Paper 11840-2016

Can you read this into SAS® for me? "Using INFILE and INPUT to Load Data Into SAS®

Peter Eberhardt, Fernwood Consulting Group Inc.;

Audrey Yeo, Athene

ABSTRACT

With all the talk of "Big Data" and "Visual Analytics" we sometimes forget how important, and often hard, it is to get external data into SAS. In this paper we will review some common data sources such as delimited sources (e.g. CSV) as well as structured flat files and the programming steps needed to successfully load these files into SAS. In addition to examining the INFILE and INPUT statements, we will look at some methods for dealing with bad data. This paper assumes only basic SAS skills, although the topic can be of interest to anyone who needs to read external files.

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.

READING DATA FROM EXTERNAL FILES

INFILE AND INPUT

Data does not always come in SAS datasets. Often times, we have to extract data from many different sources. This includes reading in data from a database such as Access, Excel spreadsheets, or text files.

In order to read in raw data with a data step, we would need to know of the following:

1. Infile – the location of the file, the name of the file, and the type of file (e.g. csv file, text file)
2. Input – the list of fields to be read in

An example of using an INFILE statement to read in the external files and an INPUT statement to list the fields to be read in is shown below:

```
DATA RAW;
   INFILE '\\Global4\...\SAS Global 2015\DATA1.TXT';
   INPUT NAME $ HEIGHT WEIGHT;
RUN;
```

```
15
16         GOPTIONS ACCESSIBLE;
17         DATA RAW;
18            INFILE '\\Global4\Marketingandbrand\Audrey\SAS Global 2015\DATA1.TXT';
19            INPUT NAME $ HEIGHT WEIGHT;
20         RUN;

NOTE: The infile '\\Global4\Marketingandbrand\Audrey\SAS Global 2015\DATA1.TXT' is:
      Filename=\\Global4\Marketingandbrand\Audrey\SAS Global 2015\DATA1.TXT,
      RECFM=V,LRECL=256,File Size (bytes)=29,
      Last Modified=16Mar2015:21:51:31,
      Create Time=16Mar2015:14:02:48

NOTE: 2 records were read from the infile '\\Global4\Marketingandbrand\Audrey\SAS Global 2015\DATA1.TXT'.
      The minimum record length was 13.
      The maximum record length was 14.
NOTE: The data set WORK.RAW has 2 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.10 seconds
      cpu time            0.01 seconds
```

Display 1. Log Output - INFILE

While we are able to read in the flat file using the code above without any issues, we are setting ourselves up for a headache if the code above is imbedded in the middle of a long code. This means that if the location or the name of the raw file changes, we would need to dig through the code to change the INFILE statement.

FILENAME AND FILEREF

Another more dynamic way of coding the example above is as follows:

```
FILENAME RAWINPUT '\\Global4\...\SAS Global 2015\DATA1.TXT';

DATA RAW;
   INFILE RAWINPUT;
   INPUT NAME $ HEIGHT WEIGHT;
RUN;
```

```
18         FILENAME RAWINPUT '\\Global4\Marketingandbrand\Audrey\SAS Global 2015\DATA1.TXT';
19         DATA RAW;
20          INFILE RAWINPUT;
21          INPUT NAME $ HEIGHT WEIGHT;
22         RUN;

NOTE: The infile RAWINPUT is:
      Filename=\\Global4\Marketingandbrand\Audrey\SAS Global 2015\DATA1.TXT,
      RECFM=V,LRECL=256,File Size (bytes)=29,
      Last Modified=16Mar2015:21:51:31,
      Create Time=16Mar2015:14:02:48

NOTE: 2 records were read from the infile RAWINPUT.
      The minimum record length was 13.
      The maximum record length was 14.
NOTE: The data set WORK.RAW has 2 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Display 2. Log Output – FILENAME, FILEREF, and INFILE

Using the FILENAME statement, we create a FILEREF (file reference) name RAWINPUT and tell SAS where the raw file is located at, the name of the file, and the type of file we are reading in.

We then use FILEREF with the INFILE statement to read in the raw file. By placing the FILENAME statement at the beginning of the code, we can change the information easily instead of digging through the whole code to change the INFILE statement.

DATALINES

Another way to read in raw data is to use the DATALINES statement. This way of reading in data is usually used when the data is small or when we want to create a test sample. The code to read in data using a DATALINES statement is shown below:

```
DATA RAW;
   INPUT NAME $ HEIGHT WEIGHT;
   DATALINES;
   ANDREW 68 153
   NATALIE   60     123
   ;
RUN;
```

```
25  16          GOPTIONS ACCESSIBLE;
26  17
27  18          DATA RAW;
28  19             INPUT NAME $ HEIGHT WEIGHT;
29  20             DATALINES;
30
31  NOTE: The data set WORK.RAW has 2 observations and 3 variables.
32  NOTE: DATA statement used (Total process time):
33        real time           0.00 seconds
34        cpu time            0.00 seconds
35
36  23             ;
37
38  24          RUN;
39  25
40  26          GOPTIONS NOACCESSIBLE;
41  27          %LET _CLIENTTASKLABEL=;
42  28          %LET _CLIENTPROJECTPATH=;
43  29          %LET _CLIENTPROJECTNAME=;
44  30          %LET _SASPROGRAMFILE=;
```

Display 3. DATALINES

SPACE DELIMITED DATA

One of the simplest ways of entering data is to separate each data values by one or more spaces, as shown in the code above. This is also known as “list input”.

While we know that SAS is able to read in both numeric and character data values, we need to let SAS know which data values are numeric data and which are character data. This is easily done by placing a dollar sign ($) after the variable names that are going to be character data (e.g. NAME). SAS will assume that variable names without a $ after it will be numeric data.

Even though there is more than one space between each data value in the second line of the data, we are still able to read in the raw data without any issues (as shown in Figure 1 below).

	NAME	HEIGHT	WEIGHT
1	ANDREW	68	153
2	NATALIE	60	123

Figure 1. Dataset Output - RAW

DELIMITED DATA

In addition to space delimited data, we can also read in files with data separated by delimiters other than spaces. It is as easy as adding a DLM= option on the INFILE statement, as demonstrated by the code below:

```
DATA DLM;
   LENGTH NAME $15.;
   INFILE DATALINES DLM='|';
   INPUT NAME $ HEIGHT WEIGHT;
   DATALINES;
   ANDREW|68|153
   NATALIE |  60  |   123
   ESMERALDA |55   |  119
   ;
RUN;
```

	NAME	HEIGHT	WEIGHT
1	ANDREW	68	153
2	NATALIE	60	123
3	ESMERALDA	55	119

Figure 2. Dataset Output - DLM

The pipe sign (|) is used as a delimiter in the example above. As you can see, by setting the DLM= option to '|', we are able to read in data that is separated by the '|' delimiter.

We also need to keep in mind that SAS defaults the length of character variables to 8 characters. Therefore, we would need to include a LENGTH statement to allow character values that are longer than 8 characters to be read in correctly.

The Comma Separated Values (CSV) file is another popular type of raw file. In addition to having the data in the CSV files separated by commas, string values are also enclosed in double quotes. Furthermore, we will get two consecutive commas if there is a missing value.

The code to read in a CSV file is demonstrated below:

```
DATA DSD;
   LENGTH NAME $15.;
   INFILE DATALINES DSD;
   INPUT NAME $ HEIGHT WEIGHT;
   DATALINES;
   "ANDREW",,153
   "NATALIE", 60 ,123
   "ESMERALDA", 55   ,  119
   "NOAH, H",66,175
   ;
RUN;
```

	NAME	HEIGHT	WEIGHT
1	ANDREW	.	153
2	NATALIE	60	123
3	ESMERALDA	55	119
4	NOAH, H	66	175

Figure 3. Dataset Output - DSD

By adding the DSD option on the INFILE statement, we are able to read in CSV files. The DSD option strips away the double quotes from the character value and assigns it to the character variable. It also interprets two consecutive commas as a missing value (as shown in Figure 3 above – HEIGHT for Andrew).DSD option also allows us to keep the comma that is within a string value.

COLUMN ALIGNED DATA

Raw data could also be aligned in columns, meaning that each variable will always be in the same location. The way to read in column aligned data is as follows:

```
DATA COLUMN1;
   INPUT NAME $ 1-10 HEIGHT 11-12 WEIGHT 14-16 ID $ 18-20;
   DATALINES;
ANDREW        153 001
NATALIE    60 123 002
ESMERALDA 55 119 003
NOAH, H    66 175 004
   ;
RUN;
```

	NAME	HEIGHT	WEIGHT	ID
1	ANDREW	.	153	001
2	NATALIE	60	123	002
3	ESMERALDA	55	119	003
4	NOAH, H	66	175	004

Figure 4. Dataset Output – COLUMN1

In order to read in column aligned data, we would need to know the starting and ending column for all the variables. The starting and ending column is then listed on the INPUT statement after the variable names.

Column aligned data allows us to read in only variables that we want. For example, if we only want the name and the weight of that person, we would only read in those two columns as demonstrated below:

```
DATA COLUMN2;
   INPUT NAME $ 1-10 WEIGHT 14-16;
   DATALINES;
ANDREW        153 001
NATALIE    60 123 002
ESMERALDA 55 119 003
NOAH, H    66 175 004
   ;
RUN;
```

	NAME	WEIGHT
1	ANDREW	153
2	NATALIE	123
3	ESMERALDA	119
4	NOAH, H	175

Figure 5. Dataset Output – COLUMN2

Column aligned data also allows us to read in variables in any order we want. For example, if we want the ID information after the name, all we need to do is to switch the order of the variable names in the INPUT statement, like the code shown below:

```
DATA COLUMN3;
   INPUT NAME $ 1-10 ID $ 18-20 HEIGHT 11-12 WEIGHT 14-16;
   DATALINES;
ANDREW       153 001
NATALIE   60 123 002
ESMERALDA 55 119 003
NOAH, H   66 175 004
   ;
RUN;
```

	NAME	ID	HEIGHT	WEIGHT
1	ANDREW	001	.	153
2	NATALIE	002	60	123
3	ESMERALDA	003	55	119
4	NOAH, H	004	66	175

Figure 6. Dataset Output – COLUMN3

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
26Apr15	Date7.	20204
26Apr2015	date9.	20204
2015/04/26	yymmdd10.	20204
04/26/14	mmddyy8.	20204
20150426	yymmdd8.	20204
150426	Yymmdd6.	20204

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:

- 2015/04/26
- 2015-04-26
- 2015 11 26

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..

ABSOLUTE COLUMN POINTERS (@)

Another way of reading in column aligned data is to use the @*n* sign (also known as column pointers) followed by the starting column. For example, @5 tells SAS to go to column 5. The way to read in column aligned data using column pointers is as follows:

```
DATA CPOINTER1;
   INPUT @1  NAME   $10.
         @18 ID     $3.
         @11 HEIGHT 2.
         @14 WEIGHT 3.
         @22 DOB mmddyy10.;
   FORMAT DOB MMDDYY10.;
   DATALINES;
ANDREW       153 001 01/20/1979
NATALIE   60 123 002 08/27/1987
ESMERALDA 55 119 003 04/25/1975
NOAH, H   66 175 004 02/28/1986
   ;
RUN;
```

	NAME	ID	HEIGHT	WEIGHT	DOB
1	ANDREW	001	.	153	01/20/1979
2	NATALIE	002	60	123	08/27/1987
3	ESMERALDA	003	55	119	04/25/1975
4	NOAH, H	004	66	175	02/28/1986

Figure 7. Dataset Output – CPOINTER1

The code above is an example of using absolute column pointers to read in raw data. Notice that we need to list the INFORMATs for each variables. The INFORMAT tells SAS how many columns to read and how to read in the data values for each variables.

RELATIVE COLUMNS POINTERS (+)

In addition to absolute column pointer, we could also use relative column pointers to read in data values as well, as shown in the example below:

```
DATA CPOINTER2;
   INPUT @1     NAME   $10.
         +7     ID     $3.
         +(-10) HEIGHT 2.
         +1     WEIGHT 3.
         +5     DOB    mmddyy10.;
   FORMAT DOB MMDDYY10.;
   DATALINES;
ANDREW       153 001 01/20/1979
NATALIE   60 123 002 08/27/1987
ESMERALDA 55 119 003 04/25/1975
NOAH, H   66 175 004 02/28/1986
   ;
RUN;
```

	NAME	ID	HEIGHT	WEIGHT	DOB
1	ANDREW	001	.	153	01/20/1979
2	NATALIE	002	60	123	08/27/1987
3	ESMERALDA	003	55	119	04/25/1975
4	NOAH, H	004	66	175	02/28/1986

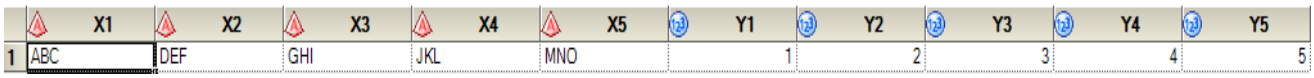
Figure 8. Dataset Output – CPOINTER2

We used the relative column pointer to read in the raw data in the code above. The INPUT statement tells SAS to start reading at column 1, then read in 10 columns and assign that data value to the variable NAME. Next, skip 7 columns ahead (+7), read in 3 columns and assign that data value to the variable ID. This is the followed by moving 10 columns back (+(-10)), read in 2 columns, and assign that data value to the variable HEIGHT.

The usefulness of relative column pointer is better shown in the example below:

```
DATA CPOINTER3;
   INPUT    (X1-X5) ($3. +3)
        @5 (Y1-Y5) (1. +5);

   DATALINES;
ABC 1 DEF 2 GHI 3 JKL 4 MNO 5
   ;
RUN;
```



	X1	X2	X3	X4	X5	Y1	Y2	Y3	Y4	Y5
1	ABC	DEF	GHI	JKL	MNO	1	2	3	4	5

Figure 9. Dataset Output – CPOINTER3

We see a pattern in the DATALINES in the example above where we have alternating 3 character string values and alternating 1 character numeric values. Instead of typing out everything into the INPUT statement as show below:

```
X1 $3. Y1 1. X2 $3. Y2 1. X3 $3. Y3 1. X4 $3. Y1 4. X5 $3. Y5 1.
```

We were able to simplify the code using a combination of absolute column pointer and relative column pointer. Notice that we do not have a @1 to tell SAS to start reading in from column 1. SAS starts reading from column 1 unless specified otherwise.

The INPUT statement above tells SAS to start reading in from column 1, and skip 3 columns (+3) after reading in 3 character value ($3.). This is to be done for X1 through X5. Then, start reading in from column 5, skip 5 columns (+5) after reading in 1 column (1.).

MORE COLUMN POINTERS

We touched on the use of column pointers earlier, showing how to use absolute pointer @n to start reading at column n and relative pointer +n to move n spaces. Both absolute and relative pointers can be more dynamic, as the tables below shoe

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.

Absolute pointers

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

Relative pointers

MULTIPLE INPUT STATEMENTS AND SINGLE TRAILING @

Sometimes we want to split a file into multiple files. The easiest way to do this is to read in the whole file, then split the datasets into the different datasets, as shown below:

```
DATA SPLIT;
   INPUT @1  NAME    $10.
         @18 ID      $3.
```

```
            @11 HEIGHT 2.
            @14 WEIGHT 3.
            @22 DOB mmddyy10.
            @33 GENDER $1.;
      FORMAT DOB MMDDYY10.;
      DATALINES;
ANDREW        153 001 01/20/1979 M
NATALIE    60 123 002 08/27/1987 F
ESMERALDA 55 119 003 04/25/1975 F
NOAH, H    66 175 004 02/28/1986 M
      ;
RUN;

DATA MALE FEMALE;
    SET SPLIT;
    IF GENDER = 'M' THEN OUTPUT MALE;
          ELSE OUTPUT FEMALE;
RUN;
```

	NAME	ID	HEIGHT	WEIGHT	DOB	GENDER
1	ANDREW	001	.	153	01/20/1979	M
2	NATALIE	002	60	123	08/27/1987	F
3	ESMERALDA	003	55	119	04/25/1975	F
4	NOAH, H	004	66	175	02/28/1986	M

Figure 10. Dataset Output - SPLIT

	NAME	ID	HEIGHT	WEIGHT	DOB	GENDER
1	ANDREW	001	.	153	01/20/1979	M
2	NOAH, H	004	66	175	02/28/1986	M

Figure 11. Dataset Output - MALE

	NAME	ID	HEIGHT	WEIGHT	DOB	GENDER
1	NATALIE	002	60	123	08/27/1987	F
2	ESMERALDA	003	55	119	04/25/1975	F

Figure 12. Dataset Output - FEMALE

In the example above, we have a raw file that contains information for both genders, male and female. To split the raw file into two separate datasets, we first read in the raw file and then subset the dataset into two different datasets.

Instead of creating a dataset and then splitting the dataset, we can split the dataset while reading the raw file, thus eliminating the need to create the full dataset. The code to do that is demonstrated below:

```
DATA MALE FEMALE;
   INPUT @33 GENDER $1. @;
   IF GENDER = 'M' THEN DO;
      INPUT @1  NAME   $10.
            @18 ID     $3.
            @11 HEIGHT 2.
            @14 WEIGHT 3.
            @22 DOB mmddyy10.;
            FORMAT DOB MMDDYY10.;
      OUTPUT MALE;
   END;
```

```
      ELSE DO;
         INPUT @1  NAME    $10.
               @18 ID      $3.
               @11 HEIGHT 2.
               @14 WEIGHT 3.
               @22 DOB mmddyy10.;
               FORMAT DOB MMDDYY10.;
      OUTPUT FEMALE;
   END;
   DATALINES;
ANDREW       153 001 01/20/1979 M
NATALIE   60 123 002 08/27/1987 F
ESMERALDA 55 119 003 04/25/1975 F
NOAH, H   66 175 004 02/28/1986 M
   ;
RUN;
```

	NAME	ID	HEIGHT	WEIGHT	DOB	GENDER
1	ANDREW	001	.	153	01/20/1979	M
2	NOAH, H	004	66	175	02/28/1986	M

Figure 13. Dataset Output - MALE

	NAME	ID	HEIGHT	WEIGHT	DOB	GENDER
1	NATALIE	002	60	123	08/27/1987	F
2	ESMERALDA	003	55	119	04/25/1975	F

Figure 14. Dataset Output - FEMALE

In the example above, using the first INPUT statement, we start by telling SAS to go to column 33 and read the single character value. Note that there is a single trailing @ at the end of the INPUT statement. The trailing @ tells SAS to stay on the same data line.

It then checks to see if GENDER is male. If it is male, we have a second INPUT statement to read in the rest of the data line; format the DOB into a readable date; and output the information into the dataset named MALE. If GENDER is female, we have a third INPUT statement to read in the rest of the data line and output the information into the dataset named FEMALE. Note that the second and third INPUT statements are the same.

So, by using multiple INPUT statements and a single trailing @, we are able to read a data line, stay on that same data line, go through a conditional check, and output the information accordingly. Without the single trailing @ to hold the data line open at the end of the first INPUT statement, we would not be able to split the data correctly.

DOUBLE TRAILING @ (@@)

Now that we know that a single trailing @ holds a data line open so that we can stay on that same data line and read in additional information (until SAS encounters another INPUT statement that does not have a trailing @ or that iteration of the data step ends), it only makes sense to go on to the next level, the double trailing @ (or @@). The @@ allows us to hold the same data line until all the record on that data line has been read before it goes to on to the next data line. In order words, @@ allows us to create multiple observations from one data line, as demonstrated below:

```
DATA DOUBLE;
   INPUT X Y @@;
   DATALINES;
1 2 3 4
5 6
```

```
;
RUN;
```

	X	Y
1	1	2
2	3	4
3	5	6

Figure 15. Dataset Output - DOUBLE

READING COLUMN DATA FROM MULTIPLE ROWS

There are times where a data line in a raw file expands to more than one row. This means that we would need to read in multiple rows of data lines in order to create one observation. The code below demonstrates how we can use row pointers (#) to do that:

```
DATA RPOINTER1;
   INPUT #1 @1  NAME   $10.
            @11 DOB    mmddyy10.
          #2 @11 GENDER $1.
            @13 HEIGHT 2.
            @16 WEIGHT 3.;
   FORMAT DOB MMDDYY10.;
   DATALINES;
ANDREW    01/20/1979
          M    153
NATALIE   08/27/1987
          F 60 123
ESMERALDA 04/25/1975
          F 55 119
NOAH, H   02/28/1986
          M 66 175
   ;
RUN;
```

	NAME	DOB	GENDER	HEIGHT	WEIGHT
1	ANDREW	01/20/1979	M	.	153
2	NATALIE	08/27/1987	F	60	123
3	ESMERALDA	04/25/1975	F	55	119
4	NOAH, H	02/28/1986	M	66	175

Figure 16. Dataset Output – RPOINTER1

By using the absolute row pointer (#), we are able to tell SAS to read in NAME and DOB from the first data line. Next, we tell SAS to move to the second data line (#2) and read in the GENDER, HEIGHT, and WEIGHT information.

Besides the absolute row pointer, we can also use the relative row pointer (/) to tell SAS to move to the next data line, as demonstrated in the example below:

```
DATA RPOINTER2;
   INPUT #1 @1  NAME   $10.
            @11 DOB    mmddyy10. /
            @11 GENDER $1.
            @13 HEIGHT 2.
            @16 WEIGHT 3.       /
            @1  ID     $3.;
   FORMAT DOB MMDDYY10.;
```

```
   DATALINES;
ANDREW    01/20/1979
          M    153
001
NATALIE   08/27/1987
          F 60 123
002
ESMERALDA 04/25/1975
          F 55 119
003
NOAH, H   02/28/1986
          M 66 175
004
   ;
RUN;
```

	NAME	DOB	GENDER	HEIGHT	WEIGHT	ID
1	ANDREW	01/20/1979	M	.	153	001
2	NATALIE	08/27/1987	F	60	123	002
3	ESMERALDA	04/25/1975	F	55	119	003
4	NOAH, H	02/28/1986	M	66	175	004

Figure 17. Dataset Output – RPOINTER2

As shown in the example above, we are able to use the relative row pointer to tell SAS to move to the next data line and read in the remaining data.

In the event that we do not need the third data line (ID), we can include a #n on the end of the INPUT statement to tell SAS that there are a total of n rows and we only need to read in the first couple rows, as shown in the example below:

```
DATA RPOINTER3;
   INPUT #1 @1  NAME   $10.
            @11 DOB    mmddyy10. /
            @11 GENDER $1.
            @13 HEIGHT 2.
            @16 WEIGHT 3.
         #3;
   FORMAT DOB MMDDYY10.;
   DATALINES;
ANDREW    01/20/1979
          M    153
001
NATALIE   08/27/1987
          F 60 123
002
ESMERALDA 04/25/1975
          F 55 119
003
NOAH, H   02/28/1986
          M 66 175
004
   ;
RUN;
```

	NAME	DOB	GENDER	HEIGHT	WEIGHT
1	ANDREW	01/20/1979	M	.	153
2	NATALIE	08/27/1987	F	60	123
3	ESMERALDA	04/25/1975	F	55	119
4	NOAH, H	02/28/1986	M	66	175

Figure 18. Dataset Output – RPOINTER3

In the example above, we told SAS that there are a total of three data lines (#3) at the end of the INPUT statement. However, we only need to read in the first two data lines (as indicated by the /). Note that the total number of data rows must be the same for each person in order for this to work.

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 '\\Global4\...\SAS Global 2015\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=\\Global4\...\SAS Global 2015\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 '\\Global4\...\SAS Global 2015\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 '\\Global4\...\SAS Global 2015\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 '\\Global4\...\SAS Global 2015\input12.dat;

DATA LineLength;
     INFILE indata LENGTH=ll; * store length 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 you use the PAD option on the INFILE statement as in:

```
INFILE indata PAD;
```

In addition to PAD, we have four other INFILE options that help us deal with rows that do not have enough data values; these options are:

- FLOWOVER: **INFILE inp FLOWOVER**;
- MISSOVER: **INFILE inp MISSOVER**;
- STOPOVER: **INFILE inp STOPOVER**;
- TRUNCOVER: **INFILE inp TRUNCOVER**;

FLOWOVER is the default behavior. The DATA step simply reads the next record into the input buffer, attempting to find values to assign to the rest of the variable names in the INPUT statement. In this case you will see the message

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

in the log. If you do not want SAS to move on to a new row if there are not enough data in the current row, you need to use one of the other INFILE options

MISSOVER prevents the DATA step from going to the next line if it does not find values in the current record for all of the variables in the INPUT statement. Instead, a missing value is assigned to all variables for which there are no input values. If you expect your data to be incomplete, this may be a good option,

STOPOVER causes the DATA step to stop processing if an INPUT statement reaches the end of the current record without finding values for all variables in the statement. If your data should be complete and the process should not continue if the data are incomplete this is a good option.

TRUNCOVER causes the DATA step to assign the raw data value to the variable even if the value is shorter than expected by the INPUT statement. If, when the DATA step encounters the end of an input record, there are variables without values, the variables are assigned missing values for that observation.

CONCLUSION

The purpose of this paper is to show how to use INFILE and INPUT statements to read in some complicated raw files. We have shown many examples and also included additional INPUT options that may be helpful. We've also included a section on what we can do to deal with bad data. We hope that with the combination of these tools that everyone will be able to successfully read in raw files and turn it into actionable datasets that they can use.

REFERENCES

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

Eberhardt, Peter "The SAS DATA Step: Where your Input Matters", *Proceedings of the 30th Annual SAS Users Group International Conference*

Howard, Neil "How SAS Thinks or Why the DATA Step Does What It Does", *Proceedings of the 29th Annual SAS Users Group International Conference*

CONTACT INFORMATION

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

Peter Eberhardt
Fernwood Consulting Group Inc
Toronto, ON, Canada
peter@fernwood.ca
twitter: rkinRobin
WeChat: peterOnDroid

Audrey Yeo
Athene USA
515-342-3759
audreyeo82@yahoo.com