

All About Strings

Rhena Seidman

How SAS* Processes Character Data

If an unwary programmer were to run the following data step:

```
DATA _NULL_;  
  A = REPEAT('X',4);  
  B = 'YYYY';  
  B = REVERSE(A);  
  PUT A= B=;
```

The log would contain:

```
A=XXXXX B=
```

Showing that, to his or her surprise, B is blank.

Understanding a few details about how SAS processes character data can help tremendously in avoiding unexpected puzzling results like this. Here are the four basic facts which, put together, explain why B ended up blank.

Fact 1: Character data takes up one byte of memory per character. For example, the constant 'YYYY' takes up 4 bytes of memory.

Fact 2: A character constant, character variable, or character expression can only be up to 200 bytes long. If a program attempts to exceed this limit, in some cases a syntax error will be generated. In other cases SAS will simply truncate the data—remove any characters past 200.

Fact 3: A character variable always has a fixed memory length which does not change during a data step. This length is determined by the SAS supervisor during the compile phase of the data step: before any data is read in, before the first observation begins. If a character variable is assigned a value longer than it can hold the value will be truncated, cut short to fit. If the variable is assigned a value shorter than the variable blank padding will result—blanks will be added on the right side to fill out the length of the variable. These principles are demonstrated below:

```
DATA _NULL_;
```

```
  LENGTH A B $8;  
  A = 'TOO LONG TO FIT';  
  B = 'SHORT';  
  * Concatenate an arrow before and after  
  the variable to show up leading or  
  trailing blanks in printout;  
  A_LINE = 'A==>' || A || '<==';  
  B_LINE = 'B==>' || B || '<==';  
  PUT A_LINE / B_LINE;
```

The log will show:

```
A==>TOO LONG<==  
B==>SHORT      <==
```

A and B are both length 8. Since the value assigned to B, 'SHORT', is 3 characters shorter, 3 blanks of blank padding were added; since the value assigned to A, 'TOO LONG TO FIT', was too long to fit, it was truncated after 8 characters.

Some confusion about length comes from the LENGTH function. The LENGTH function does not measure memory length. It measures the location in its argument of the last nonblank character, which is, essentially, the visual length. If the LENGTH function were used in the above step to evaluate the length of A and B after the assignment of the constants:

```
DATA _NULL_;  
  LENGTH A B $8;  
  A = 'TOO LONG TO FIT';  
  B = 'SHORT';  
  LENGTH_A = LENGTH(A);  
  LENGTH_B = LENGTH(B);  
  PUT LENGTH_A = LENGTH_B = ;
```

the log would show:

```
LENGTH_A=8 LENGTH_B=5
```

Note that unlike memory length, the visual length of the contents of a character variable can vary during the execution of a data step.

Fact 4: The SAS supervisor decides the length of a variable based on the context of the variable's first appearance in the data step. The first appearance can be in almost any type of statement. SAS uses a

consistent, relatively sensible methodology for inferring an appropriate length.

The following dataset has comments detailing this methodology for different types of statements:

```
DATA TEMP;
  LENGTH A $ 6;
  *A is given length 6;

  RETAIN B C '1234';
  *B and C are both length 4 because the
  constant '1234' is length 4;

  D = C;
  E = '123456';
  *If the first appearance of a variable
  is in an assignment statement SAS
  assigns it the length of the right
  side: D becomes length 4, and E, length
  6;

  F = C||E;
  *Concatenated lengths are added: F
  becomes length 10;

  G = SUBSTR(B,1,1);
  H = LEFT(B);
  I = TRIM(B);
  *Certain functions such as SUBSTR,
  LEFT, and TRIM return the same length
  as the length of their argument. G, H,
  and I all become length 4;

  J = REPEAT(B,3);
  K = SCAN(B,2);
  *Other functions such as REPEAT and
  SCAN return length 200. This is
  unfortunate, and may be fixed in a
  future SAS release. J and K both
  become length 200;

  INFILE IN;
  INPUT L $5.;
  * L is given the length of its
  associated informat: 5;
```

```
SET OLDDATA;
  *All the variables in OLDDATA will have
  the lengths they received in the data
  step that created them except variables
  which have already appeared earlier in
  this step, namely A through L;
```

There is no hierarchy of statements for resolving conflicting length hints, since SAS exclusively looks at the first appearance. Therefore, a LENGTH statement is only useful if the variables in it do not appear earlier in the step.

It is now clear why, in the original example, B is blank:

```
DATA _NULL_;
  A = REPEAT('X',4);
  * Since A first appears assigned to a
  function which returns 200, A is 200;

  B = 'YYYY';
  * Since B first appears assigned to a
  constant of length 4, B is length 4;

  B = REVERSE(A);
  *A is equal to KXXXXX followed by 195
  bytes of blank padding. The reverse of
  A is 195 blanks followed by KXXXX. B
  can only hold the first four bytes,
  which are blanks. Therefore, B is
  blank;
```

The simplest way to fix this step is to add a LENGTH statement:

```
DATA _NULL_;
  LENGTH A B $5;
  A = REPEAT('X',4);
  B = 'YYYY';
  B = REVERSE(A);
  PUT A= B=;
```

The log will now show:

A=XXXXX B=XXXXX

With the LENGTH statement added to prevent the blank padding of A and the truncation of the value assigned to B the step produces the intuitively expected results.

Finding Out About Strings

Often you will want your program to take some action conditionally depending on a characteristic of a character variable. Character strings can be tested as part of conditions in these statements:

```
IF condition THEN action;
DO WHILE(condition);
DO UNTIL(condition);
WHEN(condition) action;
IF condition;
```

Here are some expressions useful for testing strings. Use these expressions as the condition for the above statements.

To test whether a string is blank:

```
(STRING = '')
```

To test whether a string contains a blank:

```
(INDEX (STRING, ' '))
```

To test whether a string contains two consecutive blanks:

```
(INDEX (STRING, '  '))
```

To test whether a string contains an embedded blank:

```
(INDEXC (TRIM (LEFT (STRING)), ' '))
```

To test whether a string contains a lowercase letter:

```
(STRING NE UPCASE (STRING))
```

To test whether a string contains a digit:

```
(INDEXC (STRING, '0123456789'))
```

To test whether it is all digits:

```
(NOT VERIFY (STRING, '0123456789'))
```

To test whether it is a valid numeral:

```
(STRING > .Z)
```

To test whether a string falls into an alphabetic range (from Rh to Se):

```
('RH'<=: UPCASE (STRING) <=: 'SE')
```

To test whether a string is a valid SAS variable name:

```
(NOT VERIFY (UPCASE (SUBSTR (STRING, 1, 1)),  
'ABCDEFGHIJKLMNOPQRSTUVWXYZ_')  
AND  
NOT VERIFY (UPCASE (TRIM (STRING)),  
'ABCDEFGHIJKLMNOPQRSTUVWXYZ_',  
'0123456789')  
AND  
LENGTH (STRING) <= 8)
```

To test whether a string is left justified:

```
(LENGTH (STRING) = LENGTH (LEFT (STRING)))
```

To test whether it is centered:

```
(-1 <=  
LENGTH (LEFT (STRING)) +  
LENGTH (RIGHT (STRING)) -  
2*LENGTH (STRING)  
<= 1)
```

To test whether it is right justified:

```
(LENGTH (STRING) =  
LENGTH (RIGHT (STRING)))
```

To test whether a string begins with \$:

```
(STRING =: '$')
```

To test whether a word ends in the letter S:

```
(UPCASE (LEFT (REVERSE (STRING))) =: 'S')
```

To test whether it ends in ING:

```
(UPCASE (LEFT (REVERSE (STRING))) =: 'GNI')
```

To test whether a string contains CHOCOLATE:

```
(INDEX (STRING, 'CHOCOLATE'))
```

It also might be useful to measure certain things about a string. The following are some examples of measurements you can make. These measurements result in numeric values, which can be used in numeric expressions or assigned to numeric variables.

The number of blanks in a string:

```
(LENGTH (RIGHT (STRING))  
- LENGTH (COMPRESS (STRING, ' ')))
```

The number of times a particular character appears, in this case the letter E:

```
(LENGTH (COMPRESS (STRING, ' '))  
- LENGTH (COMPRESS (STRING, ' E'))  
+ (LEFT (STRING) = 'E'))
```

The number of nonblank characters in a string:

```
(LENGTH (COMPRESS (STRING, ' '))  
- (STRING = ' '))
```

The number of words in a string, if words are separated by single blanks:

```
(LENGTH (LEFT (STRING))  
- LENGTH (COMPRESS (STRING, ' '))  
+ 1)
```

And for those doing especially intensive text analysis, the ratio of consonants to vowels:

```
(  
(LENGTH (COMPRESS (STRING, ' '))  
- LENGTH (COMPRESS (UPCASE (STRING),  
'BCDFGHJKLMNPQRSTVWXYZ')))
```

```

/
(LENGTH(COMPRESS(String, ' '))
- LENGTH(COMPRESS(UPCASE(String),
' AEIOU'))))
)

```

Working With Strings

Sometimes a string contains the data you need, but not exactly in the format you prefer. For example, you might want to transform strings to make them homogeneous in some respect, for easier subsequent analysis. Use the following expressions instead of the original strings whenever the transformed string is more useful for further processing.

To remove all blanks from a string:

```
(COMPRESS(String, ' '))
```

To remove leading and trailing blanks:

```
(TRIM(LEFT(String)))
```

To shift a string 1 space to the left:

```
SUBSTR(String, 2)
```

To shift a string a variable (N) number of spaces to the right:

```
REPEAT(' ', N-1) || String
```

To left align a string:

```
LEFT(String)
```

To right align:

```
RIGHT(String)
```

To center:

```
SUBSTR(RIGHT(String),
CEIL(.5*(LENGTH(RIGHT(String)) -
LENGTH(LEFT(String)))) + 1)
```

To convert a string to only uppercase:

```
UPCASE(String)
```

To convert to only lowercase:

```
TRANSLATE(String,
'abcdefghijklmnopqrstuvwxyz',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ');
```

To remove punctuation:

```
COMPRESS(String, ',.!:;?-' )
```

Encrypting Strings

If it is important to you that certain files or parts of files be protected from unauthorized reading, you could encrypt your data.

One of the easiest ways to encrypt a string is the character substitution method. In character substitution, each character in a set of characters replaces one other character. For example, the character 7 might replace A, y would replace 7, and so on. You can use the following simple statements to implement this method:

```
RETAIN
CHARSET
'ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890ab
cdefghijklmnopqrstuvwxyz'
ENCODE
'73qXMaoltPCKObn2gBFpWhcGlsuHfm0NyIDd6k
AQVeJx9RLr8iwjvZT5SUEYz4';
```

```
CRYPT =
TRANSLATE(String, ENCODE, CHARSET);
```

Notice that all the letters in CHARSET appear in ENCODE, in scrambled order. The code characters are substituted for the characters in STRING to produce CRYPT. For example, if STRING is SASv607 then CRYPT is F7FUNdy.

To decode the data, use the following statements:

```
RETAIN
CHARSET
'ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890ab
cdefghijklmnopqrstuvwxyz'
ENCODE
'73qXMaoltPCKObn2gBFpWhcGlsuHfm0NyIDd6k
AQVeJx9RLr8iwjvZT5SUEYz4';
```

```
DECODE =
TRANSLATE(CRYPT, CHARSET, ENCODE);
```

This decodes the data by doing the substitution in the reverse direction.

An advantage of this method is that you can have different secret encoding strings for different projects. Someone would have to know or guess the encoding string used in a particular project to decode the coded data.

Another good method is the transposition method. In this method, the letters of a string are rearranged in a predetermined sequence. For example, the first character becomes the fifth character in the coded string, the fifth become the fourth, and so on. To encrypt a list of social security

numbers, you could add the following code fragment to your program:

```
LENGTH SSN SSNCODE $ 9;
ARRAY CODEVCTR{9} C1-C9;
RETAIN C1 5
      C2 9
      C3 7
      C4 2
      C5 8
      C6 4
      C7 6
      C8 1
      C9 3;

DO I = 1 TO 9;
  SUBSTR(SSNCODE, I, 1) =
    SUBSTR(SSN, CODEVCTR{I});
END;
```

Transposition is an especially effective way to encode numeric codes such as social security numbers and account numbers, or similar strings made up of a limited set of characters that are used with about equal frequency. When the social security number 070-50-8304 is coded as 043705800 all of its meaning is lost until the decryption sequence is applied.

The decryption of the encoded social security number uses these statements:

```
LENGTH SSN SSNCODE $ 9;
ARRAY CODEVCTR{9} C1-C9;
RETAIN C1 5
      C2 9
      C3 7
      C4 2
      C5 8
      C6 4
      C7 6
      C8 1
      C9 3;

DO I = 1 TO 9;
  SUBSTR(SSN, CODEVCTR{I}, 1) =
    SUBSTR(SSNCODE, I);
END;
```

The decoding process just reverses the assignment statements used in the encoding process.

To make data a bit more secure, you could combine both encrypting methods, first doing one and then the other.

* SAS is a registered trademark of SAS Institute, Inc.