Paper S1-13

The SAS Data Step: Where Your Input Matters

Peter Eberhardt, Fernwood Consulting Group Inc., Toronto, ON, Canada

ABSTRACT

Before the warehouse is stocked, before the stats are computed and the reports run, before all the fun things we do with SAS® can be done, the data need to be read into SAS. A simple statement, INPUT, and its close cousins FILENAME and INFILE, do a lot. This paper will show you how to define your input file and how to read through it, whether you have a simple flat file or a more complex formatted file.

INTRODUCTION

The paper will start with a simple list input, similar to most examples you see in SAS documents. From there, it will move on to the more typical flat file input. Here we will look first at INFORMATS when reading data, with particular emphasis on DATE variables. After examining INFORMATS, we will look at positioning within our input line. While looking at positioning, we will first show how to read at specific column locations within a line using the @ notation, followed by the method of 'holding our position' in the input line using the trailing @ notation, allowing us to perform some computational logic on the data read so far. With this small but powerful set of tools we will touch on how to read more complex input files.

COMPONENTS OF DATA INPUT

A SAS dataset is a marvelous way to store your data. It is easy to access, easy to query and can contain lots of information about itself. A raw dataset on the other hand is really none of the above; it is a potential mother lode of information waiting to be mined. So, how do you turn these raw data into the gold you want? A SAS data step and the INPUT statement.

There are three things we need to read raw data in a data step:

- A data source (INFILE)
- An INPUT statement
- A list of fields to input

Before looking at more detail at the INPUT statement, let's look at identifying the data source.

IDENTIFYING THE DATA SOURCE

In order to read raw data into a dataset, we first need a source of data, that is, a file that contains the data. In a SAS data step, we identify the source data file using the INFILE statement. The basic usage of the INFILE statement is:

```
DATA readraw;
    INFILE filereference;
    …
run;
```

SAS being SAS, we know that whatever filereference is, there is probably more than one way to define it. And in this case there is. One method is to include the full filename as in:

```
DATA readraw;
    INFILE 'c:\pharma\rawdata1.dat'; * a windows file system;
    …      * more statements go here;
run;
```

Here we have entered the file name (`rawdata1.dat`), plus the full path to that file (`c:\pharma\`). Although a convenient way to identify the input file, there are potential problems:

- What if the location changes?
- What if the file name changes?
- What if we move to a different platform (say UNIX)?

A second and more robust approach is to use a FILEREF statement to identify the file, and then use the resultant FILEREF with the INFILE statement. The code would look something like:

```
FILENAME rawdata 'c:\pharma\rawdata1.dat'; * a windows file system;
…          * other SAS statements can go here;

DATA readraw;
    INFILE rawdata; * works on any platform;
    …      * more statements go here;
run;
```

There is one other way to identify a data source that you may have seen: the DATALINES (or CARDS) statement. This form of input is often useful when just a small amount of sample data is wanted for test purposes; the data are embedded in the source code (in stream). A data step using the DATALINES statement looks like:

```
DATA readraw;
    …      * SAS statements go here;
    DATALINES;
    …      * the data are here;
    ;;     * ends with double semi-colon ;
run;
```

Note: If you would like to use double quotes in the infile statement to indicate path, make sure they are regular ones. Those "smart quotes", which Microsoft editors like to insert, may not be recognized by SAS.

READING DATA

DATALINES AND SPACE DELIMITED DATA

Once the data source has been identified, the INPUT statement is used to read the data from the file. One of the simplest forms the data source can take is a space-delimited file; this sort of input has the data values separated by one or more spaces. This type of data is commonly used with the DATALINES statement when sample data are simply entered manually. An example would be:

```
DATA results;
    INPUT height weight age;
    LABEL height = 'Height in Inches'
          weight = 'Weight in Pounds'
          age    = 'Age in Years'
    ;
DATALINES;
72 180.25     45
67     145.5 49
63 130 49
;;  * ends with double semi-colon ;
run;
```

In the above example, we just read in numeric data; note that SAS can read in both integer and fractional numbers with no special instructions. However, if there are character data in the input, we need to let SAS know; this is done by supplying an explicit input format of $ after the variable name, as in:

```
DATA results;
    INPUT height weight age name $; * note the $ after name;
    LABEL height = 'Height in Inches'
          weight = 'Weight in Pounds'
          age    = 'Age in Years'
          name   = 'Participant'
    ;
DATALINES;
73  180.25     45 Peter
67     145.5 49    Paul
63 130 49 Mary
;;  * ends with double semi-colon ;
run;
```

Be careful when using the simple $ INFORMAT; SAS will read only the first 8 characters. If the variable has more than 8 characters, we will need to give an explicit length. We will learn more about INFORMATS later.

Although useful for entering test data and for illustrating techniques, you will not normally use DATALINES. The rest of the paper will only use file data; see the appendix for the sample data.

DELIMITED DATA FILES

The examples above demonstrated reading space-delimited data. In the 'real world', delimited data files are common, however the delimiter is most commonly a comma or a tab character. Fortunately, SAS makes it easy to read files delimited by any character simply by specifying the DELIMITER= (or the short form DLM=) option on the INFILE statement. The following example shows how to read a file that is comma-delimited (see Appendix A for the sample input data):

```
FILENAME indata 'C:\pharma\input1.dat';
DATA commadelim;
     INFILE indata DLM=',';
     INPUT height weight age name $;
     LABEL height = 'Height in Inches'
           weight = 'Weight in Pounds'
           age    = 'Age in Years'
           name   = 'Participant'
     ;
run;
```

Reading delimited files is practically seamless, until we have missing data; in a comma-delimited file, that would be denoted by 2 consecutive commas. Again SAS has a solution to overcome this – the DSD (Delimiter-Sensitive Data) option on the INFILE statement; when using the DSD option, we do not need to specify the DLM option unless the delimiter is not a comma.

```
FILENAME indata 'C:\pharma\input2.dat';
DATA dsd;
     INFILE indata DSD;
     INPUT height weight age name $;
     LABEL height = 'Height in Inches'
           weight = 'Weight in Pounds'
           age    = 'Age in Years'
           name   = 'Participant'
     ;
run;
```

A popular way to transfer data between applications is to use a Comma Separated Values (CSV) format. As the name implies, commas separate the variables in the file; in addition, character values are enclosed in double quotes (“); because character variables are enclosed in quotes, it is possible that the variable can contain the delimiter (e.g.

"This, That, and Whatnot"). If you do not have SAS/ACCESS for PC File Formats, this is a common way to transfer data between SAS and EXCEL. When reading CSV files, we use the DSD option on the INFILE statement.

In the examples so far, we have used the simple $ INFORMAT for the name variable. One of the 'GOTCHAS' that often gets me is the fact that, by default, SAS assigns a length of 8 to character data. This means that on input, unless it is told otherwise, SAS will only assign the first 8 characters to a variable regardless of the length in the data. When reading a CSV file, we might try:

```
FILENAME indata 'C:\pharma\input3.dat';
DATA dsd;
     INFILE indata DSD;
     INPUT height weight age name $20.;
     LABEL height = 'Height in Inches'
           weight = 'Weight in Pounds'
           age    = 'Age in Years'
           name   = 'Participant'
     ;
run;
```

However, we probably will not get the results we expect. First, SAS will include the quotes around the character data as part of the variable. Second, if the character variables are shorter than the length specified (in the example above 20), we will get a message in the log something like:

```
NOTE: SAS went to a new row when the INPUT statement reached past the end of a line
```

To properly read character data greater than 8 characters, we need to specify a LENGTH attribute for the variable before the input statement and revert to the simple $ INFORMAT:

```
FILENAME indata 'C:\pharma\input3.dat';
DATA dsdbetter;
     INFILE indata DSD;
     LENGTH name $25;
     INPUT height weight age name $;
     LABEL height = 'Height in Inches'
           weight = 'Weight in Pounds'
           age    = 'Age in Years'
           name   = 'Participant'
     ;
run;
```

If we look at the SAS dataset `dsdbetter,` we'll see it has a subtle difference from the datasets created earlier – the variable `name` is now the first variable in the dataset, not the last as in the previous example; this is because the variable `name` is now referenced before the variables in the input list. For more information on how the order of variables is determined in a SAS data step, see *How SAS Thinks or Why the Data Step Does What It Does* by Neil Howard. To maintain the order, you could move the LABEL statement before the LENGTH statement:

```
FILENAME indata 'C:\pharma\input3.dat';
DATA dsdbetter;
     INFILE indata DSD;
     LABEL height = 'Height in Inches'
           weight = 'Weight in Pounds'
           age    = 'Age in Years'
           name   = 'Participant'
     ;
     LENGTH name $25;
     INPUT height weight age name $;

run;
```

There are a number of other INFILE options that can affect the way the delimited files are read. These options will be addressed in a later section.

COLUMN-ALIGNED DATA

Probably the most common form of input data we will encounter is column-aligned data; that is, data for variables are always in the same columns. For example, `height` is always in columns 1 to 3, `weight` in columns 4 to 10 etc. There are a number of advantages to this format, but the common ones are the ability to read only selected variables in the file as well as the ability to read the variables in any order. In later examples we will use these advantages to selectively read rows from the input.

One way to read column data is to specify the variable and the starting and ending columns as follows:

```
FILENAME indata 'C:\pharma\input4.dat';
DATA column1;
     INFILE indata;
     LABEL height = 'Height in Inches'
           weight = 'Weight in Pounds'
           age    = 'Age in Years'
           name   = 'Participant'
     ;
    INPUT height  1 - 3
           weight  4 - 10
           age    11 - 13
           name $ 14 – 31
    ;
run;
```

This example reads the data in the order they are in the file. Notice that the variables and the column indicators in the data step are aligned; not only does this make the program easy to read, having each variable on a row by itself makes it easy to re-order the way we read the data by simply cutting and pasting lines. To read the variable `name` first, the data step would look like:

```
FILENAME indata 'C:\pharma\input4.dat';
DATA column1;
     INFILE indata;
     LABEL height = 'Height in Inches'
           weight = 'Weight in Pounds'
           age    = 'Age in Years'
           name   = 'Participant'
     ;
    INPUT name $ 14 – 31
           height  1 - 3
           weight  4 - 10
           age    11 - 13
     ;
run;
```

And if we are interested in only some of the variables, we just specify those we want:

```
FILENAME indata 'C:\pharma\input4.dat';
DATA column1;
     INFILE indata;
     LABEL height = 'Height in Inches'
           name   = 'Participant'
     ;
    INPUT name $ 14 – 31
          height  1 - 3
    ;
run;
```

FORMATTED COLUMN-ALIGNED DATA

If the input data is both column-aligned and formatted (e.g. numbers contain commas and/or dollar signs), we can

use a second, and more flexible approach to reading column data - use column pointers and input formats (INFORMATS). In fact, we can use this method whether the input data are formatted or not. A column pointer is the @ sign followed by the starting column (e.g. @11). An INFORMAT tells SAS how the variable is formatted so it can be transformed into the appropriate SAS data value. We already saw a simple example of a character INFORMAT – the simple $. INFORMATS have the following basic form:

<$>informatnameW.<D>

where <> indicate optional components
$ indicates a character INFORMAT (only used for character INFORMATS)
informatname is the name of the INFORMAT
W is the total width
. is the required delimiter
D is the number of decimal places for numeric data

The INFORMAT can be intrinsic to SAS (e.g. comma8., dollar12.2, yymmdd10.) or user-defined. For an in-depth review of INFORMATS, see *(In)Formats (In)Decently Exposed* by Harry Droogendyk.

READING NUMERIC DATA

Numeric INFORMATS are of the simple form of W.D where W is the width of the field to be read and D is the number of places to the right of the decimal. When reading numeric input, SAS will do as we ask, unless the data tell it otherwise. This is explained in the following table:

Input Data	INFORMAT	SAS Data Value
12345	6.0	12345
12.345	6.0	12.345
12345	6.2	123.45
12.345	6.2	12.345

The first and third examples show how SAS does as we asked – it read in the six-byte number and assigned the appropriate number of decimals (0 and 2). The second and fourth examples show that SAS did what the data told it – it read in the six-byte number but since the data already had decimal places, SAS listened to the data and maintained the decimal places. SAS can also read in numeric data that are formatted with commas and dollar signs as is seen in the next table.

Input Data	INFORMAT	SAS Data Value
1,234,567	Comma12.0	1234567
12,345.123	Comma12.0	12345.123
$1,234,567	Comma12.2	12345.67
12.345.123	Comma12.2	12345.123
$1,234,567	Comma12.0	1234567
$12,345.123	Comma12.0	12345.123

Essentially, SAS reads in the data, removes the non-numeric characters (comma, dollar sign), then applies the input format as it did in the normal W.D examples. These examples show that care must be taken when reading in numbers that are formatted. If the numbers are integer values (e.g. $1,234,567) but an INFORMAT with decimal places is applied (comma12.2), the result may not be what we expect.

READING CHARACTER DATA

In most cases, we want to remove the leading spaces in character data, and the normal character INFORMAT $W. does this. However, there are times when we may want to keep the leading spaces; if this is the case, use $charW. instead. The following table shows the difference.

Input Data										Informat	SAS Data Value									
					P	E	T	E	R	$10.	P	E	T	E	R					
					P	E	T	E	R	$CHAR10.						P	E	T	E	R

In both of these examples the name is right justified (i.e. leading spaces) in 10 columns. In the first instance, the format $10. strips the leading blanks, and in the second $CHAR10. the leading blanks are kept.

READING DATE VARIABLES

Date (and date time) variables are often a source of confusion for new SAS users. Internally, SAS stores dates as the number of days since Jan 1, 1960. To display dates in a recognizable form, we apply a date FORMAT to the variable. Examples of intrinsic SAS date INFORMATS are:

Input data	INFORMAT	SAS Data Value
01Nov04	Date7.	16376
01Nov2004	date9.	16376
2004/11/01	yymmdd10.	16376
11/01/04	mmddyy8.	16376
20041101	yymmdd8.	16376
040101	Yymmdd6.	16376

In general, SAS is reasonably forgiving when it comes to the date separator. For example, all of the following would be read correctly using the yymmdd10. INFORMAT:

- 2004/11/01
- 2004-11-01
- 2004 11 01

And because SAS date variables are stored as the number of days since Jan 1, 1960, the variables can be displayed in any format, regardless of how they were read in. The following example shows how to read a date variable in one format (yymmdd10.), but assign a different format for display (date9.). It also shows reading a numeric value formatted as comma9..

```
FILENAME indata 'C:\UserGroups\PHARMA\input5.dat';
DATA date1;
 INFILE indata;
 LABEL height = 'Height in Inches'
       weight = 'Weight in Pounds'
       dob    = 'Date of Birth'
       name   = 'Participant'
       payment = ‘Payment’
 ;
 FORMAT dob date9.;
 INPUT @1  height         3.
       @4  weight         7.
       @11 dob     yymmdd10.
       @22 name         $18.
       @40 payment   comma9.2
;
run;
```

COLUMN POINTERS

We touched on the use of column pointers earlier, showing how to use @n to start reading at column n. This is referred to as absolute positioning since it goes to the column number specified. A column pointer is normally followed by a variable whose value is to be read starting in that column. The table below summarizes different ways to use an absolute column pointer.

Form	Example	Comments
@n	@20	Moves to column 20
@numeric-variable	@C2	C2 is a numeric variable. If C2 = 10, then the pointer would move to column 10
@(expression)	@(c2+10)	If C2 = 10, then move to column 20 (10 + 10)
@’string’	@’NOTE:’	Moves to the position in the line where there is a string ‘NOTE:’. If the string is not on the input line, then none of the record is read.

In addition to absolute column pointers, SAS provides relative column pointers, allowing us to position the pointer to a position relative to the current pointer position. The following table summarizes different ways to use a relative

column pointer.

Form	Example	Comments
+n	+1	Move one column to the right
+numeric-variable	+m2	M2 is a numeric variable. If M2 = -1, then the pointer would move back one column
+(expression)	+(c2+10)	If C2 = 10, then move to the right 20 columns

When SAS reads data into a variable, the column pointer is pointing to the column immediately after the last column read. For example, the following two lines show column positions (Cols>) and the data in a row (Data>)

```
Cols>12345678901234567890123456789
Data>123.45 678.9 here
```

If the following INPUT statement were executed, the comments show where the column pointer is after a read/position change:

```
INPUT @1 amt1 6.2  /* now at col 7, amt1 = 123.45 */
      +1           /* move one position, now at col 8 */
         amt2  5.1 /* now at col 13, amt2 = 678.0 */
      +1           /* move one position, now at col 14 */
         WhereAmI $4.  /* now at end of line, WhereAmI = 'here' */
;
```

When reading column-aligned data, we most commonly have matching pairs of variables and input formats as in:

```
filename in ' C:\UserGroups\PHARMA\datafile.dat';
data test;
infile in ;
input i 8.
      j 4.
      k 8.
      l 4.
      m 8.
      n 4.
      o 8.
      p 4.
      q 8.
      r 4.
      ;
run;
```

However, in this example we can see there is a pattern in the data – the variables are paired; that is, there are 5 sets of an 8-byte input variable followed by a 4-byte variable. When we find patterns like this, we can use some short cuts in our coding. SAS allows us to group our variables and format lists. We group our variables by enclosing the list in parentheses; similarly we put out grouped formats within parentheses. The following code reads in the same data as the example above, but uses groupings instead of explicitly listing each variable/format pair:

```
filename in ' C:\UserGroups\PHARMA\datafile.dat';
data test;
retain c9  9; /* create a variable to show column pointer with a variable */
drop c9;
infile in;
input (i k m o q) (8. +4 ) /* read the 5 8-byte numbers, skipping 4 cols   */
  @c9 (j l n p r) (4. +8 ) /* starting in col 9 read the 4-byte and skip 8 */
    ;
run;
```

The previous example also showed some examples of relative and absolute column positioning. The program reads in the five 8-byte variables by telling SAS to read in the 8 bytes then skipping over the next four columns (8. + 4). It

then positions the pointer column 9 using @c9 and tells SAS to read in the 4 bytes and skip 8 columns (4. +8). I ran a test to read in one million rows using each method and found the first way marginally more efficient on my Win 2000 machine using SAS 9.12. Personally, I prefer the first method since I like to see explicitly how I am reading each variable.

ROW HOLDERS

Up to this point, we have been reading the entire line of data in one INPUT statement. There are many times when we may not want to do this. There can be many reasons not to read the entire input row in one INPUT statement, but there are two common ones. First, we might only be interested in reading all of the data only if something in the data indicates the record is to be kept (say an asterisk in column 1). Second, the layout of the data can vary on a row-by-row basis, and there is an indicator in the row telling what type of format the row has (say a record type in column 12).

Let's look at how we might read a line of data only if the first column contains an asterisk. The first thing we could do is to read in the entire row and then keep only the rows we want:

```
filename indata 'C:\UserGroups\PHARMA\input9.dat';

data conditional1;
     infile indata;
     drop cond;
     input @1 cond $1.
              var1 3. +1
              var2 3. +1
              var3 3. +1
              ;
     if cond = '*';
run;
```

Here is an excerpt from the SAS log:

```
NOTE: The infile INDATA is:
      File Name=C:\UserGroups\PHARMA\input9.dat,
      RECFM=V,LRECL=256

NOTE: 9 records were read from the infile INDATA.
      The minimum record length was 12.
      The maximum record length was 12.
NOTE: The data set WORK.CONDITIONAL1 has 7 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds
```

The SAS log tells us that 9 records were read in and only 7 were kept. If we look at the SAS table and the input file, we see the program worked as it should. This was a small file with only a few variables, so this method may be acceptable. However, if we had to read in thousands and even millions of rows, particularly where we don't want many of the records, we would want to improve our efficiency by only reading the rows we needed. In that case, we could try something like:

```
filename indata 'C:\UserGroups\PHARMA\input9.dat';

data conditional2;
     infile indata;
     drop cond;
     input @1 cond $1.; /* read the first value in the first column */
     if cond = '*';     /* proceed only if it is ad *               */
     input @2 var1 3. +1 /* read the rest of the data              */
              var2 3. +1
              var3 3. +1
              ;
```

```
run;
```

If we look at the SAS log after submitting the above, we will see something like:

```
NOTE: The infile INDATA is:
      File Name=C:\UserGroups\PHARMA\input9.dat,
      RECFM=V,LRECL=256

NOTE: 9 records were read from the infile INDATA.
      The minimum record length was 12.
      The maximum record length was 12.
NOTE: The data set WORK.CONDITIONAL2 has 4 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.35 seconds
      cpu time            0.02 seconds
```

Something different happened. This time we have only 4 records saved, not 7 as before. And if we look at the data saved, we will see some records are from rows that were not to be read. So what happened? INPUT happened. Each time SAS encounters the INPUT statement, it goes to a new input line, unless we tell it not to. And we tell SAS not to go to a new input line by using the trailing @ on the INPUT statement. A trailing @ is simply an @ just before the semicolon. Now our program looks like:

```
filename indata 'C:\UserGroups\PHARMA\input9.dat';
data conditional3;
     infile indata;
     drop cond;
     input @1 cond $1. @; /* hold the line open */
     if cond = '*';
     input @2 var1 3. +1
              var2 3. +1
              var3 3. +1
              ;
run;
```

And now our dataset looks correct. Instead of the sub-setting `if` we used, we could have had a series of conditional `if` statements that each read in data using a different layout.

So the trailing @ holds the line open for us so we can read using more than one INPUT statement. The input line is held open until there is another INPUT statement in the same data step iteration that does not have a trailing @, or the next iteration of the data step. So, what if one input row has multiple records? That is, we want to read the variables for one record, go to the next iteration of the data step and read the variable for the next record from the same line and so on, till we have read all the records on the row. After we have read all the records on the line we go to the next line. To do that we use the double trailing @ (or @@). The following program shows how this works:

```
FILENAME indata 'C:\UserGroups\PHARMA\input10.dat';

DATA multi;
     INFILE indata;
     INPUT x
           y   @@; /* keep the line open till we run out of data */
run;
```

ROW POINTERS – READING FROM MORE THAN ONE ROW

Earlier we saw how to move the pointer within a row to allow us to specify where within a row to start reading data. When looking at how to hold a line open (the trailing @), we saw we could read more than one line of input in an iteration of the data step. Now we will see how to get the INPUT statement to open up multiple rows in each read (I will refer to these rows as a group).

There are two 'parts' to using row pointers. First, on the INFILE statement, we have to use the N= option to specify the maximum number of rows to open on each INPUT statement. Then on the INPUT statement, we use the row pointer (#) to locate the specific row within the group just opened from which we want to read. As with column pointers, there are absolute and relative variations; the following table lists the row pointer options:

Form	Example	Comments
#n	#2	Moves to the second row in the group
#numeric-variable	#C2	C2 is a numeric variable. If C2 = 4, then the pointer would move to fourth row in the group
#(expression)	#(c2+1)	If C2 = 2, then move to the second row in the group
/	/	The relative row pointer. Moves to the next row in the group

There are a couple of important things to note when using row pointers. First, we can always read multiple lines in a data step, either by using multiple INPUT statements (as we saw earlier in the trailing @ section), or by using the relative row movement of /. Using the N= option and row pointers means the INPUT can read multiple lines, in any order. Second, we should always be sure to reference the maximum row from the N= option with a row pointer on the INPUT statement. That is, if we use N=5 as an INFILE option, then we should have a #5 (or relative equivalent) on an INPUT statement. Failing to do so could lead to unpredictable results. The following program shows the use of row pointers:

```
FILENAME indata 'C:\UserGroups\PHARMA\InputMultiLine1.dat';

DATA multiline1;
     /* N=5 allows us to open 5 rows on each input
        line=lp will track the current line in the lp var */
     INFILE indata  N=5 LINE=lp;

     /* read the 5th row first, hold the line
        and print out the data and current line */
     INPUT
        #5 row5 2. data5 $7. @11 ar5 1. @;
     PUT row5= lp=;
     /* read the rest */
     INPUT
        #1 row1 2. data1 $7. @11 ar1 1.
        #2 row2 2. data2 $7. @11 ar2 1.
         ;
     PUT row1= data1= ar1= /
         row2= data2= ar2= /
         row5= data5= ar5= //
            ;
run;
```

Using row pointers is invaluable in cases where each of the records is on multiple input rows, and some key information is a row after the first row. For example, if we have 3 input rows for each of our records, and the third row has a flag telling us the layout of the first two, then using row pointers makes sense. On the other hand, if we will be building the record in the normal row order, then we can just use multiple INPUT statements as the following program shows:

```
FILENAME indata 'C:\UserGroups\PHARMA\InputMultiLine1.dat';

DATA multiline2;
     INFILE indata LINE=lp;

     /* read the rows in order */
     /* note we can still keep track of the current line */

     INPUT row1 2. data1 $7. ;
     PUT row1= lp=;
```

```
      INPUT row2 2. data2 $7. ;
      PUT row2= lp=;

      INPUT row3 2. data3 $7. ;
      PUT row3= lp=;

      INPUT row4 2. data4 $7. ;
      PUT row4= lp=;

      INPUT row5 2. data5 $7. ;
      PUT row5= lp=;
run;
```

DEALING WITH 'BAD DATA'

The examples to this point have shown various methods of reading raw data. They have also always had 'good' data; that is, no errors were encountered. In most environments errors are not unknown, so in this section we will look at some of the errors we might encounter and ways to deal with them. I will break this into two broad approaches, the first dealing with invalid data values and the second dealing with unexpected data rows.

INVALID DATA

When SAS encounters data that does not match what it is expecting during INPUT it will issue an NOTE:, set the offending variable to missing, set the automatic error variable (_ERROR_) to 1, and move along. Below is a sample log where SAS reads in a date variable where there are two dates with 'bad data':

```
429  FILENAME indata 'C:\UserGroups\PHARMA\input11.dat';
430
431  DATA invalid;
432       INFILE indata ;
433
434       /* read the rows in order */
435       /* note we can still keep track of the current line */
436
437       INPUT row 2. EndDate ddmmyy10.;
438       FORMAT EndDate date9.;
439
440  run;

NOTE: The infile INDATA is:
      File Name=C:\UserGroups\PHARMA\input11.dat,
      RECFM=V,LRECL=256

NOTE: Invalid data for EndDate in line 3 3-12.
RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6----+----7---
-+----8----+-
3         3 99/99/9999 12
row=3 EndDate=. _ERROR_=1 _N_=3
NOTE: Invalid data for EndDate in line 4 3-12.
4         4 31/11/2004 12
row=4 EndDate=. _ERROR_=1 _N_=4
NOTE: 4 records were read from the infile INDATA.
      The minimum record length was 12.
      The maximum record length was 12.
NOTE: The data set WORK.INVALID has 4 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.33 seconds
      cpu time            0.02 seconds
```

If we know there will be invalid data in some of our input, we can suppress the warning message by using the modifiers? and ??. Both modifiers suppress the `NOTE: Invalid Data` message. The ?? modifier also resets the automatic error variable to 0, eliminating the error condition flagged because of the invalid data. In both cases the offending variable will still be set to missing, but our log has been cleaned up. It is a good practice to deal with the conditions that lead to log messages since the fewer the messages, the easier it will be to verify a clean program. The program using a format modifier would look like:
FILENAME indata 'C:\UserGroups\PHARMA\input11.dat';

```
FILENAME indata 'C:\UserGroups\PHARMA\input11.dat';

DATA invalid;
     INFILE indata ;

     /* read the rows in order */
     /* note we can still keep track of the current line */

     INPUT row 2. EndDate ?? ddmmyy10.;
     FORMAT EndDate date9.;

run;
```

In our example we need to ask if using the format modifiers is the right solution. And the answer depends on the data and whether what appears to be invalid data is indeed wrong. Referring back to the example, we see that one of the input dates is `99/99/9999`; in many instances a value such as this is used to mean something like 'No End Date'. So although it is an invalid date, in some sense it is a valid value. Simply using the format modifiers will mean we will lose some potentially valuable information. We can do a bit more processing on our data to ensure we extract all we can and still suppress messages by combining the INPUT statement with the INPUT( ) function. Whereas the INPUT function reads from a data file to assign a value to a variable, the INPUT() function reads character input to assign a value to a variable. The basic form of the INPUT() function is:

```
Var = INPUT(charactervalue, format);
```

Now to read in the same data as above, suppress our log message, and deal with the 'no end date' condition, our program would look like:

```
FILENAME indata 'C:\UserGroups\PHARMA\input11.dat';

DATA invalid;
     INFILE indata ;

     /* read the rows in order */
     /* note we can still keep track of the current line */

     INPUT row 2. cEndDate $10.; * read in date as string ;
     DROP cEndDate;              * drop from the dataset  ;
     FORMAT EndDate date9.;      * a format for the date;

     EndDate = input(cEndDate, ?? ddmmyy10.); * convert to date;
     * do see if it is a missing value;
     if EndDate = .
     then
       do;
         if cEndDate = "99/99/9999"
               then EndDate = '31Dec2099'd; * an arbitrary end;
       end;


run;
```

In this example, we read in the date field as a character string, then used the INPUT() function to convert it. Note that

we also used the format modifier to suppress the message to the log and to reset the _ERROR_ variable to 0. After converting, we checked to see if we had a missing value. If the value was missing, a further check on the input string was made to see if it was our 'No End Date' value. When we encounter this 'No End Date' value, we assign a value to the variable. In this example, a future date was assigned. Assigning a valid but well into the future date would allow us to use this variable in date range checks. Although this example used invalid dates, it can be used for virtually any invalid data.

UNEXPECTED DATA ROWS

The 'Unexpected Data Rows' type of error is usually a result of the data not being quite what you expect. And since the input file itself is the root of the problem we will look at options on the INFILE statement .

One type of problem I often encounter is inconsistent date length. That is, sometimes the raw data has a four digit year and sometimes it has a 2 digit year. A similar example may be in dollar amounts. Sometimes you get the actual amount, sometimes the amount in thousands. The ideal way to deal with this is to go back to your source and make the input consistent. But since we do not live in an ideal world we have to look for acceptable workarounds. In the case of a variable length of a field we can get information about the length of the input line and use that to help us determine how to read other variables in the line. The following example shows how to use the INFILE option LENGTH= to capture the length of the input line, then by examining the length of the line, decide how to read the rest of the line:

```
FILENAME indata 'C:\UserGroups\PHARMA\input12.dat';

DATA LineLength;
     INFILE indata LENGTH=ll; * store lenght of line into ll;

    INPUT row 2. @;             * read in a variable, hold line ;
    DROP cEndDate;              * drop from the dataset  ;
    LENGTH cEnddate $10;        * 10 is longest input length;
    FORMAT EndDate date9.;      * a format for the date;
    PUT ll=;
    IF ll = 10
    THEN
      DO;
        INPUT cEndDate $8.; * read in date as string ;
        EndDate = INPUT(cEndDate, ?? ddmmyy8.);
        IF EndDate = .
        THEN
          DO;
            IF cEndDate = "99/99/99"
              THEN EndDate = '31Dec2099'd; * an arbitrary end;
          END;
      END;
    ELSE IF ll = 12
      THEN
        DO;
          INPUT cEndDate $10.; * read in date as string ;
          EndDate = INPUT(cEndDate, ?? ddmmyy10.);
          IF EndDate = .
          THEN
            DO;
             IF cEndDate = "99/99/9999"
                THEN EndDate = '31Dec2099'd; * an arbitrary end;
            END;
        END;
run;
```

In this example we read in part of the row and, using the trailing @, hold the line open for more input. Once we have read from the line, we can check the length of the current line by looking at the variable `ll` referenced in the LENGTH= option. Knowing the length of the current line we can then read in the variable using the appropriate length and convert using the appropriate INFORMAT. One important piece to note here is that you have to be careful you get the proper length of the character variable you are reading. In the example above we set a length on the

variable to 10 – the longest possible length of the variable.

Earlier we saw the log message

```
NOTE: SAS went to a new row when the INPUT statement reached past the end of a line
```

Basically this message is telling us that SAS ran out of things to read on the current line, so it went to the next line to get more data. With formatted input, this is not likely to be acceptable behaviour. The first thing we need to determine is SAS reading in the whole line with the input statement. Each operating environment has a default line length that SAS uses (256 in Unix and Windows, depends on DCB on the mainframe); if the line is longer than the default you need to tell SAS using the LRECL= option on the INFILE. For example, if the input line is 1000 characters your INFILE statement would look like

```
INFILE indata LRECL=1000;
```

This problem is normally easy to detect since every input line will generate an error. A more subtle problem may come about because spaces at the end of a line are truncated. That is, say your input ends with a text field that can have a variable length (say last name). It may be possible that the spaces that would normally be used to pad out the length of the input line are not included, so the line with a last name of JONES will be shorter than a line with a last name of EBERHARDT. In cases where this is the problem upi use the PAD option on the INFILE statement as in:

```
INFILE indata PAD;
```

CONCLUSION

SAS provides us with exceptional flexibility when reading raw data; this paper touched on some of the common aspects of reading in raw data. In addition, it looked at a few of the common problems you may encounter when reading raw data. When reading raw data the old adage 'Know Your Data' is very true. You need to be able to identify it to SAS through the INFILE statement. You need to understand its contents to properly set up your INPUT statement. And you have to be aware of the possible deviations the data may take so you can be prepared to handle them appropriately.

REFERENCES

Howard, Neil "How SAS Thinks or Why the DATA Step Does What It Does"

Droogendyk., Harry " *(In)Formats (In)Decently Exposed* "
Proceedings of the 29th Annua SAS Users Group International Conference

CONTACT INFORMATION

Peter Eberhardt
Fernwood Consulting Group Inc/
299 Laird Dr
Toronto ON M4G 3X5 Canada
Work Phone: (416)429-6705
Email: peter@fernwood.ca
Web: www.fernwood.ca

Lucheng Shao
Donald Bren School of Information
and Computer Sciences
University of California, Irvine
6210 Donald Bren Hall
Irvine, CA, 92617-3425
Email: luchengs@uci.edu

LISTINGS

The following are some sample lines along with the input filename so you can relate back to the paper. For copies of the programs and the input files e-mail me.

INPUT1.DAT

```
73,180.25,45,Peter
67,145.5,49,Paul
63,130,49,Mary
```

INPUT2.DAT

```
73,180.25,,Peter
67,145.5,49,Paul
63,130,49,Mary
```

INPUT3.DAT

```
73,180.25,,"Peter is age less"
67,145.5,49,"Paul"
63,130,49,"Mary"
```

INPUT4.DAT

```
73 180.25 45 Peter is age less
67 145.5  49 Paul
63 130    49 Mary
```

INPUT5.DAT

```
123456789012
 $1,234.122
  1,234,512
  12,123.11
```

DATAFILE.DAT

```
    1  2    3  4    2  3    4  2    3  4
    2  3    4  5    3  4    5  3    4  5
    3  4    5  6    4  5    6  4    5  6
    4  5    6  7    5  6    7  5    6  7
```

INPUT9.DAT

```
*123 123 123
*123 123 123
 456 456 456
*123 123 123
```

INPUT10.DAT

```
10 20 10 30 10 40 10 50
20 20 20 30 20 40 20 50
30 20 30 30
40 20
```

INPUTMULTILINE1.DAT

```
1 line 10 1
2 line 20 2
3 line 30 3
4 line 40 4
5 line 50 5
```

INPUT11.DAT

```
1 31/10/2004
2 01/11/2004
3 99/99/9999
4 31/11/2004
```

INPUT12.DAT

```
1 31/10/2004
2 01/11/04
3 99/99/9999
4 31/11/04
```