Why the DATA Step Does What It Does

Neil Howard, i3 Data Services, Basking Ridge, NJ

ABSTRACT

The DATA step is the most powerful tool in the SAS system. Understanding the internals of DATA step processing, what is happening and why, is crucial in mastering code and output. Concepts covered:

- Logical Program Data Vector (LPDV or PDV),
- automatic SAS variables and how they are used,
- the importance of understanding the internals of DATA step processing,
- what happens at program compile time,
- what's happening at execution time,
- how variable attributes are captured and stored, and
- handling processing defaults, data defaults, data conversions, and missing values.

This paper focuses on techniques that capitalize on the power of the DATA step and working with (and around) the default actions. By understanding DATA step processing, you can debug your programs and interpret your results with confidence.

INTRODUCTION

SAS procedures are powerful and easy to use, but the DATA step offers the programmer a tool with almost unlimited potential. In the real world, we're lucky if systems are integrated, data is clean and system interfaces are seamless. The DATA step can help you, your programmers, your program, and your users perform better in the real world – especially when you take advantage of the available features. This paper focuses on basic techniques that demonstrate the functioning of the DATA step and illustrate the default actions. Any of the topics/examples covered in this presentation have more than enough details, idiosyncrasies, and caveats to warrant its own tutorial, so selected essential processing tips and illustrative examples are provided:

- compile versus execution time activities;
- organizing your data to maximize execution;
- data defaults, data conversions;
- missing values, formatting values;
- ordering variables;
- effectively creating SAS data sets;
- the logic of the MERGE; and
- efficiency techniques.

DATA Step Compile vs. Execute

There is a distinct compile action and execution for each DATA and PROC step in a SAS program. Each step is compiled, then executed, independently and sequentially. Understanding the defaults of each activity in DATA step processing is critical to achieving accurate results. During the compilation of a DATA step, the following actions (among others) occur:

- syntax scan
- SAS source code translation to machine language
- definition of input and output files
- creation of tools:
  - input buffer (if reading any non-SAS data),
  - Logical Program Data Vector (LPDV),
  - and data set descriptor information
- determining variable attributes for output SAS data set
- capturing variables to be initialized to missing

Variables are added to the LPDV in the order seen by the compiler during parsing and interpretation of source statements. Their attributes are determined at compile time by the first reference to the compiler. For numeric variables, the length is 8 during DATA step processing for precision; length is an output property. Note that the last LENGTH or ATTRIB statement coded and compiled determines the attributes.

The variables to be output to the SAS data set are determined at compile time; the SAS automatic variables are never written to the output SAS data set, unless they have been assigned to SAS data set variables set up in the LPDV (_N_, _ERROR_, end=, in=, point=, first., last., and implicit array indices); the variables written are specified by user-written DROP and/or KEEP statements or data set options; the default being all non-automatic variables in the LPDV. The output routines are also determined at compile time.

The following statements are **compile-time only** statements. They provide information to the LPDV, and cannot by default (except in the macro language) be conditionally executed. Placement of the last statements (shown below) is critical because the attributes of variables are determined by the first reference to the compiler:

Location irrelevant:

- drop, keep, rename
- label
- retain

Location critical:

- ⇒ length
- ⇒ format, informat
- ⇒ attrib
- ⇒ array
- ⇒ by
- ⇒ where

By default, the values of variables coded in an INPUT statement and user-defined variables (e.g., with an assignment statement) are not retained across executions of the DATA step, unless referenced in a RETAIN statement. Variables whose values **are** retained include:

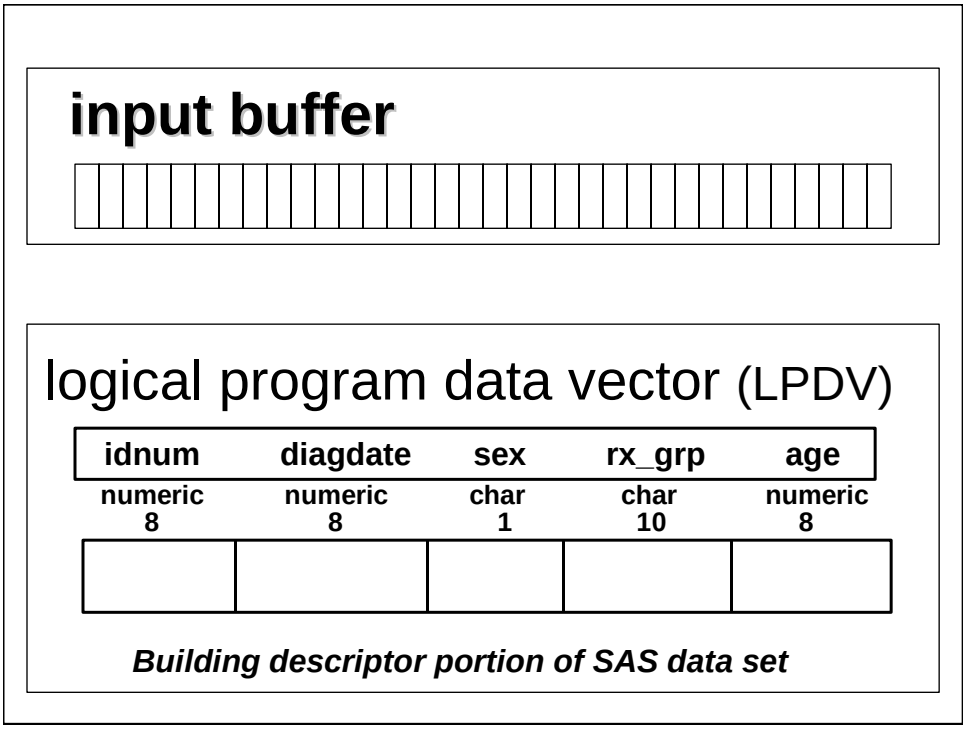
- all SAS special automatic variables
- all variables coded in a RETAIN statement
- all variables read with a SET, MERGE or UPDATE statement
- accumulator variables in a SUM statement.

Consider the following DATA step code, remembering that the statements are compiled sequentially.

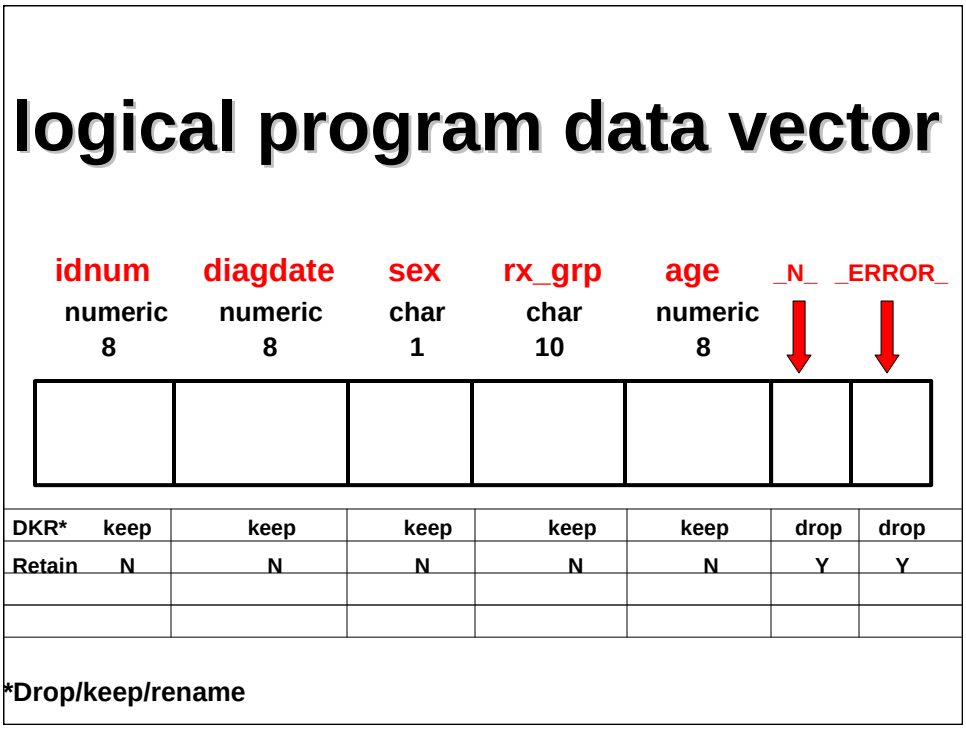
Compile Loop and LPDV

```
data a ;
    put _all_ ;      *write LPDV to LOG;
    input @1 idnum 8.
          @10 diagdate mmddyy8.
          @19 sex $1.
          @21 rx_grp $ 10. ;
    age = intck ('year', diagdate, today() ) ;
    put _all_;       *write LPDV to LOG;
cards ;
1          09-09-52 F  placebo
2          11-15-64 M 300 mg.
3          04-07-48 F  600 mg.
;
run;
```

The input buffer and LPDV are set up at compile time and variables are added to the LPDV according to the way the INPUT statement and assignment statements are written. The purpose of the PUT __ALL__ statements will become apparent at execution time, when they will dump the contents of the LPDV to the LOG before and after the user-written statements are executed.



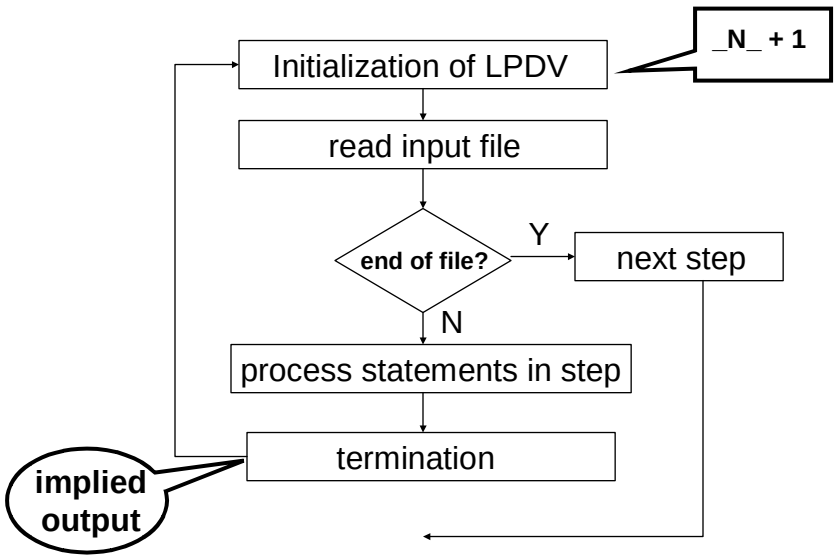
The resulting LPDV contains all the information necessary for the descriptor portion of the SAS data set, seen when PROC CONTENTS is invoked. Notice also that the LPDV contains information about which variables are written to the SAS data set being created, according to the default activities of DATA step processing. These can be overridden through the use of DROP and KEEP statements or data set options. The LPDV also tracks which variables are retained.



Once compilation has completed, the DATA step is executed: the I/O engine supervisor optimizes the executable image by controlling looping, handling the initialize-to-missing instruction, and identifying the observations to be read. Variables in the LPDV are initialized, the DATA step program is called, the user-controlled DATA step machine code statements are executed, and the default output of observations is handled.

The execution loop is shown below.

Execution of a DATA Step



When executed, the PUT __ALL__ statements in our DATA step code will illustrate how and when variables are initialized, then satisfied, before and after execution of the INPUT and assignment statements. There are 3 input records and there will be 2 dumps of the LPDV written to the LOG for each.

Execution Loop – raw data

```
data a ;
    put _all_ ;      *write LPDV to LOG;
    input @1 idnum 8.
          @10 diagdate mmddyy8.
          @19 sex $1.
          @21 rx_grp $ 10. ;
    age = intck ('year', diagdate, today() ) ;
    put _all_;       *write LPDV to LOG;
cards ;
1          09-09-52 F  placebo
2          11-15-64 M 300 mg.
3          04-07-48 F  600 mg.
;
run;
```

Note that all variables in the LPDV are initialized to missing at the beginning of each execution, except __N__ which is automatically retained. Note also that the DATA step will attempt to execute a fourth time, hence the dump of the LPDV to the LOG for __N__=4. The step will fail when the INPUT statement attempts to execute a fourth time since there is no more input data.

LPDV

IDNUM	DIAGDATE	SEX	RX_GRP	AGE	_N_
.	.			.	1
1	-2670	F	placebo	48	1
.	.			.	2
2	1780	M	300 mg.	36	2
.	.			.	3
3	-4286	F	600 mg.	52	3
.	.			.	4

Internal SAS date representation

(over all executions of DATA step……..)

The resulting LOG output is shown below:

```
83   data a ;
84   put _all_ ;      *write LPDV to LOG;
85   input @1  idnum 8.
86         @10 diagdate mmddyy8.
87         @19 sex $1.
88         @21 rx_grp $ 10. ;
89   age = intck ('year', diagdate, today() ) ;
90   put _all_;       *write LPDV to LOG;
91   cards ;

idnum=. diagdate=. sex=  rx_grp=  age=. _ERROR_=0 _N_=1
idnum=1 diagdate=-2670 sex=F rx_grp=placebo age=52 _ERROR_=0 _N_=1
idnum=. diagdate=. sex=  rx_grp=  age=. _ERROR_=0 _N_=2
idnum=2 diagdate=1780 sex=M rx_grp=300 mg. age=40 _ERROR_=0 _N_=2
idnum=. diagdate=. sex=  rx_grp=  age=. _ERROR_=0 _N_=3
idnum=3 diagdate=-4286 sex=F rx_grp=600 mg. age=56 _ERROR_=0 _N_=3
idnum=. diagdate=. sex=  rx_grp=  age=. _ERROR_=0 _N_=4


NOTE: The data set WORK.A has 3 observations and 5 variables.
NOTE: DATA statement used:
      real time           0.05 seconds
      cpu time            0.05 seconds
```

Note the results of the PROC CONTENTS run on the SAS data set created. Information from the LPDV regarding

the variables and data set attributes are reflected.

The CONTENTS Procedure

Data Set Name:	WORK.A	Observations:	3
Member Type:	DATA	Variables:	5
Engine:	V8	Indexes:	0
Created:	16:15 Thursday, March 4, 2004	Observation Length:	40
Last Modified:	16:15 Thursday, March 4, 2004	Deleted Observations:	0
Protection:		Compressed:	NO
Data Set Type:		Sorted:	NO
Label:			

-----Engine/Host Dependent Information-----

Data Set Page Size:	4096
Number of Data Set Pages:	1
First Data Page:	1
Max Obs per Page:	101
Obs in First Data Page:	3

--Alphabetic List of Variables and Attributes--

#	Variable	Type	Len	Pos
5	age	Num	8	16
2	diagdate	Num	8	8
1	idnum	Num	8	0
4	rx_grp	Char	10	25
3	sex	Char	1	24

A PROC PRINT with a format statement for DIAGDATE shows the 3 resulting observations in the new data set.

PROC PRINT

IDNUM	DIAGDATE	SEX	RX_GRP	AGE
1	09/09/52	F	placebo	48
2	11/15/64	M	300 mg.	36
3	04/07/48	F	600 mg.	52

When we apply the execution loop to the processing of SAS data, we can observe the differences. The LPDV will continue to be displayed in the LOG via the PUT __ALL__ statements. The DATA step is reading data set “a” created in the previous step.

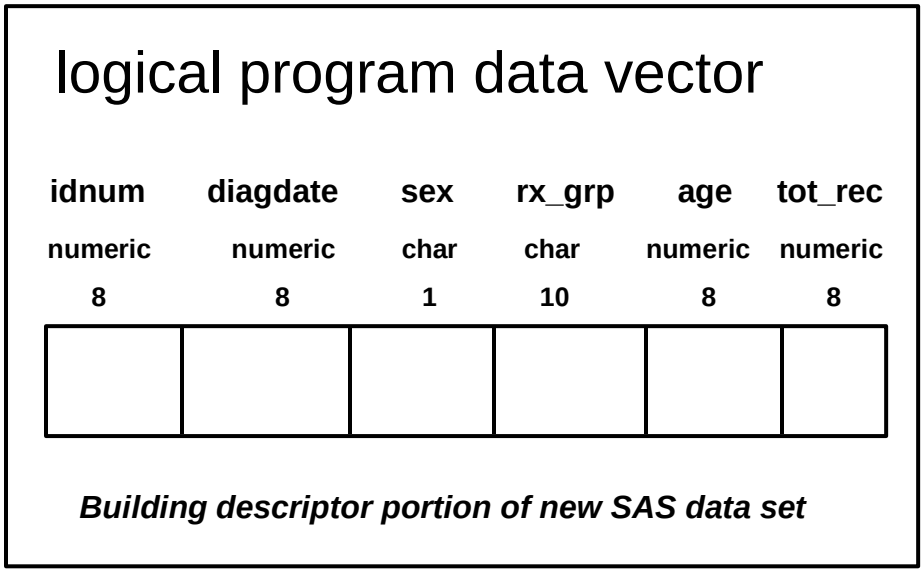
Execution Loop - SAS data

```
data  sas_a ;
  put  _all_ ;
    set  a ;
    tot_rec + 1 ;
  put  _all_ ;
run;
```

At compile time, the SET statement reads the descriptor portion of the input SAS data set in order to start building the LPDV. It retrieves the variables and their attributes. Additional variables will be added to the LPDV as their creation is encountered in subsequent SAS statements, in this case, the SUM statement to calculate TOT_REC.

Note that all variables in the LPDV will be retained. IDNUM. DIAGDATE, SEX, RX_GRP and AGE will be retained by default because they are being read in with a SET statement. TOT_REC is retained because it is the accumulator variable in a SUM statement (one of the features of the SUM statement is an implied retain).

Building LPDV from descriptor portion of old SAS data set



Given the impact of the implied retains on all variables in our code, the dumps of the LPDV to the LOG look quite different. Rather than being initialized and reinitialized to missing at the beginning of every execution, the values are retained from the previous execution and are overwritten by new values when a new observation is read by the SET

statement and TOT_REC is incremented by the SUM statement.

LPDV

IDNUM	DIAGDATE	SEX	RX_GRP	AGE	TOT_REC	_N_
.	.			.	0	1
1	-2670	F	placebo	48	1	1
1	-2670	F	placebo	48	1	2
2	1780	M	300 mg.	36	2	2
2	1780	M	300 mg.	36	2	3
3	-4286	F	600 mg.	52	3	3
3	-4286	F	600 mg.	52	3	4

(over all executions of DATA step……..)

Note the LOG notes resulting from the execution:

LOG

```
idnum=. diagdate=. sex=  rx_grp=  age=. tot_rec=0 _ERROR_=0 _N_=1
idnum=1 diagdate=-2670 sex=F rx_grp=placebo age=48 tot_rec=1 _ERROR_=0 _N_=1


idnum=1 diagdate=-2670 sex=F rx_grp=placebo age=48 tot_rec=1 _ERROR_=0 _N_=2
idnum=2 diagdate=1780 sex=M rx_grp=300 mg. age=36 tot_rec=2 _ERROR_=0 _N_=2


idnum=2 diagdate=1780 sex=M rx_grp=300 mg. age=36 tot_rec=2 _ERROR_=0 _N_=3
idnum=3 diagdate=-4286 sex=F rx_grp=600 mg. age=52 tot_rec=3 _ERROR_=0 _N_=3


idnum=3 diagdate=-4286 sex=F rx_grp=600 mg. age=52 tot_rec=3 _ERROR_=0 _N_=4
```

And the PROC PRINT of the new SAS data set:

PROC PRINT

IDNUM	DIAGDATE	SEX	RX_GRP	AGE	TOT_REC
1	09/09/52	F	placebo	48	1
2	11/15/64	M	300 mg.	36	2
3	04/07/48	F	600 mg.	52	3

Let's set up a situation where we're merging 2 SAS data sets, and notice the compile and execute activities. Below is the code to create 2 SAS data sets: LEFT with 3 numeric variables, and RIGHT with one repeat variable (ID) and 2 character variables.

```
data left;
  input ID X Y ;
cards;
1  88  99
2  66  77
3  44  55
;
```

```
data right;
  input ID A $ B $ ;
cards;
1  A14  B32
3  A53  B11
;
```

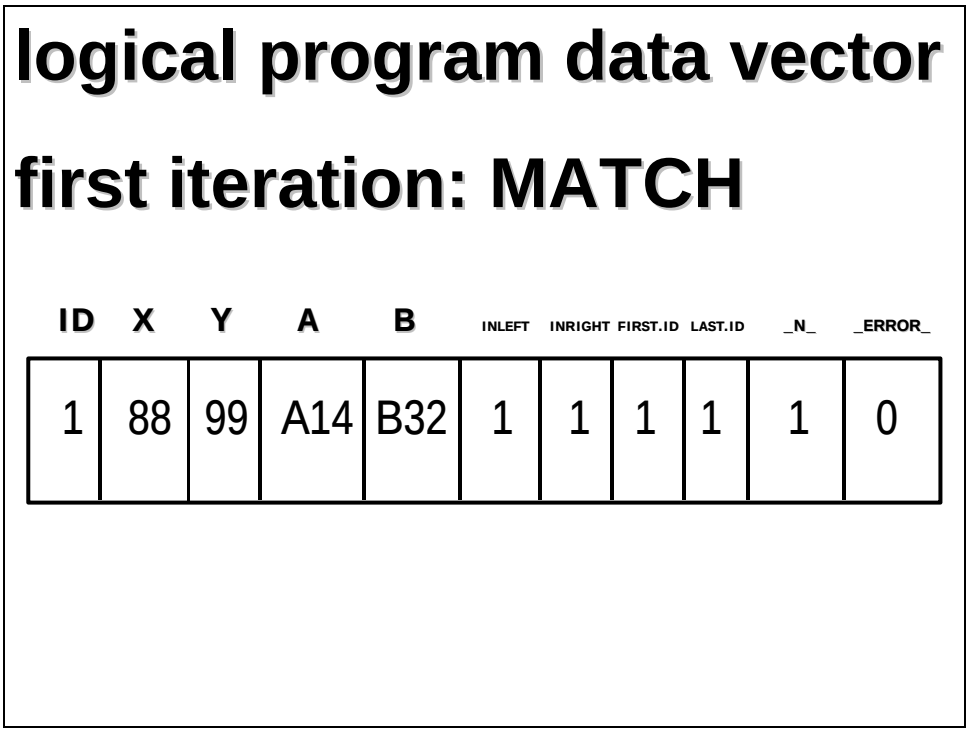
At compile time, when we invoke the MERGE statement from the following code:

```
proc sort data=left; by ID; run;
proc sort data=right; by ID; run;
data both;
   merge left (in=inleft)
         right (in=inright);
   by ID ;
run;
```

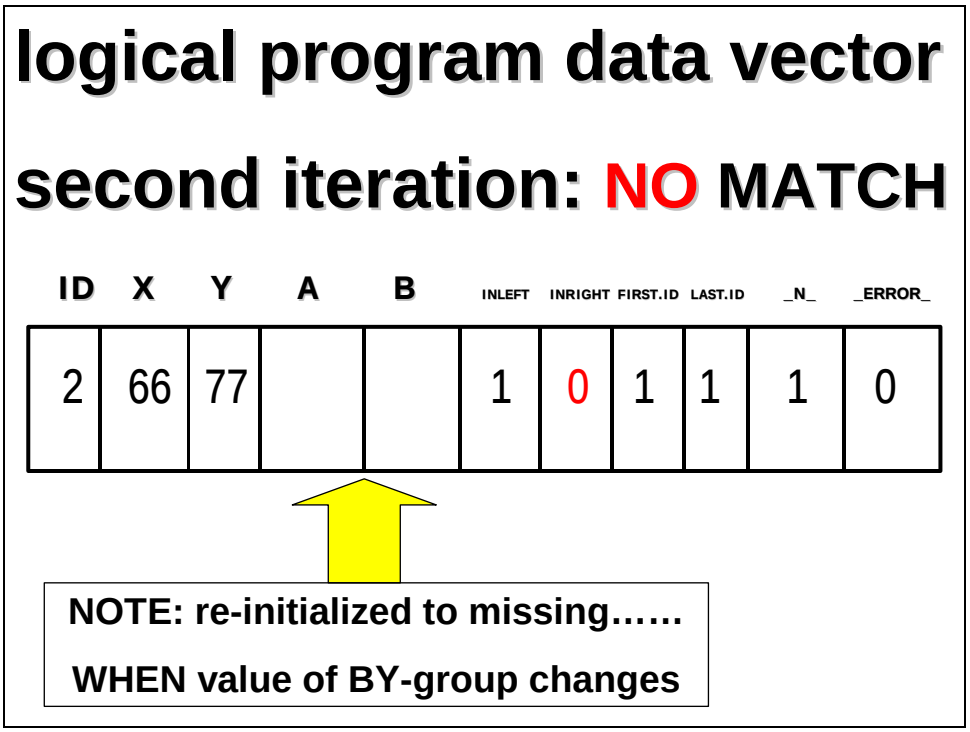
the LPDV will be built by, first, looking at the descriptor portion of the leftmost data set (LEFT) and adding its variables and attributes; then, secondly, looking at the descriptor of the next data set (RIGHT). The compiler will check to see if the variable is already in the LPDV (e.g., ID), and won't add it a second time, since it's already there. It will check to make sure the attributes (character or numeric) are the same. If not, an error message will be generated. If only the lengths are different, no error occurs.

All variables in the LPDV will be automatically retained since all are either being read using a MERGE statement or are SAS automatic variables. Note the additional automatic variables: 1) by default, because a BY statement is used, FIRST.ID and LAST.ID are created; 2) because IN= variables are coded in association with each data set on the MERGE statement, these are created. The values can be tracked using PUT __ALL__.

At execution time, the SAS supervisor sets up pointers into each SAS data set, and reads the first observation from each. If there is a match by the key variable (ID), the values from both observations are captured in the LPDV, per the first iteration example below. Note that both IN= variables are set to 1 because there is a match, and the merged observation is comprised of data from both data sets. It is both the first and last occurrence of ID=1, so FIRST.ID and LAST.ID are both set to 1.



On the second iteration, there is not a match, so the values of the variables on the observation where ID=2 overwrite the previous values in the LPDV. Note that, although there are no variable values coming from data set RIGHT, the variables A and B are missing. There is no default re-initialization to missing; however, when the value of the BY-group changes, those data set variables will be set to missing. INLEFT is 1 because there is an observation coming from data set LEFT on this execution, but there is no match from RIGHT, so its flag is 0. It is both the first and last occurrence of ID=2, so FIRST.ID and LAST.ID are set to 1.



There is a match on the 3rd iteration:

logical program data vector

third iteration: MATCH

ID	X	Y	A	B	INLEFT	INRIGHT	FIRST.ID	LAST.ID	_N_	_ERROR_
3	44	55	A53	B11	1	1	1	1	1	0

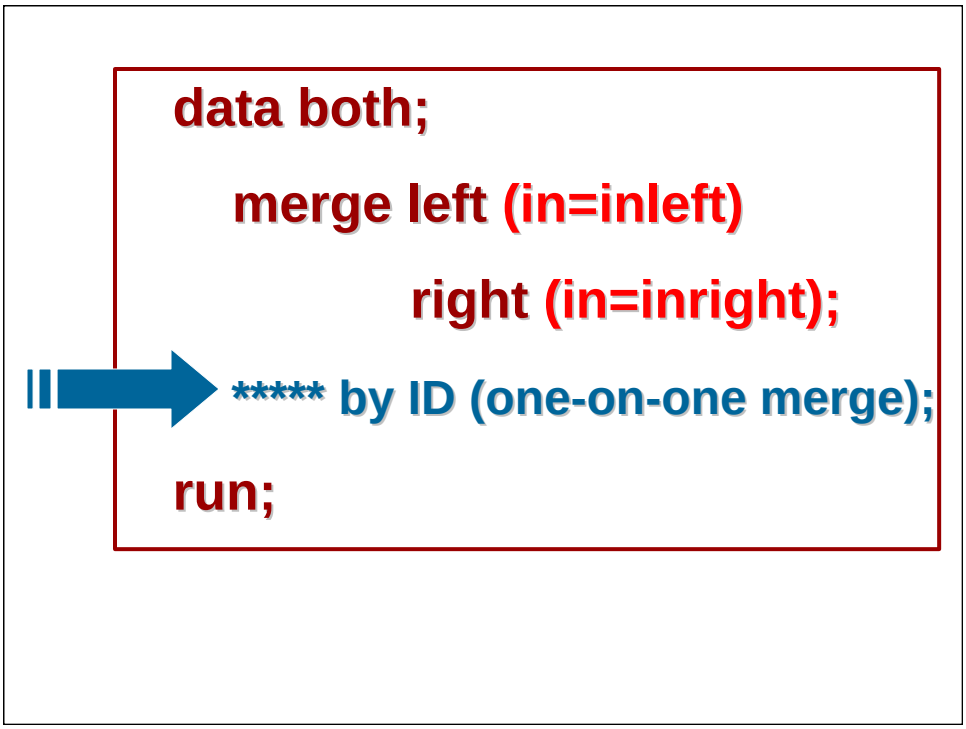
Let's try the MERGE again, without the BY statement to further examine the defaults in MERGE processing. Recall our two input data sets:

Let's try this again……

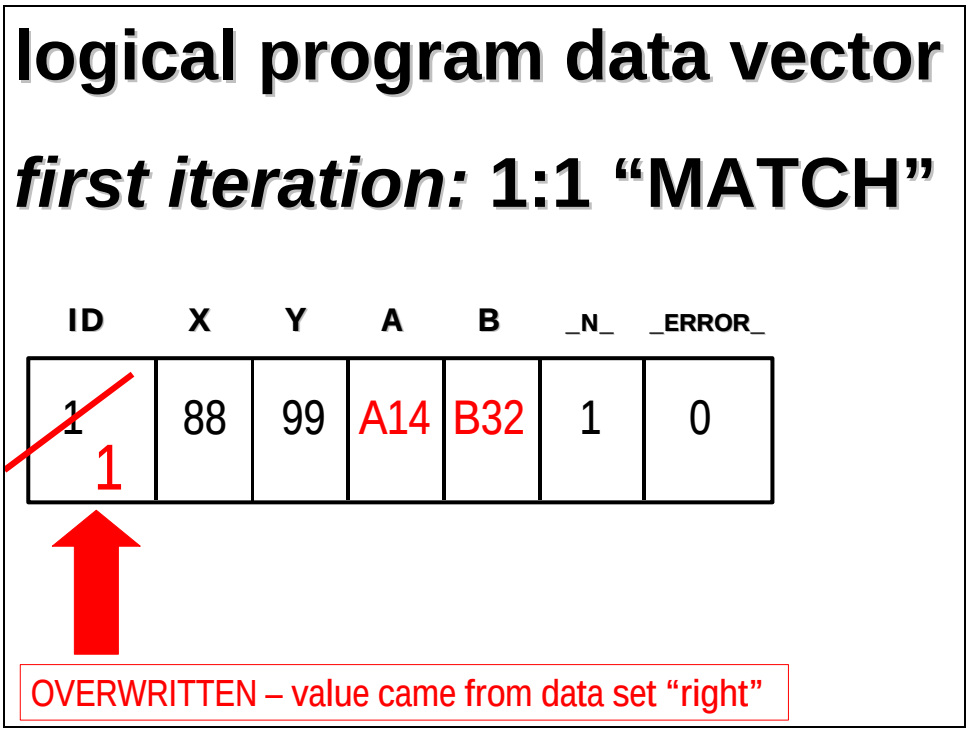
```
data left;
  input ID X Y ;
cards;
1  88  99
2  66  77
3  44  55
;
```

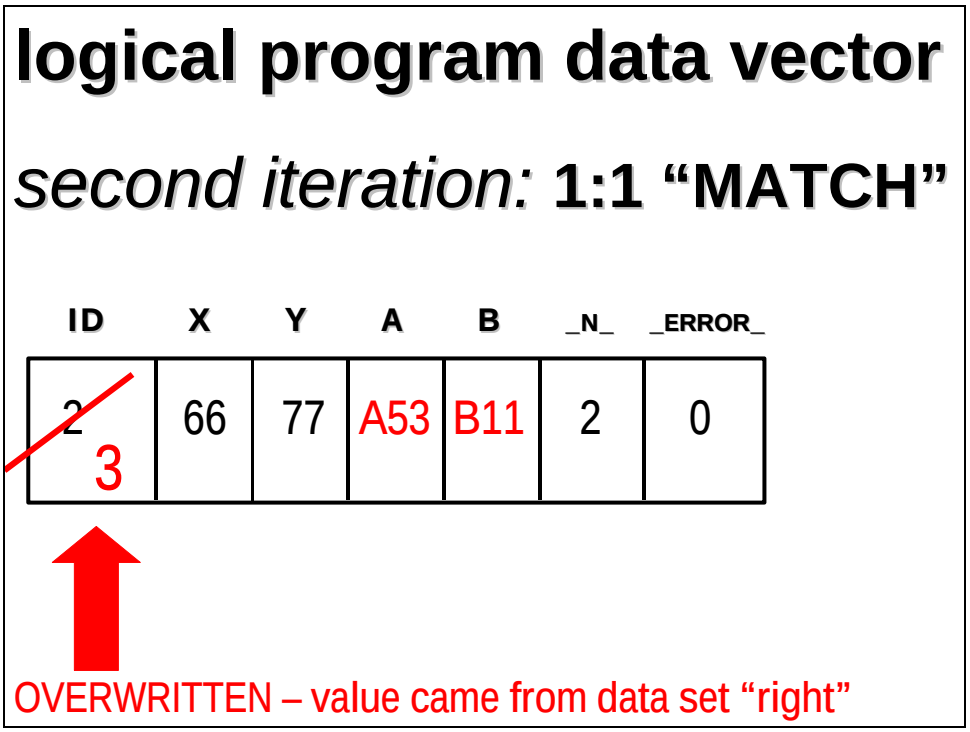
```
data right;
  input ID A $ B $ ;
cards;
1  A14  B32
3  A53  B11
;
```

The BY statement is commented out of the code, so we will be simulating a one-on-one MERGE, "matching" the first observations from each data set and writing a merged observation, matching the second observations from each, and so on, regardless of the value of ID.

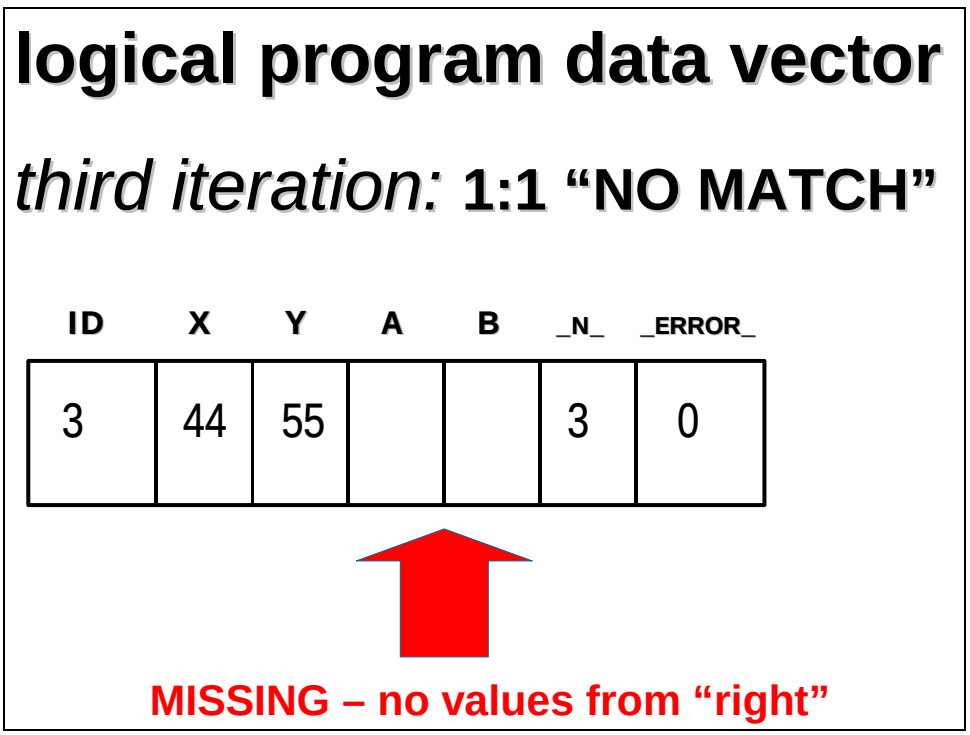


The example showing the first iteration illustrates how the values from the rightmost dataset (in red) overwrite the values from the leftmost.

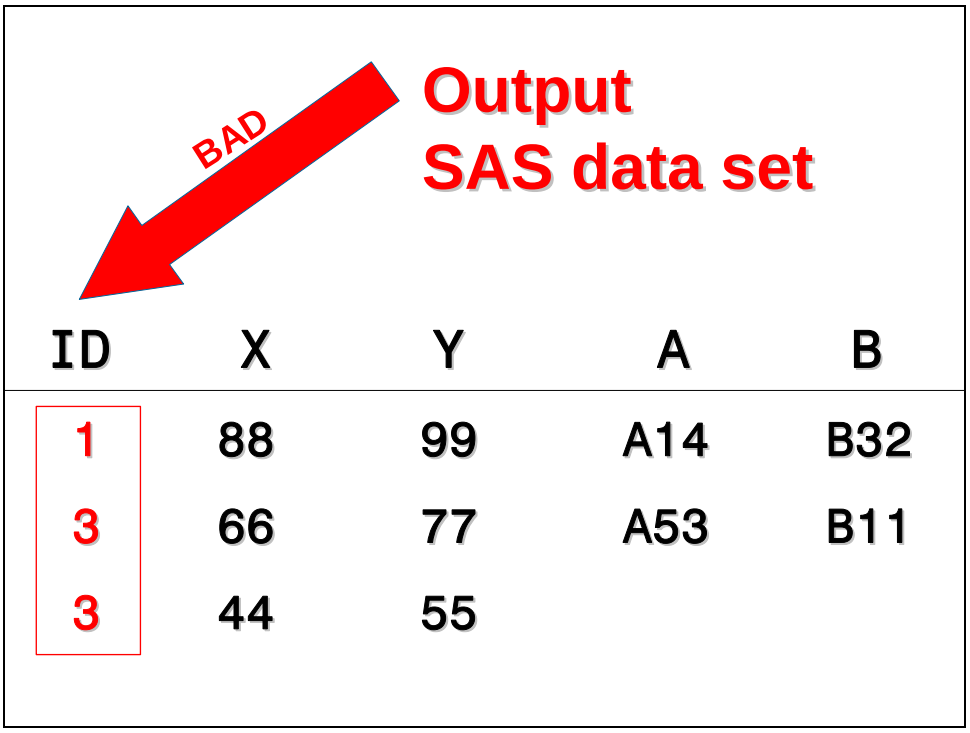




Again, note the missing values for A and B on the 3rd iteration. Another default to be aware of is the re-initialization when the data set you're reading from changes.



The output data set is not correct, but there were no error messages. SAS did exactly what you told it to do. This is why it is so important to know your data, use PUT statements to the LOG, and run PROC PRINTs before and after you execute DATA steps. By deliberately doing the wrong thing in this MERGE, the defaults of DATA step processing are more apparent. Trace the effects of processing to fully understand what is being done for you in the DATA step.



Coding Efficiencies and Maximizing Execution

The SAS system affords the programmer a multitude of choices in coding the DATA step. The key to optimizing your code lies in recognizing the options and understanding the implications. This may not feel like advanced information, but the application of these practices has far-reaching effects.

Permanently store data in SAS data sets. The SET statement is dramatically more efficient for reading data in the DATA step than any form of the INPUT statement (list, column, formatted). SAS data sets offer additional advantages, most notably the self-documenting aspects and the ability to maintain them with procedures such as DATASETS. And they can be passed directly to other program steps.

A "shell" DATA step can be useful. Code declarative, compile-only statements (LENGTH, RETAIN, ARRAY) grouped, preceding the executable statements. Block-code other non-executables like DROP, KEEP, RENAME, ATTRIB, LABEL statements following the executable statements. Use of this structure will serve as a checklist for the housekeeping chores and consistent location of important information. Use consistent case, spaces, indentation, and blank lines liberally for readability and to isolate units of code or to delineate DO-END constructions.

Use meaningful names for data sets and variables, and use labels to enhance the output. Comment as you code; titles and footnotes enhance an audit trail. Based on your understanding of the data, code IF-THEN-ELSE or SELECT statements in order of probability of execution. Execute only the statements you need, in the order that you need them. Read and write data (variables and observations) selectively, reading selection fields first, using DROP/KEEP, creating indexes. Prevent unnecessary processing. Avoid GOTOs. Simplify logical expressions and complex calculations, using parentheses to highlight precedence and for clarification. Use DATA step functions for their simplicity and arrays for their ability to compact code and data references.

Data Conversions

Character to numeric, and numeric to character, conversions occur when:

- incorrect argument types passed to function
- comparisons of unlike type variables occur
- performing type-specific operations (arithmetic) or concatenation (character)

SAS will perform default conversions where necessary and possible, but the programmer should handle all conversions to insure accuracy. The following code illustrates:

- default conversion,
- numeric-to-character conversion using PUT function,
- character-to-numeric conversion with INPUT function:

```
data convert1;
   length x $ 2 y $ 1;
   set insas; *contains numeric variables flag and code;
   x = flag;
   y = code;
run;

data convert2;
   length x $ 2 y 8
   set insas; *contains numeric variables flag and code;
   x = put(flag, 2.);
   y = input(put(code, 1.), 8.);
run;

data convert3;
   length z 2;
   set insas; *contains character variable status;
   z = input(status, 2.);
run;
```

Missing Data

The DATA step provides many opportunities for serious editing of data and handling unknown, unexpected, or missing values. When a programmer is anticipating these conditions, it is straightforward to detect and avoid missing data; treat missing data as acceptable within the scope of an application; and even capitalize on the presence of missing data. When a value is stored as "missing" in a SAS data set, its value is the equivalent of negative infinity, less than any other value that could be present. Numeric missings are represented by a "." (a period); character by " " (blank). Remember this in range checking and recoding. Explicitly handle missing data in IF-THEN-ELSE constructions; in PROC FORMATs used for recoding; and in calculations. The first statement in the following example:

```
   if age < 8 then agegroup = "child";
   if agegroup = " " then delete;
```

will include any observations where age is missing in the agegroup "child". This may or may not be appropriate for your application. A better statement might be:

```
   if (. < age < 8) then agegroup = "child";
```

Depending on the user's application, it may be appropriate to distinguish between different types of missing values encountered in the data. Take advantage of the twenty-eight special missing values:

```
. ._  .A .B .C .D .E .F .G .H .I .J .K .L .M .N .O .P .Q .R .S .T .U .V .W .X .Y .Z
```

```
   if comment = "unknown" then age = .;
   else if comment = "refused to answer" then age = .A;
   else if comment = "don't remember" then age = .B;
```

All these missing values test the same. Once a missing value has resulted or been assigned, it stays with the data, unless otherwise changed during some stage of processing. It is possible to test for the presence of missing data with the N and NMISS functions:

```
  y = nmiss(age, height, weight, name);
        ** y contains the number of missing arguments;

  z = n(a,b,c,d);
        ** z contains the number of nonmissings in the list;
```

Within the DATA step, the programmer can encounter missing data in arithmetic operations. Remember that in simple assignment statements, missing values propagate from the right side of the equal sign to the left; if any argument in the expression on right is missing, the result on the left will be missing. Watch for the "missing values generated" messages in the SAS log. Although DATA step functions assist in handling missing values, it is important to understand their defaults as well. Both the SUM and MEAN functions ignore missing values in calculations: SUM will add all the non-missing arguments and MEAN will add the nonmissings and divide by the number of nonmissings. If all the arguments to SUM or MEAN are missing, the result of the calculations will be missing. Depending on how the result will be used (in a later calculation) may determine whether or not that is acceptable:

```
x = a + b + c; * if any argument is missing, x = . ;
x = SUM(a,b,c); *with missing argument, x is sum of nonmissings;
x = SUM(a,b,c,0); * if a,b,c are missing, result will be zero;
y = (d + e + f + g) / 4; *y is missing if any variable in the calculation is missing;
y = MEAN(d,e,f,g); *number of nonmissings is divided by 4;
                    * if all arguments are missing, y = . ;
```

Since there are 90+ DATA step functions, the moral of the function story is to research how each handles missing values.

Missing values should also be taken into account when accumulating totals across iterations of the DATA step. By default, new variables created in the DATA step are initialized to missing at the beginning of each iteration of execution. Declaring a RETAIN statement overrides this default. But there are limitations.

```
      retain total 0;
      total = total + add_it;

      * this will work as long as add_it is never missing;
```

The SUM statement combines all the best features of the RETAIN statement and the SUM function:

```
     total + add_it;

     *total is automatically RETAINed;
     * add_it is added as if using the SUM function;
     * missings will not wipe out the accumulating total;
```

Missing values can also be a factor when combining data sets. Missing values from the right-most data set coded on a MERGE or UPDATE statement have different effects on the left-most data set. When there are common variables in the MERGE data sets, missings coming from the right will overwrite. However, UPDATE protects the variables in the master file (left-most) from missings coming from the transaction file. (See Real World 7 example.)

Other Data Issues

Re-Ordering Variables

SAS-L users periodically carry on the discussion of re-ordering variables as they appear in a SAS data set. Remember that as the compiler is creating the PDV, variables are added in the order they are encountered in the DATA step by the compiler. This becomes their default position order in the PDV and data set descriptor. The best way to force a specific order is with a RETAIN statement, with attention to placement. Make sure it is the first reference to the variable and the attributes are correct. It is possible to use a LENGTH statement to accomplish this, but a variable attribute could be inadvertently altered.

```
   data new;
       retain c a v; * first reference to a b c;
       set indata; * incoming position order is a b c;
       x = a || b || c;
   run;

   data new;
        length x $ 35 a $ 10 b $ 7 c $ 12; * first reference to x a b c;
        set indata; *contains c a b, in that position order;
        x = a || b || c;
   run;
```

Handling Character Data

Character-handling DATA step functions can simplify string manipulation. Understand the defaults and how each function handles missing data for optimal use.

Length of target variables

Target refers to the variable on the left of the equal sign in an assignment statement where a function is used on the right to produce a result. The default length for a numeric target is 8; however, for some character functions the default is 200, or the length of the source argument. The SCAN function operates unexpectedly:

```
data _null_;
     x= 'abcdefghijklmnopqrstuvwxyz';
     y = scan(x,1,'k');
     put y=;
run;

y=abcdefghij;    * y has length of 200;
```

The results from SUBSTR are different:

```
data old;
     a='abcdefghijklmnopqrstuvwxyz';
     b=2;   c=9;
run;

data new;
     set old;
     x=substr(a,23,4);
     y=substr(a,b,3);
     z=substr(a,9,c);
     put a= b= c= x= y= z=;
     * a is length $ 26; * x y z have length $ 26;
run;

data old;
     length idnum $ 10 name $ 25 age 8;
     idnum=substr(var1_200,1,10);
     name=substr(var1_200,11,25);
     age=substr(var1_200,36,2);
     * length statement overrides default of 200
     * for idnum, name, and age;
run;
```

SUBSTR as a pseudo-variable

Another SAS-L discussion involved the use of SUBSTR as a pseudo-variable. Note that when the function appears to the left of the equal sign in the assignment statement, text replacement occurs in the source argument:

```
data fixit;
     source = 'abcdefghijklmnopqrstuvwxyz';
     substr(source, 12, 10) = '##########';
     put source=;
run;

source=abcdefghijk##########vwxyz
```

Numeric substring

A similar function to SUBSTR if often desired for numerics. One cumbersome solution involves: 1) performing numeric to character conversion, 2) using SUBSTR to parse the string, and 3) converting the found string back to numeric. SAS would also do such conversions for you if you reference a numeric as an argument to a character function or include a character variable in a numeric calculation. See section on data conversions.

A simpler and less error-prone solution (trick) is the use of the numeric MOD function (which captures the remainder from a division operation) and INT function (which captures the integer portion of a number):

```
data new;
     a=123456;
     x = int(a/1000);
     y = mod(a,1000);
     z = mod(int(a/100),100);
     put a= x= y= z=;
run;

a=123456
x=123
y=456
z=34
```

CONCLUSIONS

The DATA step is an understandable tool. Take advantage of tracing, diagnostic and reporting features of SAS to examine the functioning of the DATA step and learn exactly what is happening at compile and execute time. The greater your understanding of DATA step activities and defaults, the easier it will be to debug code, write macros, and interpret output.

CONTACT INFORMATION

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

Neil Howard
i3 Data Services
131 Morristown Rd
Basking Ridge, NJ 07920
Work Phone: 973-348-1137
Email: neil.howard@i3data.com