

SAS®'s Various Varying Variables, or 1000+ Ways to Manipulate SAS Variables

Paul A. Choate, California State Developmental Services, Sacramento, CA

ABSTRACT

This presentation is an overview of SAS data set variables in two parts:

PART 1: VARIABLE ATTRIBUTES

- Definition of Variable Attributes and the PDV
- Changing Variable Attributes with Data Step Statements
- Indexing Variables, Changing Variable Types, and Reordering the PDV
- Changing Variable Attributes with Procedures
- Variable Information Functions and Call Routines

PART 2: VARIABLE CONTENTS AND HANDLING

- Tokens, Name Literals, and Special Constants
- Number of Variables and Variable Size Limits
- Variable Lists in Functions, Statements, Options, and Arrays
- Variable Comparisons

INTRODUCTION

There are many ways to process and manipulate variables in a SAS data set. The SAS data step and base SAS procedures are designed for the efficient manipulation of variables in data sets, but to use them it is crucial to have a solid foundation in the what, when, why, where, and how's of SAS variables. This paper surveys SAS variable attributes and variable handling, from well known to lesser known topics, with an emphasis on practical application. Although the data step is the main focus, related procedures such as SQL and DATASETS are discussed, as well as data set options available to all Base SAS procedures. Unless otherwise stated this paper refers to SAS v9.13 on Windows XP platform.

PART 1: VARIABLE ATTRIBUTES

DEFINITION OF VARIABLE ATTRIBUTES AND THE PDV

Variables are containers used to store and process character and numeric values. Variables have attributes, such as name and type, which identify them and define their qualities. Character variables contain alphabetic characters, numeric digits 0 through 9, and other special characters. Numeric variables contain numeric values stored as floating-point numbers, including dates and times. Other SAS variables, including macro and SCL variables, have corresponding characteristics but are not covered here. Transcoding and National Language Support are also not discussed.

VARIABLE ATTRIBUTES

Data step variables have eight attributes: Name, Type, Length, Format, Informat, Label, Position, and Index type.

Variable Attribute	Possible Values	Default
Name	Up to 32 characters, first character must be a letter or underscore (A, a, B, ..., Z, z, _). Other characters can be mixed case letters, numeric digits (0, 1, ..., 9), or underscores. Processed as uppercase.	External data without headers: F1...Fn or Var1...Varn Proc transpose: Col1, Col2, Coln
Type	Numeric or character	Numeric
Length	Character: 1 to 32,767 bytes Numeric: 2 or 3 to 8 bytes	Character: 8 bytes Numeric: 8 bytes
Format	Print characters or machine form	BEST12. for numeric, \$w. for character
Informat	Character or numeric	w.d for numeric, \$w. for character
Label	Up to 256 characters	None
Position in PDV	1- n	None
Index type	None, simple, composite, or both	None

Important note: If not explicitly defined, a variable's *type* and *length* are implicitly defined by its first occurrence in a data step.

PROGRAM DATA VECTOR (PDV)

The program data vector (PDV) is the logical memory area where SAS builds data set observations during processing. It is useful to consider the action of the PDV while writing or debugging code. At compilation the variable descriptor portion is built. As a program executes, SAS reads or creates data values. The values are stored during processing in the PDV. With each iteration, SAS writes the resulting observation to SAS output data sets defined in the data statement. Variable attributes are written to the output datasets from the PDV for all variables remaining after all KEEP and DROP statement are processed.

```
data class;
    set sashelp.class(obs=1); *write an observation via the PDV;
run;
```

Descriptor attributes are established when the data step is initially compiled, implicitly by order of appearance in the data step, and explicitly by data step statements and data set options. The DATASETS and SQL procedures also can change variable attributes.

The Program Data Vector								
	Attributes	User Variables					Automatic Variables	
Descriptor	Name	Name	Sex	Age	Height	Weight	_ERROR_	_N_
	Type	char	char	num	num	num	num	num
	Length	8	1	8	8	8	8	8
	Format	\$8	\$1	best12.	best12.	best12.	best12.	best12.
	Informat	\$8	\$1	12	12	12	12	12
	Label							
	Position	1	2	3	4	5	6	7
	Index type	none	none	none	none	none	none	none
Flags	Drop flag	no	no	no	no	no	yes	yes
	Retain flag	yes	yes	yes	yes	yes	yes	yes
	Values	Alfred	M	14	69	112.5	0	1

The PDV contains at least two temporary automatic variables, `_N_` and `_ERROR_`. Variable `_N_` counts the number of data step iterations and the automatic `_ERROR_` variable flags any errors during execution. Other temporary variables are statement and data set options such as `_INFILE_`, `NOBS=`, `END=`, `IN=`, `FIRST`, etc. Temporary variables are not output.

CHANGING VARIABLE ATTRIBUTES WITH DATA STEP STATEMENTS

For five out of the eight variable attributes there are data step statements to define or change that attribute: RENAME, LENGTH, FORMAT, INFORMAT, LABEL, and ATTRIB. Unfortunately the rules are different for the different attribute assignment statements, and positioning of the statement is significant for all except RENAME. The only attribute that cannot change in place during data step processing is a variable's type, although in effect the variable type can be changed during the data step. Changing the type, changing the position in the PDV, and indexing are discussed in following sections.

RENAME STATEMENT AND OPTION

To efficiently rename a variable in the data step use the RENAME statement or the RENAME data set option. Data set options are available in the data step statements DATA, SET, MERGE, MODIFY, or UPDATE, or on data set input, or on output of SAS procedures, including notably PROC SQL. RENAME is unique among attribute statements because it is also available as a data set option. For efficiency, do not rename variables with an assignment to a new variable and DROP of the old variable. The RENAME statement has the same effect regardless of position in the data step and will not cause a variable to change order in the PDV.

```
data class(rename=(Sex=Gender));          *renamed after processing;
    set sashelp.class(rename=(Age=Age_Yrs)); *renamed before processing;
    Rename Name=First_Name;              *renamed after processing;
run;
data "class" (drop=Age);                  *quoted writes to default folder;
    set sashelp.class;
    Age_Yrs=Age;                          *don't do this;
run;
data class;
    Rename Name=First_Name;                *OK - not affected by subsequent SET statement;
    set sashelp.class;
```

```
run;
```

LENGTH STATEMENT

To change a variable's length use a LENGTH statement before the variable is defined by the data step. Position of the LENGTH statement impacts the position of the variable in the PDV.

```
data class;
  length sex $6;           *defined as $6 and first variable in PDV;
  set sashelp.class;      *length changed from $1 to $6;
  if sex='M' then sex='Male';
                        else sex='Female';
run;
data class;
  set sashelp.class;      *sex defined as $1 and same position as before;
  length sex $6;         *length already set & doesn't change;
  if sex='M' then sex='Male';
                        else sex='Female';
run;
```

FORMAT AND INFORMAT STATEMENTS

To change or initialize a variable's output format or input informat use a FORMAT or INFORMAT statement anywhere in the data step, but before they are needed by put or input statements. If a FORMAT or INFORMAT is used before a variable is assigned with an assignment statement, then the FORMAT or INFORMAT sets the length of the variable. If either is the first statement to reference a variable the statement will effect the variable's position in the PDV.

To clear a pre-existing format, an empty FORMAT or INFORMAT statement must be subsequent to any SET, MERGE, or other statement defining the attribute. Alternatively, a FORMAT or INFORMAT statement explicitly defining the default before the set statement may be used. This behavior is also unique among the attribute statements.

Also note that while formats and informats may be used in PUT and INPUT statements and functions this usage does not change the format and informat variable attributes. When data are read to or from DBMS sources the external formats and compatible SAS formats are exchanged.

FORMAT statements are allowed in procedures, affecting the procedure output and the output data set. The FORMAT statement does not permanently alter the variables in the input data set.

```
data testin1;
  input tdate yymmdd.; *tdate read but default informat entered in PDV;
datalines;
20061003
;
data testin2;
  informat tdate yymmdd.;
  input tdate;        *same as above but informat yymmdd6.;
datalines;
20061003
;                               *no run needed - implied step boundary;
data testin3;
  set testin2;
  informat tdate;    *clears the incoming PDV informat;
run;
data testin3;
  informat tdate;   *doesn't clear the incoming informat, remains yymmdd.;
  set testin2;
run;
data testin4;
  input tdate;      *incorrectly read as day 20,061,003;
  informat tdate yymmdd.;
datalines;
20061003
;
```

Another important difference between how character FORMAT and INFORMAT statements act is that FORMAT statements set both the format and informat PDV descriptor values whereas INFORMAT does not always set the descriptor format. If an INFORMAT statement follows an assignment statement it only resets the informat. For FORMAT statements with formats that do not have equivalent informats, the informat of the variable inherits only the length from the format.

```
data class;
  set sashelp.class; *variable "name" has length $8;
  format name2 $2.; *name2 is length $2;
  name2=name; *both format and informat of name2 are $2;
run;
data class;
  set sashelp.class;
  informat name2 $2.; *name2 is length $2;
  name2=name; *both format and informat are $2;
run;
data class;
  set sashelp.class;
  name2=name; *both format and informat of name2 are $2;
  format name2 $2.; *name2 is length $8;
run;
data class;
  set sashelp.class;
  name2=name; *only informat is $2;
  informat name2 $2.; *name2 is length $8;
run;
```

LABEL STATEMENT

The LABEL statement works like the RENAME statement in action but is different in effect on PDV order. A variable is given the new label regardless of the statement's position in the data step, but the position of the variable in the output data set will be moved to the beginning if the statement preceded other statements. LABEL is not available as a data set option, although *data set* labels are defined via a data set option. Like the FORMAT statement, LABEL statements are allowed in procedures, affecting the procedure output and the output data set. The LABEL statement does not permanently alter the variables in the input data set.

```
data class;
  set sashelp.class;
  label age='Age of Student'; *new label w/ original position;
run;
data class;
  label age='Years of Age'; *new label w/ new position;
  set class;
run;
```

ATTRIB STATEMENT

The ATTRIB statement is a multifunction descriptor attribute modifier, allowing definition of lengths, formats, informats, and/or labels, but not allowing renaming. Position of the ATTRIB statement affects the corresponding variable descriptor options as does the positioning of the individual attribute statements.

Like the FORMAT and LABEL statements, ATTRIB statements are allowed in procedures, affecting the procedure output and the output data set. The procedure ATTRIB statement does not permanently alter the variables in the input data set. The LENGTH option has no effect in a procedure ATTRIB statement. In the following first example class1 both the label and length of Sex are set, but in the second class2 example only the label is set:

```
data class1;
  attrib sex label ='Gender'
           length =$6;
  set sashelp.class;
run;
data class2;
  set sashelp.class;
  attrib sex label ='Gender'
           length =$6;
run;
```

In this example the ATTRIB FORMAT statement is used to truncate the name value to the first initial for processing:

```
proc means data=sashelp.class(obs=11);
  class name;
  var height;
  attrib name    label  ='Initial'
           Format  = $1.
           height label  ='(in inches)';
run;
```

The MEANS Procedure
Analysis Variable : Height (in inches)

Initial	N Obs	N	Mean	Std Dev	Minimum	Maximum
A	2	2	62.7500000	8.8388348	56.5000000	69.0000000
B	1	1	65.3000000	.	65.3000000	65.3000000
C	1	1	62.8000000	.	62.8000000	62.8000000
H	1	1	63.5000000	.	63.5000000	63.5000000
J	6	6	58.7333333	4.1687728	51.3000000	62.5000000

INDEXING VARIABLES, CHANGING VARIABLE TYPES, AND REORDERING THE PDV

INDEXING VARIABLES

Indexing permits efficient lookups and by-group processing on multiple primary keys without resorting, at the cost of I/O, maintenance, and storage of a secondary index file. One method to index variables is using the output data set option.

```
data class(index=(sex agetname=(age name))); *simple and composite indexes;
  set sashelp.class;
run;
```

Indexing is also possible with the SQL and DATASETS procedures. SQL can create an index during the output of a data set as can other procedures, but SQL is uniquely similar to DATASETS in that it can also build an index on a preexisting data set, without otherwise processing the data.

```
proc sql;
  create TABLE class(index=(sex agetname=(age name))) as
  select *
  from sashelp.class;
quit;
```

```
proc sql;
  create TABLE class as
  select *
  from sashelp.class;
  create INDEX sex ON class (sex); *build index on pre-existing data set;
  create INDEX agetname ON class (age,name); *build composite index;
quit;
```

```
data class;
  set sashelp.class;
run;
```

```
proc datasets library=work nolist;
  modify class;
  index create age; *build index on pre-existing data set;
quit;
```

CHANGING VARIABLE TYPE

Once established, a variable's type attribute cannot be changed with an attribute statement. Fortunately, with a little sleight-of-hand, the type can be changed by renaming the source and writing back to the original name as the different type.

```
data class(drop=_Age);
  retain Name Sex Age Height Weight;      *RETAIN - no effect except PDV order;
  set sashelp.class(rename=(Age=_Age));    *swap out variable;
  Age=put(_Age,z2.);                       *create new type;
run;
```

Note RETAIN has no effect on variables read in a SET statement, except to establish the order in the PDV. The variable list in the RETAIN statement could be obtained from the data set's metadata, and so the variable order would not need to be explicitly coded. More on this below in the section on PROC SQL and data set metadata.

REORDERING THE PDV WITH RETAIN

The PDV is constructed when the data step is initially compiled, with each variable positioned in the PDV based on the order encountered during compilation. The PDV variable order is affected by the appearance of statements that read SAS data sets, variable assignment statements using operations, functions and call routines, and certain statements such as LENGTH and RETAIN. The RETAIN statement is unique among these statements in that it only affects a specific subset of variables and for those variables its action is easily counteracted.

The RETAIN statement is redundant for:

- variables defined by SET, MERGE, MODIFY or UPDATE statements,
- variables defined by statement options in SET, MERGE, MODIFY, UPDATE, FILE and INFILE statements,
- variables in sum statements,
- initialized array statement variables.

Or in other words, most variables in a typical data step. The RETAIN statement only changes whether the data step sets to missing values that are *created* in the data step with an INPUT or assignment statement. Thus to safely reorder the PDV of a data set for the above redundant variable types, a RETAIN statement may be used as the first statement of a data step. This will not change any part of the PDV descriptor other than its order because the RETAIN has no other effect. Input or assigned variables may also be reordered this way, and if the retain action is unwanted for those variables they may be programmatically set to missing at the start of each iteration.

```
data class;
  retain Age Height NewVar Weight Name Sex;
  NewVar=.; *set to missing to negate RETAIN;
  set sashelp.class;
  ... calculate NewVar ...
run;
```

When is reordering this useful? When data sets are merged, when new variables have been added, when exporting data, and situations where named range list processing is desired.

CHANGING VARIABLE ATTRIBUTES WITH PROCEDURES

ATTRIBUTE STATEMENTS SUMMARY

Statement	Add New Variable?	<i>What Attribute Statements Set or Change in PDV</i>						
		Rename	Type	Length	Format	Informat	Label	Position
RENAME	no	yes	no	no	no	no	no	No
LENGTH \$	yes	no	yes	yes	yes	yes	no	Yes
LENGTH n	yes	no	yes	yes	no	no	no	Yes
FORMAT \$	yes	no	yes	yes	yes	yes	no	Yes
FORMAT n	yes	no	yes	no	yes	yes	no	Yes
INFORMAT \$	yes	no	yes	yes	yes	yes	no	Yes
INFORMAT n	yes	no	yes	no	no	yes	no	Yes
LABEL	no	no	no	no	no	no	yes	Yes

The five attribute statements and their interactions might be summarized in the above table. Is this confusing? As an alternative to data step-processing of data sets to change variable properties, SAS provides three procedures to report and change variable descriptor values: DATASETS, CONTENTS and SQL.

PROC DATASETS

Aside from attribute definition statements in the data step, another approach to changing variable descriptors is through the DATASETS procedure. One important advantage is efficiency, for attributes other than variable index DATASETS only processes the header portion of the SAS data set, while in the data step the full data set is processed.

Another major advantage is that variable formats, informats, labels, and names may be directly changed without consideration of the interaction of the location of the statements as in the data step. PROC DATASETS processes each attribute defining statement completely before any subsequent statement is processed. For example, if a variable is renamed in one line then the new name is used thereafter in the same procedure.

PROC DATASETS will also build indexes on variables. In this case DATASETS processes the entire file.

```
data class;
  set sashelp.class;
run;
proc datasets library=work nolist;
  modify class;
  format height 5.2;
  informat height 6.3;
  index create age;           *causes full file to be processed;
  rename age=AgeYrs;         *also renames index;
  label AgeYrs='Years of Age';
quit;
```

PROC CONTENTS

While PROC CONTENTS will not change the variable descriptor information, it will report the descriptor settings to a file or print destination. It is also efficient, only reading the file header. CONTENTS is useful for examining foreign data sets and for reference in archival and other programming tasks. PROC DATASETS has a CONTENTS statement which produces output identical to PROC CONTENTS. Also, in some environments PROC CONTENTS will work where PROC DATASETS cannot; PROC CONTENTS can read sequential files such as from tape or an FTP server.

PROC SQL COLUMN-MODIFIERS

As noted above, SQL can index a variable on data set output or in-place on an existing data set. When SQL creates a table the usual data set options of RENAME, DROP, and KEEP are also available. PROC SQL also provides the ability to change other SAS variable descriptors on the SELECT statement through the use of FORMAT, INFORMAT, LABEL, and LENGTH column-modifiers. The SELECT statement AS operator allows a variable to be copied to a new variable with a different descriptor, similar to the function of an assignment statement in the data step. SQL column-modifiers are analogous to data step statements like ATTRIB. SQL does not modify descriptor values in place as does DATASETS. See above for variable index creation in SQL.

```
proc sql; *change Sex to Gender and modify descriptor;
  create TABLE class as
  select Name,
         Sex as Gender length=6 format=$6. label='Student Gender',
         Age, Height, Weight
  from sashelp.class;
quit;
proc sql; *SELECT AS has same effect as RENAME;
  create TABLE class as
  select Name,
         Gender length=6 format=$6. label='Student Gender',
         Age, Height, Weight
  from sashelp.class(rename=(Sex=Gender));
quit;
```

PROC SQL METDATA TABLES

The SQL procedure gives access to all the descriptor information from any defined library. The SQL table DICTIONARY.COLUMNS and its equivalent but slower twin, the data step view SASHELP.VCOLUMN, contain all variable

definitions for all defined libraries. To make things even better, PROC SQL's SELECT statement has an INTO: clause that loads data set values into a macro variable for subsequent processing. In combination DICTIONARY.COLUMNS and SELECT INTO: provide a very powerful window into data set metadata such as variable descriptor values.

From the earlier example of changing a variable's type, here is the same solution without the foreknowledge of anything but the data set name and library and the name of the variable to be changed:

```
proc sql noprint;                                *get the names from metadata;
  select name
  into: varlist separated by " "
  from dictionary.columns
  where libname = 'SASHELP' and
         memname = 'CLASS';
quit;
data class(drop=_Age);
  retain &varlist;                                *retain positions;
  set sashelp.class(rename=(Age=_Age));          *swap out variable;
  Age=put(_Age,z2.);                              *create new type;
run;
```

VARIABLE INFORMATION FUNCTIONS AND CALL ROUTINES

With v9 SAS now allows data step testing of variable attributes. The variable information functions include functions to read formats, labels, lengths, names, types, and formatted values of variables. The functions reference variable descriptor data based on either the variable's name, a text function resolving to a variable name, or the position of the variable in an array. See the documentation for the full "v" function list.

The function VLENGTH returns the variable length from a lookup based on the variable's name.

```
data _null_;
  if 0 then set sashelp.class;                    *don't read data set - load PDV only;
  vlen=vlength(name);
  put "length of name is " vlen;
  stop;
run;
```

length of name is 8

The call routine VNEXT returns the variable names, type and length based on position in the PDV.

```
data _null_;
  if 0 then set sashelp.class;
  length var_name $32 var_type $3;
  do until(var_name=' ');                        *VNEXT reads PDV left to right;
    call vnext(var_name,var_type,var_length);
    if not missing(var_name) then
      put @1 var_name= @21 var_type= @36 var_length= ;
  end;
  stop;
run;
```

```
var_name=Name      var_type=C      var_length=8
var_name=Sex       var_type=C      var_length=1
var_name=Age       var_type=N      var_length=8
var_name=Height    var_type=N      var_length=8
var_name=Weight    var_type=N      var_length=8
var_name=var_name  var_type=C      var_length=32
var_name=var_type  var_type=C      var_length=3
var_name=var_length var_type=N      var_length=8
var_name=_ERROR_   var_type=N      var_length=8
var_name=_N_       var_type=N      var_length=8
```

PART 2: VARIABLE CONTENTS AND HANDLING

TOKENS, NAME LITERALS, AND SPECIAL CONSTANTS

TOKENS

When SAS compiles a data step or procedure it must “read” the language instructions, including variable names and character constants. To do this SAS breaks the written code into *tokens*. All SAS programming language is broken into four types of tokens: names, literals, numbers, special characters. In general, *literals* are quoted strings and *numbers* are unquoted numeric strings that may contain special characters.

Names are in most cases unquoted character strings including underscores and digits, but not beginning with a number. SAS name tokens include variable names, data set names, array names, librefs, filerefs, statement names, functions names, and procedure names. When variable names are programmed in SAS they need to be distinguishable as variable names and as different from other name tokens. Name token types each have special requirements; for example procedure name tokens can be up to 32 letters long, but only the first 8 must be unique. Various parsing rule apply, such as multiple blanks and line breaks aren’t read, so embedding blanks and line breaks does not affect processing. With the exception of name literals, names always consist of letters, the underscore, and digits.

NAME LITERALS

SAS names are restricted to at most 32 characters, numbers, and underscores beginning with an alpha or underscore and without white-space. SAS *labels* sidestep these name limitations by allowing up to 256 printing characters and spaces. SAS has labels for variables and datasets, and statement labels for GOTO and LINK statement references.

Most current databases allow special characters and spaces directly in variable and table names. To allow SAS to work natively with such sources SAS *name literals* are available. A name of a variable or data set may be written as a name literal in the form ‘*Special_Name*’*n* using the standard letters, underscores, and digits.

Setting the VALIDVARNAME=ANY system option allows variable name literals to contain special characters. This works in the SAS data step and Base and Stat procedures, including the SQL procedure. DBMS data sets defined with a libname engine may also be named with special characters and spaces, *without* the special VALIDVARNAME setting. In all cases the usual macro quoting rules apply. The default value of VALIDVARNAME is v7.

The following code creates an MS Access table named “new class data” with the “name” column renamed as “name with watch-out” and the “age” column renamed as “age without ¯ovar”.

```
options validvarname=any;
libname msacc 'Data.mdb';
%let macrovar=watch-out;
data msacc.'new class data' n;
    set sashelp.class;
    rename name="name with &macrovar" n * "name with watch-out" ;
        age ='age without &macrovar' n; * "age without &macrovar" ;
run;
libname msacc clear;
options validvarname=v7;
```

Name literals are primarily useful when connecting to external databases. As an alternative to using special name literals, using options DBLABEL and DBSASLABEL SAS can read and write SAS *labels* as DBMS column headers rather than variable names. DBLABEL writes SAS labels as column headers and DBSASLABEL reads column headers as SAS labels. They are available as data set options or libname options for MS Excel, MS Access, DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata, and SAP.

SPECIAL CHARACTER CONSTANTS

SAS character constants are quoted strings of up to 32,767 characters used to assign values to character variables and other data step tasks. Example special character constants:

<i>Notation</i>	<i>Value</i>	<i>Special Constant</i>
Embedded Quotes	We'll	“We'll” or ‘We’ll’
Hexadecimal	We'll	'5765276C6C'x

The SAS processor automatically converts character constants to their alphanumeric value. In this example SAS evaluates both strings as the value "We'll" and returns a true value of 1.

```
equalstrings = ('We''ll' = '57,65,27,6C,6C'x);
```

Note the optional use of commas in hexadecimal characters.

SPECIAL NUMERIC CONSTANTS

SAS numeric constants are expressed with signed or unsigned numbers, with or without decimal points. Special constants are available for commonly used scientific and hexadecimal notation, as well as the quoted SAS date, time, and datetime notation.

<i>Notation</i>	<i>Value</i>	<i>Special Constant</i>
Scientific	-32.5	-3.25e1
Hexadecimal	23	17x
Dates	0	'01Jan1960'd
Time	0	'12:00:00am't
Datetime	0	'01Jan1960:12:00:00am'dt

The SAS processor also automatically converts numeric constants to their numeric value. In this example SAS adds 23, -0.3, and 100 to return a value of 122.7:

```
numvar = 17x + -3.e-1 + '12:01:40am't;
```

NUMBER OF VARIABLES AND VARIABLE SIZE LIMITS

NUMBER OF VARIABLES

Starting with SAS 9.1, the maximum number of variables is now greater than 32,767 and is dependent on environmental and file attributes. For example the code below creates a data set with one observation of 1,000,000 variables. Note that this is a poor design, the descriptor overhead makes this file several time larger than an equivalent file of one variable with one million rows.

```
data wide(drop=i);
  array _[1000000] 8; *variables named _1, _2, _3, ..., _1000000;
  do i = 1 to 1e6;
    _(i)=ranuni(0);
  end;
run ;
```

SIZE OF NUMERIC VARIABLES

By default numbers are stored in the maximum allowable eight bytes. The largest exact integer is found with the function CONSTANT('EXACTINT' <, nbytes>). On XP EXACTINT for eight bytes is 9,007,199,254,740,992. Larger integers may be used, but past this value more and more rounding occurs in processing and storage.

The largest double-precision integer is found with the function CONSTANT('BIG'). This returns the largest double-precision floating-point number (8-bytes) representable by SAS. For example, in scientific notation BIG=1.7976931348623E308 on an XP, or approximately 180*(1 billion**38). The actual largest integer in SAS is 1.79769313486231580849E308 minus 1. Numbers larger than this number will return an error message.

Small numbers, decimal values close to zero, have slightly less maximum precision because the negative exponent value takes storage space. The number closest to zero but not zero allowed is 2.2250738585072E-308. Large negative integers reach magnitudes equivalent to large positive values, around -1.8E308.

Numeric categorical variables, such as Likert scores, ZIP codes, and ID numbers can be stored as either numeric or character type variables. Space and I/O considerations might be significant. For large integral values of 15 digits, storing as eight byte numeric saves nearly twice as much space as 15 byte character, and will be precise up through EXACTINT8. For a single digit scale of 0-9, one byte of character storage is eight times smaller than the default 8 byte numeric, and three times smaller than the smallest three byte numeric storage.

Note that SAS date values may be safely stored in fewer than 8 bytes.

```
data _null_;
```

```

x=constant('exactint',4);
put 'largest exact 4 byte integer= ' x /
   'works as a date= ' x date9. /
   'or as a time= ' x datetime20. /;

x=constant('exactint',6);
put 'largest exact 6 byte integer= ' x /
   'works as a datetime= ' x datetime20. /;

run;

```

```

largest exact 4 byte integer= 2097152
works as a date= 23OCT7701
or as a time= 25JAN1960:06:32:32

```

```

largest exact 6 byte integer= 137438953472
works as a datetime= 09APR6315:15:04:32

```

Numeric variables with descriptor attribute length less than 8 are physically stored in less than eight bytes, with corresponding less precision, but during processing are assigned the default precision of eight bytes.

Note that since numbers are stored and processed as function of powers of the number two, most rational numbers are not stored precisely. For example the number 1/3 is stored in approximate form. During processing SAS “fuzzes” numbers so expected results are usually returned, e.g. $x=1/3+1/3+1/3$ returns 1.

SIZE OF CHARACTER VARIABLES AND SPECIAL HEXADECIMAL CHARACTERS

Character variables are strings of up to 32,767 characters, or roughly *ten single spaced typed pages* of information. Character values are stored the same as they are processed, except that SAS trims trailing blanks. Trailing blanks are implied by the variable length. Character values may include non-printing characters such as tabs or line control characters. For example, on ASCII systems blanks are represented by hex value '20x'. Non-printable characters can be manipulated using hexadecimal literals and formats, or functions such as RANK. The \$CHARw. informat preserves leading blanks.

Special characters embedded in character strings may cause problems if written to text or other external files. Some common examples are: carriage return line feeds ('0A0D'x) may cause unintended line breaks, tabs ('09'x) cause pointers to shift cells in spreadsheets, and end-of-file markers (ctrl-z or '1A'x) may cause a file to truncate.

VARIABLE LISTS IN FUNCTIONS, STATEMENTS, OPTIONS, AND ARRAYS

VARIABLE LISTS

SAS allows a list of related columns to be referred by shorthand notation rather than specifying every variable.

Variable List Type	Notation	Meaning
Standard List	VarA, VarB, VarC, ..., VarZ VarA VarB VarC ... VarZ	Simple comma or space delimited lists.
Special ALL	<u>_all_</u>	All variables.
Name Range	VarA--VarZ	Variables from VarA through VarZ, using the variable positions in the PDV.
Special Numeric	<u>_numeric_</u>	Numeric variables.
Numeric Name Range	VarA -numeric- VarZ	Numeric variables from VarA through VarZ, using the variable positions in the PDV.
Special Character	<u>_character_</u> or <u>_char_</u>	Character variables.
Character Name Range	VarA -character- VarZ or VarA -char- VarZ	Character variables from VarA through VarZ, using the variable positions in the PDV.
Numbered Range	Var1-VarN	Variables Var1 through VarN. The numbers must be consecutive.
Name Prefix	Var:	Variables beginning with the letters Var.

Only *non-automatic* variables are included in variable list references. Also note that as variables become available or values change during a data step the variables and contents of a list may change.

LISTS IN FUNCTIONS, STATEMENTS, OPTIONS, AND ARRAYS

Lists are useful wherever variables are specified in SAS.

FUNCTION EXAMPLES:

Numeric functions allowing lists require an OF modifier to precede the list.

```
data lists;
  set sashelp.class;
  sum=sum(of _numeric_, 5000);      *note the "of" operator;
  length cat $10;
  cat=cat(of _character_);
run;
```

Both statements in the data step and procedures allow list operators. The rename statement and rename data set option only accept numbered ranges.

STATEMENT EXAMPLES:

```
data list;      *note input's special list syntax "(" ;
  input date yymmdd10. +1 (var1-var3) (2.,+1);
  rename date=Day
         var1-var3=Item1-Item3; *rename only accepts numbered range;
cards;
2006-10-03 56 86 23
1961-10-03 93 48 56
;
proc freq data=sashelp.class;
  tables _character_;
run;
proc sort data=sashelp.class out=class;
  by name-char-height;
run;
```

OPTION EXAMPLES:

```
data lists(keep=_:); *keeps only the two renamed variables;
  set sashelp.class(keep=_numeric_);
  rename weight=_weight
         height=_height;
run;
```

The name prefix (colon) list style is useful for keeping track of processing variables such as do loop incremental values, flag variables and other dummy variables used in the data step but not needed in the output data set. Prefix all such variables with an underscore or other unique prefix and then use a name prefix style list in a drop statement.

ARRAY EXAMPLES:

```
data class(drop=_:);
  set sashelp.class;
  array nums _numeric_;
  array chars $ _char_;

  do _i = 1 to dim(nums);
    if nums(_i)=0 then nums(_i)=.;
  end;

  do _j = 1 to dim(chars);
    if anydigit(chars(_j)) then chars(_j)='';
  end;
run;
```

Arrays only allow `_all_`, `_numeric_`, and `_character_` list variable lists. If `_all_` is used all variables must be of the same type. The ARRAY statement also allows a special list `_TEMPORARY_` for a temporary array stored in memory.

VARIABLE COMPARISONS

In the data step variable values are often tested for equality. SAS automatically trims trailing blanks from compared character values, otherwise the strings must match byte for byte. The following returns the observation with “Janet” despite that variable Name is length \$8 and the string is \$6.

```
data class;
  set sashelp.class;
  if name = 'Janet';
run;
```

What if all names starting with “J” are desired? For this case SAS offers the often overlooked colon comparison operator.

```
data class;
  set sashelp.class;
  if Name =: 'J'; *any name starting with "J";
run;
```

The colon operator causes the comparison to be based on the length of the shorter of the two strings compared, in this case the one byte “J” string. The colon operator works with any comparison operator: EQ:, NE:, GT:, LT:, GE:, LE:, or IN: or their symbolic equivalents such as ^=: for NE:.. SQL offers an equivalent set of operators called truncation operators, where the colon is replaced by the letter “t” representing the abbreviation of *truncation*.

```
proc sql;
  create TABLE class as
  select *
  from sashelp.class
  where name EQT "J"; *any name starting with "J";
quit;
```

The data step “in” operator also allows a special numeric integer list where the integers 1, 2, 3 can be expressed 1:3.

```
data class;
  set sashelp.class;
  if age in (13:16); *test for digits 13, 14, 15, 16;
run;
```

Unfortunately, WHERE clauses and PROC SQL do not allow this integer list syntax.

When searching a list of possible values for a match to a variable or literal, usually a PROC FORMAT is more efficient than an IN list comparison, but a string search using the index function may even be faster:

```
data test; *one million three digit strings;
  do i = 1 to 1e6; drop i;
    num=put(ranuni(0)*1000,z3.);
    output;
  end;
run;
data subset; *speed: 1.46 cpu 1.45 real;
  set test;
  if num in ('003' '111' '298' '357' '461'
            '555' '610' '743' '812' '962');
run;
proc format;
value $num '003' '111' '298' '357' '461'
           '555' '610' '743' '812' '962'='Y';
run;
data subset; *speed: 0.59 cpu 0.62 real;
  set test;
  if put(num,$num.)='Y';
run;
```

```
data subset;      *speed: 0.43 cpu  0.43 real;
  set test;
  if index('003 111 298 357 461 555 610 743 812 962',num);
run;
```

SAS offers a wide variety of string functions for parsing, translating, compressing, and comparing character variables. As demonstrated above, some functions may have large efficiency advantages over others.

CONCLUSION

Although SAS variables only come in two types, the two types come in nearly countless varieties, determined by their descriptor values. Descriptor values give SAS variables meaning and usefulness, enhancing their character or numeric contents with meaningful and useful metadata. Variables may also be grouped and processed in convenient groups using lists and arrays. Variable lists can be utilized in a variety of tasks to enhance processing and speed programming. Good knowledge of variable handling forms a solid foundation for use of the rest of the SAS system.

REFERENCES

SAS 9.1.3 XP Platform
SAS Institute Inc., Cary, NC

SAS OnlineDoc 9.1.3 for the Web
SAS Institute Inc., Cary, NC

ACKNOWLEDGMENTS

The author would like to acknowledge the valuable insights and observations by the many regular contributors at SAS-L.

RECOMMENDED READING

SAS-L@LISTSERV.UGA.EDU
<http://listserv.uga.edu/archives/sas-l.html>

Documentation for SAS Products and Solutions
<http://support.sas.com/documentation/onlinedoc/index.html>

CONTACT INFORMATION

Any errors or omissions are the fault of the author. Any questions, comments or corrections are valued and encouraged. Please feel free to contact the author at:

Paul Choate, Senior Programmer Analyst
California Department of Developmental Services
1600 9th Street, Sacramento, CA 95818
Work Phone: (916) 654-2160
E-mail: pchoate@dds.ca.gov

Join the SAS-L! @ listserv.uga.edu or Google Groups

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.