider Example B.6.2. In data set SALES2, the value for UNITS in the second observation (POUNDS) has been erroneously retained for the third and fourth observations because these observations belong to the same BY group value: EMP_ID = 101. The fifth observation is not affected because it belongs to another BY group! There are four ways to resolve this problem:

1. Make sure that there is a reassignment of the variable UNIT based on all possible values of COUNTRY. For example, use
   ```sas
   IF COUNTRY='US' then UNIT='DOLLARS';
   ELSE IF COUNTRY='UK' then UNIT='POUNDS';
   ELSE UNIT='???';
   ```
2. Use another variable name. For example, use
   ```sas
   IF COUNTRY='US' then UNIT2='POUNDS';
   ELSE UNIT2=UNIT;
   ```
   UNIT2 will have the correct values!
3. Use one data step solely for the merging purpose. Use another data step to do other data processing.
(4) For one-to-one and one-to-many merges, use PROC SQL. The following code will serve the purpose:

```sas
PROC SQL;
CREATE TABLE SALES3 AS
SELECT A.*,
CASE (COUNTRY)
  WHEN ('UK') THEN 'POUNDS'
  ELSE UNIT
END AS UNIT
FROM SALES1 A, CURRENCY B
WHERE A.EMP_ID=B.EMP_ID;
QUIT;
```

**Example B.6.4** Cartesian Product Effect and Overwriting of Common in PROC SQL

(SAS Program)

```sas
PROC SQL;
CREATE TABLE FULL1 AS
SELECT * FROM A FULL JOIN B
ON A.ID = B.ID ;
CREATE TABLE FULL2 AS
SELECT COALESCE(A.ID, B.ID) AS ID,
COALESCE(A.VAL, B.VAL) AS VAL
FROM A FULL JOIN B
ON A.ID = B.ID ;
QUIT;
```

(Comments)

- The first six rows in FULL1 and FULL2 are a result of the Cartesian product effect. Since ID = 1 appears twice in A and three times in B, the resulting tables FULL1 and FULL2 have $2^3 = 6$ rows for ID = 1.
- In FULL1, all values of VAL come from the left table A. This is because the SELECT * statement is used in the creation of FULL1. Note that the ninth row in FULL1 has missing values for the columns ID and VAL. This is because the table A does not have a row with ID = 4, the values for columns ID and VAL in this row of table A_C are obtained from the table B.
- **Cartesian Product Effect of Joins in PROC SQL**

There are two basic joins: (1) The INNER JOIN in which all rows in the resulting tables have to exist in all source tables; (2) The OUTER JOIN (i.e., left join, right join or full join) in which the output table must have all rows from the left table, the right table or either of the input tables, respectively.

Regardless of the type of joins, the Cartesian product effect will always take place. For instance, if the values for the identifier columns is duplicated twice in one table and thrice in another table, by means of SQL joins, the resulting table will have $2^3 = 6$ rows with the same set of column values for the identifiers. In some cases, not only extra rows have been generated, but some rows may also have the wrong values!
• Overwriting of Common Column Values in PROC SQL

Unless clearly specified in the SELECT statement, whenever there are common columns, PROC SQL always keeps the values of the common column from the left table. This differs from the MERGE operation in the data step, which always keeps the values of the variable it last read. Because of this feature in PROC SQL, when a SELECT * statement is used, a LEFT JOIN B may not always be equivalent to B RIGHT JOIN A. To avoid confusion, clearly specify in the SELECT statement the columns you would like to keep in the output table. Moreover, use the COALESCE() function to get the first non-missing column values from a list of specified columns.

C. Macro-Processing Techniques

The SAS macro-processing facility empowers users to extend and customize the SAS system. It enhances an application, making it easier to maintain by reducing the amount of code in a program. Below are some suggestions for macro developers.

C.1 Assignment of Macro Variables

There are three methods by which a macro variable can be assigned. Each method has its own characteristics.

(1) %LET Statement

This is probably the simplest of all macro assignment statements. For instance,

%LET GRADE = PASS;

assigns the value of PASS to the macro variable GRADE. Note that %LET statement does not keep the leading and trailing blanks. If one would like to preserve these blanks, use the %STR() function. For instance, the following assignment will keep a blank before and after the value PASS in the macro GRADE:

%LET GRADE = %STR( PASS );

Another important characteristic is that %LET statements are performed during the word scanning phase, not during compilation. Hence, do not use IF ... THEN DO; %LET ... ; END; statements in a data step; only the last %LET assignment statement will be in effect.

Example C.1.1   Use of IF ... THEN %LET... Statement in a Data Statement --- An Erroneous Approach

(SAS Program)
DATA _NULL_;  
SET EXAM END=EOF;  
TOTSCORE+SCORE;  
IF EOF THEN DO;  
IF TOTSCORE >= 500 THEN DO;  
CALL SYMPUT('GRADE', 'PASS');  
END;  
ELSE DO;  
CALL SYMPUT('GRADE', 'FAIL');  
END;  
RUN;  
(Comments)  
The code in this example corrects the error in Example C.1.1 and provides the correct grade based on the total score.

(2) CALL SYMPUT(ARGUMENT1, ARGUMENT2) Statement

Unlike the %LET statement, the CALL SYMPUT assignment method keeps the leading and trailing blanks of a character variable.

Also, unlike the %LET statement, the CALL SYMPUT statement will be read by the compiler and hence, can be used to correct the error in Example C.1.1. Please see the solution in Example C.1.2.

Indeed, the CALL SYMPUT statement provides a very powerful tool! The first and second arguments can be a variable name or a character string; hence, it can produce macro variables based on variable values in a data step. However, until SAS Version 7, the assignment of a value by the CALL SYMPUT statement will be limited to 200 characters. For a macro variable of more than 200 characters in length, one has to concatenate 2 shorter macro variables. One way to accomplish this is as follows:

%LET LONG = &SHORT1 &SHORT2;

Example C.1.2   Use of CALL SYMPUT in a Data Step

(SAS Program)
DATA _NULL_;  
SET EXAM END=EOF;  
TOTSCORE+SCORE;  
IF EOF THEN DO;  
IF TOTSCORE >= 500 THEN DO;  
CALL SYMPUT('GRADE', 'PASS');  
END;  
ELSE DO;  
CALL SYMPUT('GRADE', 'FAIL');  
END;  
RUN;

(Comments)

C.2 Retrieval of Macro Variables

There are two ways to retrieve a macro variable value: the simple macro referencing and the SYMGET() function. Simple macro referencing, such as &MACVAR, resolves the value of a macro variable at compile time while the SYMGET() function gets the value at run time. When creating a code to be stored, such as creating a VIEW, it is advisable to use the SYMGET( ) function. However, beware that in SAS Version 6.12, the SYMGET( ) function truncates any macro variable values that are longer than 200 characters.

Example C.2 Simple Macro Referencing versus SYMGET Function

(SAS Program)
PROC SQL;  
*** RESOLVED AT COMPILE TIME ***;  
CREATE VIEW EXAMRPT1 AS  
SELECT NAME, COURSE, SCORE  
FROM IN.EXAM  
WHERE NAME="&STD_NAME";
C.6  Preservation of Line Size and Page Size in a Macro

Example C.6 Preservation of Line Size and Page Size in a Macro

```sas
%MACRO SUM_RPT;
  * SAVE ORIGINAL SETTING *
  DATA _NULL_;
  SET SASHELP.VOPTION;
  WHERE OPTNAME IN ('LINESIZE', 'PAGESIZE');
  CALL SYMPUT(OPTNAME, SETTING);
  RUN;
  * RESTORE ORIGINAL SETTING *
  OPTIONS PS=&PAGESIZE LS=&LINESIZE;
%MEND SUM_RPT;
```

C.7 Macro Storage

There are three ways a SAS macro can be stored. When a macro definition is submitted in a SAS session, the SAS processor compiles the macro and stores it in a SAS catalogue in the work library. This catalogue is temporary and only exists during the session. The SAS auto library facility and stored compiled macro facility enable users to store SAS macros permanently. With the compiled macro facility, one can save compiled time in each session; however, users will not be able to see the compiled code!

Once a macro is invoked, the SAS processor first searches for the macro in the SAS catalogue WORK.SASMACR in the work library. It tries to locate the macro defined in the session. If that fails and if the MSTORED option is on, the processor will search the catalogue of stored compiled SAS macros in the library defined in the SASMSTORE= option. Finally, it will attempt to locate this macro in the autocall libraries provided that the MAUTOSOURCE option is on and the SASAUTOS= option has specified the directories to be searched. The macro processor will search for the macro according to the order the directories are listed in the SASAUTOS= option. When the macro is found, the search will stop.

It is possible that there is more than one macro with the same name, but the macro processor will search according to the aforementioned hierarchy. Hence, be very, very careful and make sure you are really getting the macro you want! The SYMBLOGEN, MPRINT and MLOGIC options can help to determine whether there is a problem.

Users do have some controls over the search hierarchy. For example, NOMSTORED option instructs the processor not to search for the stored compiled macros. Make sure you know the current options setting. The MSTORED option can be specified in OPTIONS statement, options windows, autoexec files, configuration files and at invocation. The OPTIONS statement, for instance, may overwrite some options specified in the configuration file! Moreover, it is not advisable to change the SASAUTOS= option during an ongoing SAS session. The newly specified SASAUTOS= option will only be in effect when the processor encounters an uncompiled macro. To know your current option settings, type OPTIONS in the command line of the display manager to obtain the options window or look into the view SASHELP.VOPTION.

D.  Naming Convention

Names for SAS data sets, variables, macro variables, macros and formats should be descriptive so that programmers can easily identify the meaning from a given name. Special care should be taken to prevent the value of an entity from being inadvertently overwritten. AGE and ADVERSE, for instance, may have been ideal names for age and adverse event data, respectively. However, these two names are so common that they may be used by other programmers and hence, it increases the chance of being overwritten. If you are an application developer or if you rely on the OPTIONS statement with values from the original settings. For line size and page size, the original settings can be found in SASHELP.VOPTION. Please see Example C.6 for a brief illustration.

Example C.6 Preservation of Line Size and Page Size in a Macro

```sas
%MACRO SUM_RPT;
  * SAVE ORIGINAL SETTING *
  DATA _NULL_;
  SET SASHELP.VOPTION;
  WHERE OPTNAME IN ('LINESIZE', 'PAGESIZE');
  CALL SYMPUT(OPTNAME, SETTING);
  RUN;
  * RESTORE ORIGINAL SETTING *
  OPTIONS PS=&PAGESIZE LS=&LINESIZE;
%MEND SUM_RPT;
```
code from other sources, you have to be very careful of your naming convention. With a lengthy program, it can be a time-consuming task to identify such overwriting problems!

One remedy to this situation is to maintain a list of forbidden names. This list has to be updated constantly. Programmers will have to make sure that their existing programs do not contain any forbidden names in the current list, as well as, in any future listing. This can be a time-consuming process as the number of programs increases. A simpler solution is to establish some naming conventions. For example, names beginning with an X or an underline are reserved for developers only.

**Conclusion**

The paper has provided tips and techniques to achieve robustness in SAS programming. Special attention has been given to documentation practices, data processing and macro-processing methods. There is no denying that some of the recommended programming strategies may be at the expense of efficiency and programmers’ time. Hence, it is critical that readers use their good judgment in order to strike a balance between robustness and efficiency.

**Reference and Suggested Readings**


**Acknowledgment**

The author is indebted to the Department of Statistics and Statistical Laboratories of Iowa State University where she first learned SAS programming techniques. She is grateful to her friends and colleagues at Pfizer Inc. for their support and encouragement. A million thanks to

Twaha Bukenya, Pfizer Inc.; Benjamin Jackson, Pfizer Inc.; Malini Krishnan, Pfizer Inc; Gary Krynicki, SAS Institute Inc.

for sharing their expertise and/or sparing their precious leisure time to review this article. You are the reasons why this article finally appears in NESUG 99!

**Trademark**

SAS® is a registered trademark of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration.

**Author Contact Information**

Alice Monica Cheng
Pfizer Inc., 235 East 42nd Street
New York, New York 10017-5755
Work Phone: (212) 733-1328
E-mail Address: chenga@pfizer.com