Coding Tips from Too Many Years of Programming

Jimmy DeFoor, Citibank North America, Irving, Texas

ABSTRACT

This paper will review the flow of the SAS Data step, the function of SAS Macros, and provide useful tips on using SAS Macros and particular Data step techniques to make your SAS programming easier and your SAS programs more reliable and more efficient.

INTRODUCTION

We will begin with the SAS Data step, borrowing documentation from the SAS website and modifying it only slightly.

Flow of the SAS Data step

The Compilation Phase

When you submit a Data step for execution, SAS checks the syntax of the SAS statements and compiles them; that is, the processor automatically translates the statements into machine code. In this phase, SAS identifies the type and length of each new variable. During the compile phase, SAS creates the following three items:

input buffer

is a logical area in memory into which SAS reads each record of raw data as SAS executes an Input statement. Note that this buffer is created only when the Data step reads raw data. (When the Data step reads a SAS data set, SAS reads the data directly into the program data vector.)

program data vector (PDV)

is a logical area in memory where SAS constructs and manipulates the data of the SAS Data step, one observation at a time. When a program executes, SAS reads data values from the input buffer, a SAS data set, or creates them through SAS language statements. The data values are then assigned to the appropriate variables in the program data vector. From here, SAS writes the values to a SAS data set or discards them (as when a Data _Null_ is used).

Along with data set variables and computed variables, the PDV contains two automatic variables, _N_ and _Error_. The _N_ variable counts the number of times the Data step begins to iterate (loop). The _Error_ variable signals the occurrence of an error caused by the data during execution. The value of _Error_ is either 0 (indicating no errors exist), or 1 (indicating that one or more errors have occurred). SAS does not write these variables to the output data set.

descriptor information

is information that SAS creates and maintains about each SAS data set, including data set attributes and variable attributes. It contains, for example, the name of the data set and its member type, the date and time that the data set was created, and the number, names and data types (character or numeric) of the variables.

This information is available from SAS functions, such as Vname, or from the SAS Dictionary tables. Knowing how to get this info can find the latest version of a SAS data set, the various variable names that contain a particular string (such as 'Date'), or the member names in a SAS library.

The Execution Phase

By default, a simple Data step iterates once for each observation that is being created. The flow of action in the Execution Phase of a simple Data step is described as follows:

The Data step begins with a Data statement. Each time the Data statement executes, a new iteration of the Data step begins, and the _N_ automatic variable is incremented by 1.

SAS sets the newly created program variables to missing in the program data vector unless a Retain statement exists for a variable. In that case, it retains the data value from the previous iteration (loop).

SAS reads a data record from a raw data file into the input buffer, or it reads an observation from a SAS data set directly into the program data vector. You can use an Input, Merge, Set, Modify, or Update statement to read a record. Input is only for external data sets.

SAS executes any subsequent programming statements for the current record.

At the end of the statements, an output, return, and reset occur automatically. SAS writes an observation to the SAS data set and then automatically returns to the top of the Data step. Next, the values of variables created by Input and assignment statements are reset to missing in the program data vector. Note that variables that you read with a Set, Merge, Modify, or Update statement are not reset to missing here.

SAS adds another iteration, reads the next record or observation, and executes the subsequent programming statements for the current observation.

The Data step terminates when SAS encounters the end-of-file in a SAS data set or a raw data file or encounters a Stop statement.

Good Ways to Use Information from the SAS Dictionary Tables.

Above we discussed the descriptor information about variables and SAS data sets captured during the compile phase. This information is available from the SAS Dictionary Tables.

One good source of information is the location of the SAS data sets and when they were created or modified. These create and modify dates are on the Tables table and the file location is on the Members table.

Suppose we have 12 file locations for a data set, one for each month, and we don't know which location has the most current information.

First, we assign librefs to every possible file location so that all SAS data sets in those locations will have their descriptor information available to the SAS Dictionary Tables. Descriptor information from SAS data sets is not loaded into the SAS Dictionary Tables until after the librefs are defined. SAS does not know they exist until then.

```
libname  dm01 '/usr01/dm/data/stage/adf_January_stage';
libname  dm02 '/usr01/dm/data/stage/adf_February_stage';
libname  dm03 '/usr01/dm/data/stage/adf_March_stage';
libname  dm04 '/usr01/dm/data/stage/adf_April_stage';
libname  dm05 '/usr01/dm/data/stage/adf_May_stage';
libname  dm06 '/usr01/dm/data/stage/adf_June_stage';
libname  dm07 '/usr01/dm/data/stage/adf_July_stage';
libname  dm08 '/usr01/dm/data/stage/adf_August_stage';
libname  dm09 '/usr01/dm/data/stage/adf_September_stage';
libname  dm10 '/usr01/dm/data/stage/adf_October_stage';
libname  dm11 '/usr01/dm/data/stage/adf_November_stage';
libname  dm12 '/usr01/dm/data/stage/adf_December_stage';
```

Next, we join the Tables and Members tables and pull the records associated with the HH_TXN_SUMM data set.

```
proc sql;
  create table dm_tables as
     select t.*,
            l.path from
           /* this table has the create and modify dates for each data set */
           dictionary.tables  t
             join
           /* this table has the path for each library */
           dictionary.members l
         on t.memname = l.memname and
            t.libname = l.libname
      where substr(t.libname,1,2) = 'DM' and t.memname = 'HH_TXN_SUMM';
  quit;
run;
```

Then, we will sort the data by create date (crdate) so that the newest data set is at the bottom.

```
proc sort data =  dm_tables;
  by crdate;
run;
```

Notice that the paths below are not capitalized, but that SAS elements are capitalized. Remember this when searching for character strings in a Dictionary variable. You may want to use the Upcase function to ensure that you do not miss un-capitalized elements.

libname	memname	memtype	crdate	modate	nobs	path
DM01	HH_TXN_SUMM	Data	2/18/2015	2/18/2015	284592	/usr01/dm/data/stage/adf_January_stage
DM02	HH_TXN_SUMM	Data	3/19/2015	3/19/2015	283181	/usr01/dm/data/stage/adf_February_stage
DM03	HH_TXN_SUMM	Data	4/21/2015	4/21/2015	287018	/usr01/dm/data/stage/adf_March_stage
DM04	HH_TXN_SUMM	Data	5/19/2015	5/19/2015	279763	/usr01/dm/data/stage/adf_April_stage
DM05	HH_TXN_SUMM	Data	6/18/2015	6/18/2015	277418	/usr01/dm/data/stage/adf_May_stage
DM06	HH_TXN_SUMM	Data	7/17/2015	7/17/2015	279399	/usr01/dm/data/stage/adf_June_stage
DM07	HH_TXN_SUMM	Data	8/17/2015	8/17/2015	277573	/usr01/dm/data/stage/adf_July_stage
DM08	HH_TXN_SUMM	Data	9/16/2015	9/16/2015	276359	/usr01/dm/data/stage/adf_August_stage

Finally, using a Call Symput, we write out the path of the latest dataset to a Macro variable so that we can use it in a libref that points the right location for our program. We use the end-of-file (end=) option on the Set Statement to identify the last line in the data set. Call Symput will be explained later.

```
data _null_;
  set dm_tables end=eof;
  if eof then
    /* the contents of the SAS variable path is loaded into the Macro variable path */
    call symput('path',path);
run;
```

Notice the double quotes in the libref. Macro variables will not resolve inside of single quotes.

```
libname dm "&path=;
```

Now let's look at another way to find the latest data set, but this time all of the data sets are in the same library. We will use the index function to find the data sets because only their root name is the same.

```
proc sql;
  create table scorepop_files as
     select t.*,
            l.path from
           dictionary.tables  t
             join
           dictionary.members l
         on t.memname = l.memname and
            t.libname = l.libname
      where t.libname = 'PPSDL' and index(t.memname,'SCOREPOP') ne 0;
  quit;
run;
```

Again, we will sort by create date (crdate) to get the latest data set.

```
proc sort data =  scorepop_files;
  by crdate;
run;
```

libname	memname	crdate	modate	nobs	nvar	filesize
PPSDL	A202_R02_SCOREPOP_20150101	2/25/2015	2/25/2015	46578	151	47382528
PPSDL	A202_R02_SCOREPOP_20150201	3/2/2015	3/2/2015	44758	151	45613056
PPSDL	A202_R02_SCOREPOP_20150301	4/1/2015	4/1/2015	42804	151	43384832
PPSDL	A202_R02_SCOREPOP_20150401	5/1/2015	5/1/2015	44257	151	44892160
PPSDL	A202_R02_SCOREPOP_20150501	6/1/2015	6/1/2015	44041	157	45154304
PPSDL	A202_R02_SCOREPOP_20150601	7/7/2015	7/7/2015	55616	188	52953088
PPSDL	A202_R02_SCOREPOP_20150701	8/9/2015	8/9/2015	54976	249	55246848
PPSDL	A202_R02_SCOREPOP_20150801	9/1/2015	9/1/2015	54730	254	55050240
PPSDL	A202_R02_SCOREPOP_20150901	10/2/2015	10/2/2015	54380	256	55181312

Again, we will use Eof and Call Symput to export the latest memname as a Macro variable.

```
data _null_;
  set scorepop_files end=eof;
  if eof then
    call symput('memname',memname);
run;
```

Then we will use the memname Macro variable to retrieve the latest Scorepop data set.

```
Proc sort data = ppsdl.&memname
     (keep=account score fullname)
         out  = scorepop;
  by obl_no;
run;
```

Another good use for the SAS Dictionary tables is the finding of all of the SAS data sets that have a particular name or a particular character string in their name. Notice that I use the Upcase function within my Index search to ensure that I do not miss un-capitalized variable names in my search.

```
proc sql;
  create table data_sets_with_needed_variable as
     select *
           dictionary.columns
      where t.libname = 'PESDL' and index(upcase(t.name),'OPEN_DATE') ne 0;
  quit;
run;
```

You can find the Views of all Dictionary tables in the SASHELP directory created at SAS startup. They will have names like VTable, VColumn, and VLibnam. Use those views to help you understand the info you can get from dictionary tables, once you have defined the SAS Data Sets to your current session using a libref. Get the info you want from a Dictionary Table, however, because the retrieval speed is faster.

Using the Special Variable _Infile_ to Access Data in the Input Buffer.

This example shows that the special SAS variable _Infile_ contains all the characters read on that line in the input data and loaded into the Input Buffer. It also shows that the SAS default field-delimiter is a space. That is, unless SAS is told otherwise, such as the delimiter is a comma, SAS will use a space as the delimiter between fields.

This example uses the Data _Null_ approach that reads, evaluates, and modifies data, but does not create a SAS data set. A simple Input statement without any variable names or qualifiers is used to load the record into the Input Buffer.

In addition, a Cards statement is used so that the data can be read into the data set from within the program instead of coming from an external file.

```
Data _null_;
   Input;

   put _infile_; /* writes the contents of the Input Buffer to the log */

   first_field  = scan(_infile_,1); /* separates the Buffer into single words */
   secnd_field  = scan(_infile_,2);
   third_field  = scan(_infile_,3);
   forth_field  = scan(_infile_,4);
   fifth_field  = scan(_infile_,5);

   /* writes the contents of the SAS fields to the SAS log */
   put first_field= secnd_field= third_field= forth_field= fifth_field=;

   Cards; /* Tells SAS to read data within the program and that data follows.*/

       10001        Red Truck           Dallas
       10012        Green Bicycle       Fort Worth
       10033        Blue Doll House     Houston
       10057        Orange Scarecrow    Austin
       10072        Yellow Songbird     San Antonio
;
Run;
```

This is the first line written by the Put _Infile_. It shows the contents of the Input Buffer.

```
    10001       Red Truck           Dallas
```

This is contents of the fields created by the Scan functions. Notice Red and Truck are in separate fields because SAS defaults to separating phrases to words when it encounters a blank.

```
first_field=10001 secnd_field=Red third_field=Truck forth_field=Dallas fifth_field=
```

This is the second line read into the Input Buffer.

```
    10012       Green Bicycle       Fort Worth
```

Again, SAS separates the phrases into different fields.

```
first_field=10012 secnd_field=Green third_field=Bicycle forth_field=Fort fifth_field=Worth
```

We can fix this problem during the read of the input data by setting a length to be read for each field. This is different than setting a length for the field to be stored in the SAS data set. The Input statement with an Informat or the Informat statement sets the length to be read. The Length statement or the Format statement sets the length of the field to be stored in the SAS data set. When only an Informat length is specified, it also establishes the Format length.

```
Data work1;

  Length prod_id 8 prod_desc $25   city $20;
  Input  prod_id   prod_desc $25.0 city $20.0; /* $25.0 and $20.0 are Informats */

  put _infile_;
  put prod_id= prod_desc= city= ;

  Output;
  Cards;
       10001        Red Truck           Dallas
       10012        Green Bicycle       Fort Worth
       10033        Blue Doll House     Houston
       10057        Orange Scarecrow    Austin
       10072        Yellow Songbird     San Antonio
;
Run;
```

Again, this first line read into the Input Buffer.

```
    10001       Red Truck           Dallas
```

But now the contents of the fields are correct.

```
    prod_id=10001   prod_desc=Red Truck    city=Dallas
```

Had we used an Input Statement without setting a length to the reads for prod_desc and for city, the contents of the prod_desc and city on every line would have been incorrect. If this is unclear to you, execute the above program with an Input Statement of:

```
    Input  prod_id   prod_desc $   city $;
```

Prod desc will then be 'Red' and City will be 'Truck'. By the way, the default informat type for SAS is numeric. That is why the $ must be used to indicate character.

Other uses of the Input Buffer

You can scan or index the input buffer (_infile_) for any characters or words before actually populating a variable. For example, one could scan the third word of each line and chose which variable should receive the contents.

```
If      scan(_infile_,3) = 'Truck' then
 Row = '10';
Else if scan(_infile_,3) = 'Doll' then
 Row = '11';
```

This example could be used with in-stream data read with Cards, but it would be more likely to be used with an external file read with an Infile statement.

Another way of using the input buffer would be to find a particular character and then read the contents after that into a variable. To do this, add the Length= option to the Infile statement. That will load the length of the content of the current input buffer into the variable specified in the Length= option when the buffer is loaded with the Input statement.

Here, Index is used with Substr to find and then create the field of the desired content. The delimiters of the field are imagined to be pound (#) signs.

```
data work;
  length field1 $20;
  infile 'C:\Users\Jimmy\Documents\Testfile.txt' length=linelen lrecl=256;
  input;    /* Executing input statement will assign LINELEN   */
  firstpos  = index(_infile_,'#');
  if firstpos ne 0 then
    do;
      secondpos = index(substr(_infile_,firstpos+1,linelen-firstpos),'#');
      if secondpos eq 0 then
        field1 = ' ';
      else
        field1 = substr(_infile_,firstpos+1,secondpos-1);
    end;
  else
    field1 = ' ';
run;
```

Another excellent use of the Length= option is with variable-length records or with records that have an end-of-line character that is causing problems for the input statement. You can then use this approach, which holds the current line and allows you to follow with additional input statements. Pad is used as Infile option to cause field4 to be 'padded' to 180 characters for each read of the infile.

```
data work;
  infile 'C:\Users\Jimmy\Documents\Testfile.txt' length=linelen lrecl=256 pad;
  input @;    /* Executing input statement will assign LINELEN   */
  remainlen = linelen ¬ 75 ¬ 1; /* stop before bad end-of-line marker */
  Input field1 $25.0 field2 $20.0 field3 $30.0 field4 $varying180.0 remainlen;
run;
```

And, just so you know that I can learn something new, SAS offers this method of starting an input statement when a particular string is found, which is 'phone' in this case. This method reads the content immediately after the string 'phone' is found for the length defined in the informat. The Scanover option allows the scan for 'phone'. The Truncover option enables you to read variable-length records without error when some records are shorter than the Input statement expects.

```
data _null_;
   infile phonebk truncover scanover;
   input @'phone:' phone $32.;
run;
```

SAS Macros

SAS Macros begin with a %Macro and end with %Mend and have a name.

They have a name so that they can be retrieved from the Macro catalog. In this case, the name is Example.

```
%Macro Example;
   If today() eq '01Jan1999'd then
     Stop;
%Mend Example;
```

SAS Macros are compiled when encountered in SAS program. They are executed when called, which happens when the name of the macro is encountered following a percent sign (%).

%Example

This means that SAS Macros are compiled or executed regardless of their location within or without a SAS Data step or Procedure. Thus, the actions performed or the code generated by a SAS Macro is not known by the SAS Data step or Procedure until the Macro is executed. So, this structure would be confusing to the reader, but it would work just fine.

```
Data work2;
   Set work1;

   %Macro Example;
   If today() eq '01Jan1999'd then
     Stop;
   %Mend Example;

   %example;
   output;
Run;
```

It would resolve to this before the Data step was compiled.

```
Data work2;
   Set work1;
   If today() eq '01Jan1999'd then
    stop;
   output;
Run;
```

Of course, a much more understandable structure would be

```
%Macro Example;
   If today() eq '01Jan1999'd then
     Stop;
%Mend Example;
```

```
Data work2;
   Set work1;
   %Example;
   output;
Run;
```

Now, let's discuss using SAS macros and SAS macro variables to make your SAS coding easier and execute your SAS programs more efficiently. SAS Macros can be especially good for generating SAS code from SAS data values.

Creating Macro Variables That Contain Data Values

This example shows how to transform data into the contents of SAS macro variables that can be evaluated or written within a Macro %Do Loop. The SAS code generated can be If Statements, Keep Statements, Select Statements, etc.

The first step is to create a character count of each line so that it can be used to create a unique macro variable of each data value. Then concatenate the count variable to the same string so that each data value is a unique macro variable, but starts with the same string as the other macro variables of that characteristic, such as prod_id1, prod_id2, prod_id3, etc. .

Use the End= variable to write out a macro variable of the total line count at end-of-file so that it can be used to control the use of all macro variables in a %Do Loop.

```
Data _null_;
  Set work1 end=eof; /* End-of-file variable will be EOF */
  cnt + 1;
  charcnt  = strip(put(cnt,2.0));  /* Create character forms of the numeric values */
  charprod = strip(put(prod_id,6.0));

  /* Call Symput writes out a Macro Variable from inside a Data step.
     The first argument is the macro variable to be created. The
     Second argument is the data value that will be stored in the
     macro variable. */

  call symput('prod_id'||charcnt,charprod);
  call symput('prod_desc'||charcnt,strip(prod_desc));
  call symput('city'||charcnt,strip(city));

  /* Write out the number of macro variables created at end-of-file */
  if eof then
    call symput('cnt',strip(charcnt));
run;
```

The %Put statement displays the content of macro variables to the SAS log and can be used in open code.

```
%put cnt=&cnt
     prod_id1   =&prod_id1     prod_desc1   =&prod_desc1
     prod_id&cnt=&&prod_id&cnt prod_desc&cnt=&&prod_desc&cnt;

cnt=5 prod_id1=10001 prod_desc1=Red Truck prod_id5=10072  prod_desc5=Yellow Songbird
```

```
Remember that our data is

       10001        Red Truck           Dallas
       10012        Green Bicycle       Fort Worth
       10033        Blue Doll House     Houston
       10057        Orange Scarecrow    Austin
       10072        Yellow Songbird     San Antonio
```

Count is clearly five because five records of variables were read from work1. A double ampersand (&&) resolves to a single ampersand (&) in the first pass of macro resolution. Thus, &&prod_id&cnt first becomes &prod_id5, which then becomes 10072 in the second pass. This double pass by the Macro processor is called double resolution or indirect referencing.

Creating a Macro that Generates If-Then-Else Statements

Again, Macros are code that are compiled and executed before SAS code is compiled and executed. The keyword %macro starts a macro compilation. The keyword %mend closes a macro compilation.

First, the Ifstat macro is compiled and stored in the Macro catalog.

```
%macro ifstat;
  /* Execute the %Do loop according to the number of lines of data */
  %Do j = 1 %to &cnt;
    %if &j ge 2 %then
       %do;
          Else
       %end;
          If prod_id = &&prod_id&j then
            prod_desc = "&&prod_desc&j";
    %if &j = &cnt %then
       %do;
          Else prod_desc = 'Unknown';
       %end;
  %end;
%mend ifstat;
```

Second, it is placed in the Data step in the location of where the IF statements will be executed.

SAS starts compiling the Data step when it sees the word Data. When it encounters the macro call, it executes the macro - which places the SAS code in the location of the macro. SAS then compiles the generated text as if a macro never existed.

```
data _null_;
  set work1;
  %ifstat ;
  output;
run;
```

The SAS Log reveals the code as compiled and executed by the Data step. Options Mprint guarantees that the SAS code generated by a macro will be displayed in the SAS log.

```
options mprint;
```

When the SAS code generated by a macro is displayed in the SAS log, Mprint and the name of the macro is displayed.

```
183  data _null_;
184    set work1;
185    %ifstat ;
MPRINT(IFSTAT):   If prod_id = 10001 then prod_desc = "Red Truck";
MPRINT(IFSTAT):   Else If prod_id = 10012 then prod_desc = "Green Bicycle";
MPRINT(IFSTAT):   Else If prod_id = 10033 then prod_desc = "Blue Doll House";
MPRINT(IFSTAT):   Else If prod_id = 10057 then prod_desc = "Orange Scarecrow";
MPRINT(IFSTAT):   Else If prod_id = 10072 then prod_desc = "Yellow Songbird";
MPRINT(IFSTAT):   Else prod_desc = 'Unknown';
186    output;
187  run;

NOTE: There were 5 observations read from the data set WORK.WORK1.
```

Creating a Macro that Creates a User Format

This macro will be named ‘format’ and will create a user format named Prddesc. User formats are very efficient ways of assigning values to variables and can be stored in a format library, if desired, so that they can be used in many programs.

This macro will use a %Do loop to generate an assignment statement for each prod_id and prod_desc. It will also assign a description of ‘Unknown’ when a prod_id is encountered that does not have a product description.

```
%macro format;
  /* Execute the %Do loop according to the number of lines of data. */
  %Do j = 1 %to &cnt;
    %if &j = 1 %then
       %do;
          /* semi-colons within a Macro loop are ignored by the Macro compiler */
          Proc Format;
            Value Prddesc
       %end;
            &&prod_id&j  =  "&&prod_desc&j"
    %if &j = &cnt %then
       %do;
              other = 'Unknown'
           ;
          Run;
       %end;
  %end;
%mend format;
```

The format macro will be passed to the Macro processor when %format is seen by the SAS compiler. The generated SAS code will then be passed to the SAS compiler.

```
%format ;
```

Again, the log shows the code seen and compiled by SAS, but this time by the Procedure compiler instead of the Data step compiler.

```
MPRINT(FORMAT):   Proc Format;
MPRINT(FORMAT):   Value Prddesc
                      10001 = "Red Truck"
                      10012 = "Green Bicycle"
                      10033 = "Blue Doll House"
                      10057 = "Orange Scarecrow"
                      10072 = "Yellow Songbird"
                      other = 'Unknown' ;
MPRINT(FORMAT):   Run;
NOTE: Format PRDDESC has been output.
```

User formats are executed with a Put function that relates the description to the product id. Here is an example.

```
data work2;
  set work1;
  prod_desc = put(prod_id, prddesc.0);
  output;
run;
```

All of the assignment work performed by the user format is done in the background using a binary search that is quite efficient for assignments that have eight or more characteristics that could be assigned. Otherwise, an If-Then-Else assignment will do just as well.

Generating SAS comments in the SAS log from a Macro

There may be times when displaying comments in a SAS log from a macro may be helpful for understanding the actions taken by a macro, such as when a user format was created much earlier in the program with a SAS macro than its current use in a Data step.

This is done by placing using the same macro structure that created the user format, but putting a %Put statement within the %Do Loop. The %Put statement will display each ‘assignment’ and its ending semicolon (;). A %STR statement is used to differentiate the semicolon for the comment from the semicolon for the %Put statement. The %STR statement hides the semicolon of the comment from the Macro processor. Otherwise, the macro compiler would think that the semi-colon was ending the %Put statement.

```
%macro comment;
   /* a macro to display toy product descriptions. */
   %put;
   %put * This comment shows the descriptions assigned to each product id %str(;) ;
   %do j = 1 %to &cnt;
      %put * "&&prod_id&j" = "&&prod_desc&j" %str(;) ;
   %end;
   %put * Other = "Unknown" %str(;) ;
%mend comment;
```

```
%comment ;
```

The SAS log shows the comments that are generated.

```
* This comment shows the descriptions assigned to each product id;
* "10001 " = "Red Truck" ;
* "10012 " = "Green Bicycle" ;
* "10033 " = "Blue Doll House" ;
* "10057 " = "Orange Scarecrow" ;
* "10072 " = "Yellow Songbird" ;
*  Other   = "Unknown" ;
```

Using a Macro to Generate a Variable List for Keep statements or Where Clauses or If Statements.

First, compile the macro that will use the prod_id macro variables created earlier by the Call Symput statements in a Data step.

```
%macro vargrp;
   %do j = 1 %to &cnt;
      %let varlist = %trim(&varlist &&prod_id&j);
   %end;
   ;
%mend vargrp;
```

Second, initialize the macro variable for the varlist to blank.

```
%let varlist = ;
```

Third, execute the macro to populate the varlist.

```
%vargrp;
```

Fourth, verify that the varlist was generated using a %Put statement.

```
%put varlist=&varlist;

varlist=10001 10012 10033 10057 10072
```

Fifth, use the varlist where appropriate.

a) In a Set statement where clause that limits records read into a Data step.

```
Data work3;
  set work1(where=(prod_id in (&varlist)));
  output;
run;
```

b) In a Proc SQL where statement that limits records read from a table.

```
Proc sql;
   create table work3 as
     select * from work1
       where prod_id in (&varlist);
  quit;
```

c) In a Data step to limit the records output to a data set.

```
Data work3;
  set work1;
  if prod_id in (&varlist) then
    output;
run;
```

d) In a Proc SQL to limit the records output to a data set.

```
Proc sql;
  create table work3 as
    select * from work1
      having prod_id in (&varlist);
  quit;
```

Note, a Having statement is executed when the results of the query is written to the SAS data set just as does the If statement in the Data step above.

The _Numeric_ and _Character_ variable lists created by the Data step

Two ways of managing SAS variables in a Data step are the _Numeric_ and _Character_ lists.

The _Numeric_ list includes all numeric variables that are defined in the Data step.
The _Character_ list includes all character variables that are defined in the Data step.

These two lists allow the creation of arrays that are only numeric or only character.

```
array numvars{*}  _Numeric_;
array charvars{*} _Character_;
```

This allows the manipulation of all character variables or all numeric variables in a Data step using an array that employs the Dim function, which returns the count of elements in an array.
A simple use would be to set all missing numeric values to zero.

```
Do j = 1 to dim(numvars);
  If numvars(j) eq . then
    Numvars(j) eq 0;
End;
```

But a very powerful use is to find the character variables that contain a particular string.

```
Data foundvars;
  Keep foundvar rowcnt;
  Set work1;
  Rowcnt + 1;
  Array charvars(*) _character_;
  Do j = 1 to dim(charvars);
    If index(upcase(charvars(j)),'DALLAS') ne 0 then
      do;
        Foundvar = vname(charvars(j));
        Output;
      end;
  End;
Run;
```

CONCLUSION

My goal has been to provide coding guidelines and suggestions that would enable you to code more efficiently and effectively. I hope I have done so.

TRADEMARKS

REFERENCES

SAS Institute Inc., *SAS(R) 9.2 Language Reference: Concepts, Second Edition*

Overview of Data step Processing
http://support.sas.com/documentation/cdl/en/lrcon/62955/HTML/default/viewer.htm#a000985872.htm

SAS Institute Inc., *SAS(R) 9.2 Language Reference: Dictionary, Fourth Edition*

Infile Statement,
http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a000146932.htm

ARRAY Statement
http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a000201956.htm

SAS Institute Inc., *SAS(R) 9.2 Macro Language: Reference*

Call Symput
http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/viewer.htm#a000210266.htm

How SAS Processes Statements with Macro Activity
http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/viewer.htm#a001071824.htm

Referencing Macro Variables Indirectly
http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/viewer.htm#a001071915.htm

%Do, Iterative Statement
http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/viewer.htm#a000543755.htm

CONTACT INFORMATION

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

Name: Jimmy DeFoor
Enterprise: Citibank National Bank
E-mail: thefoor@sbcglobal.net