ABSTRACT
The SAS DATA STEP supports quite a few statements and functions. Many of these are usually accompanied with acceptable default values. Due to the nature of SAS programming, however, the function defaults are sometimes overly generous. This is particularly true of length specifications for both the numeric and character variables. With the default of 8 bytes for numeric variables (the equivalent of double-float notation in other languages) and the default length of character variables sometimes as long as 200 bytes, the potential for shortening variable lengths is quite high. Some standard SAS functions and procedures enable you to shorten variable lengths quite easily. Shorter variable lengths mean shorter observation lengths, and shorter observation lengths enable faster I/O processing.

INTRODUCTION
The SAS system encompasses a slew of procedures and elements that can be quite effective when their features are fully utilized and used in combination. The data set options that are available for use with both the DATA STEP’s data and set statements are capable of customizing a data set while being processed by the DATA STEP, and the DATA STEP’s support of array processing allows for the processing of many variables with a relatively small amount of code. Likewise, PROC SQL is capable of generating a significant amount of variable information using a relatively small amount of code. PROC TRANSPOSE provides the means of transforming a horizontally orientated data set into a vertically oriented dataset. Fortunately, the SAS Macro Language provides the means to glue information from the DATA STEP, information from PROC SQL, and the functionality of PROC TRANSPOSE together.

These three components of the SAS system are among the first a novice SAS programmer should start learning once the basic elements of a DATA STEP are mastered. Competency with these elements is essential for the intermediate SAS programmer. Using these elements as a means to optimize the lengths of observation variables in a data set makes for an extremely effective learning exercise and is a good start on intermediate SAS programming.

MACRO VARIABLES ARE USED AS ABBREVIATIONS FOR LISTS OF SPECIFIC VARIABLES
The SAS Macro Language can be considered a programming language itself. It mirrors DATA STEP programming quite closely with one major exception: it is entirely alphanumeric character based. Basic numeric calculations can be made using the %eval() macro function, but that is a rare exception. Macro variables, consequently, are variables that contain strings of characters. When a macro variable stores a number, it stores the alphanumeric representation of the number—not the value of the number. The following macro language statement uses the ‘%let’ statement to assign the numeric string ‘3.14159265’ to the macro variable ‘pi’:

\[
\text{\%let pi = 3.14159265 ;}
\]

The value string can now be referred to with the character string ‘&PI’ (upper or lower case).

With a few exceptions, string delimiters(‘ ‘ and ‘”) are not used as the macro language uses only alphanumeric characters. Many more detailed explanations and uses of the SAS Macro Language are available in several excellent books and papers available on the SAS web site.

As macro variables store strings and the length of macro variables are based on the length of the string values they are assigned, macro variables are extremely well suited for storing variable lists.

PROC SQL AND SAS DICTIONARY TABLES ARE USED TO GENERATE THE REQUIRED VARIABLE LISTS
PROC SQL is SAS’ implementation of the ANSI SQL standard. As such, it supports all of the functionality of the ANSI standard as well as many features of the SAS language. The SAS system also maintains about two dozen dictionary tables which comprise all the metadata relevant to the SAS system as it is implemented in a SAS session. One of the features supported by PROC SQL is the use of macro variables. Consequently, we can use PROC SQL to generate the lists of specific variables we want to optimize.

The following PROC SQL code will generate lists of all character and numeric variables from the CLASS sample data set respectively:
Similarly, we can generate the count of each type of variable:

```sql
PROC SQL is also used to generate a table comprised of variable length metadata:
```
USE ARRAYS TO ACCESS THE LISTED VARIABLES WITHIN A DATA STEP

The array DATA STEP element will allow us access to each variable in a list of variables. Arrays are lists of DATA STEP variables. The variables can be either from data sets or created during the DATA STEP. Although we could use the global _CHARACTER_ and _NUMERIC_ variable lists as abbreviations for the character variable list and numeric variable list respectively, we will instead use the previously-generated specific variable lists. The arrays themselves, however, only exist in the DATA STEP. There are methods to maintain the list of variables in an array beyond a DATA STEP, but detailing those methods are beyond the scope of this paper. The syntax for declaring an array is:

```
ARRAY array-name { subscript } <$><length> <array-elements> <(initial-value-list)>;
```

Our declaration for the character variable array would then be:

```
array char_vars(*) &_CHARACTER_VARIABLES ;   *NO VALUES ASSIGNED ;
```

And our declaration for the numeric variable array would then be:

```
array num_vars(*) &_NUMERIC_VARIABLES ;   *NO VALUES ASSIGNED ;
```

The ‘*’ specifies that SAS will generate the subscript by counting the elements assigned to the array.

CORE CODE FOR GENERATING VALUES FOR USE IN THE SUBSEQUENT OPTIMIZATION OF A DATA SET

Data set optimization requires the knowledge of how much we can change the variables without truncating the values the variables hold. A DATA STEP is used to step through the records and the variables of the data set intended for optimization to first generate and then retain the optimal variable lengths. A distinct data set is generated for a decimal value flag summary, an absolute value summary, and a character length summary. The drop or keep data set option is used with each summary data set specification so that variables that do not belong with the corresponding variables types are not retained in the summary data sets:

```
* GENERATE DATA SETS COMPRised OF MAXIMUM ABSOLUTE NUMERIC VALUES,
  DECIMAL PRECISION FLAGS AND CHARACTER VARIABLE VALUE LENGTHS ;
data
  work._decimal_summary (drop = i &_CHARACTER_VARIABLES)
  work._maximum_abs_values (drop = i &_CHARACTER_VARIABLES)
  work._maximum_char_lengths (keep = &CHAR_LENGTH_VARIABLES)
; 
```

These variables are numeric variables that will contain the maximum length for each character variable.

The end data set option is used on our set statement to specify when we are ready to save the values we will use to subsequently optimize the data set:

```
set &_DS
  end = end_of_ds
; 
```
Once we initialize the input and output data sets, we are ready to initialize the arrays:

```plaintext
* INITIALIZE ARRAYS FOR EACH TYPE OF VARIABLE, LENGTH SUMMARIES FOR
  EACH TYPE OF VARIABLE, AND A FLAG ARRAY FOR NON-INTEGER VALUES ;
array num_vars{*} &_NUMERIC_VARIABLES ;
array char_vars{*} &_CHARACTER_VARIABLES ;
array max_vals{&_NUM_N} _TEMPORARY ;
array max_chars{*} &_CHAR_LENGTH_VARIABLES ;
array dec_vals{&_NUM_N} _TEMPORARY ;
```

We use the `retain` statement to both initialize the values for the listed variables to zero and hold the values they are subsequently assigned from one observation to the next observation. Assigning a value of zero insures we do not finish with a missing value for any of these variables.

```plaintext
retain dec_vals max_vals &_CHAR_LENGTH_VARIABLES 0 ;
```

The `DATA STEP`'s `do` statement will cycle through each variable in each variable array. We can use either the macro variable count values or the `dim()` function to generate the count of elements for each array.

The `abs()` and `max()` functions are used to generate and assign the maximum absolute value for each numeric variable.

```plaintext
* CYCLE THROUGH AND RETAIN THE MAXIMUM ABSOLUTE VALUE AND
  NON-INTEGER FLAGS OF NUMERIC VARIABLES ;
do i = 1 to &_NUM_N ;
  max_vals[i] = max(max_vals[i], abs(num_vars[i])) ;
dec_vals[i] = input(put(max(dec_vals[i], abs((num_vars[i])-int(num_vars[i]))),BOOLEAN.),1.0) ;
end ;
```

The `int()`, `abs()`, and `max()` functions and a BOOLEAN format are used to flag whether a numeric value utilizes decimal precision. SAS utilizes a double float storage scheme for all numeric variables and allocates the maximum of 8 bytes for each numeric variable by default. The SAS programmer has the ability to lower the number of allocated bytes to 2 or 3, depending upon the Operating System. Doing so, however, may result in a significant loss in decimal precision and should be avoided.

A detailed explanation of SAS’ storage of numeric values may be found at [http://support.sas.com/documentation/cdl/en/lrcon/61722/HTML/default/a000695157.htm](http://support.sas.com/documentation/cdl/en/lrcon/61722/HTML/default/a000695157.htm)

The maximum lengths for character values are generated and assigned in a similar manner:

```plaintext
* CYCLE THROUGH AND RETAIN THE LENGTH OF CHARACTER VARIABLES ;
do i = 1 to dim(char_vars) ;
  max_chars[i] = max(max_chars[i], length(strip(char_vars[i]))) ;
end ;
```

Now that we have cycled through all of the numeric variables and all of the character variables for each observation in the original data set (yes, this can be a lengthy process for larger data sets), we are ready to output our summary values to the summary data sets:

The maximum absolute values for each numeric variable are assigned to the original numeric variables. These numeric variables, however, are output to the _maximum_abs_values data set; not the original data set.

```plaintext
if (end_of_ds) then do ;
  * COPY MAXIMUM ABSOLUTE VALUES TO ORIGINAL NUMERIC VARIABLES AND
    OUTPUT SUMMARY RECORD ;
do i = 1 to &_NUM_N ;
  num_vars[i] = max_vals[i] ;
end ;
output work._maximum_abs_values ;
```
The decimal precision flags are output to their own data set in a similar manner.

```sas
* COPY DECIMAL FLAGS TO ORIGINAL NUMERIC VARIABLES AND OUTPUT SUMMARY RECORD ;
  do i = 1 to &_NUM_N ;
    num_vars[i] = dec_vals[i] ;
  end ;
  output work._decimal_summary ;
```

The generation of the `_maximum_char_lengths` data set is less complex.

```sas
* OUTPUT CHARACTER VARIABLE LENGTH SUMMARY RECORD ;
output work._maximum_char_lengths ;
```

We finish our ‘end of data set’ processing and …

```sas
end ;   * if (end_of_ds) ;
```

We are ready to .

```sas
run ;
```

**GENERATING TRANSPOSED DATA SETS COMPRISED OF OPTIMIZED VARIABLE VALUES**

Cloning the original data set and saving only the required length values leaves us with three data sets each comprised of only one record. We need to transpose these data sets so that we have three data sets comprised of an observation for each variable, with each observation containing a variable name character variable and a numeric length variable. The following PROC TRANSPOSE code generates exactly what we need:

```sas
* RECTANGULARIZE THE SUMMARY RECORD DATA SETS ;
proc transpose
  data = work._decimal_summary
  out = work._decimal_flags (rename = (
    _NAME_ = NAME
    COL1 = NON_INT
  )) ;
proc transpose
  data = work._maximum_abs_values
  out = work._rectd_maximum (rename = (
    _NAME_ = NAME
    COL1 = ABS_MAX_VALUE
  )) ;
proc transpose
  data = work._maximum_char_lengths
  out = work._rectd_char_lengths (rename = (
    _NAME_ = NAME
    COL1 = MAX_CHAR_LENGTH
  )) ;
```

5
We previously had to generate new numeric variables that contained the lengths of the character variable values. We needed new names for the new variables, so we merely appended `_l` to the original variable names. Now that we are not working with the original data set we are able to drop the appended string:

```
* CHANGE VALUES OF CHARACTER VARIABLE LENGTH VARIABLE BACK TO
ORIGINAL VARIABLE NAME ;
data
  work._rectd_char_lengths ( index = (name /unique) ) ;
attrib
  name label = 'Variable Name' length = $ 32 format = $CHAR32.
set
  work._rectd_char_lengths ;
  name = substr(name,1,length(strip(name))-%length(&_CHAR_VAR_SUFFIX)) ;
run ;
```

Our PROC TRANSPOSE code did not assign labels to the transposed variables. We also need a better name for our maximum absolute values data set. Let’s use PROC DATASETS!

```
* STANDARDIZE TRANSPOSED DATA SETS ;
proc datasets library = work nolist ;
age _rectd_maximum _maximum_abs_values ;
run ;
modify _decimal_flags ;
  label name = 'NAME OF VARIABLE' ;
modify _maximum_abs_values ;
  label name = 'NAME OF VARIABLE' ;
quit ;
```

We now generate a single table comprised of all the information needed to optimize the original data set by joining the original `variable_lengths` table with the three transposed data sets using `PROC SQL`. Use of the `variable_lengths` table as the first table in a series of left joins assures us of an ordered observation for each variable in the original data set. A where statement excludes the numeric variables with values of decimal precision.

```
* GENERATE DATA SET COMPRISED OF ALL VARIABLE NAMES AND
  OPTIMIZED BYTE VALUES FOR NUMERIC VARIABLES ;
proc sql ;
create table
  work._required_bytes
as select
  lengths.varnum, lengths.name, lengths.label, lengths.type,
  input(put(values.abs_max_value, BYTES_NEEDED.),1.) as REQUIRED_BYTES label =
  'REQUIRED BYTES'
from
  work._variable_lengths lengths
left join work._decimal_flags flagged on
  (lengths.name EQ flagged.name)
left join work._maximum_abs_values values on
  (flagged.name EQ values.name)
where
  flagged.non_int NE 1
order by
  varnum
;
create unique index name on work._required_bytes;
quit; * proc sql;

We subsequently update the _required_bytes data set first with character variable length values and then with the required bytes for numeric values.

* UPDATE REQUIRED BYTES DATA SET WITH THE REQUIRED BYTES FOR CHARACTER VARIABLES;
data
  work._required_bytes (  
    index = (name /unique)  
  )  ;
update
  work._required_bytes
  work._rectd_char_lengths (  
    rename = (max_char_length = required_bytes)  
  )
by
  name
;  
if (missing(required_bytes)) then
  error 'ERROR: NO REQUIRED BYTES: ' name= ;
run;

* UPDATE THE VARIABLE LENGTHS DATA SET WITH THE REQUIRED BYTES DATA SET;
data
  work._variable_lengths ;
update
  work._variable_lengths
  work._required_bytes (  
    rename = (required_bytes = length)  
  )
by
  name
;  
proc sort ;
by varnum ;
run;

Finally! All the information we need to generate an optimally-sized data set in a second data set. All we need to do now is to use this information in a final DATA STEP. We will do this by first generating a valid length statement argument for each variable and then export the valid argument to an indexed macro variable (remember: macro variable values are comprised of strings) Here, the strings will be valid length statement arguments.

* EXPORT VARIABLE NAMES AND REQUIRED BYTES TO INDEXED MACRO VARIABLES;
data
  _NULL_ ;
attrib
  bytes          label = 'Bytes Required String'     length = $ 5
  length_str     label = 'Length Statement String'   length = $ 64
;  
set
  work._variable_lengths
  end = end_of_ds
;  
bytes = strip(put(length,5.0)) ;
CONCLUSION

The previous snippets of code are from a Macro program that not only optimizes a data set but also reports on the relative results of the optimization (percentages, etc.). The snippets, however, demonstrate the use of SAS system procedures and elements that can be among the first used by a novice SAS programmer.

We used the data set options `keep` and `drop` in the DATA STEP’s `data` and `set` statements to customize the composition of the data set within the set statement, and we used the `set` statement’s `end` option to delimit DATA STEP code we did not want executed until all of the data set’s observations were read.

We used PROC SQL to generate macro variables comprised of character variable and numeric variable lists.

We used arrays of variables to assign values generated by several different DATA STEP functions to values of another array of variables.

We used PROC TRANSPOSE to transpose horizontally oriented informational data sets into vertically oriented datasets.

We used PROC DATASETS to prepare three informational datasets for subsequent merging.

And, going full circle, we used PROC SQL to first merge the three informational datasets and then used macro code within a DATA STEP’s length statement to generate a data set comprised of observations whose variables were of an optimal length.

The author hopes this paper will encourage the novice SAS programmer to quickly move beyond their comfort zone and discover, on their own terms and in their own manner, the benefits and advantages of using elements of the SAS system both together and to their fullest extent.
REFERENCES

  http://www2.sas.com/proceedings/sugi30/031-30.pdf
  http://www2.sas.com/proceedings/sugi30/070-30.pdf

ACKNOWLEDGMENTS

I would like to thank the SAS users in the Survey Research Operations section of the Survey Research Center at the Institute for Social Research for the initial presentation of this paper. I would especially like to thank Frost Hubbard for his encouragement and Megan Turf for her valuable feedback.

RECOMMENDED READING


Varney, Brian (1999), “Creating Data Driven Programs with the Macro Language,” *Proceedings of the 24th Annual SAS Users Group Conference*


CONTACT INFORMATION

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

Name: Philip A. Wright
Enterprise: Inter-university Consortium for Political and Social Research (ICPSR),
The Institute for Social Research (ISR),
University of Michigan
Address: P.O. Box 1248
City, State ZIP: Ann Arbor, Michigan 48106-1248
Work Phone: 734-615-7886
Fax: 734-647-8200
E-mail: pawright@umich.edu
Web: http://www.icpsr.umich.edu

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.