

Generating new variables from variable names and values in a DATA step

Bruce Gilson, Federal Reserve Board

ABSTRACT

Writing DATA step code to transform data when new variable names are constructed from variable values can be difficult, in part because using variable values on the left-hand side of assignment statements to create new variables isn't straightforward. It's sometimes easiest to write SAS code to a file in a DATA step and then %INCLUDE the SAS code to execute it, as shown in this paper.

The following techniques used in the code are described in detail:

1. Using the PROC SQL INTO clause to copy data set values to a macro variable or series of macro variables.
2. Writing intermediate results to a temporary file (including why this is beneficial).
3. Using the little-known SAS-provided TSLIT macro to avoid needing double quotation marks around literal text that contains quotes, greatly simplifying processing.
4. Processing all elements of an array in a DATA step function or CALL routine with the OF operator.

INTRODUCTION

Writing DATA step code to transform data when new variable names are constructed from variable values can be difficult, in part because using variable values on the left-hand side of assignment statements to create new variables isn't straightforward. It's sometimes easiest to write SAS code to a file in a DATA step and then %INCLUDE the SAS code to execute it, as shown in this paper.

The following techniques used in the code are described in detail:

1. Using the PROC SQL INTO clause to copy data set values to a macro variable or series of macro variables.
2. Writing intermediate results to a temporary file (including why this is beneficial).
3. Using the little-known SAS-provided TSLIT macro to avoid needing double quotation marks around literal text that contains quotes, greatly simplifying processing.
4. Processing all elements of an array in a DATA step function or CALL routine with the OF operator.

A SMALL EXAMPLE

Data set ONE contains the following values. TYPE is character, and the other variables are numeric.

customer	type	month	balance
1	cd	1	11
1	checking	1	101
1	savings	1	1001
1	cd	2	12
1	checking	2	102
1	cd	3	13
1	checking	3	103
2	cd	1	21
2	checking	1	201
2	savings	1	2001
2	cd	2	22
2	checking	2	202
2	savings	2	2002
2	cd	3	23

2	checking	3	203
2	savings	3	2003
3	checking	1	301
3	savings	1	3001
3	checking	2	302
3	savings	2	3002
3	checking	3	303
3	savings	3	3003

We want to do a "Many to one" transformation where the resulting data set, TWO, has one observation per customer, as follows.

customer	cd1	cd2	cd3	checking1	checking2	checking3	savings1	savings2	savings3
1	11	12	13	101	102	103	1001	.	.
2	21	22	23	201	202	203	2001	2002	2003
3	.	.	.	301	302	303	3001	3002	3003

Note the following about the new data set.

- Each observation contains all account balances for one customer, with the variable names for each account balance constructed from the data with the form of *typemonth*. For example, the BALANCE for CUSTOMER 2 when TYPE=checking and MONTH=3 is 203, so CHECKING3 in observation 2 of the new data set is 203.
- Not all customers have all account types in every month. For example, customer 1 only has a SAVINGS account in month 1. The new data set includes variables SAVINGS_2 and SAVINGS_3 because some customers have savings accounts in those months, but these two variables have missing values in the first observation for customer 1.

In real life, the data set would be much larger, and manually writing enough IF/THEN/ELSE clauses to cover all possible cases is unrealistic. An automated method for transforming the data is needed.

The following sections show how to code this application.

CODE PART 1: SORT THE DATA SET BY THE KEY VARIABLES

We sort data set ONE by CUSTOMER and MONTH, the variables whose values we want to consolidate in one observation.

```
proc sort data=one;
  by customer month;
run;
```

We're overwriting data set ONE but to preserve it if necessary, use the OUT= option to create a new, sorted data set, with code like the following:

```
proc sort data=one out=one_sorted;
  by customer month;
run;
```

CODE PART 2: READ VALUES FROM DATA SET ONE TO CREATE MACRO VARIABLES

Each pair of statements does the following.

- The PROC SQL SELECT statement with the INTO clause copies data set values to a macro variable or series of macro variables.
- After each SELECT statement, the automatic macro variable SQLOBS, which contains the number of rows selected, is copied to a macro variable before it's overwritten.

In this example, the macro variables we create are small, but for applications with large amounts of data, it's important to remember that the maximum size of a macro variable is 65,534 bytes (characters).

Details for each pair of statements are provided after the code.

```
proc sql noprint;
    /* Create macro variable ALL_TYPES containing comma-separated
       values of TYPE, with the TYPE values quoted */
    select distinct quote(strip(type))
    into :all_types separated by ','
    from one;
    /* Create macro variable with number of distinct TYPES */
    %let num_types=&sqllobs;

    /* Create macro variables ALL_CUSTOMER1, ALL_CUSTOMER2,..
       containing distinct CUSTOMER values */
    select distinct customer
    into :all_customer1-
    from one;
    /* Create macro variable with distinct number of CUSTOMERs */
    %let num_customers=&sqllobs;

    /* Create macro variables with first and last month values */
    select min(month), max(month)
    into :month_start trimmed, :month_end trimmed
    from one;
    /* Create macro variable with range of months */
    %let num_months=%eval(&month_end-&month_start+1);

    /* Create macro variable with space delimited list
       of new variable names */
    select distinct cats(type,month)
    into :all_vars separated by ' '
    from one;
    /* Create macro variable with number of new variable names */
    %let num_vars=&sqllobs;
quit; /* exit from PROC SQL */
```

FIRST SELECT CLAUSE

```
select distinct quote(strip(type))
into :all_types separated by ','
from one;
%let num_types=&sqllobs;
```

Macro variable ALL_TYPES contains comma-separated values of TYPE, with the TYPE values surrounded by quotes.

The STRIP function removes leading and trailing blanks from each value of TYPE.

Macro variable NUM_TYPES contains the number of distinct values of TYPE.

Macro variable values in this example are as follows:

- ALL_TYPES="cd","checking","savings"
- NUM_TYPES=3

SECOND SELECT CLAUSE

```
select distinct customer
```

```

        into :all_customer1-
        from one;
    %let num_customers=&sqlobs;

```

The odd-looking syntax ALL_CUSTOMER1- causes SAS to create as many macro variables as needed with ascending numeric suffixes (ALL_CUSTOMER1, ALL_CUSTOMER2, and ALL_CUSTOMER3 in our example).

Macro variable NUM_CUSTOMERS contains the number of distinct values of CUSTOMER.

Macro variable values in this example are as follows:

- ALL_CUSTOMER1=1
- ALL_CUSTOMER2=2
- ALL_CUSTOMER3=3
- NUM_CUSTOMERS=3

THIRD SELECT CLAUSE

```

        select min(month), max(month)
        into :month_start trimmed, :month_end trimmed
        from one;
    %let num_months=%eval(&month_end-&month_start+1);

```

Macro variables MONTH_START and MONTH_END contain the first (earliest) and last (latest) month. The TRIMMED argument removes leading and trailing blanks from the results.

Macro variable NUM_MONTHS contains the total range of months.

- The macro processor treats all values as text (character) by default, so integer arithmetic must be done with the %EVAL function.
- If the %EVAL function is omitted, the macro variables would resolve (be replaced by their values) but nothing else would happen.
 - Code: `%let num_months=&month_end-&month_start+1;`
 - NUM_MONTHS value: 3-1+1
- The %SYSEVALF function, not needed here, must be used for non-integer arithmetic.

Macro variable values in this example are as follows:

- MONTH_START=1
- MONTH_END=3
- NUM_MONTHS=3

FOURTH SELECT CLAUSE

```

        select distinct cats(type,month)
        into :all_vars separated by ' '
        from one;
    %let num_vars=&sqlobs;

```

Macro variable ALL_VARS contains the new variable names, space-separated.

- The CATS function concatenates character values, removing leading and trailing blanks. In this case, it returns all combinations of TYPE and MONTH. The number of values returned is the product of the number of TYPEs (3 in our example) and the number of MONTHs (3 in our example).

Macro variable NUM_VARS contains the number of new variable names.

Macro variable values in this example are as follows:

- ALL_VARS=cd1 cd2 cd3 checking1 checking2 checking3 savings1 savings2 savings3
- NUM_VARS=9

CODE PART 3: SPECIFY A TEMPORARY FILE TO CONTAIN THE GENERATED SAS STATEMENTS

```
filename myfile temp;
```

The generated SAS code is written to a temporary file, as specified by the TEMP keyword, which has the following advantages.

- The program is portable.
- Issues with file permissions or running out of disk space are avoided.
- The temporary file is automatically deleted at the end of the SAS session, avoiding clutter.

During application development, you might want to inspect the file containing the SAS code. If so, change the FILENAME statement to reference an actual file in an already existing folder or directory rather than TEMP, as in the following:

```
filename myfile 'c:\myfolder\sascode.txt';
```

CODE PART 4: WRITE THE SAS CODE TO THE TEMPORARY FILE

```
data _null_;
  file myfile; /* Specify location where SAS statements are written */
  put
    @1 "data two;" /
    @5 "set one;"/
    @5 "by customer;"/
    @5 "array out_vars [&num_types,&num_months] &all_vars;" /
    @1 "      " %tslit(array types[&num_types] $32 _temporary_ (&all_types));)/
    @5 "keep customer &all_vars;"/
    @5 "retain &all_vars;"/
    @5 "if first.customer then call missing(of out_vars[*]);"/
    @5 "_this_type=0;"/
    @5 "do _i=1 to dim(types) while (_this_type=0);"/
    @9 "if type=types[_i] then _this_type=_i;"/
    @5 "end;"/
    @5 "if _this_type then out_vars[_this_type,month]=balance;" /
    @5 "if last.customer then output;"/
    @1 "run;"/
  ;
run;
```

The PUT statement writes the DATA step that generates the new data set to the temporary file. Let's review some key sections of the code.

ARRAY CONTAINING THE NEW VARIABLES

```
"array out_vars [&num_types,&num_months] &all_vars;"
```

OUT_VARS is a 2-dimensional array. The number of rows equals the number of TYPEs and the number of columns equals the number of MONTHs.

In this example, the code resolves as follows:

```
"array out_vars [3,3] cd1 cd2 cd3 checking1 checking2 checking3 savings1
savings2 savings3";
```

In this example, the array has 3 rows and 3 columns, and variables are arranged in the array as follows:

cd1	cd2	cd3
checking1	checking2	checking3
savings1	savings2	savings3

THE TSLIT MACRO

```
"      " %tslit(array types[&num_types] $32 _temporary_ (&all_types);)
```

The TSLIT macro allows you to surround text with single quotes after macro variables are resolved (replaced with their values), which can greatly simplify processing of text containing quotes.

Starting in SAS 9.4, TSLIT is provided with SAS software as an autocall macro, so it's automatically available.

In this example, the code resolves as follows (note the single quotes, which are inserted after macro variables NUM_TYPES and ALL_TYPES are resolved):

```
"      " 'array types[3] $32 _temporary_ ("cd","checking","savings");'
```

The pair of double quotes at the start of the statement write four blank spaces to the temporary file to align this statement with the other statements.

To help understand why the TSLIT macro is useful, let's briefly consider a few other versions of the code.

- Surround the text with single quotes
 - 'array types[&num_types] \$32 _temporary_ (&all_types);'
 - The single quotes prevent macro variable resolution, and the following is written to the temporary file:
 - array types[&num_types] \$32 _temporary_ (&all_types);
- Surround the text with double quotes
 - "array types[&num_types] \$32 _temporary_ (&all_types);"
 - This resolves to the following:
 - "array types[3] \$32 _temporary_ ("cd","checking","savings")":
 - Errors occur because the double quotes pair up improperly. For example, first quoted expression is as follows:
 - "array types[3] \$32 _temporary_ ("

THE RETAIN STATEMENT

```
"retain &all_vars;"
```

We read all observations for a given CUSTOMER from the original data set, then write a single observation for that CUSTOMER. As we assign a value to one new variable per observation, the RETAIN statement ensures that the new variable values are retained across observations until we're ready to write out the observation.

In this example, the code resolves as follows:

```
retain cd1 cd2 cd3 checking1 checking2 checking3 savings1 savings2  
savings3;
```

PROCESS ALL ELEMENTS OF AN ARRAY WITH THE OF OPERATOR

```
"if first.customer then call missing(of out_vars[*]);"
```

The OF operator lets you process all elements of an array in a DATA step function or CALL routine. It has the following syntax: `of array-name[*]`

When we begin processing a new CUSTOMER, the CALL MISSING routine sets all the new variables (CD1-CD3, CHECKING1-CHECKING3, and SAVINGS1-SAVINGS3 in our example) to missing.

- Because the new variables are retained (see previous section), we must prevent values from the prior CUSTOMER from being used for the current CUSTOMER.
- To illustrate by example why this is important, CUSTOMER 3 has no CDs, so this prevents CUSTOMER 3 from inheriting values for CD1, CD2, and CD3 from CUSTOMER 2.

ASSIGN VALUES TO THE NEW VARIABLES

```
"_this_type=0;"/  
"do _i=1 to dim(types) while (_this_type=0);"  
"if type=types[_i] then _this_type=_i;"  
"end;"  
"if _this_type then out_vars[_this_type,month]=balance;"
```

We compare the current value of TYPE to the values in the TYPES array, which contains all account types ("cd", "checking", "savings" in our example), until one of the following occurs:

- TYPE in the current observation matches one of the account types, and the following occurs:
 - The WHILE clause causes the DO loop to stop.
 - The IF statement is true, so we assign BALANCE to the appropriate new variable.
- All TYPES array values ("cd", "checking", "savings" in our example) are checked but none match the current value of TYPE. The IF statement is false, so no assignment takes place. This shouldn't happen because we generated the TYPES array values from the data in the first PROC SQL SELECT INTO statement in Part 2 of the code, but this is an example of defensive programming.

To illustrate by example, consider observation 15 in data set ONE.

- CUSTOMER=2, TYPE=checking, MONTH=3, BALANCE=203.
- For the first loop iteration, _i=1, and TYPES[1] is "cd" which doesn't match TYPE.
- For the second loop iteration, _i=2, and TYPES[2] is "checking", which matches TYPE.
- When the third loop iteration starts, the WHILE clause is not true so loop processing ends.
- The IF statement is true, so OUT_VARS[2,3], which maps to the variable CHECKING3 (see the section "Array containing the new variables" above), is set to 203.

WRITE AN OBSERVATION TO THE NEW DATA SET

```
"if last.customer then output;"
```

After processing all observations for a CUSTOMER and copying balances to the appropriate new variable, write an observation to the new data set.

CODE PART 5: INCLUDE AND RUN THE CODE WE CREATED, SHOW RESULT

%INCLUDE brings the SAS code written to the temporary file above into the current program. The SOURCE2 argument causes the included code to be written to the SAS log. This is self-documenting but not required.

PROC PRINT without a data set name prints the most recently processed data set, which in this case is the data set created in the included code.

```
%include myfile/source2;
title "Data set with one observation per customer";
proc print;
run;
title;
```

CODE GENERATED IN THE TEMPORARY FILE

The following code is generated. The PUT statement that wrote to the temporary file used column pointers (for example, @5 for column 5) to indent the SAS statements in the temporary file for readability. Three records in the temporary file are wider than the page width for this paper and wrap below.

```
data two;
  set one;
  by customer;
  array out_vars [3,3] cd1 cd2 cd3 checking1 checking2 checking3 savings1
savings2 savings3;
  array types[3] $32 _temporary_ ("cd","checking","savings");
  keep customer cd1 cd2 cd3 checking1 checking2 checking3 savings1
savings2 savings3;
  retain cd1 cd2 cd3 checking1 checking2 checking3 savings1 savings2
savings3;
  if first.customer then call missing(of out_vars[*]);
  _this_type=0;
  do _i=1 to dim(types) while (_this_type=0);
    if type=types[_i] then _this_type=_i;
  end;
  if _this_type then out_vars[_this_type,month]=balance;
  if last.customer then output;
run;
```

CONCLUSION

This paper presented a simple data transformation technique that could be useful when using variable values to construct new variable names. It showed how to generate and write SAS code to a file in a DATA step and then %INCLUDE the SAS code to execute it.

I've seen similar examples where date variables (e.g., YEAR, MONTH, and DAY were used to construct the new variables. Other examples are easy to imagine.

REFERENCES

Carpenter, Arthur (2017), " Quotes within Quotes: When Single (') and Double (") Quotes are not Enough," Proceedings of the MidWest SAS Users Group 2017 Conference.
<<https://www.mwsug.org/proceedings/2017/RF/MWSUG-2017-RF02.pdf>>

SAS Institute Inc. (2017), "Base SAS® 9.4 Procedures Guide, Seventh Edition, Second Edition," Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2023. " The OF Operator with Functions and Variable Lists". In *SAS® 9.4 Language Reference: Concepts, Sixth Edition*. Cary, NC: SAS Institute Inc. Available at <https://documentation.sas.com/doc/en/lrcon/9.4/n1t04bwpt1jkadn1a5mk35we2yif.htm>

SAS Institute Inc. 2020. "%TSLIT Macro". In *Base SAS® 9.4 Utilities: Reference*. Cary, NC: SAS Institute Inc. Available at https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lebaseutilref/n1phgnraodvpln1bm941n44yq7q.htm

ACKNOWLEDGMENTS

Thanks to Mark Jordan at SAS Institute for reviewing early drafts of this paper, providing much better data and code, and encouraging me to dive into the details of tools like PROC SQL SELECT INTO, the TSLIT macro, and the OF operator. And thanks to Zoriana Kurzeja for reviewing the paper and providing suggestions. Their support is greatly appreciated.

CONTACT INFORMATION

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

Bruce Gilsen
bruce.gilsen@frb.gov

Any brand and product names are trademarks of their respective companies.

APPENDIX 1: DATA SET USED IN THIS PAPER

The data set used in this paper can be easily generated as follows.

```
data one;
  length type $8.;
  input customer type $ month balance;
  datalines;
1 cd 1 11
1 checking 1 101
1 savings 1 1001
1 cd 2 12
1 checking 2 102
1 cd 3 13
1 checking 3 103
2 cd 1 21
2 checking 1 201
2 savings 1 2001
2 cd 2 22
2 checking 2 202
2 savings 2 2002
2 cd 3 23
2 checking 3 203
2 savings 3 2003
3 checking 1 301
3 savings 1 3001
3 checking 2 302
3 savings 2 3002
3 checking 3 303
3 savings 3 3003
;
```

APPENDIX 2: THE ENTIRE PROGRAM USED IN THIS PAPER

```
/****** CODE PART 1: Sort the data set by the key variables *****/

proc sort data=one;
  by customer month;
run;
```

```

/***** CODE PART 2: Read values from data set ONE to create macro
        variables *****/

proc sql noprint;
    /* Create macro variable ALL_TYPES containing comma-separated
       values of TYPE, with the TYPE values quoted */
    select distinct quote(strip(type))
    into :all_types separated by ','
    from one;
    /* Create macro variable with number of distinct TYPES */
    %let num_types=&sqllobs;

    /* Create macro variables ALL_CUSTOMER1, ALL_CUSTOMER2,..
       containing distinct CUSTOMER values */
    select distinct customer
    into :all_customer1-
    from one;
    /* Create macro variable with distinct number of CUSTOMERS */
    %let num_customers=&sqllobs;

    /* Create macro variables with first and last month values */
    select min(month), max(month)
    into :month_start trimmed, :month_end trimmed
    from one;
    /* Create macro variable with range of months */
    %let num_months=%eval(&month_end-&month_start+1);

    /* Create macro variable with space delimited list
       of new variable names */
    select distinct cats(type,month)
    into :all_vars separated by ' '
    from one;
    /* Create macro variable with number of new variable names */
    %let num_vars=&sqllobs;
quit; /* exit from PROC SQL */

/***** CODE PART 3: Specify a temporary file to contain the generated SAS
        statements *****/

filename myfile temp;

/***** CODE PART 4: Write the SAS code to the temporary file *****/

data _null_;
    file myfile; /* Specify location where SAS statements are written */
    put
    @1 "data two;" /
    @5 "set one;"/
    @5 "by customer;"/
    @5 "array out_vars [&num_types,&num_months] &all_vars;" /
    @1 " " %tslit(array types[&num_types] $32 _temporary_ (&all_types));)/
    @5 "keep customer &all_vars;"/
    @5 "retain &all_vars;"/
    @5 "if first.customer then call missing(of out_vars[*]);"/
    @5 "_this_type=0;"/

```

```

@5 "do _i=1 to dim(types) while (_this_type=0);"/
@9 "if type=types[_i] then _this_type=_i;"/
@5 "end;"/
@5 "if _this_type then out_vars[_this_type,month]=balance;" /
@5 "if last.customer then output;"/
@1 "run;"/
;
run;

/***** CODE PART 5: Include and run the code we created, show result *****/

%include myfile/source2;
title "Data set with one observation per customer";
proc print;
run;
title;

```