A Sampler of Code Techniques

Ian Whitlock, Westat, Rockville, MD

Abstract

Several short coding examples will be presented. The examples include a problem in updating a master file, the use of the ODS OUTPUT statement to build a control data set for the selection of a set of random samples, the use of NAMED INPUT, and some macro examples. Principles of simplicity, maintainability, and reuse will be emphasized.

The examples are suitable to a wide range of programming experience. They require only base SAS and are operating system independent except for file definitions.

Named Input

Although it is documented, named input is one of the better kept secrets of the INPUT statement. The problem is that the examples do not reveal why one might want the tool. SAS goes into named input mode whenever it finds a variable name followed by an equal sign in an INPUT statement. In this mode the record is searched for each variable in the Program Data Vector (PDV), i.e., the memory area of the DATA step.

The online documentation for version 8 indicates that once you enter named INPUT mode, you should list any variables not yet picked up in the form variable=. Whether that is true or not depends on how the PDV is defined. For example,

```
data w ;
   retain x y ;
   input x= ;
cards ;
x=1 y=2
;
```

produces an error message because the type of Y has not been determined. While

```
data w ;
   length y 8 ;
   input x= ;
cards ;
x=1 y=2
;
```

produces just an uninitialized message. And

```
data w ;
   retain x y .;
   input x= ;
cards ;
x=1 y=2
;
```

produces no indications in the log that anything is questionable. Of course, when Y does not appear on a record, the currently retained value of Y is used.

If an informat is to be used then it should be specified in an INFORMAT statement or with the colon modifier to avoid the danger of running into the next variable.

Should one retain the named input variables? One advantage is that when the following observation has the same value for a variable, then that variable may be omitted on the corresponding input record. On the other hand one must be careful to use this feature wisely.

Another thing that you should know is that the "/" may be used to tell SAS to continue to the next record for the same observation. I generally add a dummy character variable END with the value "*" to make the end of each record stand out. For example,

```
data w ( drop = end );
   retain x y .;
   input x= ;
cards ;
x=1 /
y=2 /
end=*
;
```

produces just one observation.

These features make named input very handy for hand tailored test data. Suppose the real data has 50 variables, but each test case needs only a few of those variables. List, column, and formatted input would require wide records with little help as to what values go with what variables. With named input one can easily see what value goes with what variable and omit any variables that do not matter. The "/" allows the physical records to fit on a screen. Typically, in addition to the END variable I added a character variable NOTE indicating why the record is needed. This helps in looking at a print when verifying that each case came out as predicted.

UPDATE

Another thing that named input is useful for is constructing transactions for updating a file. Here it is important to know about the UPDATE statement and how it works. UPDATE takes the form

```
update master trans ;
by id ;
```

Here MASTER is the SAS data set to be updated and TRANS is the data set indicating the new values. The variable (could be multiple variables) ID is the key to a unique observation in MASTER so that it's value can indicate exactly which observation to change. On the other hand, there many be many observations with the same ID in TRANS. The changes indicated are applied one observation after another for the single corresponding MASTER observation. The standard missing value in TRANS indicates that the value in MASTER is not to be changed. Consequently only the ID and changed variables need be named. Note that it was a rather similar situation that made named input good for constructing test data.

Here is a simple example. Suppose that MASTER has the variable FIRSTNAME, LASTNAME, and MARITAL_STATUS. Now consider:

```
data stdtrans ;
   length id $5
          firstname $20
          lastname $ 30
          marital_status $ 1
   ;
   input (id firstname lastname
          marital_status) (=) ;
cards ;
id=00001 firstname=John
id=00002 lastname=Smith
id=00002 marital_status=M
;
data new_master ;
   update old_master stdtrans;
   by id ;
run ;
```

In the previous example, the transactions came from an external data set. Now suppose we have the transaction data in the form of a SAS data set, TRANS, containing the variables ID, VARNAME, and VALUE. How can one use it with the UPDATE statement? Here TRANS has to be converted into the standard form. This can be done in a two step process to write a file suitable for reading with named input and then a step to do the reading.

```
filename temp temp ;
data _null_ ;
   set trans ;
   file temp lrecl = 32767 ;
   put id= varname "=" value ;
run ;

data stdtrans ;
   length .... ;
   infile temp recl=32767 ;
   input id= ;
run ;
```

STDTRANS is now ready to use in an update step similar to the one previously shown.

The FILENAME statement uses the TEMP option to write to the work directory. SAS generates the filename and cleans up when SAS is exited.

In the above steps LRECL was set large enough to accommodate many variables. In practice there is little problem with giving a large LRECL and not using a lot of it.

MERGE

Let's take a moment to consider the MERGE statement. Two simple rules avoid a great many problems with merges.

> Never allow a variable in common between the data sets in a merge other than the BY variables.
>
> Never allow more than one of the data sets in a merge to have multiple observations for the BY group.

Assuming the corresponding BY-variables have common characteristics, the first rule guarantees that the data sets in the merge can appear in any order. Hence the code is more stable when changes are introduced, and you don't have to worry about one variable overwriting another.

Many SAS programmers argue with the first rule, but when pressed to give an example where they must violate it, they usually offer the update situation described above or an example of interleaving, which is better done with a SET statement. In other words, the violations come from failing to know which form of SAS input to use.

Violation of the second principle is so bad that SAS writes a note to the log. Why isn't it an error? SAS was designed to do a one-to-one merge within the BY-group, so it isn't a error, but it is rarely right to do so, and can appear to manufacture data when an unwary person sees the resulting output. Remember that in a merge data is retained in the BY-group. Hence a value on a short group will carry over to other records when the group in the other file is longer. Another problem is that the records are

simply matched one-to-one within the BY-group and there may be no basis for that particular match.

ODS OUTPUT

While working on a project, I was asked for each variable to draw a sample of observations where the sample size was to be 2% of the observations with positive values. Since the variables might change over time and the data certainly would, I wanted to start with a list of variables and determine the count of positive values for each variable rather than read frequency reports and modify the program based on the counts.

The first step in this process is to realize that the problem contains a clear cut sub-problem. Make a SAS data set containing the variable name, value, and count for the value. This is a useful service that extends beyond the boundaries of the original problem, hence I could write a the general macro independent of the project to supply the control data I needed for my project.

In the past I had written some macros of this sort, using the OUT= option of the TABLE statement, but I considered such a plan rather ugly. One has to worry about data type, getting the name of the variable and the associated format to obtain the formatted values. Could the ODS OUTPUT statement produce a simpler macro?

Unfortunately the data set output by ODS violates an important database principle - do not store data information in the names of the variables. ODS uses the variable name to hold the value, rather than a common name like VALUE. This means that the name of this column changes with each variable, so that one cannot simply concatenate the tables. In the same manner, the name of the column holding the formatted value is of the form F_var, where var is the name of the variable. Consequently, not much work is saved, in combining these tables, but this method does save the need to do the formatting and hence considering the type of each variable.

First let's consider an example with one variable INCOME_CAT. Here is the code.

```
ods listing close ;
ods output onewayfreqs(match_all) =
     __fq1(keep= table f_: frequency ) ;
   proc freq data = study ;
      table income_cat / missing ;
      format income_cat inc. ;
   run ;
ods listing ;
ods output close ;
```

It produces the data set __FQ1 with the variables TABLE holding the name of the variable, F_INCOME_CAT holding the formatted values (always character) of INCOME_CAT and FREQUENCY giving the count.

Now with more variables there will be more tables, __FQ1, __FQ2, etc. All of these data sets can all be read with one monster SET statement. Then the separate F_-variables can be combined into one with a SELECT block on an IN= variable.

One problem that occurs with multiple variables is that the size of the formatted values can change from table to table, hence it is necessary to calculate the longest such variable and assign that length to the new variable holding all formatted values of all the variables in the TABLE statement.

Here is the finished complete code in an extended version.

```
/* FreqOut.sas - create file
   (variable,value,frequency)
*/
%macro FreqOut
  (data=&syslast /* input dataset name*/
  ,out=freqout  /* output dataset name*/
  ,vars=         /* list of variables */
  ,by=         /* list of by variables*/
  ,fmtassign= /* var fmt var var fmt..*/
              /* . to override formats*/
  ,debugging=NO    /* YES or NO       */
  ) ;

  %let data = &data ; /* force eval */

  %local
    i        /* looping variable      */
    nv       /* number of variables   */
    ng       /* number of by groups   */
    svnotes  /* option value for notes*/
    svmprint /* option mprint value   */
  ;

  /* check for missing parameter */
  %if %length(&vars) = 0 %then
  %do ;
    %put ERROR(&sysmacroname): No
variables to process - halting ;
    %goto mexit ;
  %end ;

  /* save original option settings */
  %if %upcase(&debugging) ^= YES %then
  %do ;
    %let svnotes =
            %sysfunc(getoption(notes));
```

```
  %let svmprint =
          %sysfunc(getoption(mprint));
  options nonotes nomprint ;
%end ;

%if &fmtassign = . %then
    %let fmtassign = &vars ;

/* get number of variables in &vars */
%let vars = %sysfunc(compbl(&vars)) ;
%let nv = %length (&vars)
  -%length(%sysfunc(compress(&vars)))
  + 1 ;
%let nv = %eval ( &nv ) ;

/* get number of by-group values */
%if %length(&by) = 0 %then
  %let ng = 1 ;
%else
%do ;
  /* calculate number of
     by-group values
  */
  data _null_ ;
    if __eof then
      call symput("ng"
                  ,put(__ng, 5. -l)) ;
    set &data end = __eof ;
    by &by ;
    if first.%scan(&by,-1) then
      __ng + 1 ;
  run ;
%end ;

/* make the frequency tables */
ods listing close ;
ods output onewayfreqs(match_all) =
 __fq1(keep= &by table f_: frequency);
  proc freq data = &data ;
    %if %length (&by) > 0 %then
    %do ; by &by ; %end ;
    table &vars / missing ;
    %if %length (&fmtassign) > 0 %then
    %do ; format &fmtassign ; %end ;
  run ;
ods listing ;
ods output close ;

/* get the longest fomatted length */
proc sql noprint ;
  select max(length) into :maxlen
    from dictionary.columns
   where libname = "WORK"
     and substr(memname,1,4) = "__FQ"
     and substr(upcase(name),1,2)="F_"
    ;
 quit ;

/* --------------------------------
   make the final data set
   --------------------------------
*/
data &out ( keep = &by Variable Value
                   Frequency ) ;
  set
    %do i = 1 %to &ng*&nv ;
       __fq&i(rename=(table=Variable))
    %end ;
  ;

  length  Variable $ 32
          Value $ &maxlen ;

    select ;
      %do i = 1 %to &nv ;
         when (upcase(Variable) =
           "%upcase(%scan(&vars, &i))"
              ) Value =
              left(f_%scan(&vars,&i));
      %end ;
      otherwise error ;
    end ;
 run ;

%if %upcase(&debugging) ^= YES %then
%do ;
  proc sql ;
    %do i = 1 %to &ng*&nv ;
      drop table __fq&i ;
    %end ;
  quit ;
  options &svnotes &svmprint ;
%end ;

%mexit:
%mend  FreqOut ;
```

Let's look at some features to note. First the parameters are keyword parameters. This means that the code to call the macro will be a little easier to read with having a copy of the macro in front of you, since the parameter names have been chosen to reflect their meaning and will appear in the calling code. Note the comments that go with each parameter so that the macro header almost documents how the macro is to be used.

The DATA parameter has the default value &SYSLAST. This takes a little understanding. Remember that no macro instructions are executed between the %MACRO

statement and the %MEND statement. Hence &SYSLAST is not evaluated when the macro is compiled. Instead, the instruction to reference the automatic variable SYSLAST is stored. This means that latter in the program when DATA is referenced, the last created SAS data set will replace the reference. This is a good idea, since it preserves the SAS default way of working, but it would be unfortunate if someone happened to place a step at the top of the macro creating a new data set, which would then take the place of the one that the caller had in mind when using the default. Thus it is almost obligatory to force the evaluation of SYSLAST at the top of the macro with:

```
%let data = &data ; /* force eval */
```

This removes the reference to SYSLAST and fixes the value of DATA with the correct data set name.

The default value for FMTASSIGN is empty. This means that by default any previously assigned formats will be used in the PROC FREQ. However the consumer of the macro can override the formats with temporary ones if he desires to do so. This raises the question of how simply cancel the format assignments. This is done with the missing dot and the code:

```
%if &fmtassign = . %then
      %let fmtassign = &vars ;
```

Now the macro can generate the correct format assignment for PROC FORMAT.

Note that the VARS parameter is refused when left empty, in contrast to the handling of the FMTASSIGN parameter. Why? Because we need the explicit list of variables here. If one wanted more protection one could code checks for _ALL_, _NUMERIC_, _CHARACTER_ and the list symbols, "-" and ":". However, I chose not to do so.

NV, the number of variables in the VAR parameter is calculated using the COMPBL and COMPRESS DATA step function. How do we count words in a space separated list of words? Standardize to one space between each word, then count the spaces and add one because there is one more word than spaces. What about no words? Well we already eliminated that possibility in the parameter checking.

The DATA step functions are executed using the macro function %SYSFUNC. The COMPESS is used to remove blanks. Now we can bind the number of blanks by comparing the lengths of the list of words with one blank between each word and the list with no blanks. This forms a very fast and efficient method of counting the elements in a space separated list, because all of the looping normally required has been shoved down to the function level.

Note the declaration of all local variables. This makes the code easier to read and guarantees the consumer that none of his macro variables will ever be changed as a side effect of calling the macro. Again the comments for each variable help the reader to see how the local variable will be used in the macro.

The block of code:

```
%if %upcase(&debugging) ^= YES %then
  %do ;
    %let svnotes =
            %sysfunc(getoption(notes));
    %let svmprint =
            %sysfunc(getoption(mprint));
    options nonotes nomprint ;
  %end ;
```

raises an interesting question because it represents a common problem often faced by the macro programmer. The macro needs to store a small number of option settings that will be changed to meet the requirements of the macro, so that the original values can be restored. Now imagine that we had a macro OPTLIST (LIST) that would return a list of settings for restoring the original values. In this case, the code would reduce to the following.

```
%let optlist = notes mprint ;

...
%let svopts = %optlist ( &optlist ) ;
...
options &svopts ;
```

I leave it to you as a practice problem to work out the details of how you will loop through the list of options and concatenate the results for the final return value. The one hint you may need to know is that GETOPTIONS has a second argument. When the value is "KEYWORD" the function returns with a value suitable for direct use in an OPTIONS statement. For example, given LINESIZE, the function would return LINESIZE=*number,* and given MPRINT, it might return NOMPRINT.

The next block of code assigns NG the number of BY-group values. This is needed later when we combine all of the frequency data sets. If BY parameter is empty then the whole file is one BY-group. Otherwise we have to count them.

I assumed that if consumer assigns a value to the BY parameter, then he knows that the input data set is sorted in that order. Hence I can use a DATA step to increment by one at the beginning of each BY-group. A relatively new feature of the %SCAN function is to search backward by using negative numbers, hence

```
%scan(&by,-1)
```

finds the last variable listed in the BY parameter. The test for end-of-file is placed at the top of the step as cheap insurance against someone later adding some for of subsetting to this step and subsetting away the last record.

Note that this step follows the simple rule that any data variable created by the macro on a temporary basis will have a name beginning with a double underscore. This means that if the consuming code never creates data variables beginning with a double underscore, then there will never be any variable name clashes between the users' names and the macro author's names. (Macro variable name clashes have already been avoided by always declaring all variable local unless the macro has been given specific permission to use the variable.)

The ODS step has already been explained before the macro was presented.

As explained in the paragraph preceding the macro, we need to know the longest formatted value in the frequency tables in order to insure that none of these values will be truncated when we combine all of the frequency tables and reduce the set of formatted value variables to one variable called VALUE. To do this we use the SQL dictionary file, DICTIONARY.COLUMNS to look at the lengths and obtain the longest one. This value is then exported to macro variable, MAXLEN, using the INTO operator.

We now have all the pieces in place for the final DATA step to create the output data set.

Basically it is a very simple step consisting of three parts - the SET statement combining the frequency tables, a LENGTH statement to assign the length of VALUE, and a SELECT block to decide which variable to use in assigning VALUE the correct formatted value. The OUTPUT is by default at the bottom of the step.

How many data sets are to be combined? If there is no BY-processing then the one data set for each variable and that number is &NV. With BY-processing we have to multiply by &NG, the number of BY-groups. When there is no BY-processing we assigned NG the value one, so now we can just multiply. Why is no %EVAL needed? Because %EVAL is automatically called by the macro facility in all places where a number is expected.

So now we have a finished macro serving a useful function.

Conclusion

We have looked at some of the features of named input. When you understand how the named input works, it can be surprisingly handy for some special tasks. We took a quick look at the MERGE statement and some common rules that one rarely sees, but are responsible for avoiding a great many of the problems SAS programmers fall into when doing merges. Finally we took a detailed look at the construction of an interesting macro to see how one should think about developing a macro and what problems should be considered.

Contact Information

Ian Whitlock

Westat

1650 Research Boulevard

Rockville, MD 20850

IanWhitlock@westat.com