Paper BB-05

Using SAS® Variable Lists Effectively

Howard Schreier, Arlington VA

ABSTRACT

"SAS variable lists" are a SAS language feature providing shortcuts for declaring and referencing variables. Instead of enumerating the variables, SAS variable lists allow the specification of rules to implicitly generate the needed variable names. The feature is useful, but there are nuances and non-intuitive aspects to be considered. This paper outlines the capabilities of SAS variable lists, then goes on to consider issues including timing (compilation vs. execution), differences between DATA step and PROC step usage, variable ordering, behavior of numeric suffixes, use of SAS variable lists in function calls, and contexts which do not fully support SAS variable lists.

INTRODUCTION

Many of the examples in this paper will make use of the small data set created by running

```
DATA demo ;
INPUT   Order Name $ Native $ Next   V1   V2   V3 ;
CARDS ;
           10 Larry  AA        101  1.1  1.2  1.3
           20 Moe    BB        202  2.1  2.2  2.3
           30 Curly  CC        303  3.1  3.2  3.3
;
```

As a SAS data set, it has a header holding structural information about variables, including

- name
- creation ordering (1, 2, 3 etc.)
- type (numeric or character)
- length

SAS can access that metadata directly from the header. We can employ PROC CONTENTS to expose it for review.

```
PROC CONTENTS DATA=demo VARNUM ;
RUN ;
```

```
Variables in Creation Order

#    Variable    Type    Len

1    Order       Num       8
2    Name        Char      8
3    Native      Char      8
4    Next        Num       8
5    V1          Num       8
6    V2          Num       8
7    V3          Num       8
```

WHAT IS A SAS VARIABLE LIST

Let's start with something that is **not** a SAS variable list:

```
PROC PRINT DATA=demo ;
VAR Name V1 V2 ;
RUN ;
```

There are three variable names in the VAR statement. Because they are enumerated explicitly, without abbreviation, they do not by definition constitute a SAS variable list. A SAS variable list is a construct which uses abbreviation and implication to name or reference variables. SAS then uses appropriate rules to expand these lists. In other words, SAS variable lists are shortcuts. There are four basic types of SAS variable list:

- Numbered Range Lists
- Name Range Lists
- Name Prefix Lists
- Special SAS Name Lists

The documentation for SAS variable lists is somewhat tucked away in the *Language Concepts* manual, as

SAS System Concepts
SAS Variables
SAS Variable Lists.

TYPICAL USAGE

Before delving into the details and subtleties, let's look at some commonplace examples.

NUMBERED RANGE LISTS

```
PROC PRINT DATA=demo ;
VAR Name V1-V3 ;
RUN ;
```

Obs	Name	V1	V2	V3
1	Larry	1.1	1.2	1.3
2	Moe	2.1	2.2	2.3
3	Curly	3.1	3.2	3.3

The two explicit variables have a common root (V), and non-negative integer suffixes (1 and 3) supply the range boundaries. The intervening hyphen implies inclusion of the variable(s) (V2) that will fill in and make the expanded list consecutive.

NAME RANGE LISTS

```
PROC PRINT DATA=demo ;
VAR Name--V1 ;
RUN ;
```

Obs	Name	Native	Next	V1
1	Larry	AA	101	1.1
2	Moe	BB	202	2.1
3	Curly	CC	303	3.1

The name range list, like the numbered range list, requires two variable names to establish the boundaries. Notice that here they are separated by **two** hyphens. The expansion involves including all of the variables that fall within the boundaries when the variables are considered in order, as reported in the PROC CONTENTS output (see above). In this example, the boundary variables (NAME and V1) are in positions 2 and 5, so the expansion includes them as well as NATIVE and NEXT from positions 3 and 4 respectively.

NAME PREFIX LISTS

```
PROC PRINT DATA=demo ;
VAR na: ;
RUN ;
```

Obs	Name	Native
1	Larry	AA
2	Moe	BB
3	Curly	CC

Like the numbered range list, the name prefix list involves roots. A colon follows the root (NA in this case), and the rule is to include variables in the data set (DEMO) whose names begin with that root. The ordering again follows that reported by PROC CONTENTS (see above).

SPECIAL SAS NAME LISTS

```
PROC PRINT DATA=demo ;
VAR _ALL_ ;
RUN ;
```

Obs	Order	Name	Native	Next	V1	V2	V3
1	10	Larry	AA	101	1.1	1.2	1.3
2	20	Moe	BB	202	2.1	2.2	2.3
3	30	Curly	CC	303	3.1	3.2	3.3

Here keywords are employed instead of variable names and their roots. _ALL_ indicates that every variable in the data set is to be included in the expansion, in the pre-existing order (as reported by PROC CONTENTS, above).

FRAMES OF REFERENCE

The examples we've just seen are useful, and they do illustrate the essence of SAS variable lists. However, they are rather simplified. In particular, we cannot always look at the existing structure of just one data set to determine which variables will be included in the expansion of a SAS variable list and how those variables will be ordered.

PROCEDURE STEPS

Consider this PROC PRINT step:

```
PROC PRINT DATA=demo(DROP = V1 RENAME = (Next=vNext) ) ;
VAR v: ;
RUN ;
```

It's similar to the example of a name prefix list presented above. However, the DROP option will exclude the variable V1 from the expansion, and the renaming of variable NEXT will cause it to be included in the expansion using its new name (but maintaining its original position). The result:

Obs	vNext	V2	V3
1	101	1.2	1.3
2	202	2.2	2.3
3	303	3.2	3.3

So we've seen that things can be a bit complicated in a PROC step.

DATA STEP

When we turn to the DATA step, things are **considerably** more complicated. First of all, the entire frame of reference for SAS variable lists shifts away from the structure of a single input data set. That's reasonable; after all, a DATA step may have no input data sets at all, or it may have more than one. Instead, the frame of reference is the program data vector (PDV), the space that holds variables in memory during DATA step processing. A consequence is that PROC CONTENTS alone does not supply enough information to predict how a SAS variable list will be expanded in a DATA step; we have to analyze the code.

We will also see that the expansion of SAS variable lists takes place during compilation (the phase during which SAS figures out what a DATA step is supposed to do, before it actually begins processing). Here is an example.

```
DATA _NULL_ ;
CALL MISSING(V3) ;
SET demo(DROP = V1 RENAME = (Next= vNext) ) end=Last ;
v_Again = 123 ;
ARRAY vv[*] v: ;
v_Last  = 456 ;
PUT vv[*]= ;
RUN ;
```

The first variable created is V3, in the CALL statement, so V3 is placed first in the PDV. SAS then turns to the SET statement and opens the header of the data set DEMO to learn the names, positions, and characteristics of the variables within. It honors the DROP option and so does not add V1 to the PDV. V3 is already in the PDV, so the presence of V3 in the data set does not affect the PDV. All of the other data set variables enter the PDV in order, and NEXT becomes VNEXT in the PDV, reflecting the RENAME option. The process of building the PDV continues with the assignment statement for V_AGAIN. Then we come to the ARRAY statement which includes the name prefix list "V:". It is expanded to include all of the variables having names beginning with the letter "V" and known at that point in the compilation. It does not include the variable V_LAST, because that variable is not introduced and added to the PDV until **after** the name prefix list in the ARRAY statement. The PUT statement displays all of the array elements, which in this situation means all of the variables in the expanded name prefix list. In the log we see

```
v3=1.3 vnext=101 v2=1.2 v_again=123
v3=2.3 vnext=202 v2=2.2 v_again=123
v3=3.3 vnext=303 v2=3.2 v_again=123
```

Note the absence of V_LAST. The lesson is that the location of a SAS variable list in the DATA step code can be critical.

Also worth noting is the absence of the variables _ERROR_ and _N_ from the expanded list. SAS variable lists do not include such automatic variables. The variable LAST is also excluded. It is user-named, but indirectly so, by means of the END= option in the SET statement. Such variables are excluded in the expansion of SAS variable lists.

DECLARATION VS. REFERENCE

A numbered range list can declare new variables, at least in some contexts. For example, consider

```
DATA ;
INPUT Var1-Var4 ;
CARDS ;
100 200 300 400
;
```

When the INPUT statement is compiled, SAS expands the numbered range list and sees that the variables do not exist, so they are added to the PDV at that point. Here's a more complicated example.

```
DATA ;
CALL MISSING(Var2) ;
INPUT Var1-Var4 ;
CARDS ;
100 200 300 400
;
PROC PRINT DATA=_LAST_;
RUN ;
```

The reference to VAR2 in the CALL statement causes it to be placed immediately in the PDV. When SAS gets to the INPUT statement and expands the numbered range list, it realizes that VAR2 already exists and that only the other three variables have to be added to the PDV. Running PROC PRINT against the output data set, we get

Obs	Var2	Var1	Var3	Var4
1	200	100	300	400

The values were read in the order specified in the numbered range list, so VAR1 has the value 100, and so forth. The PROC PRINT step lacks a VAR statement, so by default the presentation follows creation (PDV) order

The numbered range list is the only SAS variable list that can trigger variable creation. It is also the only one that can override PDV ordering in the expansion process. The other varieties are, by their nature, limited to referencing existing variables; they also lack features for changing variable order.

The use of numbered range lists to create new variables typically takes place in DATA steps. That isn't a restriction however. Here's an example of a PROC step that employs a numbered range list to declare variables.

```
PROC SUMMARY DATA=demo ;
VAR V1 V2 V3 ;
OUTPUT OUT=summout MEAN = Avg1-Avg3 ;
RUN ;
```

VARIABLE TYPE RESTRICTIONS

Name range lists and special SAS name lists have additional features to restrict expansion to one or the other variable type (character or numeric). In name range lists the desired type is spelled out between the two hyphens. For example, in

```
PROC PRINT DATA=demo ;
VAR Name-NUMERIC-V1 ;
RUN ;
```

the expansion contains all numeric variables between NAME and V1, inclusive. Here's the result:

```
Obs    Next     V1

 1      101    1.1
 2      202    2.1
 3      303    3.1
```

Similarly, running

```
PROC PRINT DATA=demo ;
VAR Name-CHARACTER-V1 ;
RUN ;
```

gives us

```
Obs    Name     Native

 1     Larry      AA
 2     Moe        BB
 3     Curly      CC
```

SAS special name lists have available, in addition to _ALL_, type-specific keywords. In this step:

```
PROC PRINT DATA=demo ;
VAR _CHARACTER_  ;
RUN ;
```

the keyword _CHARACTER_ causes all character variables to be included in the VAR statement, so the output looks like this:

```
Obs    Name     Native

 1     Larry      AA
 2     Moe        BB
 3     Curly      CC
```

Analogously, this code:

```
PROC PRINT DATA=demo ;
VAR  _NUMERIC_ ;
RUN ;
```

restricts processing to numeric variables, giving us

```
Obs    Order    Next     V1     V2     V3

 1       10      101    1.1    1.2    1.3
 2       20      202    2.1    2.2    2.3
 3       30      303    3.1    3.2    3.3
```

CONTEXTS

SAS variable lists are not supported in all contexts. In some places they are just partly supported. Moreover, the limitations are for the most part not documented. You often have to experiment.

GOOD NEWS

There are a number of important contexts where SAS variable lists are well supported and quite useful. Earlier we saw an example in which a prefix name list was used to build an array. Here is another ARRAY statement example, this one using a numbered range list.

```
DATA round(DROP = i) ;
SET demo ;
ARRAY vv[*] V1-V3 ;
DO i = 1 TO DIM(vv) ;
   vv[i] = ROUND(vv[i]) ;
   END ;
RUN ;
PROC PRINT DATA=round;
RUN ;
```

The output looks like this:

Obs	Order	Name	Native	Next	V1	V2	V3
1	10	Larry	AA	101	1	1	1
2	20	Moe	BB	202	2	2	2
3	30	Curly	CC	303	3	3	3

Function arguments are another context essentially made-to-order for SAS variable lists. For example, if we run

```
DATA plus ;
SET demo (KEEP = v:) ;
plus_A = SUM(0 ,    V1 , V2 , V3) ;
plus_B = SUM(0 ,    V1   -    V3) ;
plus_C = SUM(0 , OF V1   -    V3) ;
plus_D = SUM(0 , OF V1   V2   V3) ;
RUN ;
PROC PRINT DATA=plus ;
RUN ;
```

we get

Obs	V1	V2	V3	plus_A	plus_B	plus_C	plus_D
1	1.1	1.2	1.3	3.6	-0.2	3.6	3.6
2	2.1	2.2	2.3	6.6	-0.2	6.6	6.6
3	3.1	3.2	3.3	9.6	-0.2	9.6	9.6

All four formulas are in fact syntactically correct. PLUS_A uses commas to separate the individual function arguments. That works well here but would be cumbersome if there were say

40 rather than four arguments. PLUS_B is an attempt to use a numbered range list by coding a hyphen between the lower and upper boundary variables (V1 and V3, respectively). However, the hyphen is also the subtraction operator, and SAS resolves the ambiguity by assuming subtraction to be the intent; hence the incorrect results (-0.2 instead of 3.6 etc.). PLUS_C delivers the solution: If you want to use a SAS variable list inside the argument list of a function call, precede it with the keyword "OF". PLUS_D shows us that this keyword can also be used with explicit individual variable names (that is, in the absence of a SAS variable list).

SAS variable lists can be used with just about any function in the DATA step. Of course you must take care to provide the correct number of arguments, and arguments of the right type. This can get tricky when the arguments aren't explicit. It turns out that SAS variable lists are particularly useful in establishing arguments for functions that don't have a limited number of arguments. This would include summary functions (as in the example above), character concatenation functions (CAT, etc.) and COALESCE and COALESCEC.

BAD NEWS

SAS variable lists do not work in PROC SQL, at least not directly. Consider this example.

```
PROC SQL ;
SELECT _ALL_
 FROM demo(OBS=1) ;
SELECT *
 FROM demo(OBS=1) ;
SELECT *
 FROM demo(OBS=1 KEEP=_CHARACTER_) ;
QUIT ;
```

The first SELECT statement, which attempts to use a special SAS name list, fails. The log message is

```
ERROR: The following columns were not found in the contributing tables:
_ALL_.
```

Of course SQL offers its own more-or-less equivalent wild card, the asterisk. That is demonstrated in the second SELECT statement, which generates

```
Order   Name      Native     Next        V1        V2        V3
-------------------------------------------------------------
   10   Larry     AA          101       1.1       1.2       1.3
```

Also, SAS variable lists can be used in data set options coded within SQL statements. The final SELECT statement illustrates such usage and produces this result:

```
Name      Native
-----------------
Larry     AA
```

PARTIAL SUPPORT OF SAS VARIABLE LISTS

We've seen notable examples of contexts in which SAS variable lists do, and do not, work as you might hope. There are also places in which the support is incomplete.

Here's one example: the INPUT statement. If we run this code:

```
DATA _NULL_ ;
ARRAY abc[*] a b c (8 88 888);
INPUT a--c ;
PUT abc[*] ;
CARDS ;
9 99 999
;
```

the log shows

```
9 99 999
```

which is evidence that the INPUT statement operated on all three variables, including B. If we change the code to use the special SAS name list _NUMERIC_ in place of the name range list, we might expect the same outcome since in this example both SAS variable lists would expand to include the same variables in the same order (A,B,C). The modified code is

```
DATA _NULL_ ;
ARRAY abc[*] a b c (8 88 888);
INPUT _NUMERIC_ ;
PUT abc[*] ;
CARDS ;
9 99 999
;
```

but the result is

```
ERROR: Cannot use _NUMERIC_ as a variable name.
```

SAS VARIABLE LISTS IN THE PUT STATEMENT

The PUT statement is another context where support for SAS variable lists is just partial. Because the situation is rather complicated, and because the use of such shortcuts in the PUT statement can be convenient, it's worthwhile examining the limits thoroughly.

It's important to understand that there are two distinctly different ways to specify variables in a PUT statement. One syntax is the "variable-list", as it is called in the documentation. That sounds a lot like other terms we've been using, so it will be explicitly referred to here as the "PUT statement variable list". A PUT statement variable list comprises references to one or more variables, contained in a pair of parentheses. So, it may start like this:

```
PUT . . . (q  w  e  r  t  y)
```

A PUT statement variable list cannot itself include any of the other specifications commonly used in a PUT statement, such as formats or pointer controls. However, it **must** be followed by a second parentheses container, called a "format-list", containing not just formats but other specifications used in PUT statements (such as pointer controls). Moreover, the format list cannot be empty; something has to be coded. So, it may look like this:

```
PUT . . . (q  w  e  r  t  y)(5.2 +1) . . . ;
```

The specifications in the format list are applied to each of the variables in the PUT statement variable list, providing a significant shortcut. However, the variables here are still named individually. It's time to turn back to our main subject, SAS variable lists, in quest of more shortcutting.

It turns out that SAS variable lists are fully supported when used within PUT statement variable lists. Here is evidence.

```
DATA _NULL_ ;
SET demo(OBS=1) ;
PUT '[1] ' ( V1-V3                    ) ( +0 ) ;
PUT '[2] ' ( Order--Native            ) ( +0 ) ;
PUT '[3] ' ( Order-CHARACTER-Native   ) ( +0 ) ;
PUT '[4] ' ( Order-NUMERIC-Native     ) ( +0 ) ;
PUT '[5] ' ( n:                       ) ( +0 ) ;
PUT '[6] ' ( _ALL_                    ) ( +0 ) ;
PUT '[7] ' ( _CHARACTER_              ) ( +0 ) ;
PUT '[8] ' ( _NUMERIC_                ) ( +0 ) ;
RUN ;
```

Here we've exercised all eight varieties of SAS variable list (including the type-specific versions of the name range list and the SAS special name list). Why the "+0" specification? Recall that the format lists have to contain something. Since we don't need really anything, we keep the compiler happy by calling for a do-nothing pointer control. Here are the results that appear in the log.

```
[1] 1.1 1.2 1.3
[2] 10 Larry AA
[3] Larry AA
[4] 10
[5] Larry AA 101
[6] 10 Larry AA 101 1.1 1.2 1.3
[7] Larry AA
[8] 10 101 1.1 1.2 1.3
```

There are no failures or misinterpretations; all eight of the SAS variable lists work as advertised.

Now we turn to the other syntax for specifying variables in a PUT statement. The documentation refers to it simply as "variable". Note the singular noun, but also keep in mind that PUT statement "ingredients" can be re-used. In particular, two or more variables can be coded back to back. Such a statement can look like this:

```
PUT . . . Q  W  E  R  T  Y . . . ;
```

Note the absence of parentheses.

Now of course we want to know whether SAS variable lists work in this context. There is no problem with either numbered range lists or name range lists, as evidenced by running this code:

```
DATA _NULL_ ;
SET demo(OBS =1) ;
PUT'[1] ' V1-V3                    ;
PUT'[2] ' Order--Native            ;
PUT'[3] ' Order-CHARACTER-Native   ;
PUT'[4] ' Order-NUMERIC-Native     ;
RUN ;
```

The results are

```
[1] 1.1 1.2 1.3
[2] 10 Larry AA
[3] Larry AA
[4] 10
```

as expected. When we turn to prefix name lists, it's a different story. We run

```
DATA _NULL_ ;
SET demo(OBS=1) ;
PUT N: ;
RUN ;
```

and in the log we see

```
NOTE: Variable N is uninitialized.
.
```

The prefix name list was not recognized, and "N" was taken to be a simple variable name. This happens because the colon has a different meaning in the PUT statement; it indicates that a format is to be applied to a variable being written in list style. If parenthesized PUT statement variable lists and format lists were being used, as in

```
PUT ( v: )( : 6.1 ) ;
```

the compiler can distinguish between the two parentheses containers and thus knows the significance of any colon it encounters. That's not the case here.

Finally, we want to know whether SAS special name lists will work outside parentheses. We can start with character variables, by running

```
DATA _NULL_ ;
SET demo(OBS=1) ;
PUT _CHARACTER_ ;
RUN ;
```

In the log we see

```
NOTE: Variable _CHARACTER_ is uninitialized.
.
```

There is no ambiguity. The keyword "_CHARACTER_" is simply not recognized as such and is taken to be an ordinary variable name.

Now we'll turn to numeric variables and try

```
DATA _NULL_ ;
SET demo(OBS=1) ;
PUT _NUMERIC_ ;
RUN ;
```

The result is

```
ERROR: Cannot use _NUMERIC_ as a variable name.
```

The compiler sees that "_NUMERIC_" is special, but it thinks we're attempting to use it, inappropriately, as an ordinary variable. Why_CHARACTER_ and _NUMERIC_ are not at least treated consistently is left as a point of curiosity[1].

Finally we take up the type-independent SAS special name list. The code is

```
DATA _NULL_ ;
SET demo(OBS=1) ;
PUT _ALL_ ;
RUN ;
```

and in the log we see

```
Order=10 Name=Larry Native=AA Next=101 V1=1.1 V2=1.2 V3=1.3 _ERROR_=0 _N_=1
```

At first glance it seems that the SAS special name list _ALL_ can be used in this context, but we have to wonder why we are getting name-style output, and why the automatic variables _ERROR_ and _N_ appear. To examine things more thoroughly, let's use a more complete example:

```
DATA _NULL_ ;
SET demo(OBS=1 KEEP = na:) END=Last;
P8 = 88 ;
PUT'[1] '  +7 (_ALL_)(=) ;
PUT'[2] '      _ALL_     ;
P9 = 99 ;
RUN ;
```

This time we have two PUT statements, one of which, [1], has the PUT statement variable list and format list, while the other, [2], has just the bare keyword _ALL_. We've used a KEEP= option to reduce the number of variables entering the program data vector from the input data set, but we've also added some new variables via two carefully placed assignment statements and the specification of an END= option. Now we'll look at the results:

```
[1]        Name=Larry Native=AA P8=88
[2] Last=1 Name=Larry Native=AA P8=88 P9=. _ERROR_=0 _N_=1
```

It turns out that the bare _ALL_ in [2] is in fact a PUT statement feature, essentially unrelated to the SAS special name list _ALL_ that we can code inside the parentheses (as in [1]). What are the differences other than context?

- The bare _ALL_ automatically presents its output in named style, whereas we had to

code an equal sign to tell the SAS special name list _ALL_ to do that.

- The bare _ALL_ reports on automatic variables (here, _ERROR_ and _N_) as well as on variables created by options (here, LAST).
- The bare _ALL_ is expanded at run time, not during compilation.

The last conclusion is supported by the fact that the variable P9, not yet in the program data vector when the PUT statements are compiled, nevertheless is reported by the bare _ALL_.

OTHER ISSUES AND POINTS OF CURIOSITY

We've now covered the basic and more important aspects of SAS variable lists and their usage. We are left with a few caveats and curiosities.

ORDER OF COMPILATION EVENTS

Earlier we saw that, in DATA steps, SAS variable lists are expanded during compilation, and that only those variables already known to the compiler (or already implied in a numbered range list) are included. This makes the ordering of statements very important. In fact, it turns out that even the ordering of compilation events **within** a statement can affect results. Here's an example, showing both code and results from the log.

```
514  DATA _NULL_ ;
515  Num1 = 1 ;
516  Num2 = 2 ;
517  CatA = CAT(OF Num1 Num2) ;
518  CatB = CAT(OF _NUMERIC_) ;
519  TypeA = VTYPE(CatA) ;
520  TypeB = VTYPE(CatB) ;
521  PUT CatA= TypeA= / CatB= TypeB= ;
522  RUN ;

NOTE: Character values have been converted to numeric values at the places
given by:
      (Line):(Column).
      518:8
CatA=12 TypeA=C
CatB=12 TypeB=N
```

Variables CatA and CatB appear to evaluate to the same result (12). But notice the NOTE about type conversion, and that CatA is character whereas CatB is numeric (as detected by the VTYPE function). This is despite the fact that lines 515 through 517 create exactly two numeric variables, Num1 and Num2. Here's what happens. When the compiler analyzes line 517, it starts on the left side of the assignment statement and appends the receiving variable CatA to its list, but defers typing it (as either numeric or character). It then turns to the right side, sees that the function call returns a character result, and consequently establishes CatA as a character variable before turning to line 518. In examining line 518, SAS again adds the receiving variable CatB to its list and defers typing. When it moves on to the right side, it detects the SAS special name list (_NUMERIC_) and realizes that it must expand it before it can take up the expression in which it appears. However, it cannot expand _NUMERIC_ as long as the type of CatB is undetermined.

Having at this point no indication of the appropriate type, the compiler by default makes CatB numeric. So, the expansion of _NUMERIC_ yields three variables (Num1, Num2, and CatB). Since the CAT function returns a character result while the receiving variable CatB is numeric, type conversion to take place. At execution time, the CAT function strings together the three values (1, 2, and a missing value for CatB). This intermediate result ("12.") is converted to numeric (12) and stored in CatB.

A human reader of the code would probably see the intent and make CatB a character variable, but the DATA step compiler is not that clever. The lesson, in the context of SAS variable lists, is to pay attention to their **exact** location in the DATA step. Another lesson, not specific to the subject of this paper, is that it's almost always a really good idea to declare character variables and their lengths explicitly, rather than trusting the defaults and decision rules on which the compiler depends.

RANGES OF ONE

In either a numbered range list or a name range list, it's allowable for the lower and upper range boundaries to be the same, so that the expansion (contraction, actually) in effect is equivalent to naming a single variable. Here's a double example.

```
DATA _NULL_ ;
SET demo(OBS=1 KEEP = na: v:) ;
PUT'[1] '  _ALL_ ;
CALL MISSING (OF Name--Name V2-V2) ;
PUT'[2] '  _ALL_ ;
RUN ;
```

The result seen in the log is

```
[1] Name=Larry Native=AA V1=1.1 V2=1.2 V3=1.3 _ERROR_=0 _N_=1
[2] Name=  Native=AA V1=1.1 V2=. V3=1.3 _ERROR_=0 _N_=1
```

The CALL MISSING statement has just two arguments: NAME (from the name range list) and V2 (from the numbered range list). We can see that those are the only two variables processed by CALL MISSING and thus having missing values in the second PUT statement [2].

This construct may seem rather pointless, but it is advantageous in macro programming because it eliminates the need to treat ranges of one as special cases.

INVERTED RANGES

SAS will allow the opening boundary of a numbered range list to have a higher suffix than the closing boundary, as in

```
PROC PRINT DATA=demo ;
VAR V3-V1 ;
RUN ;
```

The output is

```
Obs     v3     v2     v1

 1     1.3    1.2    1.1
 2     2.3    2.2    2.1
 3     3.3    3.2    3.1
```

Such inversion does not work for name range lists. If we try running

```
PROC PRINT DATA=demo ;
VAR V3--V1 ;
RUN ;
```

the result is

```
ERROR: Starting variable after ending variable in data set.
```

GAPS IN NUMBERED RANGE LISTS

The consequences of referencing a non-existent variable by means of a numbered range list depend on the context. To illustrate, let's first derive a data set that lacks variable V2.

```
DATA gap ;
SET demo(OBS=1 DROP= na: V2) ;
RUN ;
PROC PRINT DATA=gap ;
RUN ;
```

The output looks like this:

```
Obs    Order    Next     V1     V3

 1       10      101    1.1    1.3
```

Now let's see what happens if we include a VAR statement implicitly referencing V2 in the PROC PRINT step:

```
PROC PRINT DATA=gap ;
VAR V1-V3 ;
RUN ;
```

In the log we see

```
 ERROR: Variable V2 in suffix list not in data set.
```

In a DATA step the outcome is different. Consider this code

```
DATA _NULL_ ;
SET gap ;
PUT'[1] ' (_ALL_)(=) ;
ARRAY vv[*] V1-V3 ( 2.1 2.2 2.3 );
PUT'[2] ' (_ALL_)(=) ;
RUN ;
```

It produces (in the log)

```
[1] Order=10 Next=101 V1=1.1 V3=1.3
[2] Order=10 Next=101 V1=1.1 V3=1.3 V2=2.2
```

The first PUT statement [1] reflects only the variables coming from the data set, including V1 and V3 but not V2. When the compiler encounters the numbered range list in the ARRAY statement, it sees the implicit and previously unknown variable V2 and tacks it onto the end of the program data vector, as reported in the second PUT statement [2]. We also note that at run time V1 and V2 receive values (1.1 and 1.2) from GAP, while in V2 the initial value (2.2) declared in the ARRAY statement persists.

CONSISTENT USE OF LEADING ZEROES IN NUMBERED RANGE LISTS

Consider this code:

```
DATA one ;
INPUT v_008-v_10 ;
PUT _ALL_ ;
CARDS ;
8.1 9.1 10.1
;
```

The leading zeroes in the lower boundary of the numbered range list could present a problem. Incrementation of the suffixes would generate variables named V_009 and V_010, but not V_10 as specified. In fact, SAS automatically prevents this. The output from the PUT statement is

```
v_08=8.1 v_09=9.1 v_10=10.1 _ERROR_=0 _N_=1
```

Notice that the extra zero in v_008 has been ignored and the variable receives the name V_08 instead. Then suppose you run this code:

```
DATA two ;
INPUT v_8-v_10 ;
PUT _ALL_ ;
CARDS ;
8.1 9.1 10.1
;
```

to generate a second data set. This time there are no leading zeroes in the numbered range list. That's fine; the compiler will increment from V_8 to produce V_9 and V_10, just as specified.

However there is still room for error. Look at what happens if we use this DATA step:

```
DATA both ;
SET one two ;
RUN ;
```

to concatenate the two data sets. We get

```
Obs    v_08    v_09    v_10    v_8    v_9

 1      8.1     9.1    10.1     .      .
 2       .       .     10.1    8.1    9.1
```

SAS does not "remember" that these variables were declared by means of numbered range lists and does not appreciate any intent to align almost-like-named variables. It just follows its rules. So it's wise to follow consistent practice in naming variables.

CHAR AS A PARTIAL ALIAS FOR CHARACTER

It appears to be undocumented, but the string "CHAR" can be substituted for "CHARACTER" in name range lists and special SAS name lists. For example, if we run

```
DATA _NULL_ ;
SET demo(OBS=1) ;
PUT'[1] ' (Name-CHARACTER-Next)(=) ;
PUT'[2] ' (Name-CHAR      -Next)(=) ;
RUN ;
```

we see this in the log.

```
[1] Name=Larry Native=AA
[2] Name=Larry Native=AA
```

Note that the two PUT statements generate the same results; the alias works. Turning to SAS special name lists, the test code is

```
DATA _NULL_ ;
SET demo(OBS=1) ;
PUT'[1] ' (_CHARACTER_)(=) ;
PUT'[2] ' (_CHAR_     )(=) ;
RUN ;
```

and the results of the PUT statements are

```
[1] Name=Larry Native=AA
[2] Name=Larry Native=AA
```

Again we have identical outcomes. Note that the PUT statements in the twp preceding examples employed parentheses containers. When these are not used, there is some divergence. To illustrate, we start with

```
DATA _NULL_ ;
SET demo(OBS=1) ;
PUT _CHARACTER_ = ;
RUN ;
```

We know from an earlier example that SAS does not recognize the keyword _CHARACTER_ in this context. So the results are predictable:

```
NOTE: Variable _CHARACTER_ is uninitialized.
_CHARACTER_=.
```

But when we use the shortened form (_CHAR_) as in

```
DATA _NULL_ ;
SET demo(OBS=1) ;
PUT _CHAR_ = ;
RUN ;
```

there is divergence (_CHAR_ is not an alias for _CHARACTER_) as the result is

```
ERROR: Cannot use _CHAR_ as a variable name.
```

In this respect, the effect of _CHAR_ is analogous to that of _NUMERIC_ but dissimilar to that of _CHARACTER_[1].

RESERVED, OR NOT

We know that three[1] of the four keywords used in special SAS name lists, are reserved in the DATA step. That is, they cannot be used as variables. To demonstrate this again, we can run

```
DATA _NULL_ ;
CALL MISSING(_ALL_, _CHARACTER_, _CHAR_, _NUMERIC_) ;
RUN ;
```

which produces

```
ERROR: Cannot use _ALL_ as a variable name.
ERROR: Cannot use _CHAR_ as a variable name.
ERROR: Cannot use _NUMERIC_ as a variable name.
```

However, the same protection does not necessarily exist in PROC steps. This code:

```
PROC SUMMARY DATA=demo ;
OUTPUT OUT= badnames
 MIN    (Order) = _NUMERIC_
 MAX    (Order) = _CHARACTER_
 MEDIAN(Order) = _CHAR_
 MODE   (Order )= _ALL_
;
RUN ;
```

runs with no problem, and PROC PRINT shows us that BADNAMES looks like this:

```
Obs   _TYPE_   _FREQ_   _NUMERIC_   _CHARACTER_   _CHAR_   _ALL_

 1       0        3         10            30         20       .
```

This is certainly an odd thing to do, but it is permitted. However, when we try to use the data set (BADNAMES) in a DATA step, as in

```
DATA _NULL_ ;
SET badnames ;
RUN ;
```

all we see is

```
ERROR: Cannot use _NUMERIC_ as a variable name.
ERROR: Cannot use _CHAR_ as a variable name.
ERROR: Cannot use _ALL_ as a variable name.
```

That's because these names are reserved, but only in the DATA step.

CONCLUSION

SAS variable lists provide shortcuts that are often handy. The numbered range list construct is arguably the most versatile form of SAS variable list, being able to create as well as reference variables, and being able to override pre-existing variable order.

REFERENCES

SAS Institute Inc. 2010. *SAS® 9.2 Language Reference: Concepts, Second Edition*. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2011. *SAS® 9.2 Language Reference: Dictionary, Fourth Edition*. Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

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

Howard Schreier
Arlington VA
703-979-2720
hs AT howles DOT com
http://www.sascommunity.org/wiki/User:Howles

Contact other readers of this paper via the paper's home page,
http://www.sascommunity.org/wiki/Using_SAS_Variable_Lists_Effectively

[1] The examples in this paper were all produced using SAS 9.2. Shortly before the publication due date, the author was informed by SAS Technical Support that the behavior of the keyword _CHARACTER_ is different in SAS 9.3. Specifically, _CHARACTER_ will produce the same results as _CHAR_ and results analogous to those produced by _NUMERIC_. That should eliminate some anomalies described and illustrated in this paper.