SYMPUT and SYMGET:
Getting DATA Step Variables and Macro Variables to Share

Christianna Williams, PhD
The University of North Carolina at Chapel Hill

ABSTRACT
Because one of the most powerful incentives to use the SAS® macro language is to allow SAS programs to be more data-driven, it is critical for the DATA step and the macro facility to “talk” to each other. The SYMPUT routine and the SYMGET function provide two mechanisms to facilitate this communication. Specifically, SYMPUT provides a way to turn DATA step variables into macro variables, and SYMGET does the converse, grabbing macro variable values and assigning them to DATA step variables. The purpose of this tutorial-style paper is to provide several examples of the use of these tools, using the different types of arguments that each accepts, and a few rules that may prevent a headache or two as one starts to write programs in which the DATA step and the macro facility share information. The focus will be on SYMPUT because of its greater utility.

INTRODUCTION
It is often desirable to pass information from one part of a program to another. This might be putting the value of a DATA step variable into a title, or it might be preserving the value of some system macro variables in a data set. A third example is selecting observations or doing some other processing in one data set based on data values in a different data set. As is usually the case in SAS, there may be multiple ways to achieve each of these goals, but frequently an efficient, reproducible method will involve interface between the DATA step and the Macro facility. In this paper, I am going to introduce a CALL routine and a SAS function that can facilitate and automate this communication. A schematic representation is shown in Figure 1.

![Figure 1. Schematic representation of communication between the DATA step and the macro facility](image)

Part of the utility of these tools is that they can be used both with the DATA step and macro variables that you or other users define as well as the automatic variables that SAS defines. Examples of both directions of communication using both types of DATA step and macro variables will be provided. I should point out that there are other mechanisms of achieving this communication (for example the CALL EXECUTE routine and the RESOLVE function and the INTO clause in PROC SQL). I’ll focus on SYMPUT and SYMGET because they are used in the DATA step and deal specifically with variables (both macro and DATA step), while RESOLVE can
accept other macro expressions as arguments and CALL EXECUTE deals with execution of macro programs. (In other words, they are a topic for another day…).

**DEFINITIONS AND SYNTAX**

What are SYMPUT and SYMGET? Simply put, SYMPUT is a SAS language routine that assigns a value produced in a DATA step to a macro variable. Conversely, SYMGET is a SAS language function that returns the value of a macro variable to the DATA step during DATA step execution. Next, let’s get the syntax out of the way. This will provide a good way to start introducing some examples.

**SYMGET SYNTAX**

The syntax for SYMGET is the following:

```sas
myvar = SYMGET(argument) ;
```

In this statement the argument can be one of the following three things:

1) The name of a macro variable, enclosed in quotes but with no ampersand, as follows:

```sas
myvar = SYMGET('macrovar');
```

So, here the DATA step variable MYVAR takes on the value of the macro variable &MACROVAR. MYVAR will be a character variable, and unless specified otherwise, it will have a length of 200.

2) The name of a DATA step character variable, specified without quotation marks, which contains the names of one or more macro variables. The correct functioning of the statement below requires that the value of the DATA step variable CHARVAR is the name of a macro variable at each iteration of the DATA step:

```sas
myvar = SYMGET(charvar);
```

So, in each iteration of DATA, the value of the DATA step variable CHARVAR supplies the name of a macro variable – and the value of that macro variable gets assigned to the DATA step variable MYVAR.

3) A character expression that constructs a macro variable name. The following statement assumes that there are macro variables &V1, &V2, &V3, . . . &Vx for each iteration of the DATA step:

```sas
myvar = SYMGET('v'||LEFT(_N_));
```

**SYMGET EXAMPLES**

**SYMGET EXAMPLE 1a – Using SYMGET with the name of a user-defined macro variable as the argument**

We’ll use this simple (minded) example to illustrate a couple caveats with SYMGET, and to raise a question about its usefulness. Consider the following code:

```sas
%LET ok=yes;
DATA example_1a ;
LENGTH check2 $5;
check1 = SYMGET('ok') ;
check2 = SYMGET('ok') ;
check3 = "&ok";
check4 = SYMGET('ok') ;
LENGTH check4 $3 ;
RUN;
```
The first statement assigns the value ‘yes’ to a macro variable &ok. Note that this may replace a value that &ok already had or may be creating a new macro variable. In the DATA step, we assign values based on &ok to each of three DATA step variables, twice using SYMGET and once with a simple assignment statement. Let’s look at a PROC PRINT of the data set TEST.

```
check1  check2  check3  check4
yes     yes     yes     yes
```

So, we conclude that the four different assignment statements have identical results, right? WRONG! Here’s a portion of the PROC CONTENTS output.

```
-----Alphabetic List of Variables and Attributes-----
#  Variable   Type  Len  Pos
--------------------------
 2  check1    Char  200  5
 1  check2    Char  5    0
 3  check3    Char  3    205
 4  check4    Char  200  208
```

Why do these four variables have different LENGTHs when they all have the same value – and three of them obtained those values from identical assignment statements? Well, in the absence of other instructions the length of a new character variable will be 200. So, the variable CHECK1 has a length of 200. Using a LENGTH statement we specified that CHECK2 have a length of $5, which it does. But then why does CHECK4 have a length of 200? The reason is that the LENGTH statement will only affect the length of a variable if that variable hasn’t already gotten its length assigned in some other way. Here, CHECK4 has already gotten its value from the SYMGET function, which at that point didn’t know any better than to give it the default length of 200. Helpfully, the SAS log gives us a warning “Length of character variable has already been set”, although it doesn’t tell us which character variable. So, what about CHECK3, which didn’t use SYMGET at all...Well, the truth is you don’t really need the SYMGET function to assign the value of a macro variable to a DATA step variable; an regular assignment statement works just fine in this instance, provided you use the double quotes and the ampersand. Further, CHECK4 gets the LENGTH of 3 because DATA step variables created by assignment statements take on the length of the value they are assigned at the first iteration of the data step – here 3 because the word “yes” has 3 characters.

Of course, this might make you wonder about what would be the effect of a few variations on this assignment statement; so as an aside I can’t resist (i.e. this doesn’t have much to do with how SYMGET works, but it might help you on a NESUG problem-solving contest some time...), let’s add a few more statements to our code:

```
%LET ok=yes;
DATA getting_ridiculous ;
LENGTH check2 $5;
check1 = SYMGET('ok');
check2 = SYMGET('ok');
```
Here is a PRINT of our new silly data set.

```
check2  check1  check3  check4  check5  ok  check6  yes  check7
  yes     yes     yes     yes     .     .     .     .     &ok
```

PROC CONTENTS indicates that CHECK5 and CHECK6 are each numeric with length 8, as are OK and YES, while CHECK7 is a character variable with length 3. We also get notes in the log indicating that the variables OK and YES are uninitialized. So, what's going on? Well, we've already discussed CHECK1 – CHECK4. Because there is nothing in the assignment statement for CHECK5 to tell SAS that OK is a macro variable, this statement tries to give CHECK5 the value of a DATA step variable called OK. Unfortunately, OK doesn't (yet) exist, so CHECK5 is missing. Further, because we have just (unintentionally) told SAS that there is a variable called OK on our data set, it creates one, but of course it is missing – hence the log note about this uninitialized variable. The assignment statement for CHECK6 tries to assign the value of the macro variable &OK to a DATA Step variable, but because there are no quotes around the &OK, SAS assumes CHECK6 is intended to be numeric, but when &OK resolves it has the value 'yes', which is clearly not numeric, so CHECK6 ends up missing as well. The value of CHECK7 is explained by the fact that SAS doesn't consider ampersands within single quotes to indicate a macro variable, so CHECK7 simply gets assigned the character string &ok, which has three characters so CHECK7 becomes a character variable with length 3. Can you think of other screwy variations on this theme? I can. See the example code and output that are also on the CD.

**SYMGET EXAMPLE 1b – Using SYMGET with the name of an automatic macro variable as the argument**

I won't belabor this one quite so much, but for symmetry, I include an example of using SYMGET to store the value of an automatic macro variable as a DATA step variable.

```
DATA example_1b ;
LENGTH today1 $9 ;
today1 = SYMGET('SYSDATE9') ;
today2 = "&SYSDATE9" ;
RUN;
```

PROC CONTENTS indicates that both TODAY1 and TODAY2 are character variables with length 9, and PROC PRINT shows the following:

```
today1       today2
  17JUL2004    17JUL2004
```
What if we wanted to store the date as a numeric variable (i.e. the way SAS stores a date – as the number of days since January 1, 1960)? We could use the following code:

```sas
DATA another_1b;
today3 = INPUT(SYMGET('SYSDATE9'),date9.);
today4 = INPUT("&SYSDATE9",date9.);
RUN;
```

The result is identical for TODAY3 and TODAY4 – both are numeric, have length 9 and have the value 16269 (the number of days from January 1, 1960 until July 17, 2004). Of course, we could add a FORMAT so that the date would print out looking like a date. All of the examples given so far seem to point to a dirty little secret…SYMGET seems to be unnecessary. We can accomplish the same result with a garden-variety assignment statement, and often this is simpler because, in the case of the character variables, no LENGTH statement is needed. Let’s see what else SYMGET can do for us.

**SYMGET EXAMPLE 2 – Using SYMGET with the name of a DATA step character variable as argument**

The second type of argument that can be provided to the SYMGET function is the name of a DATA step character variable. The statement will look something like this:

```sas
myvar = SYMGET(charvar);
```

Using SYMGET in this way assumes that the value of the character variable CHARVAR is the name of a macro variable at each iteration of the DATA step. This statement then assigns the value of those macro variables to the DATA step variable MYVAR for each loop through the DATA step. In the following code, let’s say we have macro variables &HARD, &MODERATE and &EASY, each of which contains the number of kilocalories burned for each hour of strenuous, moderate and easy exercise. These values might be changed depending on the weight of the individual concerned.

```sas
%LET hard=650;
%LET moderate=375;
%LET easy=240;
DATA exercise;
INPUT sport $ intensity $;
CARDS;
golf easy
dance moderate
bike hard
jog hard
swim hard
walk moderate
stretch easy ;
RUN;
DATA calories;
SET exercise;
cal_char = SYMGET(intensity);
cal_num = INPUT(SYMGET(intensity),4.) ;
RUN;
```
The three %LET statements above assign the calorie expenditure values for this run of the program. Of course, these macro variables could get their values in some other way...such as SQL INTO: or user input through %WINDOW or from a different data set using SYMPUT. Then, in the first DATA step (DATA exercise), we read in a list of types of exercise and the intensity category for each. Finally, in the second DATA step (DATA calories), we use SYMGET twice. In the first instance, the SYMGET function will take the value of the character variable INTENSITY (which is either “easy”, “moderate” or “hard” depending on the observation), find the value of the associated macro variable (i.e. 240 if INTENSITY is “easy”, 375 if “moderate” and 650 if “hard”) and assign that value to the variable CAL_CHAR, which is initialized here as a character variable with length 200 (in the absence of a LENGTH or ATTRIB specifying otherwise). We might prefer to have this information stored in a numeric variable – to simplify later calculations, for example. The second assignment statement does this, by using the INPUT function to convert the character value that the SYMGET function delivers to a numeric value. CAL_NUM will thus be a numeric variable with length 8. See the PRINT below:

```
  sport   intensity cal_char cal_num
  golf    easy      240      240
  dance   moderate  375      375
  bike    hard      650      650
  jog     hard      650      650
  swim    hard      650      650
  walk    moderate  375      375
  stretch easy      240      240
```

So, CAL_CHAR and CAL_NUM look the same in the listing here, but one is character the other numeric. As noted above, the correct functioning of SYMGET in this situation depends on the fact that there was a macro variable corresponding to each value of INTENSITY encountered in the data. What if this weren’t the case – if, for example the data set EXERCISE had an additional observation, with SPORT = “pilates” and intensity = “painful”? There would be no error, but you’d get notes in the log during the execution of the DATA step for CALORIES something like the following:

```
NOTE: Invalid argument to function SYMGET at line xx column 12.
NOTE: Invalid argument to function SYMGET at line xx column 17.
sport=pilates intensity=painful cal_char= cal_num=._ERROR_=1 _N_=8
```

And for the eighth observation in CALORIES, where SPORT = “pilates” and intensity = “painful”, CAL_CHAR and CAL_NUM would both be missing. As shown below, the preceding example could very readily be done without SYMGET, but it might take a little more code, and this solution might be less desirable than having the values associated with each intensity established (for example by user input) at the beginning of the program (or in a file to be %INCLUDED).
SYMGET EXAMPLE 3 – Using SYMGET with a character expression as the argument

The third type of argument that can be provided to the SYMGET function is a character expression that, when evaluated, constructs the name of a macro variable. For example,

```
myvar = SYMGET('v'||LEFT(_N_));
```

For this one, let’s expand on the previous example. We are going to add some more macro variables with values corresponding to the amount of time an industrious individual intends to exercise on each day of a busy week and the type of exercise she plans to do:

```
* Duration of exercise each day of week;
%LET time1=2.0;
%LET time2=1.0;
%LET time3=0.5;
%LET time4=0.6;
%LET time5=0.2;
%LET time6=1.0;
%LET time7=1.2;

* Exercise planned for each day of week;
%let sport1 =bike;
%let sport2 =walk;
%let sport3 =jog;
%let sport4 =stretch;
%let sport5 =bike;
%let sport6 =swim;
%let sport7 =walk;

DATA eachday
wholeweek (KEEP = calsofar timesofar
RENAME = (calsofar=calperweek timesofar=hrsperweek));
DO day = 1 TO 7;
LEN
```
total time and calorie expenditure for the week (WHOLEWEEK). Through the DO loop, we'll get some info for each day. For each DAY (observation), by concatenating the string 'sport' with the DAY number, the DATA step variable SPORT will get a value corresponding to the appropriate macro variable (i.e. &SPORT1 when DAY = 1, &SPORT2 when DAY = 2, etc...). The IF-THEN/ELSE block identifies the appropriate INTENSITY level according to the indicated SPORT. The next statement (assignment of CAL_NUM) uses the SYMGET function in the same way as in the previous example. The TIME variable gets its value in a similar way to SPORT, except that, because we want TIME to be numeric, we enclose pass the result of SYMGET (which will always be character) to the INPUT function, along with an INFORMAT that will preserve the decimal info. The remaining statements simply compute the calorie expenditure, and perform the desired summarization. I rename the summary variables on the WHOLEWEEK data set. PRINTs of EACHDAY and WHOLEWEEK are shown below:

<table>
<thead>
<tr>
<th>PRINT of EACHDAY:</th>
</tr>
</thead>
<tbody>
<tr>
<td>day</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRINT of WHOLEWEEK:</th>
</tr>
</thead>
<tbody>
<tr>
<td>calperweek</td>
</tr>
<tr>
<td>3374</td>
</tr>
</tbody>
</table>

Now, I’ll turn to the SYMPUT routine, which passes information in the other direction – from the DATA step to the macro processor.

SYMPUT SYNTAX
The syntax for SYMPUT is the following:

```
CALL SYMPUT(macrovar, value) ;
```

The first argument will yield the name of the macro variable being created or assigned a value and the second argument will provide the value. However, each of these arguments can take three different forms. The first argument (denoted “macrovar” above) can be one of the following:

1) a character string enclosed in quotes (must be a valid macro variable name), as in...

```
CALL SYMPUT('macrovar', value) ;
```

This statement would assign the value given by the second argument to a macro variable called MACROVAR, creating &MACROVAR if it didn’t already exist.

2) the name of a DATA step character variable, as in...

```
CALL SYMPUT(charvar, value) ;
```
Here, on each iteration of the DATA step, a macro variable is created (or assigned a value) the name of which corresponds to the value of the DATA step character variable CHARVAR. The value of each macro variable will be determined by second argument (details on this part later).

3) a character expression that, when evaluated, yields a valid macro variable name, such as...

\[
\text{CALL SYMPUT('v'||LEFT(_N_), value);} 
\]

Here, macro variables with names &V1, &V2, &V3, and so on will be created (or assigned values), one for each value of the automatic DATA step variable _N_, which corresponds to the number of times the DATA step has iterated.

The second argument (denoted “value” above) can be one of the following:

1) a string enclosed in quotes, as in...

\[
\text{CALL SYMPUT('check', 'yes')} ;
\]

This statement would assign the value ‘yes’ to a macro variable &CHECK, creating &CHECK if it didn’t already exist.

2) the name of a DATA step variable (numeric or character), the values of which will be assigned to the macro variable(s) in question, as in

\[
\text{CALL SYMPUT('check', code)} ;
\]

For each iteration of the DATA step, the macro variable &CHECK would take on the value of the DATA step variable CODE. Note that this type of second argument to the SYMPUT routine might be more useful in conjunction with a different type of first argument (e.g. one that can vary from one iteration of the DATA step to the next); we’ll see some examples below.

3) a DATA step expression, such as.

\[
\text{CALL SYMPUT('check', 'v'||LEFT(_N_))} ;
\]

Here, &CHECK would get values ‘v1’, ‘v2’, ‘v3’, and so on with the _N_ incrementing by one for each trip through the DATA step.

OK! Let’s move on to some more concrete examples.

**SYMPUT EXAMPLES**

**SYMPUT EXAMPLE 1 – Putting the number of observations in a data set into the TITLE**

Let’s assume that we have a data set that contains a bunch of X-Y observations, and we want to do some analyses of the relationship between X and Y, starting with a simple scatter plot. Further, I want my title to include the number of observations analyzed. This week my data set has 100 valid observations, but next week it might have 75 – or 7500, so I don’t want to “hard code” the value 100 into the title. This is a perfect situation for putting the number of observations into a macro variable, which can then go in the title – and SYMPUT is perfectly suited for the task. Of course, we’ll do variations on the theme to get it just right – consider the following code:

```sas
DATA _NULL_ ;
   SET mydata (KEEP = x y WHERE=(x NE . AND y NE .)) END=lastobs ;
   IF lastobs THEN CALL SYMPUT('numobs1',_N_) ;
```
The sole purpose of this DATA _NULL_ step is to determine the number of observations in the MYDATA data set that will be included in the PLOT. (Aside: The reason for the WHERE= option on the SET statement is to ensure that I am counting only observations that will be included in the analysis – that is, are non-missing for both X and Y; we want to be sure these are the only observations read – and therefore counted as _N_ gets incremented.) When the last observation is read, the automatic variable _N_ will contain the number of observations that will be plotted; so, I want to store this in a macro variable, which can then be passed to the TITLE statement. The three different CALL SYMPUTs are just to help illustrate how it works, as we’ll see below. Three macro variables, called &NUMOBS1, &NUMOBS2, and &NUMOBS3 are created, and each is assigned a value that is some permutation of the value of _N_, which as we’ve said had the number of observations that will be plotted. The output below shows the results for the three titles:

| Plot of 100 observations | Plot of 100 observations | Plot of 100 observations |

Only the third one gives us what we really want – without extra blanks before or after the 100 gets inserted into the title. To accomplish this, for &NUMOBS3 we use the LEFT function to left-justify the value of _N_ (i.e. get rid of leading blanks) and then use TRIM to eliminate the trailing blanks.

**SYMPUT EXAMPLE 2 – Storing information about DATA step variables to direct later processing**

In this example, I want to determine the earliest and latest dates that interviews were conducted to collect data for a research study. I’d like to write these values to the SAS log for documentation, but further I want to select and analyze observations from a different data set, which contains information about hospitalizations, only if the admission date falls within the window specified by the interviews.

```sas
DATA _NULL_;  
  SET interviews (KEEP = intvdate) END=lastobs ;  
  IF _N_ = 1 THEN DO;  
    start = intvdate;  
    finish = intvdate ;  
  END;  
  RETAIN start finish;  
  start = MIN(start,intvdate) ;  
  finish = MAX(finish,intvdate) ;  
  IF lastobs THEN DO;  
    CALL SYMPUT('start',LEFT(PUT(start,DATE9.))) ;  
    CALL SYMPUT('finish',LEFT(PUT(finish,DATE9.))) ;  
  END;  
RUN;
```
Most of this code is devoted to identifying the earliest and latest interview dates and keeping track of this information, so that, when the last observation in the INTERVIEWS data set is read, the DATA step variables START and FINISH will contain the desired information. The two SYMPUT CALLs will create macro variables &START and &FINISH that contain these values, formatted as DATE9 so that they'll be readily interpretable as dates when written to the log:

```sas
490 %PUT The earliest date is &start.. ;
The earliest date is 14MAR2000.
491 %PUT The last date is &finish.. ;
The last date is 28OCT2002.
```

As noted, the next thing we want to do is select observations from a data set of hospitalizations for which the admission date falls between these two dates. The code is pretty straightforward:

```sas
DATA select ;
SET hospitalizations;
   IF "&start"D LE admitdate LE "&finish"D;
RUN;
```

Without a MERGE or join of any type I can thus transfer information from one DATA step to another. Here the macro variables &START and &FINISH are resolved by placing their names in double quotes. We need them to be actual SAS dates (not text as macro variables are always stored), and the D appended to the character strings tells SAS to convert the strings to SAS dates so that the needed comparison can be evaluated. I have the SYMBOLGEN system OPTION turned ON so that the log also shows the values of the macro variables:

```sas
523     DATA select ;
524     SET hospitalizations;
      SYMBOLGEN: Macro variable START resolves to 14MAR2000
525     IF "&start"D LE admitdate LE "&finish"D;
      SYMBOLGEN: Macro variable FINISH resolves to 28OCT2002
526     RUN;

      NOTE: There were 6411 observations read from the data set
      WORK.HOSPITALIZATIONS.
      NOTE: The data set WORK.SELECT has 3931 observations and 4 variables.
```

Note that this information could also be transferred to a PROC, in order to select observations for analysis. For example, I could compute univariate statistics on the length of stay (LOS) for observations falling within the window:

```sas
PROC UNIVARIATE DATA = hospitalizations PLOT;
WHERE "&start"D LE admitdate LE "&finish"D;
```
VAR los;
RUN;

SYMPUT EXAMPLE 3 – Using a “CONTROL” data set to direct later processing.
The last SYMPUT example uses what I call a “control” data set. By this I mean a small data set that contains information that will dictate the manner in which another part of the program will be carried out. In this example, I have a 4-observation data set called CUTPOINTS, which contains the 1%, 5%, 95% and 99% -ile points for two performance tests (a WALKing speed test – seconds to walk 20 feet and a GRIP strength test – measured in kilograms). Let’s say I want to select observations from another, much larger data set based on these cutpoints. Here is the control data set, CUTPOINTS:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Q_1</th>
<th>Q_5</th>
<th>Q_95</th>
<th>Q_99</th>
</tr>
</thead>
<tbody>
<tr>
<td>WALK</td>
<td>F</td>
<td>4.2</td>
<td>5.7</td>
<td>20.5</td>
<td>43.5</td>
</tr>
<tr>
<td>WALK</td>
<td>M</td>
<td>4.0</td>
<td>5.2</td>
<td>16.8</td>
<td>37.8</td>
</tr>
<tr>
<td>GRIP</td>
<td>F</td>
<td>6.0</td>
<td>9.0</td>
<td>28.0</td>
<td>31.0</td>
</tr>
<tr>
<td>GRIP</td>
<td>M</td>
<td>10.0</td>
<td>16.0</td>
<td>44.0</td>
<td>50.0</td>
</tr>
</tbody>
</table>

I could merge this information into my large data set, but another way to accomplish the selection, which might be more efficient, is to use the information in CUTPOINTS to create several macro variables, specifically one for each percentile for each test for men and for women. SYMPUT can handle this beautifully:

DATA _NULL_;  
  SET cutpoints;  
  ARRAY q{4} q_1 q_5 q_95 q_99;  
  DO i = 1 to 4;  
    CALL SYMPUT(test||gender||TRIM(LEFT(i)),q{i});  
  END;  
RUN;  
%PUT _USER_;

In this DATA _NULL_ step, 16 macro variables get created and assigned values. In the CALL SYMPUT statement, to the left of the comma is an expression that will resolve to be the name of the macro variable. The value of the DATA step variable TEST (‘GRIP’ or ‘WALK’) will be concatenated with the value of gender (‘M’ or ‘F’), which will in turn be concatenated with a 1,2,3, or 4 – for the 1\textsuperscript{st}, 5\textsuperscript{th}, 95\textsuperscript{th} and 99\textsuperscript{th} percentiles, respectively. These macro variables will be assigned the values corresponding to each of these percentiles for each of the 4 test-gender combinations, each of which corresponds to an observation in the CUTPOINTS data set. The %PUT _USER_ statement puts some very useful information into the log:

<table>
<thead>
<tr>
<th>GLOBAL</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GRIPM2</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>WALKM4</td>
<td>37.8</td>
<td></td>
</tr>
<tr>
<td>GRIPM3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>GRIPM1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>WALKF1</td>
<td>4.2</td>
<td></td>
</tr>
</tbody>
</table>
dictate the appropriate macro variable to retrieve for the proper cutpoint. The DATA step below will select observations in the best and worst 1% by gender and test from a large data set containing test results from several thousand men and women. Note that it uses SYMGET to retrieve the appropriate cutpoints. There are existing DATA step variables WALKTIME and GRIP, which correspond to the test performance for the individual represented in each observation of the BASELINE data set; BASELINE is also presumed to have a variable GENDER (coded 'M' or 'F') that indicates the individual’s gender – and thus helps to dictate the appropriate macro variable to retrieve for the proper cutpoint.

```
DATA best1 poorest1;
  SET baseline ;
  IF ( walktime GE SYMGET('walk'||LEFT(gender)||'4') ) OR
    ( grip LE SYMGET('grip'||LEFT(gender)||'1') ) THEN OUTPUT poorest1 ;
  ELSE IF ( walktime LE SYMGET('walk'||LEFT(gender)||'1') ) OR
    ( grip GE SYMGET('grip'||LEFT(gender)||'4') ) THEN OUTPUT best1;
RUN;
```

The log indicates that some observations have been selected and an inspection of a PRINT of the resulting data set (BEST1 is shown) suggests we got what we wanted.

<table>
<thead>
<tr>
<th>walktime</th>
<th>grip</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>26</td>
<td>F</td>
</tr>
<tr>
<td>15.4</td>
<td>32</td>
<td>F</td>
</tr>
<tr>
<td>7.4</td>
<td>31</td>
<td>F</td>
</tr>
<tr>
<td>5.8</td>
<td>50</td>
<td>M</td>
</tr>
<tr>
<td>5.7</td>
<td>52</td>
<td>M</td>
</tr>
<tr>
<td>3.4</td>
<td>49</td>
<td>M</td>
</tr>
<tr>
<td>4.2</td>
<td>20</td>
<td>F</td>
</tr>
<tr>
<td>1.8</td>
<td>21</td>
<td>F</td>
</tr>
<tr>
<td>3.1</td>
<td>19</td>
<td>F</td>
</tr>
<tr>
<td>5.6</td>
<td>52</td>
<td>M</td>
</tr>
<tr>
<td>6.0</td>
<td>54</td>
<td>M</td>
</tr>
<tr>
<td>4.0</td>
<td>50</td>
<td>M</td>
</tr>
<tr>
<td>6.1</td>
<td>32</td>
<td>F</td>
</tr>
<tr>
<td>5.5</td>
<td>33</td>
<td>F</td>
</tr>
<tr>
<td>9.4</td>
<td>32</td>
<td>F</td>
</tr>
<tr>
<td>3.6</td>
<td>35</td>
<td>M</td>
</tr>
<tr>
<td>5.3</td>
<td>34</td>
<td>F</td>
</tr>
</tbody>
</table>
The program could be readily modified to keep track of whether the observation was selected for walking speed, grip strength or both.

CONCLUSIONS

I find the SYMPUT routine to be extremely useful, and I frequently use it in ways similar to those I've shown here in the examples – as a convenient way to move DATA step information from one DATA step to another or from a DATA step to a PROC, TITLE or FOOTNOTE. The beauty of this is that it allows your SAS programs to be more flexible – their correct functioning does not depend so much on the exact conditions of the input data. I love this because I am always trying to re-use code for data processing and analyses that I've written before with the validity of the results not being overly dependent on the things about the data that are likely to change from one run to the next. Further, the SYMPUT routine itself is quite flexible, allowing a variety of types of arguments – the values of which may themselves be data-driven.

I myself don't use SYMGET quite so much, as often a simple assignment statement can do the job of transferring macro variable values to DATA step variables. However, SYMGET does have its uses in the DATA step, and its SYMmetry with SYMPUT made its inclusion in this paper appealing. Also, SYMGET can be useful with Screen Control Language (SCL) programs, in which macro variables frequently need to be referenced without ampersands (examples in Carpenter, 1998).

I hope this paper has taught you at least one thing you didn't know, and that it can perhaps serve as a reference when you incorporate interfaces between the DATA step and the macro processor into your programs.

REFERENCES


ACKNOWLEDGMENTS

SAS is a Registered Trademark of the SAS Institute, Inc. of Cary, North Carolina. ® indicates US registration.

CONTACT INFORMATION

I welcome your comments, suggestions or questions at:

Christianna S. Williams, PhD
Cecil G. Sheps Center for Health Services Research
University of North Carolina at Chapel Hill
Campus Box # 7590
Chapel Hill, North Carolina 27599
Email: Christianna_Williams@unc.edu