

Building Macro-based Data Step Functions Using Random Macro Aliases to Simulate Local Scope of Internal Function Variables

Paul M. Dorfman, Citibank Universal Card Services, Jacksonville, FL

ABSTRACT

In lieu of user-defined SAS® DATA step functions, SAS Macro language is a powerful means of developing functions and language statements to be used in the DATA step. However, care should be taken to properly localize variables used inside such constructs, so they would not interfere with other variables used in the DATA step where they may be invoked. In this paper, issues related to developing macro-based DATA step functions are discussed, and a technique of creating random macro aliases that practically eliminates interference between the variables is presented. The method is illustrated by a real-world example of building macro-based DATA step function where local scope of internal variables is randomly simulated by a stand-alone macro.

THE PROBLEM

Creating user-defined functions to modularize DATA step code is an important aspect of SAS application development. Functions provide for a better program structure, help standardize it, improve its reusability, ease maintenance, and eliminate repetitive code. Since DATA step user-defined functions are not available as an intrinsic part of base SAS software, using SAS Macro Language to create DATA step functions and statements has become a common practice.

One problem associated with such routines is related to the fact that unlike real user-defined functions, they lack an internal mechanism of making the scope of their internal variables local. Once the macro implementing a function has been compiled, these variables become an integral part of the caller's program data vector. If at least one of them happens to share its name with a variable belonging to the caller or other functions it has invoked, it can render the program of the calling DATA step corrupt.

To an even greater extent, these considerations pertain to the names of `_TEMPORARY_` arrays used as internal function constructs. In this case, a coincidence between an array name and the name of a non-array variable would simply prevent the calling DATA step from compilation because of an illegal array reference.

Using labels as GOTO and LINK destinations inside such functions is another potential time bomb. If, as it often happens, a code-assembling macro function is invoked in a DATA step more than once, the caller will abend with the warm greeting in the log: 'ERROR 305-185: Label has already been used.'

DO WE REALLY NEED INTERNAL VARIABLES?

In some cases of function development, the problem can be eliminated by getting rid of internal variables and labels altogether. Let us discuss the issue of variables first.

Instead of assigning intermediate results to variables, we can make use of combinations of nested SAS functions and boolean expressions. As an example, if the purpose of `%NWORDS(STR)` macro function is to return the number of words in a string separated by one or more blanks, the function may be written as

```
%MACRO NWORDS(STR);
  (COMPRESS(&STR) NE ' ') *
  (LENGTH(LEFT(COMPBL(&STR))) -
   LENGTH(COMPRESS(&STR)) + 1)
%MEND NWORDS;
```

In this program, the boolean expression `COMPRESS(&STR) NE ''` eliminates an IF-THEN-ELSE structure that otherwise we would need to return 0 if the STR is blank. Combining nested SAS functions `COMPRESS`, `LENGTH`, `LEFT`, `COMPBL` makes auxiliary variables unnecessary. `%NWORDS` could then be called from a DATA step as an ordinary SAS function:

```
NO_WORDS = %NWORDS(SOME_STR);
```

For simple macro functions like `%NWORDS`, this approach may work just fine. However, in more complex cases, the program of a routine overloaded with nested SAS functions and booleans is difficult to read, its purpose is not self-explanatory, and it is hard to maintain. So, even when this way of function design is feasible, it may still be desirable to use some internal variables for better clarity.

Another technique of eradicating internal variables is to store temporary values in the macro memory, rather than in the program data vector, during DATA step execution, so that later on in the DATA step, the values can be retrieved and reused. For instance, let us consider the function `%SWAP(VAR1,VAR2)` frequently used inside sorting routines to make two variables exchange their values:

```
%MACRO SWAP(VAR1,VAR2);
  TEMP = &VAR1;
  &VAR1 = &VAR2;
  &VAR2 = TEMP;
%MEND SWAP;
```

For the time being, let us limit our discussion to equal length character variables. To avoid contaminating the program data vector of a caller by the auxiliary variable `TEMP`, we could code instead

```
%MACRO SWAP(VAR1,VAR2);
  CALL SYMPUT('TEMP',&VAR1);
  &VAR1 = &VAR2;
  &VAR2 = SYMGET('TEMP');
%MEND SWAP;
```

where the macrovariable `TEMP`, rather than the variable `TEMP`, would be used as an intermediate storage location.

APPLICATIONS DEVELOPMENT

However, the advantage thus gained comes at a steep price. First, this approach presents serious performance problems, because the data exchange rate within the program data vector far exceeds that of between the program data vector and the macro memory. Monitoring the following DATA step,

```
DATA _NULL_;  
  RETAIN A 'A' B 'B';  
  DO I=1 TO 100000;  
    %SWAP(A,B);  
  END;  
RUN;
```

that was run on mainframe with the FULLSTIMER option reveals that the step uses 2.41 and 0.03 CPU seconds when TEMP is coded as a macrovariable and a variable, respectively. With numeric variables, the loss of performance would be even more dramatic, as extra CPU time would be required for data type conversions. Besides, since all macrovariable values are strings, additional code would be needed to preserve the precision of numeric values involved.

Speaking of labels as GOTO destinations, they can be eradicated together with GOTOs by choosing alternative language statements. It is hardly desirable, however, to be compelled to replace all LINK statements in the function with repetitive code.

As far as _TEMPORARY_ arrays are concerned, their use is vital in DATA step function development.

Fortunately, coding functions without any of these language elements is not really necessary. All we need is a little name lottery.

RANDOM ALIASES

In most situations, coding 'variableless' DATA step functions is not possible at all, so using internal function variables becomes imperative. Within a SAS application system, any DATA step or other function can potentially be a caller. Therefore, if the system is meant to be robust, every user-defined function it incorporates - or may incorporate in the future - must have a completely unique set of variables, not shared with any other function or caller.

To ensure the uniqueness of function variables, programmers often resort to concocting variable names so odd that the code of potential callers would be highly unlikely to duplicate them. However, this technique has a number of drawbacks. First, it still provides no guarantee that commonly named variables will not be encountered. In fact, even different coders have the propensity to use similar patterns trying to make variable names look wild. For instance, contrary to the intent of replacing 'standard' loop counter names like I or J with uncommon ones, it is quite common to surround them with series of underscores. Second, writing and debugging a function with meaningless variable names can be a very painful experience. For this reason, a part of a name, say, the first four characters, is often reserved to retain some meaning. The remaining three are then used to make the name as odd as possible which does not leave a lot of room to play with.

However, the idea of making identifier names weird has an enormous potential. If all 8 positions allowed for a variable name were used to assemble the name totally at random, what would be the chance of choosing a duplicate name? Since we have 27 characters for the first position and 37 for each of the remaining seven, there are 2,563,160,682,591 possible full 8-character names. Hence, the probability of creating a duplicate 8-character identifier purely by chance is about $3.9E-13$, and that of picking up a duplicate name shorter than 8 characters is strictly 0. In practical terms, it means that a randomly chosen 8-character valid SAS name will never match any other SAS name in the entire application system, and will never be matched in the future.

Implemented programmatically, this approach would eliminate the problem of duplicate names. However, if we choose random identifiers for a function prior to or in the process of actually coding it, the inconvenience of programming using meaningless names will persist. What if we reverse the process: first code, test and debug the function using perfectly meaningful names, and only then automatically replace them with random aliases during macro compilation? In this case, a set of 'normal' names could be enjoyed as the function is being developed, whereas the DATA step compiler would only see a set of corresponding random aliases. In other words, the following recipe could be offered:

1. Code the macro to be used as a DATA step function using meaningful names for internal function variables, _TEMPORARY_ arrays and labels. Debug the code in a test DATA step, and make sure that the function works as intended.
2. Throughout the code of the function, precede the names with ampersands. It will make it possible, during macro compilation time, to replace each name with the value of the macrovariable having the same name.
3. Pass the lists of the names of variables, _TEMPORARY_ arrays and labels to the appropriate keyword parameters of a stand-alone macro routine designed to populate respective macrovariables with 8-character random aliases. Invoke the routine as the very first program line of the function.

To accomplish the last step, we need a generic macro routine, which, according to the recipe above, would accept the name lists, create macrovariables with the same names, and assign each of them an 8-character random alias. It would be also desirable to write a kind of alias-to-name cross-reference to the log. One way to code such a macro, let us call it %RANVAR, is

```
%MACRO RANVAR(VARNAMES=,ARRNAMES=,LBLNAMES=);  
  %LET STRFIRST =  
    _ABCDEFGHIJKLMOPQRSTUVWXYZ;  
  %LET STRALL =  
    0123456789&STRFIRST;  
  %PUT;  
  %PUT X-REFERENCE:;  
  %PUT;  
  %MACRO ALIAS(PLIST,DROP);  
    %LET PARMNUM = 1;  
    %IF &DROP %THEN DROP;  
    %LET VNAME = %SCAN(&PLIST,&PARMNUM);  
    %DO %WHILE(&VNAME NE);  
      %LET FLN =
```

```

%SCAN(%SYSEVALF(%SYSFUNC(
  RANUNI(0))*26+1),1,%STR(.));
%LET FL = %SUBSTR(&STRFIRST,&FLN,1);
%LET ALL = ;
%DO N=1 %TO 7;
  %LET ALLEN =
  %SCAN(%SYSEVALF(%SYSFUNC(
    RANUNI(0))*36+1),1,%STR(.));
  %LET ALL = &ALL%SUBSTR(
    &STRALL,&ALLEN,1);
%END;
%GLOBAL &VNAME;
%LET &VNAME = &FL&ALL;
%IF &DROP %THEN &FL&ALL;
%PUT -----;
%PUT &FL&ALL | &VNAME;
%LET PARMNUM = %EVAL(&PARMNUM+1);
%LET VNAME = %SCAN(&PLIST,&PARMNUM);
%END;
%IF &DROP %THEN %STR(;;);
%PUT;
%MEND ALIAS;
%IF &VARNAMES NE %THEN %DO;
  %PUT ALIAS | VARIABLE;
  %ALIAS (&VARNAMES,1);
%END;
%IF &ARRNAMES NE %THEN %DO;
  %PUT ALIAS | ARRAY;
  %ALIAS (&ARRNAMES,0);
%END;
%IF &LBLNAMES NE %THEN %DO;
  %PUT ALIAS | LABEL;
  %ALIAS (&LBLNAMES,0);
%END;
%MEND RANVAR;

```

The idea of this code is simple. For every name supplied with the parameter list, the macro: a) creates a macrovariable with that exact name; b) assembles a string whose first character is chosen at random from all the letters and an underscore, whereas each of the remaining seven characters is chosen, also at random, from all letters, an underscore, and all digits; c) assigns the string to the macrovariable. If the list of names being processed pertains to variables, then compose a DROP statement accordingly; do not create any such statement for _TEMPORARY_ arrays and labels to avoid errors and log warnings.

A USAGE EXAMPLE

As an example of this approach, let us create a DATA step function %BISEARCH to perform a binary search of a single-subscripted array. The function should accept the name of the array and a key value to search for. It should return the index of the matching element into a variable whose name we should be able to supply, and a missing value if the key has no match. All the random aliases, as well as the index variable supplied, must be dropped. It would also be nice to be able to call the function in the style of a language statement.

According to the plan outlined above, we should first code the function using a set of meaningful names:

```

%MACRO BISEARCH(ARY=,KEY=,INDEX=)/STMT;
  DROP &INDEX;
  FOUND = .;
  &INDEX = .;
  BOT = LBOUND(&ARY);
  TOP = HBOUND(&ARY);
  DO WHILE (NOT FOUND);
    IF &ARY(BOT) > &KEY OR
    &ARY(TOP) < &KEY
    THEN LEAVE;
    MID = FLOOR((TOP + BOT)/2);

```

```

  IF &KEY = &ARY(MID) THEN DO;
    FOUND = 1;
    GOTO LABELONE;
  END;
  IF TOP - BOT = 0 THEN LEAVE;
  IF &KEY > &ARY(MID)
  THEN BOT = MID + 1;
  ELSE TOP = MID - 1;
END;
LABELONE:
  IF FOUND THEN &INDEX = MID;
%MEND BISEARCH;

```

The variable names TOP, MID, BOT, FOUND are commonly encountered in SAS code, especially in implementations of searching and sorting algorithms, so they are very likely to be replicated elsewhere in the system. Note that the reference to the label LABELONE is used in the code instead of LEAVE statement solely for the purpose of demonstration.

The next steps of the plan are to: a) replace the variables TOP, MID, BOT, FOUND with &TOP, &MID, &BOT, &FOUND; b) replace the label LABELONE with &LABELONE; c) pass the list the variable names and the label name to the respective keyword parameters; d) put the call to %RANVAR on the top of the program. This results in the following:

```

%MACRO BISEARCH(ARY=,KEY=,INDEX=)/STMT;
  %RNDVAR(VARNAMES=TOP MID BOT FOUND,
    LBLNAMES=LABELONE);
  DROP &INDEX;
  &FOUND = .;
  &INDEX = .;
  &BOT = LBOUND(&ARY);
  &TOP = HBOUND(&ARY);
  DO WHILE (NOT FOUND);
    IF &ARY(&BOT) > &KEY OR
    &ARY(&TOP) < &KEY
    THEN LEAVE;
    &MID = FLOOR((&TOP + &BOT)/2);
    IF &KEY = &ARY(&MID) THEN DO;
      &FOUND = 1;
      GOTO &LABELONE;
    END;
    IF &TOP - &BOT = 0 THEN LEAVE;
    IF &KEY > &ARY(&MID)
    THEN &BOT = &MID + 1;
    ELSE &TOP = &MID - 1;
  END;
  &LABELONE:
  IF &FOUND THEN &INDEX = &MID;
%MEND BISEARCH;

```

Note that no care needs to be taken to drop random aliases in %BISEARCH, because the appropriate DROP statement is placed into the calling DATA step code by the nested macro %RNDVAR once %BISEARCH has been successfully compiled.

A SAMPLE FUNCTION CALL

Now that the binary search function design is complete, it can be invoked in a DATA step. Suppose we have a numeric array EVEN(1000000) populated with even integers (presorted in ascending order, so that the binary search algorithm could be used). Let us choose X_FOUND for the INDEX=. If the function works correctly, searching for the values 22222 and 22221 must return 11111 and a missing value, respectively, into X_FOUND. Calling %BISEARCH statement-style results in the following log excerpt (for better clarity, the cross-reference has been moved after the notes):

```
67 DATA TEST;
```

APPLICATIONS DEVELOPMENT

```
68  ARRAY EVEN(1000000) _TEMPORARY_ ;
69  DROP INTEGER SRCHFOR;
70  DO INTEGER=1 TO DIM(EVEN);
71  EVEN(INTEGER) = INTEGER * 2;
72  END;
73  DO SRCHFOR = 22222, 22221;
74  BISEARCH ARY=EVEN KEY=SRCHFOR
    INDEX=X_FOUND;
75  PUT X_FOUND=Q;
76  END;
77  RUN;
```

X_FOUND=11111 X_FOUND=.

NOTE: The data set WORK.TEST has 1 observations and 0 variables.

NOTE: The DATA statement used 0.75 CPU seconds and 5112K.

X-REFERENCE:

ALIAS	VARIABLE
EV3_OK6R	BOT
LBZZ9_D6	MID
PD4A3BCL	TOP
OMI6QJR5	FOUND

ALIAS	LABEL
CT73EUTY	LABELONE

The cross-reference produced by %RANVAR shows what kind of random aliases it has concocted. The fact that the data set TEST contains 0 variables means that, indeed, no 'junk' variables have been kept, even though no explicit DROP statement was coded to drop them. Instead, the required DROP statement containing the aliases was generated by %RANVAR macro for every 'localized' meaningful name included in the parameter list.

MPRINT output (indented for clarity) demonstrates the actual code assembled by %BISEARCH and executed as a part of the compiled DATA step:

```
DROP EV3_OK6R LBZZ9_D6 PD4A3BCL OMI6QJR5 ;
DROP X_FOUND;
OMI6QJR5 = .;
X_FOUND = .;
EV3_OK6R = LBOUND(EVEN);
PD4A3BCL = HBOUND(EVEN);
DO WHILE (NOT OMI6QJR5);
  IF EVEN(EV3_OK6R) > SRCHFOR OR
     EVEN(PD4A3BCL) < SRCHFOR
    THEN LEAVE;
  LBZZ9_D6 = FLOOR((PD4A3BCL + EV3_OK6R)/2);
  IF SRCHFOR = EVEN(LBZZ9_D6) THEN DO;
    OMI6QJR5 = 1;
    GOTO CT73EUTY;
  END;
  IF PD4A3BCL - EV3_OK6R = 0 THEN LEAVE;
  IF SRCHFOR > EVEN(LBZZ9_D6)
    THEN EV3_OK6R = LBZZ9_D6 + 1;
  ELSE PD4A3BCL = LBZZ9_D6 - 1;
END;
CT73EUTY:
IF OMI6QJR5 THEN X_FOUND = LBZZ9_D6;
```

Note that the first DROP statement is the one generated by %RANVAR macro, and it does not reference the label alias. The same would happen to any _TEMPORARY_ array alias if the array were present in the program of the function. The macro %RANVAR containing the label name reference creates a fresh label alias every time %BISEARCH is invoked.

Thus, if the latter were called twice from the same DATA step, any chance of the duplicate label error cited above would be eliminated.

CONCLUSION

1. The method of random aliases presented above has been used to code DATA step user-written functions in a number of SAS application systems. Even though some of the systems contain thousands of variables, no code corruption due to duplicate variable or label names has been observed.
2. Because the names eventually used by a calling DATA step are totally 'crazy', function developers are free to use any names they like, and it is unnecessary to cross-check whether a certain name already exists in the system, or is being coded in the process of parallel development. The benefits include peace of mind, more rapid development, and higher reliability.
3. The advent of 'long' identifiers in the SAS version 7 does not make the technique obsolete: the long-term COBOL experience shows that even long names fail to eliminate the problem of duplicate names completely. On the contrary, the availability of long names will make the method of random aliases absolutely, positively bulletproof. With 8-character names and 10,000 variables in the system, the probability of picking up a duplicate name by pure chance is about one billionth. Should one would think it is not small enough, then increasing the length of a random alias to, say, 20 would make it 5.9E-26 in a system incorporating 1,000,000 variables. The only alteration required to accommodate the change will be replacing 7 by 20 in the macro %RANVAR.

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

ACKNOWLEDGEMENTS

The author would like to acknowledge the contributions of the following individuals in the preparation of this paper.

Doris Bogar
Victor Dorfman
Yuri Katsnelson
Vladimir Kirillov
Eugenia Kravchenko
Vera Voloshin
Matthew Wilson

AUTHOR CONTACT INFORMATION

Paul M. Dorfman
Citibank UCS
7777 Baymeadows Way West
4-4-A325
(904)954-2419
pdorfma@ucs.att.com