Paper 246-31

How to Think Through the SAS® DATA Step

Ian Whitlock, Kennett Square, PA

Abstract

You are relatively new to SAS and interested in writing SAS code. You have probably written DATA step code; some works, some doesn't, sometimes you know why, and sometimes you don't. If this describes you, then this talk is for you.

You are a programmer, new to SAS. The DATA step language looks similar to PL1 and has some characteristics that look like languages you are familiar with. However, after learning the syntax your code feels awkward and you feel like you are fighting the system. The DATA step has been very well designed, but it is different, and it is worth learning how it works, if you want elegant code, this talk is for you.

Even people who have been SAS programmers who have been working with SAS for years should find, an "Ah ha, so that's why", in this talk.

SAS is Step Oriented

Before looking at the DATA step in detail, it would be good to take an overall look at SAS. SAS is a step oriented language rather than a routine oriented language. Routine oriented languages pass information between routines in a vector of arguments, while SAS steps typically communicate to future steps by SAS data sets. This means that much of SAS is an IO bound language. Consequently, the biggest improvements in run time efficiency, in terms of SAS programming, come from limiting data either by choosing which variables (columns) belong to a data set or which observations (rows or records) to keep on a data set, or by limiting the number of steps where data is read or created. From the programmer's point of view both of these options require good planning. From the SAS system's point of view the biggest improvement will come from the ability to parallel process the reading of files. One of the reasons version 9 is so exciting, is that it has introduced such processing for some procedures, has put the SPDE engine in the BASE product, and promises more parallel processing in the future.

Procedural steps are executable packages of machine code designed to be used with a few options in a PROC statement, and a very specialized language of statements that can be quickly parsed for control information. Consequently programming efficiency is gained mainly though the use of procedures. Hence it is probably best to take a look at the introduction to each procedure that looks at all like something you may use at sometime. The details may be safely left to a later date when you have a better overall view of what can be done with SAS. Procedural steps also tend to execute rather efficiently because they have been carefully developed to do so. It also means that SAS is a very extensible language by simply adding more procedures to the language.

Role of the DATA Step

The DATA step is a separate language for performing programming tasks such as data manipulation, i.e. cleaning and editing, and data restructuring. The tasks of combining and transforming either cannot be done with procedures or should not be done with them because the requirements are too complex or because of some overriding consideration. Often a requirement of efficiency in data passing or a desire to minimize code blocks plays a role in choosing the DATA step when a procedure would also work. It is important to realize that for the most part statements that can be put in the DATA step cannot go in procedures and vice versa. In contrast to other

programming languages, SAS is better seen as a group of related interacting languages than as a single language. It also means that the programmer can quickly become proficient in an area of interest without the need to invest time in all areas. However, the DATA step is really the glue that holds all these languages together and is, therefore, particularly important for the SAS programmer as opposed to the SAS user.

Step Boundaries

Either kind of step does not execute when submitted, until a step boundary is encountered. Typically the RUN statement is the explicit boundary statement, but sometimes it is the QUIT statement in procedures where either statements execute immediately upon completion (e.g. PROC SQL), or the RUN statement serves as a boundary for a group of statements within the procedure (e.g. PROC DATASETS). One can also create an implicit boundary by continuing with a new step. However, good SAS programmers rarely rely on an implicit boundary because explicit boundaries make the code more readable by emphasizing steps. Moreover, it is almost never seen in the code of a programmer proficient in SAS macro because of the dangerous side effects that become possible when macro instructions may be wrongly executed while a step is compiled or parsed, rather than as intended after the step finishes execution.

The fact that steps execute when a step boundary is reached means that the remaining code has not yet been read. Consequently, earlier steps can execute when later steps have errors or may not even have been written yet. This feature is important for code development, in the debugging process, and in simply getting fast feedback about how the code is working. In contrast, many program development in compiled languages often require a complete program and compilation before a part of the program can be executed. Thus SAS programs are often easier to develop. It also means that information gained in the early steps of the program may actually be used to write and then execute subsequent step code via the %INCLUDE statement.

Global Statements

In addition to the separate languages of steps, SAS has global statements which can appear in any step and execute as soon as the semi-colon is reached. The OPTIONS, LIBNAME, FILENAME, TITLE, and FOOTNOTE statements are the most prominent examples. These statements not only execute immediately, but their effect continues throughout all subsequent steps until similar statements override them with new settings. Consequently, it is clearer to place such statements between steps rather than inside them. They are a property of the global program rather than an individual step. However this policy makes boundary statements even more important.

Data Influences SAS Program Structure

In addition to the organization of SAS code the SAS programmer must become aware of the data world. It is divided into two parts - SAS stuff, i.e. data (not code) automatically recognized by SAS, and other stuff (external to SAS) which can only be utilized in SAS via special interfacing statements.

SAS data values come in only two types, numeric (floating point numbers, i.e. the computer equivalent of the mathematical real numbers), and fixed length character strings. There is a significant cost in time converting digits and decimals to the floating point form used in SAS; consequently, the first lesson in execution time efficiency from the data world is to learn to work with SAS data sets where the data need no conversion upon use. SAS data sets are stored in SAS libraries, roughly equivalent to the portion of a directory using special SAS extensions.

One of the main purposes of the DATA step is to provide a means of reading external data and creating SAS data sets for later use in procedures.

SAS is a name oriented language in the sense that the variable names are stored with the SAS data. This means simplicity in the sense that the programmer need not define all variables in all programs because the information travels with the data. Consequently it is very easy to write fast one-use SAS programs and to modify SAS code from one program to use in another. However, it also means that careful thought, planning, and techniques are required when writing code that will work without modification when ported from acting on one SAS data set to another.

Compile versus Execute

The DATA step is compiled like many other programming languages. However the executable module is discarded when it finishes execution. This means that there is never a question about which execution module goes with what code, but it also means that anyone running SAS programs also usually has access to the code. Version 9 has introduced some features that help to balance this situation for the programmer who requires it. It may also mean that the consumer of this code must license SAS in order to use the code. SAS, as a user's language, is currently in a state of transition where client computers, without SAS, can execute SAS programs by accessing a centrally located version of SAS via the SAS Enterprise Guide, the internet or an intranet.

In learning to program with the DATA step, it is very important to learn how to distinguish compile time, when the code is read, from execution time, when some form of corresponding executable machine code is actually executed.

Compilation is done in a single pass of the code reading each line in the physical order in which it appears. Consequently there will be times when the order of lines doesn't matter for SAS execution, but it does matter to the compiler because things will be set up differently depending on which line is viewed first. Many of the DATA step statements are executable in the sense that the compiler does something to make the corresponding machine code available for execution at the time the code is executed. However, some statements are non-executable in the sense that they provide information to the compiler on how to set things up, but there is no corresponding action to take place at execution time. Such statements are often called compile time directives. Typically these statements can appear in any physical order as long as they are not needed to determine variable characteristics. Finally, some statements are both. Some action is indicated at compile time, and some action is compiled for execution time. Consequently the DATA step language is richer than many languages in how various lines interact at various times, i.e. the beginner can have "fun" digging out the undocumented features and learning new techniques not encountered in previous computer language experience.

The Implied Loop of the DATA Step (ILDS)

The next most important thing to understand about the DATA step is that there is an implied loop driven by reading data. If there is no line indicating that data are to be read, then the lines are executed once as indicated by the appearance of the code. If there are lines that indicate data are to be read, but they are conditional so that no such line is actually executed, then the step stops with a NOTE that the step stopped due to looping. Without an explicit command to do otherwise, the DATA step stops executing, when it is requested to read from a file and there are no more data in that file, i.e. the end of file indicator has been reached there is a request to read more data. Consequently, the DATA step usually stops implicitly on some line indicating some form of reading data.

The words "implied loop of the DATA step" appear so often that I abbreviate it as ILDS.

One important consequence of the ILDS is that it may make sense to place code before the line that reads data. The most important example would be to check for end of file so that some action can take place just before the step stops. If the test is placed below, then it may never be executed when some subsetting statement deletes the last record.

```
if eof then
do ;   /* final reporting code */  end ;
set mydata end = eof ;
```

Another example might be to do something at the beginning of execution of the step. The variable _N_ is automatically created and indicates the iteration number of the loop. Hence one might use

```
if _n_ = 1 then
do ;   /* housekeeping code before reading */ end ;
```

Remember that each of the examples above is a snippet of DATA step code, i.e. this code must be housed in a DATA step for it to have any meaning to SAS. However, in SAS, the examples indicated are not as common as might be expected, since the system does a lot of the preparatory work that is needed in other languages and often does the correct thing when there are no more data to read for a given request.

_N_ is often used to count observations, but is important to realize that this is only a good observation counter when one record is read per iteration of the ILDS since it is really counting iterations of the loop not observations. _N_ is the first example of a variable set up by the system for the DATA step programmer's use. Typically, variables set up and assigned by the system as opposed to the programmer begin and end with an underscore. It is not a bad idea to avoid such names because the convention has a meaning to DATA step programmers. However, there is no required separation of this system's name space from that of the programmer. In fact, you can even change the value _N_ to any numeric value, but on the next iteration of the ILDS its value will revert to the value assigned by the system. As a system variable, _N_ is not kept on any output. Hence if you want to have this value on your output data set it must be copied to a new variable.

Default SAS Output

Another important fact about the ILDS is that on the last line of execution in the step, data are automatically written to a SAS data set by default. The exceptions are when the programmer explicitly uses the OUTPUT statement somewhere in the step. The DELETE statement returns to the top of the ILDS without writing a default output, and the RETURN statement returns to the top of the ILDS with a default OUTPUT. The lone keyword _NULL_ on the DATA statement indicates no SAS output is to be written. Since a large part of time spent executing a DATA step is spent in writing data, it is important to use _NULL_ whenever you do not need SAS output. DATA steps that write external files or reports are typical of those that may not need SAS output.

SAS is typically a language with many default activities. If you don't specify OUTPUT explicitly, you will get anyway with one observation per implied loop of the DATA step. If you don't name the data set then SAS will do that, choosing the first available name in the list DATA1, DATA2, DATA3, ….

More than one data set may be listed in the DATA statement. In this case,

```
output ;
```

writes to all the data sets listed. If you want to be specific then

```
output <sub-list> ;
```

is required. I tend to think it is a best practice to always specify the sub-list, usually one data set, because explicit name provides stability to the code whenever extra data sets are later added to the DATA statement.

The Program Data Vector

The program data vector, abbreviated as PDV, is a logical concept to help you think about the manipulation of variables in the DATA step. Prior to version 6 it was also a physical concept referring to the specific area in memory where DATA step variables were stored. In version 6 the memory structure was reorganized to speed up the system time spent outside the ILDS, but the PDV concept continues to be a good way for the programmer to think about what is important to him in variable manipulation. The PDV can be thought of as one long list of storage areas for all the named variables of the DATA step.

One way to get in the PDV is by assignment, as shown in the following example.

```
data w ;
   x = 3 ;
   c = "ABC   " ;
run ;
```

W is the name of the output data set. I typically use the name W to indicate that it is a play data set used for investigation or instruction. At the bottom of the step, i.e. when the compiler reaches the step boundary, the PDV looks like

X	C	_N_	_ERROR_
3	ABC	1	0

Actually the PDV is the stored values, but it is convenient to add the names and write the values in a readable form so that we can think about what is important. Remember that 3 is really stored in 8 byte floating point notation and the variable C is stored in 6 bytes. Why? Well X comes first because the compiler saw it first and it is 8 bytes because all numeric processing in the DATA step is done in 8 bytes. C comes next because it was encountered next in reading the DATA step. It is 6 bytes long because the value on the right contains 3 letters and 3 blanks. If C were later assigned a longer value it would be truncated to fit the same 6 byte area in the PDV. If C were later assigned the one letter "X" then the value would be extended with 5 blanks to fit that same 6-byte area.

SAS has fixed length character data. The first occurrence where the compiler can assume a length or has a rule for determining a default length will be the place where the fixed storage length is assigned. Usually this is also the place that it is added to the PDV.[1] Character data can be from one byte to 32,767 bytes (32K). There is no 0 length string in the DATA step. Without any further aid the compiler will probably choose 8 or 200 characters.

A more subtle consequence of the fact that SAS is a fixed length string language appears in the handling of string data by character manipulating functions. If you are coming from a programming language that allows variable length strings, you will have to learn the way of fixed lengths. You will find that SAS is very predictable and reasonable once you understand the fixed length philosophy.

One policy to control string lengths is to use a LENGTH statement to assign all character variable lengths at the top of the DATA step. Then you do not have to worry about what the compiler will do because you told it what to do. Consider:

```
data w ;
   length c $ 3 x 3 ;
   x = 3 ;
   c = "ABC   " ;
```

[1] A RETAIN statement will force the variables named which are not already in the PDV to then be put in the PDV. However the type and storage length is not yet assigned when the RETAIN statement does not initialize these variables.

```
run ;
```

Now the PDV is

C	X	_N_	_ERROR_
ABC	3	1	0

Although C is assigned after X, it comes before in the LENGTH statement, so C was met first by the compiler. Note the trailing blanks were eliminated because they were too much for the shortened length. X still has the length 8 in the PDV. For numeric variables the LENGTH statement applies only to the length stored on output DATA sets.

The length of X could be a problem. In the example, X has the value 3 and this value can be stored precisely in 3 byte floating point form. Hence the squeeze to 3 bytes in W and any subsequent expansion to 8 bytes in a later step will not hurt for small integer values. What small means, depends on the precise form of floating point notation used by computer system. The integer range -255 to +255 is always safe for the smallest possible numeric length and it could be much larger. A good policy is to use the standard length of 8 whenever a value can be a large integer or need not be an integer. Even the simple decimal, 0.1, can cause no end of programming problems when it is stored in less than 8 bytes.

What happens to the 3 in the box marked X if we add another line after the assignment of C to assign, say 7, to X? Then the 3 is replaced by the 7 and the 3 is gone. If you need to keep the value 3, then you need a second box, i.e. variable, to save the original value of X. This may seem obvious, but many questions about why something happens come from the failure to understand this principle when changes in the PDV are made.

The PDV is a valuable aid to following in detail what is happening in a SAS DATA step. We will return many times to looking at the PDV to understand what happens at compile time when the PDV is set up and at execution time as the values in the PDV change.

External I/O

The important statements of external I/O are the FILENAME, INFILE, INPUT, FILE, and PUT statements. Each of these statements is rather complex. They have to be complex to handle the variety of situations encountered in the world external to SAS. INPUT reads data from a buffer and PUT writes data to a buffer. Think of a buffer as something in the SAS system to move data from a file to the PDV or from the PDV to a file. How it works doesn't matter to us. However, the options we can specify to control this process can matter very much depending on the nature of what is done in a SAS program. By default INPUT statements refer to a buffer called either CARDS (old name) or DATALINES (new name) meaning that the data is in the program following a statement indicating the step is finished and data is coming. By default PUT statements refer to a buffer called LOG which writes to the SAS log.

As a first example, consider:

```
data _null_ ;
   put "before: " _all_ ;
   input x ;
   put "after : " _all_ ;
cards ;
1
2
3
;
```

The CARDS statement comes in four types - CARDS, CARDS4, DATALINES, and DATALINES4. Those ending in "4" indicate the end of data by 4 consecutive semi-colons starting in column1; hence, SAS code may be included in the data. The other two simply require a line containing a semicolon somewhere on the line. For any type the buffer may be referenced by any of the four words.

In practice relatively few SAS programs actually use this input buffer. It is there mainly for simple testing, and to communicate the data with the code for examples in books, talks, or when teaching SAS. In general, it is a bad idea to write SAS programs with the data imbedded in the program because the data are more likely to change than the program and because typical volumes of data would swamp the code making the program difficult to read.

Here are the messages produced by the example code.

```
before: x=. _ERROR_=0 _N_=1
after:  x=1 _ERROR_=0 _N_=1
before: x=. _ERROR_=0 _N_=2
after:  x=2 _ERROR_=0 _N_=2
before: x=. _ERROR_=0 _N_=3
after:  x=3 _ERROR_=0 _N_=3
before: x=. _ERROR_=0 _N_=4
```

In the first line "X=." indicates that X has the standard missing value. Before each iteration of the ILDS, certain variables are set to missing by the system. Note that with each new value of _N_, X is set to missing. For each second line with the same value of _N_, X gets a value from the INPUT statement, but then loses it on the next iteration. This action frequently causes problems for the beginner, but experienced SAS programmers come to depend on it and find it useful. We will give comprehensive rules for which variables are covered, but we need to see more SAS statements before the rules can make sense.

Finally, note the last line when _N_=4. The "before" line is written so new iteration has begun, but there is no "after" line because the step stopped on the INPUT statement and there was no input data left to read. It is this natural rhythm of the DATA step that suggests that the test for end of file should come before the read statement rather than after it, as was mentioned in the section on the implied loop of the DATA step.[2]

How can one take counts or sums over multiple records? To be specific let's add code to obtain the sum of X's as SUMX and the number of times the INPUT command was read as INPCOUNT. Counting can be thought of as incrementing or adding 1 so both problems could be solved if we knew how to get the step to remember values from one iteration to the next. Since this sort of thing is so essential to programming, SAS supplies a special SUM statement which includes a request to not set the obtained value back to missing. Let's also make a SAS data set W. Here is the code.

```
data W ;
   put "before: " _all_ ;
   input x ;
   inpcount + 1 ;
   sumx + x ;
   put "after : " _all_ ;
cards ;
1
2
3
;
```

[2] The code in DO-group for the test for end-of-file actually executes just before the step would be stopped on the request to read past the end of file.

The additional statements say to add the expression on the right to the variable named on the left and of course, do not automatically set this variable to missing. What about initialization? No problem, the initial value is set to 0 at the beginning of the step. In addition to this the SUM statement says to treat the addition of missing as 0. Here is the result.

```
before: x=. inpcount=0 sumx=0 _ERROR_=0 _N_=1
after : x=1 inpcount=1 sumx=1 _ERROR_=0 _N_=1
before: x=. inpcount=1 sumx=1 _ERROR_=0 _N_=2
after : x=2 inpcount=2 sumx=3 _ERROR_=0 _N_=2
before: x=. inpcount=2 sumx=3 _ERROR_=0 _N_=3
after : x=3 inpcount=3 sumx=6 _ERROR_=0 _N_=3
before: x=. inpcount=3 sumx=6 _ERROR_=0 _N_=4
```

Note that W has the variables X, INPCOUNT, and SUMX, but not _ERROR_ or _N_ because variables created by the system do not go on output data sets. Also note that W has 3 observations, not 4, since the fourth iteration never reached the bottom of the ILDS for the implied output to take place.

The FILENAME statement ties a file reference (fileref), i.e. an ad hoc name to a particular file which also becomes the name of the corresponding buffer for handling the file. There are various options for spelling out things about the file and there are device types for making other objects appear as files to the SAS program. Here are some examples illustrated with files from a Windows operating system.

```
filename fref "c:\test\mydata.dat" ;
filename prj "c:\project" ; /*requires further spec by INFILE statement*/
filename all "c:\project\f*.txt" ;          /* wild card concatenation */
filename gp ("c:\f1.txt" "h:\other.dat") ;  /* explicit concatenation  */
filename code temp ;    /* temporary work file typically used for code */
```

The INFILE statement requests that a buffer for reading be set up. To this extent it is a compile time directive. The buffer can have a name. The name is the fileref when the INFILE statement is tied to a FILENAME statement via the fileref. In this case all INFILE statements referring to that fileref refer to the same buffer. Hence one can assign options to the buffer through multiple INFILE statements or one INFILE statement because it is the buffer, not the statement, that posses the property. The fileref PRJ requires that a specific data set be determined. For example,

```
infile prj(maydata.dat) ; /* mydata.dat is a member of "c:\project" */
```

For all the other fileref's mentioned

```
infile <fileref> ;
```

would do when no further options are needed. Remember that the INFILE statement must be within a DATA step, while the FILENAME statement is global and preferably put near the beginning of the program whenever other considerations do not take priority.

The INFILE statement also points to that buffer. All subsequent INPUT statements refer to that buffer until something changes to point INPUT to a different buffer. Consequently, the INFILE statement is also executable, in that something can be changed at execution time, it is usually best to place it just before (or at least near) the INPUT statement that will use this buffer.

The INFILE statement can sometimes be used to mention a file directly without reference to a FILENAME statement. In general, this is a bad idea because the data file used in a program is much more likely to change than the code; hence placing this information in one place near the beginning of the program makes a lot of sense. In addition, each INFILE statement that references a file directly requests a new buffer so that this form of INFILE statement behaves quite differently from one that is tied to a FILENAME.

For example, here the problem is to make a list of files in the directory C:\JUNK that have the CSV extension. The INPUT statement assumes member names no longer that 8 characters including the extension and no spaces in the names. To remedy this you need a more sophisticated INPUT statement than we have discussed. The FILENAME statement illustrates the use of the PIPE device to make a DOS host command look like a file.

```
filename q pipe 'dir "c:\junk\*.csv" /b' ;

data csvlist ;
   infile q ;
   input mem $ ;
run ;
```

The FILE statement is much like the INFILE statement, but it refers to files that will be written to rather than read from. Of course, this means some of the options will be different for the two statements. In the same way PUT is very much like INPUT.

There are three types of INPUT and PUT: - list, column and formatted[3]. List INPUT/PUT simply takes a list of variables (with "$" after any character variable for INPUT). It is assumed that a space separates each field for reading and should be separated by a space when writing. The default length for input character variables is 8 bytes. Its main virtue is simplicity and it is useful for simple testing and examples. For example,

```
input id $ fname $ lname $ weight ;
```

could be used to read

```
123 Mary Fox 130
```

Note that 123 looks like a number, but we have specified that it is to be character because identifiers should in general, be character since there is no intent to do arithmetic on them. Remember that the names must be short unless a preceding LENGTH statement is used to allow for longer names, and there can be no spaces in the names.

Column INPUT/PUT is somewhat more flexible in that the length of character variables are determined by the specified width of the field and the fields do not need a separator. For example,

```
input
  id     $  1-3
  lname  $ 15-35
  fname  $  4-14
  weight   43-45
;
```

might read

```
123MaryJane   Foxworthington       xxx    130
```

Note the extra control over the length of character variables, the ability to change the order of reading, and the ability to skip over extraneous information. I improved the format by putting one variable on a line to make it easier to read.

Formatted input provides a completely flexibility and is easier to modify when the record structure changes; hence, although column input is often used, I prefer formatted INPUT/PUT. Informats specify how data is to be read and formats specify how it is to be written. Remember, SAS has only two data types - real numbers and fixed length

---

[3] There is a fourth type named INPUT/PUT. Named PUT is used at the beginning of the section on control statements.

character strings. For example, suppose a date is just before the weight and we want to pick up that date as BIRTHDAY,

```
123MaryJane   Foxworthington       10/23/1981130
```

Now the formatted INPUT statement could be

```
input
     id       $char3.
   @4 fname    $char11.
     lname    $char21.
     birthday mmddyy10.
     weight   3.
;
```

The first three informats are named $CHAR. The numbers 3., 11., and 21. are width specifiers. The number may be left out leaving the width to a default value, but the dot is required to distinguish informat/format names from variable names which cannot have the dot. This informat says to get the data as is without any form of translation. $21. is a informat that specifies getting the data, but deleting leading blanks and treating a single dot as missing; hence changing it to blank. BIRTHDAY is stored as the number 7966 which is the number days from January 1, 1960. Think of the beginning of January 1, 1960 as the zero point on an axis i.e. the first day. Then 7966 is the 7966th day. It is important to store dates in this form by using an appropriate date informat because SAS supplies many functions, and formats for manipulating and displaying dates when stored in this form.

The instructions for reading a line can even be buried in the line. In the example below, the first "a" might indicate that the value for the variable CODE, "abc", is to be found starting in column 5.

```
a5   abc
```

In this case we would need to read the first two columns and hold the line for reading the remainder in another INPUT statement. A trailing @-sign is used to indicate holding the line.

```
input type $char1. pos 1. @ ;

if type = "a" then
   input @pos code $char3. ;
```

In the second INPUT statement "@pos" indicates "go to the column indicated by POS" . Omitting the trailing @-sign indicates the line is to be released. If the line is still held with a trailing @-sign at the bottom of the ILDS, then the line is still released by the system before the next iteration. A double trailing @@-sign indicates do not release this line until it is explicitly released by being forced to the next line or by an INPUT statement without trailing @-signs.

There is still much to learn about external I/O but much of it is just learning the complete syntax of the five statements we have introduced in this section.

Control Statements

SAS supplies typical statements for making decisions and looping. IF is used to make a decision. It has the form

```
if <condition> then <consequent> ;
else <alternative consequent> ;
```

Note that the consequents are single SAS statements. To perform a set of statements they must be housed in a DO-group. For example, consider:

```
if x = 1 then
do ;
   count + 1 ;
   if count >= 7 then stop ;
   else put z= ;
end ;
```

The DO statement is the consequent, but it begins a block that does not end until the END statement. Remember the first statement in this block indicates that the variable COUNT is to be incremented by 1. COUNT is then tested to see if it is greater or equal to 7. In that case the consequent is the STOP statement. This means to explicitly stop executing the DATA step at this point. It does not mean to stop compiling the step. SAS will read code to the step boundary and then may execute that code many times, but when inside this DO-group whenever COUNT is 7 or larger the step execution will halt and SAS will go on to process the next step. When COUNT is less than 7 the ELSE consequent requests SAS to write the name of the variable Z followed by an equal sign followed by the value of Z at that moment. Where does the PUT statement write? Remember that by default it writes to the LOG and otherwise writes to the last buffer pointed to by a FILE statement. The line indicates still another form of PUT called named-put. It is very useful for writing messages to the log either for information or for debugging since it allows you to say include the name and =-sign with the value. In this example it is most likely in a debugging step where we want to see the value of Z the first 6 times that X is 1 and bring the whole step to a stop on the seventh time.

Note that the ELSE in the DO-group applies to the immediately preceding IF statement and that there is no ELSE to the first IF statement in the example.

The condition in an IF statement is something capable of being true or false. In the example the first IF condition is, X = 1. How can truth values be represented in SAS? The data type numeric is used. Zero and all missing values are false, and everything else is true. When SAS assigns a truth value it uses the canonical values 0 for false and 1 for true. This means that truth values can be used in expressions for assignments. For example,

```
tv = ( y > 3 ) and ( z = "abc" ) ;
```

assigns a value of 1 when both parts are true and 0 otherwise. Truth value assignments can often simplify the code of a complex decision making process. The parentheses were not needed in the above but it is a wise idea to include some parentheses to help the reader see that a truth value assignment is being made.

In SAS there is a special form of IF called the subsetting IF. For example,

```
if good ;
```

Good is a numeric variable. If the condition is true, i.e. the value is not 0 or missing, the code below will be reached and executed. If it is false, then the iteration of the ILDS will end without default output and the next iteration of that loop will begin. There are other ways to write the code, but the subsetting IF is very SAS idiomatic and much preferred by most experienced SAS programmers.

An iterative DO-loop names an index variable and range of values. The code in the DO-block is repeated for each value in the range. Consider:

```
do i = 1 to 3 , 8, 20 ;
   put i= ;
end ;
```

I is the index variable. It gets the values 1, 2, 3, 8, and 20 in that order so the loop is iterated a total of five times. The incrimination term is +1 but it can be changed with a BY clause.

I, J, and K are typical loop indices inherited from the fact that Fortran used variable names beginning with these letters to indicate integer values. The names are good in the sense that they convey that the variable is simply an index and has no more than local significance to the loop. Consequently it is important to drop such variables to prevent them from going on any output files or to use more meaningful and significant names when the index is to be kept on the output file or plays a role in other parts of the DATA step.

There are also the conditional DO-Loops:

```
do while ( <expression is true> );
```

and

```
do until ( <expression is false> ) ;
```

For the WHILE loop testing is done at the top of each iteration. Consequently the block of code will not execute if the expression is false on the first iteration. For the UNTIL loop testing is done at the bottom so the loop is always iterated at least once. Remember that all numeric expressions, including missing values, have a truth value, but sometimes it is particular important to initialize the loop-expression before going into the loop. Moreover, usually some code in the loop should be capable of changing the truth value of the expression.

The GOTO statement offers a general branching tool

```
goto <label> ;
```

that could be used for branch out of a loop. However, general branching can be a dangerous tool, so SAS also supplies the LEAVE statement for simply exiting a loop and the CONTINUE statement for branching to the beginning of the next iteration of the loop. Consequently sometimes it is desirable to create a loop that is not stopped by the DO-condition. For example, assume the value of LIST is `"a b c d"` but at code time we do not know how many words to expect in the list. Then one might code

```
i = 0 ;
do while ( 1 ) ;
   i + 1 ;
   word = scan ( list , i ) ;
   if word = " " then leave ;
   /* code to use word */
end ;
```

Remember that the loop expression, 1, is always true so that in principle the loop will iterate forever. This is all right because the SCAN function must run out of words and thus trigger exiting the loop with a LEAVE. In general, you should always ask what brings the loop to an end and try to make it clear in the code for other readers.

SAS I/O

The LIBNAME statement ties a group of SAS stuff, i.e. data sets with special SAS extensions, to a tag called the libref. It serves a similar purpose to the FILENAME statement except that it always works on a group of data sets rather than an individual data set, although the group may contain only one member. For example,

```
libname prj "c:\project\data" ;
```

references all the SAS stuff in the indicated directory. Note that the extension ".SAS" indicates a SAS program but it is not recognized as data stuff and hence is not part of any library created with a LIBNAME statement. All SAS data sets are then referenced by a two level name of the form <libref.member>. For example, suppose the data for Arizona are collected in the SAS data set known to Windows as

```
"c:\project\dataAZ.sas7bdat" ;
```

Then in a SAS program containing the above LIBNAME statement, this data set would be known as PRJ.AZ. Of course, the LIBNAME statement must precede any use of the libref. Consequently, the libref is a property of the program and has no intrinsic meaning across job sessions. One can concatenate the SAS stuff in different directories using a notation similar to that given for the FILENAME so that a library can cross directories. The set of extensions recognized is dependent on an engine where the default engine depends on the version. Consequently, one can even have multiple libraries in one directory, but it is better to keep them in separate libraries.

A special library called the WORK library is provided by SAS for temporary data sets that by default will be deleted when SAS closes. If a libref is not given, then there is a default assumed. The systems option USER controls this default, and by default it is WORK. In all that follows one level names, e.g. W, are used and can be thought of the two level name beginning with WORK, e.g. WORK.W. Beginners tend to want two level names, but it is simpler and better to adopt the SAS way and allow the options to specify what the distinguished library is.

Now suppose we have a SAS dataset W with the variable X and want to make a new data set Q, my other popular name for testing and examples, with INTX holding the integer part of X. Here is the DATA step.

```
data q ;
   set w ;
   intx = int ( x ) ;
run ;
```

Remember there is always an implied loop of the DATA step. The second line is the SET statement. At execution time it means get an observation from the named data set, i.e. move the values of the variables into the PDV. How did the variables get into the PDV? At compile time this instruction tells the compiler to find the data set W header, obtain all the variable names and there characteristics so that they can be added to the PDV, and to set up a buffer for reading the data at execution time. INT is a function that takes the integer part of a real number. There is an implied output to Q at the bottom. Q contains all the variables from W plus the newly created INTX. When and where does the loop stop? It stops on the SET statement when it is asked to get an observation and there are none, i.e. when it is asked to read past the end-of-file.

SAS does not reset values of variables coming from a SAS data set to missing at the beginning of each iteration of the ILDS. However, it does initialize them to missing once before execution of the step begins. One interesting use of this principle is the calculation of percents. Suppose, TOTAL contains one observation with the variable TOTX, the total of X in the data set W, and you want to add a variable PCT to W. One solution is

```
data w ;
   if _n_ = 1 then set total ;
   set w ;
   pct = 100 * x / tot ;
run ;
```

Just as the DATA statement can list more than one output data set, the SET statement can read more than one data set. For example,

```
set w1 w2 ;
```

first reads the set W1 then the set W2. Separate buffers are set up to read each set. Remember that at compile time the header of W1 will be looked at first. Any specified properties of the variables in W1 will determine the properties for these variables. However default properties may be changed by the corresponding variable in W2. For example, X in W1 may have the default label "X" and X in W2 may have the assigned label "BIG X". In this case the second label wins. However, if X in W1 had the label "LITTLE X" then that would win. The question of which data set determines the variable property is particularly important for the length of character variables. W2 can also have new variables, in which case, their properties are determined by the header information in W2.

As usual in SAS, it is left to the programmer to make sure that s/he knows what code is determining the properties of the variables in the PDV. You can always force the properties by declaring them before allowing the compiler to see the SET statement.

A great deal of DATA step processing is based on one record in and one record out. In fact, it is a tribute to the design of the language that this structure works as well as it does because it is not so true of many languages. However there is a significant class of problems where the natural structure of the problem demands one record in and many out or many records in and one out. In these situations one can achieve simpler code by placing the OUTPUT statement in a loop or the input statement in a loop. It is well worth looking for such examples and being ready to use an explicit loop whenever the situation arises.

The fact that a separate buffer is set up for each physical mention of a SAS data set has interesting consequences. The code

```
set w1 w1 ;
```

requests that the data be read twice. This is sort of obvious, but so does

```
set w1 ;
set w1 ;
```

In this case the second SET statement does not read the next record. It reads the same record from the second buffer. If you do have a need to read two records from the same data set it must be done with one SET statement. One possibility is

```
do i = 1 to 2 ;
   set w1 ;
   if i = 1 then
   do ; /* whatever for the observation */ end ;
   else
   do ; /* whatever for the second observation */ end ;
end ;
```

The same logic can be handled with a subroutine using the LINK statement. At execution time control jumps to a labeled set of statements and returns at the RETURN statement to the line following the LINK statement.

```
link readw ;
/* do whatever for the observation */
link readw ;
/* do whatever for the observation */
. . .
return ;

readw:
   set w1 ;
return ;
```

The only disadvantage to this form is that the SET statement now appears rather late in the DATA step so a length statement may be needed to get the correct length and type of variables before the compiler gets down to the SET statement. Note that SAS subroutines have no data hiding, i.e. all DATA step variables are known to all parts of the DATA step including routines.

Sometimes it is important to know which data set is contributing information. SAS provides a data set option, really an option of the input buffer specified as a property of the set, used as follows:

```
set w1 ( in = w1 ) w2 ( in = w2 ) ;
```

This requests SAS to provide 0/1 variables named W1 and W2 which indicate whether data is contributed by the corresponding data set. I often use the data set member name as the name of the IN= variable because it can easily be identified with the data set and because the name is usually available, i.e. not otherwise used. These variables do not get set to missing on each iteration of the ILDS and they will not go on any output data sets, but they are in the PDV as named variables.

In general with SAS input there can be BY-processing. This means specially named variables will be set up to indicate when a group with common BY-values starts and ends. For example,

```
set w1 ;
by state city ;
```

requests that the variables named FIRST.STATE, LAST.STATE, FIRST.CITY, and LAST.CITY be added to the PDV after the variables in W1. Assuming this code is in a DATA step and the SET statement has just been executed, the PDV might look like the following table. W1 is grouped in state city order and there are exactly two observations in AL and 4 in AZ.

state	city	...	first.state	last.state	first.city	last.city	_ERROR_	_N_
AL	Birmingham		1	0	1	1	0	1
AL	Flagstaff		0	1	1	1	0	2
AZ	Flagstaff		1	0	1	1	0	3
AZ	Tuscon		0	0	1	0	0	4
AZ	Tuscon		0	0	0	0	0	5
AZ	Tuscon		0	1	0	1	0	6

At _N_ = 1, FIRST.STATE is true since it is the first record; LAST.STATE is false since the second record is also in AL; and both FIRST.CITY and LAST.CITY are true since the second observation contains a different city, Flagstaff. At _N_ = 2, LAST.STATE is true because the next observation comes from AZ; and most importantly, LAST.CITY is also true. Note that the next observation also has CITY="Flagstaff", but since the outer group, STATE, changes, SAS forces the inner group CITY to change. Many beginners spend hours trying to code this situation when SAS handles it automatically. You should be able to convince yourself of the remaining values for the first dot and last dot variables.

The significance of these variables means that the SAS programmer can know when a group begins and ends automatically. This is useful information, but it also means that the DATA step processing can usually be done on a record by record basis. You don't need all the data at once to determine the group end points. You don't even need to look ahead or behind by one record. Consequently many little programming tasks are simplified. Suppose we want split W1 into two data sets, UNIQUE should contain those observations where state and city are unique to one record, and DUPS should contain all observations where there is more than one state city combination. If you study the example you should understand that the following code will do the job.

```
data unique dups ;
```

```
      set w1 ;
      by state city ;
      if first.city and last.city then output unique ;
      else output dups ;
run ;
```

The code is almost trivial, but it is surprising how many programmers trip over this problem when first encountered. The code

```
set w1 w2;
by state city ;
```

is also important. Without the BY statement the entire set W1 is read before the set W2, i.e. the data are concatenated. With the BY statement a group in W1 is read before the corresponding group in W2, i.e. the data are group interleaved. If each group contains at most one observation then the data are interleaved.

Before leaving the SET statement we should look at how SAS provides the end-of-file indicator.

```
set w1 end = eof ;
```

The END= clause is an option of the SET statement. It requests SAS to provide a variable called EOF, my choice, that will be 0 until the last record is read. It is time to go back to the initial section of the ILDS and understand why you should test the value of EOF before the SET statement rather than after.

After the SET statement, MERGE is the most common for SAS input. When the system option MERGENOBY is set to NOWARN a MERGE statement with one data set works just like a SET statement, but this is rarely used in practice. Usually a MERGE statement is used to bring data from multiple data sets together matching on variables specified in a BY statement.

With no BY-statement, SAS does a one-to-one match of records between the data sets listed. If a variable is unique to a short data set, the value of that variable will be set to missing once, at the point where the short data set no longer contributes values. Sometimes a one-to-one merge is needed, but it can always be avoided through the introduction of a sequence variable. I, therefore, tend to see a one-to-merge as a either a mistake or laziness on the part of the programmer. However, many will differ strongly on the question.

With a BY-statement, SAS does a one-to-one match of records within a by group of the data sets listed. It is important to realize that observations are read only once into the PDV. Variables from SAS data sets are set to missing once instead of at the beginning of every iteration of the implied loop. Consequently a value contributed in a BY-group continues to exist in the PDV after a data set runs out of records. However this means that if you change such a value then it will stay changed.

There are two basic important situations. One is that all the data sets listed on the MERGE statement have at most one observation per BY-group. The other is that exactly one of the data sets may have many observations of the BY-group. SAS will issue a warning if another situation arises, but it is not considered an error since one can sometimes construct examples where it might be useful. However most experienced SAS programmers would and should be very suspicious of any situation where this warning does not indicate an error because in almost every situation it is a programmer error.

Since the PDV can hold only one value for a given variable name, the only variables that should be (and must be) in common among the data sets listed on the MERGE statement are those in the BY group. If this situation is met then the data sets of the MERGE statement may be listed in any order.

You can use DROP=, KEEP=, and RENAME= data set options to enforce the rule that there should be no common variable in two or more data sets other than the BY-variables. First consider:

```
set w ( keep = x y z rename = (z=wz) ) ;
```

In this case W may have many variables but only the values of X, Y, and Z will be transferred to the PDV because of the KEEP= option. The RENAME= option specifies the name to be use in the PDV. DROP= has the opposite effect. Sometimes it is handy when almost all of a great many variables would need to be listed using KEEP=. Generally code using KEEP= is easier to read because the reader can then know explicitly what is coming into the PDV from where. DROP= may accomplish the same work, but it does not inform the reader of what variables are in the DATA step. I typically reserve DROP= for dropping names on output data sets and then usually only for trivial loop indices and obviously local temporary variables.

Suppose we have two datasets ENG and HIST reporting the students and grades taking these two classes. It would be natural to have common variables GRADE and STUDID in both data sets and we might want to get both grades for each student taking both classes. The code to solve this problem might be

```
data eng_hist ;
   merge eng  ( keep = studid grade rename = ( grade=enggrade ))
         hist ( keep = studid grade rename = ( grade=histgrade ))
   ;
   by studid ;
run ;
```

Suppose we have MAJOR in addition. Again you would want only one major variable but where should it come from? The very question suggest that the code should verify that the major is consistent and report any errors. Hence:

```
data eng_hist ( keep = studid enggrade histgrade major )
     errs     ( keep = studid emajor hmajor )
   ;
   merge eng  ( keep = studid grade major
                rename = ( grade=enggrade major=emajor ))
         hist ( keep = studid grade major
                rename = ( grade=histgrade major=hmajor))
   ;
   by studid ;
run ;
```

There are legitimate times when many common variables are needed, but those times are an indication that MERGE is the wrong tool for the task. The rule - no variable in common except the BY variables which must be in common - has the advantage that it is simpler to maintain and avoids many of the merge problems one encounters when the rule fails. Unfortunately, the SAS Institute and many SAS programmers have emphasized that PDV is filled by the last data set to contribute a value. This leads to the common pitfall that the first data set may be last when the second data set runs out of matching records. Hence one cannot guarantee that one data set is sufficient to always fill the role of being the last data set to contribute a value.

If you remember three principles:

1. Only one value can be held under one name in the PDV at any one time

2. No common variable between two or more data sets listed in the merge with the exception of the BY variables that must be common to all the data sets

3. No more than one data set with multiple records for any combination of BY values

then all merge errors are a consequence of violating one of these principles or of having a common BY variable which has different properties in two or more of the data sets.

I emphasized the need to study programs where more that one observation is read at a time. One example that comes as a surprise is using BY processing. Suppose you need to output all the records in a group if and only if the value X=1 is achieved at least twice for observations in the group. So you need to know something at the end of each group to be used at the beginning of the group. This can be done with multiple steps, but there are times when it is handy to do it in one step.

```
data wanted ( drop = need2 );
   need2 = 0 ;
   do until ( last.group ) ;
      set w ;
      by group ;
      need2 + ( x=1 ) ;
   end ;
   do until ( last.group ) ;
      set w ;
      by group ;
      if need2 >= 2 then output ;
   end ;
run ;
```

Note that this example depends on the use of two separate buffers for reading W. The UNTIL condition is used because the correct value for LAST.GROUP is not assigned until after the SET statement executes.

SAS I/O is also performed by the MODIFY and UPDATE statements, but these will be left for your investigation.

ARRAYS

One traditional topic we have not yet looked at is the concept of an array. In other languages an array is usually a contiguous section of memory where each element has the same size hence the elements can be manipulated efficiently with pointers either by the processing or by providing the concept in the language so that user can do so. SAS, as a name based language is different. Historically a SAS array is simply a list of variables whose values can be obtained though an index applied to an array name. The most common usage is to replace a sequence of similar statements where only the variable of interest is changed by loop where the variable of interest can be reference by a common array name and either the index of the loop or a simple expression involving the index of the loop.

For example assume SALES is a data set in which the variables SALES1 through SALES53 specify an amount from one of 53 different entities for each observation. Then

```
array sales (53) sales1 - sales53 ;
```

defines an array SALES. After the ARRAY statement one can reference any of the sale amounts by[4]

```
sales[i] or sales(i) or sales{i}
```

We might choose to accumulate these values with

[4] Historically parentheses were used on the IBM mainframe. When SAS was ported to other computers there was a problem with parentheses and brackets were used. Finally braces were introduced as a common symbol. Today all computers running SAS allow all the possibilities.

```
totsales = 0 ;
do I = 1 to dim ( sales ) ;
   totsales + sales[i] ;
end ;
```

This particular example might be better handled by a one line solution making use of the fact that the SUM function allows the keyword OF to indicate a list of variables and that the dash in this context indicates the list SALES1, SALES2, … , SALES53

```
totsales = sum(of sales1-sales53) ;
```

However, add that only sales over some limit should be accumulated and the one line solution falls apart while making the accumulation conditional in an IF statement is a trivial change the loop solution with an array.

In version 6 SAS introduced the key word _TEMPORARY_ to indicate a contiguous block of memory where each element has the same size and the elements have no names so they can only be referenced through the corresponding array name. For example,

```
array names (51) $20 _temporary_ ;
```

Since the variables have no names, they do not belong to the PDV. Moreover they are not set to missing in each iteration of the implied loop.

Missing values

Throughout I have emphasized that SAS sets some variables to missing on each iteration of the ILDS. There are other times when SAS sets a variable to missing without an explicit request. Beginners and even experts can have a hard time stating the precise rules. We now have the machinery to discuss the problem. First we need a classification of variables in the DATA step. There are four disjoint types:

1. Variables defined in an ARRAY statement using _TEMPORARY_
2. Variables assigned by SAS either at the programmers request via an option or implicitly
3. Variables that come from a SAS data set
4. All remaining variables

Nothing yet says that a variable cannot appear to be in more than one category, but the categories have been arranged so that it can be required that a variable can only be in first possible category. For example, X might appear in one DATA step with an assignment, in a SAS data set, and as the name requested by some option, so categories 2, 3, and 4 all appear possible, but the last rule says X must be in category 2 and therefore cannot be in categories 3 and 4. Of course, many of these conflicts would lead to downright dangerous programming practices and some, while not wrong, could be highly misleading to many programmers.

All variables are initialized once before the DATA step begins. SAS uses missing as the initial value when a compile time directive or some other rule does not give a better initial value. For the first three categories SAS does not reset the value of a variable to missing at the beginning of the implied loop of the DATA step. Hence the question of when SAS set the value to missing at the beginning of each iteration of the implied loop pertains only to the catch all others fourth category.

The RETAIN statement allows the programmer to specifically set aside the action of setting to missing at the beginning of each iteration. For example,

```
retain x y z 7 a b "empty" ;
```

says to initialize the variables X, Y, and Z to 7, and the variables A and B to the 5 characters "empty". It also says to drop the automatic activity of resetting the variables to missing on each iteration. Several other commands allow an implied retain. For example we saw that the SUM statement has a built-in implied retain. An ARRAY statement has an option to initialize the variables in an array. Whenever this option is used to initialize an array variable all the variables in the array are automatically retained. I am not aware of any other statements that imply a retain.

Now for the hard part: when does SAS set variable values to missing at times other than the beginning of each iteration of the implied loop in the DATA step? Consider

```
Merge w1 w2 ;
by group ;
```

where W1 has one observation per value of GROUP and W2 may have multiple observations for a value of GROUP. Remember that records are read only once. For a matching GROUP value the variables from W1 go to the PDV once and do not change unless the code makes an explicit change. What should happen when a value of GROUP is only in W2? We do not want the old values for the variables in W1. At the beginning of each BY group SAS resets the variables to missing so that the failure to read an observation does not cause problems.

A similar sort of problem arises with

```
set w1 w2 w3 ;
by group ;
```

What should happen to the values in the PDV of variables from W1 when an observation from W2 or W3 is read? It would be a problem to have the values from the last read observation in W1 still in the PDV. Again SAS saves the situation so that the values in the PDV coming from W1 are set to missing once each time the buffer is switched. However, note that if W2 has many observations for a value of GROUP and there is explicit code to set a value for a variable from W1 to something, this value will stay in the PDV until the buffer is switched for reading from W3 for the same value of GROUP. While this activity seems reasonable and necessary when you think about it, it still comes as a surprise to many even experienced SAS programmers when they change the value of a variable in W1.

Conclusion

The DATA step is a beautiful language. I have covered some of the problems that often cause confusion, but I have not tried to be complete in this task, nor have I tried to be complete in the introduction of any SAS statement.

Contact information

Ian Whitlock

29 Lonsdale Lane
Kennett Square, PA 19348
Ian.Whitlock@comcast.net