Paper 242

Betcha Didn't Know …

Richard D. Langston, SAS Institute Inc., Cary, NC

ABSTRACT

This poster attempts to describe several DATA step features that users may not be aware exist. Some of these features are first being documented with Version 7 of the SAS® System, although they are present in Version 6.

DYNAMIC OPENS OF SAS DATA SETS

One of the features that I've always wanted to see in the DATA step was a way to dynamically specify SAS data sets. You've been able to dynamically specify input and output files via the FILEVAR= option, but in the past there was no way to specify SAS data sets except by explicit reference. However, starting in Release 6.11 of the SAS System (and 6.09E on the mainframes), the functions that used to only exist in SCL were made available to the DATA step. These functions, OPEN, SET, and FETCH, allow you to dynamically open and read observations from arbitrary SAS data sets. As an example, consider three SAS data sets, each with the variable X. The data sets DSET1, DSET2, and DSET3, have the same prefix but a different numeric suffix. We can read each of these by constructing their names, using the OPEN function to open a data set, SET to establish an association with variables in the DATA step, and FETCH to read from the data set. Here is some sample code to do so:

```
data dset1; x=1; run;
data dset2; x=2; run;
data dset3; x=3; run;

data temp;
     length x 8;
     do i=1 to 3;
        dsname='dset'||put(i,1.);
        dsid = open(dsname,'I');
        call set(dsid);
        obsnum=0;
        do while(fetch(dsid) = 0);
           obsnum+1;
           put 'reading obs ' obsnum
               ' from ' dsname;
           output;
           end;
        rc = close(dsid);
        end;
     drop i dsname dsid rc;
     run;
data _null_; set temp; put _all_; run;
```

The output on the log is as follows:

```
NOTE: Variable X is uninitialized.
reading obs 1  from dset1
reading obs 1  from dset2
reading obs 1  from dset3
```

The note about X being uninitialized is generated because at compile time, the DATA step cannot tell that the variable X will indeed be set.

The OPEN function is called with a character expression indicating the data set name, and 'I' to indicate an open for input. You are returned a "handle" as a numeric value that must be used when calling the SET and FETCH functions. The SET function is called to make this the current data set, and the FETCH function is called to actually read the observation. Note that the DATA step will not know at compile time what variables will come from these data sets. You must provide this information so that the program data vector can be filled with the variables. In the example above, the LENGTH statement indicates that the variable X is being provided for the program data vector.

This feature is especially useful for multiple tape data sets that reside on different volumes. If you want to defer the mounting of tape volumes, you can use these functions to accomplish this. For example, here is an MVS jobstream:

```
//  EXEC SAS609
//TAPE1 DD UNIT=TAPE,
//  VOL=SER=<volser>,DSN=<dsn>,
//  DISP=(OLD,PASS)
//TAPE2  DD  UNIT=AFF=TAPE1,
//  VOL=SER=<volser>,DSN=<dsn>,
//  DISP=(OLD,PASS)
//TAPE3  DD  UNIT=AFF=TAPE2,
//  VOL=SER=<volser>,DSN=<dsn>,
//  DISP=(OLD,PASS)
//SYSIN  DD  *

data temp;
     length x 8;
     do i=1 to 3;
        dsname='tape'||put(I,1.)||
               '.daily';
        dsid = open(dsname,'I');
        call set(dsid);
        do while(fetch(dsid) = 0);
           output;
           end;
        rc = close(dsid);
        end;
     stop;
     drop i dsname dsid rc;
     run;
```

In this example, we defer the opening of a data set until the open function is called. Because we control the opening and closing of each data set, we can allow each of the tapes to be individually mounted on the same tape drive. If instead we'd used the following SET statement:

```
data temp;
  set tape1.daily tape2.daily tape3.daily;
```

we would have needed three different tape drives, since all three data sets must be opened simultaneously for the SET statement to gather all the attribute information.

WHERE= OPTION FOR OUTPUT DATA SET

You may already be aware of the WHERE= option for input data sets. This allows you to subset the observations you want to process for input. However, starting in Release 6.11 of the SAS System, you can also use the WHERE= option for output data sets. This option is quite useful if you would otherwise produce large numbers of observations in a SAS data set.

To demonstrate how to use the WHERE= option, consider this example that also shows how to use the PUT function in a WHERE clause in conjunction with a CNTLOUT= data set for PROC FORMAT.

In this example, we have a format library in WORK.FORMATS containing the formats X1X, X2X, and X3X. We are interested in creating a CNTLOUT= data set with only the observations for only the X1X and X3X formats. We create a new format called $ACCEPT that indicates that those with certain names will be mapped to Y while all others are mapped to N. If we use the PUT function with these format, using a character expression corresponding to the format name, only the ones we want will return the character value of Y. We run PROC FORMAT with the CNTLOUT= option, and the resultant output data set will contain observations for X1X, X2X, X3X, and $ACCEPT. But by using the WHERE= option on the data set specification, and by using the PUT function, only the observations we want will be written out. In a real-world case, this would be very useful if the format library was large and only a few of the formats were wanted. (Of course, the SELECT statement of PROC FORMAT could also be used here, but this is a simple example for showing WHERE= and PUT functions in WHERE clauses).

```
proc format;
     value x1x 1='yes' 2='no';
     value x2x 1='oui' 2='non';
     value x3x 1='si' 2='no';
     run;

proc format;
     value $accept 'X1X','X3X'='Y' other='N';
     run;

proc format
      cntlout=new
        (where=(put(fmtname,$accept.)='Y'));
     run;

data _null_; set new;
     put _all_;
     run;
```

The log output for the last DATA step is as follows:

```
FMTNAME=X1X START=1 END=1 LABEL=yes MIN=1 MAX=40
DEFAULT=3 LENGTH=3 FUZZ=1E-12 PREFIX=  MULT=0
FILL=  NOEDIT=0 TYPE=N SEXCL=N
EEXCL=N HLO=  _ERROR_=0 _N_=1

FMTNAME=X1X START=2 END=2 LABEL=no MIN=1 MAX=40
DEFAULT=3 LENGTH=3 FUZZ=1E-12 PREFIX=  MULT=0
FILL=  NOEDIT=0 TYPE=N SEXCL=N EEXCL=N
HLO=  _ERROR_=0 _N_=2

FMTNAME=X3X START=1 END=1 LABEL=si MIN=1 MAX=40
DEFAULT=2 LENGTH=2 FUZZ=1E-12 PREFIX=  MULT=0
FILL=  NOEDIT=0 TYPE=N SEXCL=N EEXCL=N
HLO=  _ERROR_=0 _N_=3

FMTNAME=X3X START=2 END=2 LABEL=no MIN=1 MAX=40
DEFAULT=2 LENGTH=2 FUZZ=1E-12 PREFIX=  MULT=0
FILL=  NOEDIT=0 TYPE=N SEXCL=N EEXCL=N HLO=
_ERROR_=0 _N_=4
```

We can see that only the two observations each for X1X and X3X are placed into the output data set.

OBTAINING OPTIONS VIA THE GETOPTIONS FUNCTION

If you want to obtain the value of an arbitrary option, you could use PROC PRINTTO and redirect the log output and post-process it. But that's a real pain. But starting in Release 6.11 (and Release 6.09E for the mainframes), you can use the GETOPTION function in the DATA step to get the value of an arbitrary option. In this example, we show how to use the function.

On some hosts, the TEMP keyword can be specified in the FILENAME statement to indicate a temporary file. For example:

```
FILENAME XXX TEMP;
```

would associate the fileref XXX with a temporary file.

But this feature is not available on the Windows and OS/2® platforms (prior to Version 7) for some reason. In order to code around it, one needs to learn the value of the WORK= option. That option specifies where the work data sets are stored, and that same directory can be used for storing temporary files. Here is the SAS code to do this. In addition, we use the &SYSSCP macro variable to determine the operating system name so that the extra work will be needed only for PC platforms.

```
data _null_;
     if substr("&sysscp.",1,3) in
        ('WIN' , 'OS2') then do;
        value =  "'" ||
          trim( getoption ('WORK'))||
                "\TEMPFILE.DAT'" ;
        end;
     else do;
        value = 'TEMP';
        end;
     call symput('workfile',trim(value));
     run;

filename tempfile &workfile.;
```

The GETOPTION function is called with the expression 'WORK' in order to determine the directory name. We append \TEMPFILE.DAT so that we have a fully-qualified file name, in quotes, in the variable called VALUE. If we are not running on a PC platform, we set VALUE to TEMP (note that quotes are not used in that case). We can then unconditionally issue the FILENAME statement with the fileref TEMPFILE and with the next word being either TEMP (for those platforms supporting the TEMP keyword) or the fully-qualified file name in quotes.

PROGRESSIVE SOLUTION USING VARIOUS FEATURES

In this example, we solve an interesting word problem using three different methods, with each method improving on the former.

Problem: Since the inauguration of George Washington as the first president of the United States, at which times have there been 4 or more presidents and ex-presidents alive at the same time?

Assume we have a SAS data set called INFO that contains the SAS data variable INAUG, indicating the date of inauguration, and LASTDAY, indicating the date of death (or today's date, for those still alive as of today's date).

The first method I chose for solving this is to create a SAS data set containing one observation for each date between INAUG and LASTDAY for each president. We also have a character variable PRESNUM which has the value P*nn* where *nn* is the president number (George Washington=1 etc.), and the variable Y, which is always 1. Once the data set is sorted by DATE, we can use PROC TRANSPOSE to create a SAS data set containing the variables P01-Pnn (for each of the nn presidents - 41 as of Bill Clinton), and with one observation for each date since Washington's inauguration. The values of any P*nn* variable will be 1 if the president were alive on that day, and otherwise missing (that's why the variable Y is always 1). We can then create the data set BYDAY containing only those dates with 4 or more P*nn* variables with non-missing values, determined via the NMISS function.

```
data prezinfo(keep=presnum date y);
     set info(keep=inaug lastday) end=eof;
     length presnum $13 y 3;
     presnum='P'||trim(left(put(_n_,best12.)));
     y = 1;
     do date=inaug to lastday;
        output;
        end;
     if eof;
     call symput('lastpres',
                 trim(left(put(_n_,best12.))));
     run;

proc sort data=prezinfo; by date;
proc transpose data=prezinfo out=byday;
     id presnum; by date; var y;
     run;
data byday; set byday;
     living=&lastpres.-nmiss(of p1-p&lastpres.);
     if living>=4;
     run;
```

This approach was interesting, but its big disadvantage is that there is an observation per "entity day" (i.e., one observation for each president for each day the president was alive), which means over 226,000 observations when run in 1999. If this example were instead based on a database with thousands of entries instead of only 41, this would make for a very large data set and might be an impractical approach.

Here's a revised way. We can use a temporary array. I have found that I often forget that temporary arrays are available, and they can be a very nice feature for accumulation as long as you know the dimensions of the array. In this method, we know the starting date is INAUG for the first observation and the ending date is today's date. We can create a macro variable &START to indicate the first day, and &RANGE to indicate the total number of days to process. We can then create a temporary array called ALLDATES to hold a counter for each day. For every president day, the counter is incremented. When we come to end-of-file, we write out one observation for each date, setting a variable STATUS to 1 if there were 4 or more presidents alive on that day, and to 0 otherwise. We then run PROC MEANS to get the first and last day of each segment. We are only interested in those segments with STATUS=0, so we use our new friend the WHERE= option for output data sets to subset the observations we want.

```
data _null_; set info(obs=1);
     call symput ('start',
                  put(inaug,best12.));
     call symput ( 'range' ,
         put(today()-inaug+1,best12.));
     run;

data temp; set info end=eof;
     array alldates{&range.} _temporary_;
     do i=inaug to lastday;
        alldates{i-&start.+1}+1;
        end;
     if eof;
     date= &start. - 1;
     do i=1 to &range.;
        status=alldates{i}>=4;
        output;
        date+1;
        end;
     keep date status;
     run;
proc means min max noprint;
     by status notsorted;
     var date;
     output out=new(where=(status=1))
       min=start max=end;
     run;
```

This was a much nicer solution, in that we didn't create all those observations. However, we still write out one observation for each day since Washington's inauguration. This might still be undesirable depending on the real-world case you're analyzing. So we refine the solution further, and only write out the observations we really want.

In this example we still use the &START and &RANGE macro variables (see the SAS code above), and we still use a temporary array. But when we hit end-of-file, we run back through the temporary array and determine which observations are to be written. We make use of the LAG function here. Many users may think that the LAG function only works in conjunction with observations, but in fact the LAG function simply returns the value of the argument for the last time that particular LAG function was called. Also, we use the peculiar-looking but useful "double not". This is because the first time the LAG function is called for a variable, it will return a missing value. The "not" of a missing value is 1, and the "not" of 1 is 0. Likewise, if the LAG function returns a 0, a double not results in a 0, so a missing value and a 0 cause the same value to be returned. This is useful when you want to perform arithmetic on the value. Performing arithmetic on a missing value results in a missing value, which is not what we want. Our variable TRANS will be either 00, 01, 10, or 11, and we are only interested in 01 and 10, indicating a transition from one status to another. We set our START and END variables accordingly. We end up with the same data set that PROC MEANS produced in the previous example, but without the large intervening data set and without the processing time of PROC MEANS.

```
data temp; set info end=eof;
     array alldates{&range.} _temporary_;
     do i=inaug to lastday;
```

```
      alldates{i-&start.+1}+1;
      end;
   if eof;
   start= &start. + 1;
   end= start;
   do i=1 to &range.;
      status=alldates{i}>=4;
      trans=(^^lag(status))*10 + status;
      if trans=10 then do;
         output;
         end;
      else if trans=01 then do;
         start=end;
         end;
      end+1;
      end;
   output;
   keep start end;
   run;
```

Just for completeness' sake, here are the 13 date ranges determined by the three different approaches to this problem. (The program was last run on 20JAN1999).

```
START=05MAR1817 END=06JUL1826
START=05MAR1829 END=06JUL1831
START=05MAR1841 END=06APR1841
START=07APR1841 END=25FEB1849
START=06MAR1849 END=17JUN1849
START=05MAR1853 END=03JUN1868
START=05MAR1869 END=10OCT1869
START=05MAR1885 END=25JUL1885
START=05MAR1897 END=15MAR1901
START=15SEP1901 END=16SEP1901
START=21JAN1961 END=22OCT1964
START=21JAN1969 END=28MAR1969
START=21JAN1981 END=20JAN1999
```

CONTACT INFORMATION

Richard D. Langston
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
(919) 677-8000
sasrdl@wnt.sas.com (Rick Langston)