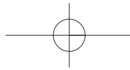


Manipulating Data

CHAPTERS

1 INPUT and INFILE
Building a SAS Data Set from Raw Data

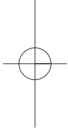
2 Data Recoding
Grouping Data Values

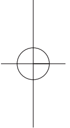
3 SET, MERGE, and UPDATE
Reading and Combining SAS Data Sets

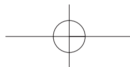
4 Table Lookup Tools
Relating Information from Multiple Sources

5 SAS Functions
Data Translation Tools

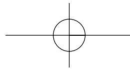
6 SAS Dates
Reading, Writing, and 'Rithmetic with Date Values

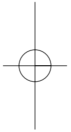


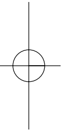


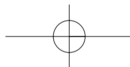


Cody_55126Chap1 6/22/99 4:10 PM Page 2









Chapter 1 INPUT and INFILE

Building a SAS Data Set from Raw Data

EXAMPLES

1 *Reading Raw Data Separated by Spaces* 4
2 *Reading Data Values Separated by Commas or Other Delimiters* 7
3 *Applying an INFORMAT Statement to List Input* 9
4 *Reading Character Values That Contain Blanks* 11
5 *Reading Data Arranged in Columns* 12
6 *Reading Column Data That Require Informats* 17
7 *Reading Two Lines (Records) per Observation* 19
8 *Reading Parts of Your Data More Than Once* 23
9 *Using Informat Lists and Relative Pointer Controls* 25
10 *Reading a Mixture of Record Types in One DATA Step* 28
11 *Holding the Data Line through Multiple Iterations of the DATA Step* 30
12 *Suppressing Error Messages* 34
13 *Reading Data from External Files* 36
14 *Reading in Parts of Raw Data Files* 37
15 *Reading Data from Multiple External Files* 40
16 *Reading Long Records from an External File* 42

Introduction

SAS System procedures can operate only on SAS data sets. Quite often, however, the data that you need to process are in a raw form. The first step is, therefore, to transform the raw data into a SAS data set. The work of manufacturing this is done in a SAS DATA step through the use of a DATA statement. This statement names the SAS data set you are creating. The raw data are then read into the data set via an INPUT statement. The seemingly simple INPUT statement is really a SAS System powerhouse in that it can create a SAS data set from raw data existing in a wide variety of formats. The raw data may exist in a file external to the environment in which the SAS code is being prepared (in which case they are usually referred to by an INFILE statement), or they can be entered instream along with the SAS code by means of a DATALINES statement. SAS software will also recognize the older CARDS statement as the beginning of raw data input. In this first chapter, you will see the power, ease of use, and flexibility of this key DATA statement.

Cody_55126Chap1 6/22/99 4:10 PM Page 4

Example 1

Reading Raw Data Separated by Spaces

FEATURES: **DATA, INPUT, and DATALINES Statements, List Input, Missing Data**

There are a variety of different styles of INPUT code that can be used to read raw data. List input reads data into a SAS data set using a "space delimited" form of data entry. This method can be used when each raw data value is separated from the next one by one or more spaces. This form of data entry has its limitations to be sure, but let us first lay it out via an example before we pick it apart.

Suppose you have the following raw data values and you want to create a SAS data set from them:

ID	HEIGHT	WEIGHT	GENDER	AGE
1	68	144	M	23
2	78	202	M	34
3	62	99	F	37
4	61	101	F	45

This can be done using list input as follows:

Example 1

```
DATA LISTINP;
   INPUT ID HEIGHT WEIGHT GENDER $ AGE;
DATALINES;
1 68 144 M 23
2 78 202 M 34
3 62 99 F 37
4 61 101 F 45
;

PROC PRINT DATA=LISTINP;
   TITLE 'Example 1';
RUN;
```

See Chapter 9, "PROC PRINT," Example 2, for an explanation of TITLE statements. Now, on to the program.

The previous code produces the following output:

Output from Example 1 - Reading Raw Data Separated by Spaces

```
                              Example 1

                OBS  ID  HEIGHT  WEIGHT    GENDER  AGE

                 1    1    68      144        M     23
                 2    2    78      202        M     34
                 3    3    62       99        F     37
                 4    4    61      101        F     45
```

There are several important points to notice about this basic example. Raw data lines are read beginning with the line immediately following the DATALINES statement. You signify the end of your data input with a lone semicolon (called a NULL statement) on the line following your last line of data. Some programmers prefer a RUN statement followed by a semicolon instead. Take your choice.

Next, the INPUT statement lists the variables you wish to create in the same order as the corresponding data values listed below the DATALINES statement. You cannot skip over any data values with this simple form of list input. Later in this chapter we'll demonstrate how to jump around the raw data line when reading in the data.

The SAS System reads data as either character or numeric, and then stores them as such. Numeric data can contain numbers or numeric missing values (see below), while character data can contain numbers, letters, character missing values, and any special characters (e.g. _, #, &). In this example, GENDER is a character variable because it contains the alphanumeric characters, M or F. We indicate that GENDER is a character variable by following it with a dollar sign ($) in the INPUT statement. Without the dollar sign, the program would be expecting numerical values for GENDER (and would get really upset when it encountered M's and F's).

Another point to notice is that the data values have to be separated by at least one blank space as they all are in the previous example. Data values can be separated by more than one space (possibly to improve readability). The lines of data in the following code could be substituted for the original four lines of data with no change in the resulting data set:

```
DATALINES;
1   68 144   M 23
  2    78    202 M    34
   3   62  99 F  37
4   61 101 F 45
```

Messy isn't it? It will, however, work just fine.

The PRINT procedure statements are included in this program so that you can see that the DATA step reads the data as expected. For now, all you need to know about PROC PRINT is that it is a procedure that will print the contents of a SAS data set. Chapter 9, "PROC PRINT," contains more information on PROC PRINT.

Cody_55126Chap1 6/22/99 4:10 PM Page 6

Handling Missing Values

Now suppose you have a missing value of HEIGHT for observation 2, and enter your data as follows:

```
DATALINES;
1 68 144 M 23
2    202 M 34
3 62 99 F 37
4 61 101 F 45
```

The data item is missing so why not just leave it out? Although it looks right, this will get you into big trouble with list input. If you leave the value blank, the program will just look for the next value it finds, after at least one space and read it as the data for the second variable.

In this case, after reading in a value of `2` for ID, the SAS System looks for a value for HEIGHT. It finds `202` and accepts it as a value for HEIGHT. It then attempts to read `M` as the value for WEIGHT. WEIGHT is a numeric variable and cannot have a non-number as a value. The result will be an error message in the log and a missing value for WEIGHT.

Next, `34` is read as the value for GENDER. This is legal because GENDER is a character variable and can have any alphanumeric content. The program is still looking for a value for AGE, so it goes to the next line to read the first data value on that line, `3`, as AGE. Since the value for AGE is the last value the INPUT statement looks for, the program completes building the current observation, brings in the next input line, and starts building the next observation. It's really amazing how wrong things can get when you make one simple innocent mistake.

So how do you solve this problem when reading in list input data? Use a period (.), separated by one or more blanks from surrounding data to indicate that a data value is missing. The period acts as a place holder. It tells the INPUT statement that there is no value to be read here, and to get on about its business. In our example, the correct way to indicate a missing value for HEIGHT for observation 2 is as follows:

```
DATALINES;
1 68 144 M 23
2 . 202 M 34
3 62 99 F 37
4 61 101 F 45
```

Cody_55126Chap1 6/22/99 4:10 PM Page 7

Example 2

Reading Data Values Separated by Commas or Other Delimiters

FEATURES: **INFILE DATALINES options DLM= and DSD, Comma-Delimited List Input**

A fairly common practice is to separate adjacent data values with a comma (,). These comma-delimited data can be easily read by a SAS program as long as you tell the program what to expect. In this case you do so through the use of an INFILE statement as follows (output would be identical to the previous output, with the exception of the title):

Example 2.1

```
DATA COMMAS;
   INFILE DATALINES DLM=',';
   INPUT ID HEIGHT WEIGHT GENDER $ AGE;
DATALINES;
1,68,144,M,23
2,78,202,M,34
3,62,99,F,37
4,61,101,F,45
;

PROC PRINT DATA=COMMAS;
   TITLE 'Example 2.1';
RUN;
```

As you will see later in this chapter, an INFILE statement is usually used to indicate that the raw data are stored in, and are being read from, an external file. The location, or source, or file specification of the external data is named in the INFILE statement, and a number of options that control how the data are read can also be included. By using the reserved filename `DATALINES`, you can apply some of these options to instream data. (Note: the older term CARDS still works even if you use a DATALINES statement to begin your data.) In the present example, the option DLM=',' tells the program to use commas rather than spaces as data delimiters. You may choose any data delimiter you wish with this option. You can even choose multiple characters such as DLM='XX' for your delimiter.

An improvement to the DLM= option was made available in Release 6.07 of the SAS System. A new option, DSD, allows you to treat two consecutive delimiters as containing a missing value. In addition, you can read a text string that contains the delimiter if it is contained in quotes. Further, quoted text strings can also be read. In both cases, the quotes surrounding the text string are not included in the stored value. If the DSD option is used without the DLM= option, the SAS System assumes that you are using commas as your delimiter.

Cody_55126Chap1 6/22/99 4:10 PM Page 8

The following program demonstrates the use of the DSD option:

Example 2.2

```
DATA COMMAS;
   INFILE DATALINES DSD;
   INPUT X Y TEXT;
DATALINES;
1,2,XYZ
3,,STRING
4,5,"TESTING"
6,,"ABC,XYZ"
;

PROC PRINT DATA=COMMAS;
   TITLE 'Example 2.2';
RUN;
```

The output from Example 2.2 is as follows:

Output from Example 2.2 - Reading Data Values Separated by Commas or Other Delimiters

```
                    Example 2.2

                OBS    X    Y    TEXT

                 1     1    2    XYZ
                 2     3    .    STRING
                 3     4    5    TESTING
                 4     6    .    ABC,XYZ
```

Notice that the SAS System treats the consecutive commas as containing a missing value, omits the quotes from the data values, and allows you to include a comma in a text string. The DSD option is probably most useful in reading files produced by spreadsheet and database programs that produce DIF (data interchange format) files.

Cody_55126Chap1 6/22/99 4:10 PM Page 9

Example 3

Applying an INFORMAT Statement to List Input

FEATURES: **INFORMAT Statement, : Format Modifier**

When you use a simple list INPUT statement such as the one in Example 1, the default length for character variables is 8. This means that all character variables are created and stored with a length of 8 bytes. This creates two potential problems. First, if you are reading small length values as you did for GENDER (1 byte), you are wasting storage space using 8 bytes when 1 will suffice. Second, if you read character values longer than 8 bytes, the stored value will be truncated to 8. Another shortcoming of default list input is that data cannot be read that occur in standard configurations, such as dates in MM/DD/YY format. In this example, you can modify your list input by including an INFORMAT statement to define certain patterns or informats in which the raw data occur. Let's read two additional variables, LASTNAME and DOB, which needs special attention (and you can save some storage space as well). Here is the example:

Example 3.1

```
DATA INFORMS;
   INFORMAT LASTNAME $20. DOB MMDDYY8. GENDER $1.;
   INPUT ID LASTNAME DOB HEIGHT WEIGHT GENDER AGE;
   FORMAT DOB MMDDYY8.;
DATALINES;
1 SMITH 1/23/66 68 144 M 26
2 JONES 3/14/60 78 202 M 32
3 DOE 11/26/47 62 99 F 45
4 WASHINGTON 8/1/70 66 101 F 22
;

PROC PRINT DATA=INFORMS;
   TITLE 'Example 3.1';
RUN;
```

Cody_55126Chap1 6/22/99 4:10 PM Page 10

This code produces the following output:

Output from Example 3.1 - Applying an INFORMAT Statement to List Input

```
                              Example 3.1

OBS   LASTNAME        DOB      GENDER   ID    HEIGHT    WEIGHT    AGE

 1    SMITH        01/23/66      M       1      68        144      26
 2    JONES        03/14/60      M       2      78        202      32
 3    DOE          11/26/47      F       3      62         99      45
 4    WASHINGTON   08/01/70      F       4      66        101      22
```

Note that the order of the variables in the output is not the same as the order in the INPUT statement. When the SAS System builds a data set, it stores its variables in the order in which they are encountered in the DATA step. Since the first three variables encountered in the DATA step are LASTNAME, DOB, and GENDER (in the INFORMAT statement), they are the first three variables stored in the SAS data set. The other variables, ID, HEIGHT, WEIGHT, and AGE follow the order in the INPUT statement.

Here the INFORMAT statement gives the following information about the patterns in which some of the raw data elements are found:

- the length of LASTNAME can be up to 20 characters
- the data for DOB are found in MM/DD/YY form
- GENDER is only one character long.

The MMDDYY8. specification after DOB instructs the program to recognize these raw data in MM/DD/YY form and to translate and store them as SAS date values. You also use a FORMAT statement to associate an output pattern, or format with DOB. If you didn't do this, the program would have printed the DOB variable in a SAS date value format. The DOB for SMITH, for example, would have printed as 2214. We cover the fascinating and mysterious world of SAS date values in depth in Chapter 6, "SAS Dates." (Bet you just can't wait!)

You could have accomplished the same goal as above by supplying your informats directly in the INPUT statement. This is called modified list input. You simply follow any variable name you wish to modify by a colon (:) and an informat. The colon tells the program to read the next

Cody_55126Chap1 6/22/99 4:10 PM Page 11

non-blank value it finds with the specified informat. The previous program could have been written as follows, yielding the same output as the previous example (except for the title and the order of the variables):

Example 3.2

```
DATA COLONS;
   INPUT ID LASTNAME : $20. DOB : MMDDYY8.
   HEIGHT WEIGHT GENDER : $1. AGE;
FORMAT DOB MMDDYY8.;
DATALINES;
1 SMITH 01/23/66 68 144 M 26
2 JONES 3/14/60 78 202 M 32
3 DOE 11/26/47 62 99 F 45
4 WASHINGTON 8/1/70 66 101 F 22
;

PROC PRINT DATA=COLONS;
   TITLE 'Example 3.2';
RUN;
```

In this example, the SAS System

- reads the second non-blank value it finds as the value for LASTNAME, but it allows up to 20 characters for the value instead of only the default eight characters.
- reads the next non-blank value as DOB, but it realizes that the data being read is a date that occurs in MM/DD/YY form.
- knows that the data for GENDER always occurs as a 1-byte value, and therefore does not use up an extra 7 bytes to save it.

In Chapter 11, “PROC FORMAT,” we show you how to modify data values as they are read in. Stay tuned.

Example 4

Reading Character Values That Contain Blanks

FEATURE: **& Format Modifier**

The key property of list input is that at least one blank space separates each data value from the next. But what if the data value contains blanks, like a first and last name combination, or a multi-word city name like New York? All is not lost. With a little help, a SAS INPUT statement can read data values that contain one or more single blanks. You do this by following the variable

name that contains the blank spaces with the ampersand (&) format modifier. You can then also use an informat if you wish, as you did with the colon modifier. The rule now is that there must be at least two consecutive blank spaces separating data values. So, in order to read data containing a 25-byte character variable NAME, which could be made up of multiple words, use the following code:

Example 4

```
DATA AMPERS;
   INPUT NAME & $25. AGE GENDER : $1.;
DATALINES;
RASPUTIN    45 M
BETSY ROSS  62 F
ROBERT LOUIS STEVENSON  75 M
;

PROC PRINT DATA=AMPERS;
   TITLE 'Example 4';
RUN;
```

Notice that there are at least two spaces after each complete name. In fact, there are four spaces after `RASPUTIN`. The output for this example follows:

Output from Example 4 - Reading Character Values That Contain Blanks

```
                    Example 4

     NAME                    AGE    GENDER

     RASPUTIN                 45       M
     BETSY ROSS               62       F
     ROBERT LOUIS STEVENSON   75       M
```

Example 5

Reading Data Arranged in Columns

FEATURE: **INPUT Column Specification**

In addition to being able to read raw data values that are separated from each other by one or more spaces, the SAS System provides two methods of reading data values that are uniformly aligned in columns: column input and formatted input. Both provide the ability to read data from

fixed locations in the input record, and both therefore expect to find the data in those locations. Formatted input provides the additional feature of allowing you to read data that occur in other than standard numeric or character formats, but this is one of those "beyond the scope of this book" topics. Column input and formatted input, as well as list input, can be freely intermixed within the same INPUT statement, as you will see in later examples in this chapter.

A column INPUT statement can be used to read lines of data that are aligned in uniform columns. With this method, the name of the variable being read is followed by the column, or column range (starting and ending columns), containing the data for that variable. If you are defining a character variable, the identifying "$" comes before the column numbers. Here is an example.

Example 5.1

```
DATA COLINPUT;
   INPUT ID 1 HEIGHT 2-3 WEIGHT 4-6 GENDER $ 7 AGE 8-9;
DATALINES;
168144M23
278202M34
362 99F37
461101F45
;

PROC PRINT DATA=COLINPUT;
   TITLE 'Example 5.1';
RUN;
```

This code produces the following output, Example 5.1 (identical to that displayed in Example 1 except for the title.)

Output from Example 5.1 - Reading Data Arranged in Columns

```
                    Example 5.1

          OBS  ID  HEIGHT  WEIGHT    GENDER  AGE

            1   1    68      144        M     23
            2   2    78      202        M     34
            3   3    62       99        F     37
            4   4    61      101        F     45
```

In this example, you do not leave any spaces between data values. You can if you wish, but unlike list input, it is not necessary to delimit the data values in any way. The column specifications in the INPUT statement provide instructions as to where to find the data values. Also, notice that we placed the value 99 (for the variable WEIGHT for observation 3) in columns 5-6 rather than in columns 4-5 as we did in previous examples. Numbers placed right-most in a

field are called right adjusted; this is the standard convention for numbers in most computer systems. You could have placed the 99 in columns 4-5 here as well because your instructions were to read the value for AGE anywhere in columns 4-6. The SAS System correctly reads the value even if it is not right adjusted, but it is a good habit to right adjust numbers in general since other computer programs aren't quite as smart as SAS software.

Making Your Program More Readable

Speaking of good habits, let's adopt another one. (If these habits truly yield more productive programming, then they will be easy to make and hard to break.) When using column and formatted input, it's worth the extra effort to code the variables in the INPUT statement in a uniform columnar fashion. It makes for easier code proofreading and maintainability. Here is another version of the previous program that will yield exactly the same output (except for the title):

Example 5.2

```
DATA COLINPUT;
   INPUT ID          1
         HEIGHT    2-3
         WEIGHT    4-6
         GENDER  $   7
         AGE       8-9;
DATALINES;
168144M23
278202M34
362 99F37
461101F45
;

PROC PRINT DATA=COLINPUT;
   TITLE 'Example 5.2';
RUN;
```

Notice that each variable name in the INPUT statement is on a separate line and that the column specifications all line up. This makes for a neater, easier to read, program.

Reading Selected Variables from Your Data

When you read data in columns, you indicate missing values by leaving the columns blank. You also have the freedom to skip any columns you wish and read only those variables of interest

Cody_55126Chap1 6/22/99 4:10 PM Page 15

to you. If, for example, you only wanted to read ID and AGE from the previous data, you could use the following code:

Example 5.3

```
DATA COLINPUT;
   INPUT  ID  1
          AGE 8-9;
DATALINES;
168144M23
278202M34
362 99F37
461101F45
;

PROC PRINT DATA=COLINPUT;
   TITLE 'Example 5.3';
RUN;
```

This code produces the following output:

Output from Example 5.3 - Reading Selected Variables from Your Data

```
                    Example 5.3

                   OBS  ID   AGE

                    1    1    23
                    2    2    34
                    3    3    37
                    4    4    45
```

Reading Values in Different Order

In this example you did not eliminate any data from the lines of data, but you chose to read only part of each line, specifically columns 1 (ID) and 8-9 (AGE). When using column or formatted input, you can read data fields in any order you want to. You do not have to read them in order, from left to right, in ascending column order. You can also read column ranges more than once, or read parts of previously read ranges, or even read overlapping column ranges as different variables. The next set of code shows an example of reading the same data that you have been working with, but by jumping around the input record.

Cody_55126Chap1 6/22/99 4:10 PM Page 16

Example 5.4

```
DATA COLINPUT;
   INPUT  AGE      8-9
          ID       1
          WEIGHT   4-6
          HEIGHT   2-3
          GENDER $ 7;
DATALINES;
168144M23
278202M34
362 99F37
461101F45
;

PROC PRINT DATA=COLINPUT;
   TITLE 'Example 5.4';
RUN;
```

This code produces the following output:

Output from Example 5.4 - Reading Values in Different Order

```
                         Example 5.4

              OBS  AGE  ID  WEIGHT  HEIGHT  GENDER

                1  23    1    144     68       M
                2  34    2    202     78       M
                3  37    3     99     62       F
                4  45    4    101     61       F
```

Notice that the variables exist in the data set COLINPUT, and are therefore displayed in the output, in the same order in which they are read via the INPUT statement.

Cody_55126Chap1 6/22/99 4:10 PM Page 17

Example 6

Reading Column Data That Require Informats

FEATURES: **@ Column Pointer, SAS Informats**

Instead of using starting and ending column numbers to describe the location of the data, you can use the starting column number, the length of the data value (number of columns it occupies), and a SAS informat. A typical data description containing information about the previous collection of data might look like the following:

```
Variable   Starting
  Name      Column   Length  Format      Description
-----------------------------------------------------------
ID            1         3    Numeric     Subject ID
GENDER        4         1    Char        M=Male, F=Female
AGE           9         2    Numeric     Age of subject
HEIGHT        11        2    Numeric     Height in inches
DOB           13        6    MMDDYY6     Date of birth
```

The following code uses pointers and informats to read instream data that occur in the pattern described above.

Example 6

```
DATA POINTER;
   INPUT @1  ID       3.
         @5  GENDER  $1.
         @7  AGE      2.
         @10 HEIGHT   2.
         @13 DOB      MMDDYY6.;
FORMAT DOB MMDDYY8.;
DATALINES;
101 M 26 68 012366
102 M 32 78 031460
103 F 45 62 112647
104 F 22 66 080170
;

PROC PRINT DATA=POINTER;
   TITLE 'Example 6';
RUN;
```

The @ characters (at signs) are called column pointers; they indicate the starting column for an action. When they appear before a variable name in an INPUT statement, they tell the SAS System to go to a certain column. Following the name of the variable, you can use an informat to tell the program how to read the data.

There are many types of SAS informats, but we only use a few common ones here. Numeric variables use an informat of the form *w.d,* where *w* is the width of the field (number of columns) containing the data, and *d* is the number of places to the right of the decimal point in the value. When *d* is omitted, it is assumed to be 0. If the data values actually contain decimal points, the *d* part of the specification is ignored. A minus sign (–) may be included in a negative value, but it must immediately precede the value with no intervening space. The width, *w*, must be large enough to include any decimal points or minus signs found in the data. Character informats are of the form $*w*., where *w* is the width of the field (number of columns) containing the data.

There are a large number of date informats in the SAS System. The one used here, MMDDYY6., instructs the software to read a data value from six columns, the first two being the month, the next two the day of the month, and the last two being the year. If the values contain special characters (typically slashes or dashes) separating the three parts of the date, you can use MMDDYY8. instead.

All SAS informats contain a period (.), either as the last character in the format or immediately preceding the number of decimal places contained in a data value. Omitting this period, like omitting the "sacred semicolon," can, under the right circumstances, cause countless hours of head-scratching while trying to discover why the obtained results are so wrong! It is quite possible to omit a period or semicolon and still have "syntactically correct" code. Of course the results may be pure garbage. Two words to the wise are all that we can offer. BE CAREFUL!!

Output from the previous code in Example 6 is as follows:

Output from Example 6 - Reading Column Data That Require Informats

```
                         Example 6

     OBS     ID    GENDER    AGE    HEIGHT       DOB

      1     101      M        26      68      01/23/66
      2     102      M        32      78      03/14/60
      3     103      F        45      62      11/26/47
      4     104      F        22      66      08/01/70
```

Notice that the values for DOB are printed in MM/DD/YY format. This is accomplished with the line of code, `FORMAT DOB MMDDYY8.` (See the writeup accompanying Example 3.1 for an explanation).

Cody_55126Chap1 6/22/99 4:10 PM Page 19

Reading Multiple Lines of Data per Observation

Example 7

Reading Two Lines (Records) per Observation

FEATURES: **#, / Line Pointers**

You now know how to specify column ranges of raw data when reading values into SAS variables, and you can even jump around within a line of data. But what if a set of data for an observation spans multiple lines (records) on the raw data input file? You can easily tell the SAS System which line contains the next data value to read by using a line pointer (# or /). We will only cover the basic situation here where each observation contains the same number of lines; however, real life situations can get much more complicated where there can be different numbers of records for different observations. We leave this and other advanced tasks as lookup assignments for you, to be completed when the need arises.

Suppose we extend the last example by adding a second line of data per observation. The new data description is as follows:

```
  Variable   Starting
    Name      Column     Length    Format     Description
----------------------------------------------------------------------
Record 1
  ID           1           3      Numeric     Subject ID

  GENDER       4           1      Char        M=Male, F=Female

  AGE          9           2      Numeric     Age of subject

  HEIGHT       11          2      Numeric     Height in inches

  DOB          13          6      MMDDYY6     Date of birth

Record 2
  ID           1           3      Numeric     Subject ID

  SBP          5           3      Numeric     Systolic blood pressure

  DBP          9           3      Numeric     Diastolic blood pressure

  HP           13          3      Numeric     Heart rate
```

Notice that both lines of raw data contain the subject ID number. This is a good policy in general and will aid in data integrity and validity checking. Although you could read the ID number from both records for each subject (with a different variable name for each), and then check one against the other for validity before proceeding to the next observation, you do not do so here. The following code reads the data, two records per observation.

Example 7.1

```
DATA POINTER;
   INPUT #1 @1  ID       3.
            @5  GENDER  $1.
            @7  AGE      2.
            @10 HEIGHT   2.
            @13 DOB      MMDDYY6.
         #2 @5  SBP      3.
            @9  DBP      3.
            @13 HR       3.;
FORMAT DOB MMDDYY8.;
DATALINES;
101 M 26 68 012366
101 120  80 68
102 M 32 78 031460
102 162  92 74
103 F 45 62 112647
103 134  86 74
104 F 22 66 080170
104 116  72 67
;

PROC PRINT DATA=POINTER;
   TITLE 'Example 7.1';
RUN;
```

The #'s in the INPUT statement tell the SAS System which raw data lines to access when reading in values. In this case, the instructions are to obtain values for ID, GENDER, AGE, HEIGHT and DOB from line 1 (#1) for each observation, and to obtain values for SBP, DBP and HR from line 2 (#2) of each observation. Although values for the ID number are present on both records for each observation, you only read them from line 1 in this example.

Output for this code is as follows:

Output from Example 7.1 - Reading Two Lines (Records) per Observation

```
                         Example 7.1

     OBS  ID   GENDER  AGE  HEIGHT   DOB      SBP  DBP HR

       1  101    M      26    68   01/23/66   120  80  68
       2  102    M      32    78   03/14/60   162  92  74
       3  103    F      45    62   11/26/47   134  86  74
       4  104    F      22    66   08/01/70   116  72  67
```

Cody_55126Chap1 6/22/99 4:10 PM Page 21

If the raw data consist of the same number of records per observation, and if all records are to be read for each observation, as is the case in the current example, then instead of explicitly denoting which numbered record to go to for each subset of variables, you can just tell the system to go to the next line after reading the final value from the current line. This is accomplished by using the relative line pointer (/) indicator. The following INPUT statement could be used instead of the previous one, and the output would be identical:

```
INPUT    @1  ID       3.
         @5  GENDER  $1.
         @7  AGE      2.
         @10 HEIGHT   2.
         @13 DOB      MMDDYY6.
       / @5  SBP      3.
         @9  DBP      3.
         @13 HR       3.;
```

In this case, the SAS System begins to read data from the first raw data input record. After reading values for ID through DOB, it moves to the next raw data record for three more variables (SBP, DBP, HR). These variables are all part of the same observation being built in the SAS data set. When the system finishes building the current observation, it advances to the next raw data record and starts to build the next one.

Using the absolute line pointer specifications in Example 7.1 is preferable to using the relative control shown above. Absolute line pointer control allows you to go forward or backward, makes it less likely to miscount the number of slashes, and makes for code that is easier to read. We show you the relative line pointer method since you may encounter programs that use it.

Skipping Selected Input Records

And now, one last wrinkle before we abandon the topic of multiple raw data input records per observation. Suppose there are many (or even two) lines of raw data, and you only wish to read from a few lines (or even one) per observation. You might wonder why you should type in extra lines to begin with. One answer is that you may be wrapping SAS code around an existing file of raw data that contains more than you need for the current application, but there are compelling reasons not to reshape the data (extra effort involved, chance of mutilating perfectly good valid data, etc.). How can you not read certain lines? There are two methods. Either use multiple slashes (///) to skip unwanted lines, or explicitly direct the INPUT statement to only the desired lines by using numbered #'s. In either case, if there are unwanted records at the end of each set of raw data lines, they must be accounted for.

Suppose you have a file of raw data consisting of four records per observation, and you only want to read two variables from the second record of the four. The following code accomplishes this.

Cody_55126Chap1 6/22/99 4:10 PM Page 22

Example 7.2

```
DATA SKIPSOME;
   INPUT #2 @1  ID   3.
            @12 SEX $6.
         #4;
DATALINES;
101 256 RED   9870980
101 898245 FEMALE 7987644
101 BIG   9887987
101 CAT 397  BOAT 68
102 809 BLUE  7918787
102 732866 MALE   6856976
102 SMALL 3884987
102 DOG 111  CAR  14
;

PROC PRINT DATA=SKIPSOME;
   TITLE 'Example 7.2';
RUN;
```

The previous INPUT statement instructs the system to go directly to the second available line of input data (#2), read data for two variables from that line (ID, SEX), and then go directly to the fourth line of input (#4). It is essential to include the #4 pointer even though you are not reading any data from line 4. It is needed so that the correct number of lines are skipped, and the program reads the correct line of data for the beginning of each observation. On each iteration of the DATA step, line 2 is read and the pointer then moves to the fourth line.

The previous code yields the following output:

Output from Example 7.2 - Skipping Selected Input Records

```
                    Example 7.2

                  OBS   ID     SEX

                   1   101    FEMALE
                   2   102    MALE
```

As expected, the only data that are read and converted into data set variables, and subsequently printed out, are those for ID and SEX.

Cody_55126Chap1 6/22/99 4:10 PM Page 23

Example 8

Reading Parts of Your Data More Than Once

FEATURES: **@ Pointer Control**

There are often times when it is useful to read the same raw data columns more than once to create different SAS data set variables. You can read the exact same range of columns multiple times and create variables of different types (e.g. numeric, character, date, etc.), or read a subset of a previously read column range into a new variable, or even read overlapping ranges into different variables.

Suppose you are dealing with inventory data where it is important to know in which state a part was manufactured, the weight of the part, and the year in which it was made. Each part has a 14-character part ID made up of four components: a two-character state code, a three-digit part number, a three-digit part weight, and a six-digit manufacture date in MMDDYY format. Each part also has a description and a quantity-on-hand value.

You have to build a SAS data set that includes the following independent variables: PARTID (part ID), ST (state of manufacture), WT (part weight), YR (year of manufacture), PARTDESC (part description), and QUANT (quantity on hand.) PARTID must be read as a character variable because it contains alphanumeric characters (the state abbreviations). You also want ST read separately. WT must be read as a numeric variable because you have to do arithmetic calculations with it (e.g. total weight for all pieces in the warehouse). You only need the year at present for YR, so you don't read the entire date.

Once again, there are many ways to accomplish your task. The following code represents one method:

Example 8

```
DATA PARTS;
   INPUT @1   PARTID    $14.
         @1   ST         $2.
         @6   WT          3.
         @13  YR          2.
         @16  PARTDESC  $24.
         @41  QUANT       4.;
DATALINES;
NY101110060172 LEFT-HANDED WHIZZER       233
MA102085112885 FULL-NOSE BLINK TRAP     1423
CA112216111291 DOUBLE TONE SAND BIT       45
NC222845071970 REVERSE SPIRAL RIPSHANK   876
;

PROC PRINT DATA=PARTS;
   TITLE 'Example 8';
RUN;
```

PARTID is first read as a 14-byte character variable starting in column 1. By re-reading columns 1 and 2, you obtain a value for the 2-byte character variable, ST. You then read data from within the same column range that you read as character for PARTID and obtain two numeric variables: WT from columns 6-8 and YR from columns 13-14. The INPUT statement completes the process by reading in a 24-byte character variable PARTDESC starting in column 16 and a 4-byte numeric variable QUANT starting in column 41.

The previous code yields the following output:

Output from Example 8 - Reading Parts of Your Data More Than Once

```
                        Example 8

OBS    PARTID        ST   WT  YR    PARTDESC               QUANT

1    NY101110060172  NY  110  72    LEFT-HANDED WHIZZER      233
2    MA102085112885  MA   85  85    FULL-NOSE BLINK TRAP   1,423
3    CA112216111291  CA  216  91    DOUBLE TONE SAND BIT      45
4    NC222845071970  NC  845  90    REVERSE SPIRAL RIPSHANK  876
```

There are actually other ways to accomplish these goals (but there aren't better ways to get basic concepts across.) You could use character substring functions and date functions and PUT and INPUT functions, etc., but let's wait until Chapter 5, "SAS Functions," to meet these powerful tools.

Example 9

Using Informat Lists and Relative Pointer Controls

FEATURES: **Informat Lists, +*n* Relative Pointer Controls**

Repetition, repetition, repetition. Repetitious isn't it? It definitely has its place in certain areas, such as teaching if it's done right, but the whole basis of "computing" in general is to let the machine do the repetitive work, right? Take a look at the following code (typical for a beginning SAS System programmer):

Example 9.1

```
DATA LONGWAY;
   INPUT ID     1-3
         Q1     4
         Q2     5
         Q3     6
         Q4     7
         Q5     8
         Q6     9-10
         Q7     11-12
         Q8     13-14
         Q9     15-16
         Q10    17-18
         HEIGHT 19-20
         AGE    21-22;
DATALINES;
1011132410161415156823
1021433212121413167221
1032334214141212106628
1041553216161314126622
;

PROC PRINT DATA=LONGWAY;
   TITLE 'Example 9.1';
RUN;
```

The objective here is obviously to read data consisting of an ID number, answers to ten questions, and height and age for each subject. There is a better way. The SAS System provides the ability to read a repetitive series of data items by using a variable list and an informat list.

The variable list contains the variables to be read; the informat list contains the informat(s) for these variables. The previous code can be rewritten, using a variable list and an informat list, as follows:

Example 9.2

```
DATA SHORTWAY;
   INPUT ID 1-3
      @4 (Q1-Q5)(1.)
      @9 (Q6-Q10 HEIGHT AGE)(2.);
DATALINES;
1011132410161415156823
1021433212121413167221
1032334214141212106628
1041553216161314126622
;

PROC PRINT DATA=SHORTWAY;
   TITLE 'Example 9.2';
RUN;
```

The INPUT statement here works as follows: after reading in a value for ID, five values are read for variables Q1, Q2, Q3, Q4, and Q5. They are all read with a 1. informat, and they are all contiguous in the raw data. (A small digression is in order. A list of variables, all having the same base and each one having a sequential numeric suffix, can be abbreviated as BASE#-BASE#. In this case, Q1, Q2, Q3, Q4, and Q5, can be written as Q1-Q5. End of small digression.) After the last variable in the list, Q5, is read, a new list is initiated. This one consists of variables Q6-Q10, HEIGHT and AGE. This time they are all read with a 2. informat.

An alternate coding for the previous INPUT statement is:

```
INPUT @1 (ID Q1-Q10 HEIGHT AGE)(3. 5*1. 7*2.);
```

In this case there is only one variable list and one informat list, but the instructions are identical to the last example. The *n** denotes how many times a particular informat is to be used; `5*1.` means, "use the 1. informat five times, i.e. for the next five variables."

Aside from the title, either of the previous two sets of code produces the following output.

Output from Example 9.2 - Using Informat Lists and Relative Pointer Controls

```
                         Example 9.2

OBS  ID   Q1  Q2  Q3  Q4  Q5  Q6  Q7  Q8  Q9  Q10  HEIGHT  AGE

1    101  1   1   3   2   4   10  16  14  15   15    68     23
2    102  1   4   3   3   2   12  12  14  13   16    72     21
3    103  2   3   3   4   2   14  14  12  12   10    66     28
4    104  1   5   5   3   2   16  16  13  14   12    66     22
```

When using informat lists, the raw data do not have to be all contiguous and of the same type, as they are in the last example. Any informats can be used and intermixed, and blank spaces can be skipped with the relative +*n* pointer controls. These pointer controls can be used anywhere in an INPUT statement and merely move the column pointer forward or backward +(-*n*) the designated (*n*) number of spaces. Since there is no negative pointer control available, in order to go backwards, you actually have to advance a negative amount. Silly looking at first, but it is logical.

Suppose your 10 questions occur in five pairs, each pair consisting of a numerically answered question and then a characterly answered question (not really sure about “characterly”, but you get the point.) Suppose further that all pairs are separated from other pairs, and from other variables, by two spaces. The following code handles this situation:

Example 9.3

```
DATA PAIRS;
   INPUT @1 ID 3.               ❶
    @6  (QN1-QN5)(1. +3)        ❷
    @7  (QC1-QC5)($1. +3)       ❸
    @26 (HEIGHT AGE)(2. +1 2.); ❹
DATALINES;
101  1A  3A  4B  4A  6A  68 26
102  1A  3B  2B  2A  2B  78 32
103  2B  3D  2C  4C  4B  62 45
104  1C  5C  2D  6A  6A  66 22
;

PROC PRINT DATA=PAIRS;
   TITLE 'Example 9.3';
RUN;
```

OK, what’s happening here? It’s really pretty straightforward. The INPUT statement performs the following tasks:

❶ Start at column 1 and read a 3-byte numeric field into variable ID.

Cody_55126Chap1 6/22/99 4:11 PM Page 28

❷ Go to column 6 and repeat the following five times: read a 1-byte numeric field into a variable and then move forward 3 columns from the current position to get ready for the next variable in the list. Name the variables QN1-QN5.
❸ Go back to column 7 and repeat the following 5 times: read a 1-byte character field into a variable and then move forward 3 columns to get ready for the next variable in the list. Name the variables QC1-QC5.
❹ Go to column 26 and read a 2-byte field into the numeric variable HEIGHT. Advance the column pointer 1 column, and read another 2-byte field into the numeric variable AGE.

That's it. Powerful and efficient. The resulting output is as follows:

Output from Example 9.3 - Using Informat Lists and Relative Pointer Controls

```
                              Example 9.3

OBS  ID  QN1  QN2  QN3  QN4  QN5  QC1  QC2  QC3  QC4  QC5  HEIGHT  AGE

1   101    1    3    4    4    6    A    A    B    A    A      68   26
2   102    1    3    2    2    2    A    B    B    A    B      78   32
3   103    2    3    2    4    4    B    D    C    C    B      62   45
4   104    1    5    2    6    6    C    C    D    A    A      66   22
```

The key to using variable and informat lists is patterns. If you can arrange your data in repeating patterns, then these repetitions can be put to your advantage. Look for them.

Example 10

Reading a Mixture of Record Types in One DATA Step

FEATURES: **Reading Data Conditionally, @ (Single Trailing At Sign)**

Consider the following situation: you've been given a set of raw data which are a combination of data lines from different sources. They all contain the same data fields, but they are in different positions in each raw data line depending on the source of the data. There is an identifying value in each observation that denotes the source of the data and, therefore, the format of the data values for that observation.

This is not at all an unusual situation, and it is one that the SAS System can handle readily. By using a trailing @, the INPUT statement gives you the ability to read a part of your raw data line, test it, and then decide how to read additional data from the same record.

Cody_55126Chap1 6/22/99 4:11 PM Page 29

Background: How a DATA Step Builds an Observation

Before you proceed further, you have to know a little about how a DATA step builds an observation in a SAS data set and how an INPUT statement operates with multiple lines of raw data. A DATA step begins when the DATA keyword is encountered, and ends when a DATALINES statement, a RUN statement, another DATA keyword, or a PROC keyword is encountered. In all the examples so far, each time an INPUT statement executed, a pointer moved to a new record. If, however, you include a single @ at the end of the INPUT statement (before the semicolon), the next INPUT statement in the same DATA step does not bring a new record into the input buffer but continues reading from the same raw data line as the preceding one. At the end of the DATA step an observation is written to the SAS data set (unless you explicitly use an OUTPUT statement somewhere in the DATA step—see Example 15.2 in this chapter and Example 6.1 in Chapter 3). On the next iteration of the DATA step, the pointer moves to the next record and the INPUT statement begins processing again.

Back to Our Reading Mixed Records Example

Now back to our example with mixed records. A 1 in column 20 specifies that your data contain values for ID in columns 1-3, AGE in columns 4-5, and WEIGHT in columns 6-8; a 2 in column 20 specifies that the value for ID is in columns 1-3, AGE is in columns 10-11, and WEIGHT is in columns 15- 17. The following code correctly reads the data:

Example 10

```
DATA MIXED;
   INPUT @20 TYPE $1. @; ❶
   IF TYPE = '1' THEN   ❷
      INPUT ID     1-3
            AGE    4-5
            WEIGHT 6-8;
   ELSE IF TYPE = '2' THEN  ❸
      INPUT ID     1-3
            AGE    10-11
            WEIGHT 15-17;
DATALINES;
00134168           1
00245155           1
003      23   220  2
00467180           1
005      35   190  2
;

PROC PRINT DATA=MIXED;
   TITLE 'Example 10';
RUN;
```

Cody_55126Chap1 6/22/99 4:11 PM Page 30

The program works as follows:

❶ After reading a value for TYPE in the first INPUT statement, the single trailing @ says, "hold the line," that is, do not go to a new data line if you encounter another INPUT statement.
❷ The IF-THEN/ELSE code tests the current value of TYPE and proceeds accordingly. If the value of TYPE is 1, then the program uses the next INPUT statement to read ID, AGE, and WEIGHT.
❸ If TYPE = 2, then an alternate INPUT statement is used.

When a second INPUT statement (one without a trailing @) is encountered, the data line is released and you are ready for the next iteration of the DATA step.

The code produces the following output:

Output from Example 10 - Reading a Mixture of Record Types in One DATA Step

```
                         Example 10

                  OBS    TYPE  ID  AGE  WEIGHT

                   1       1    1   34    168
                   2       1    2   45    155
                   3       2    3   23    220
                   4       1    4   67    180
                   5       2    5   35    190
```

As you can see, all values are assigned to their proper data set variables, regardless of which columns they are read from. Now if you think that a single trailing @ was neat stuff, just wait till the next example.

Example 11

Holding the Data Line through Multiple Iterations of the DATA Step

***FEATURE:* @@ (Double Trailing At Sign)**

If a single trailing @ tells the system to "hold the line", what do you suppose a double trailing @ would instruct it to do? Why, "hold the line more strongly", of course! What does this translate into? Remember that under normal conditions, a complete iteration of the DATA step constructs one observation in a SAS data set from one raw data line. The DATA step then repeats this process, again and again, until there are no raw data lines left to read. Each time an INPUT statement ending with a semicolon (and no trailing @) is executed, the pointer moves to the next record. By using a double trailing @ (@@), you can instruct the SAS System to use multiple

Cody_55126Chap1 6/22/99 4:11 PM Page 31

iterations of the DATA step to build multiple observations from each record of raw data.

An INPUT statement ending with @@ instructs the program to release the current raw data line only when there are no data values left to be read from that line. The @@, therefore, holds an input record *even across multiple iterations of the DATA step*. This is different from the single trailing @, which holds the line for the next INPUT statement but releases it when an INPUT statement is executed in the next iteration of the DATA step.

The next two programs both accomplish the same result; they build identical SAS data sets. Notice that you are using list input in these examples. You can use @@ in other kinds of INPUT code, but it makes the most sense with list input. As a matter of fact, we're hard pressed to think of a non-esoteric situation in which you would want to use @@ with column input. The first following program does not use @@ so that you can see the comparison:

Example 11.1

```
DATA LONGWAY;
   INPUT X Y;
DATALINES;
1 2
3 4
5 6
6 9
10 12
13 14
;

PROC PRINT DATA=LONGWAY;
   TITLE 'Example 11.1';
RUN;
```

Now here is the short way, using @@, with considerably fewer data lines:

Example 11.2

```
DATA SHORTWAY;
   INPUT X Y @@;
DATALINES;
1 2 3 4 5 6
6 9 10 12 13 14
;

PROC PRINT DATA=SHORTWAY;
   TITLE 'Example 11.2';
RUN;
```

Cody_55126Chap1 6/22/99 4:11 PM Page 32

Here's how it works. Data values are read in pairs. Three pairs are read from the first data line, and then the INPUT statement goes to the next data line for more data. The important thing to realize is that, although there are only two raw data lines, the DATA step actually iterates six times, one for each set of variables (X and Y) named on the INPUT statement. The @@ stops the system from going to a new raw data line each time the INPUT statement executes. In effect, all the data values can be thought of as strung out in one continuous line of data. Using @@ causes the DATA step to keep reading from a data line until there are no more data values to read from that record (reaches an end-of-record marker), or until a subsequent INPUT statement (that does not have a single trailing @) executes. Here is the output.

Output from Example 11.2 - Holding the Data Line through Multiple Executions of the DATA Step

```
Example 11.2

OBS   X    Y

 1    1    2
 2    3    4
 3    5    6
 4    6    9
 5   10   12
 6   13   14
```

Extra Caution with Missing Values and @@

Remember what happened way back at the beginning of this monumental chapter when you were missing input data for a single variable and didn't use a period to represent the missing value? Two raw input data lines were incorrectly merged into one very wrong data set observation. Only a small amount of data was affected because each new execution of the DATA step started with a new data line. The system "caught up" with itself and then got back on track. If the same missing data situation were present when using @@, all succeeding values in all succeeding observations would be in error. (This is not to say that one incorrect data value is better than many!)

To illustrate what we mean about compounding errors, suppose you inadvertently omitted the second X value `3` in the previous code and entered the first line of data as `1 2 4 5 6`. From that point on, the program would be "out of whack" and would be reading Y values for X's, and vice versa, until it reached the end of the raw data where it would look in vain for that last Y value.

Cody_55126Chap1 6/22/99 4:11 PM Page 33

The output would be as follows:

Output from Example 11.2 - When a Data Value is Left Out

```
                    Example 11.2

                     OBS   X    Y

                      1    1    2
                      2    4    5
                      3    6    6
                      4    9   10
                      5   12   13
```

Read the SAS Log!

If you just skim the output, everything might look right. Don't stop there! All SAS System jobs are accompanied by a SAS log that documents the processing of the SAS statements and the manipulation of SAS data sets, and presents notification that various procedures were executed. The SAS log accompanying the previous SAS program looks (in part) something like this:

```
1      DATA SHORTWAY;
2         INPUT X Y @@;
3      DATALINES;

NOTE: LOST CARD.
RULE: ----+----1----+----2----+----3----+----4----+---5----+
6      ;
X=14 Y=. _ERROR_=1 _N_=6
NOTE: SAS went to a new line when INPUT statement reached
      past the end of a line.
NOTE: The data set WORK.SHORTWAY has 5 observations and 2
      variables.
NOTE: The DATA statement used 0.06 CPU seconds and 2389K.
```

The `LOST CARD` note in the SAS log is the system's way of telling you that a problem has occurred. In this case, it should lead to an examination of the raw data input values where the problem could be easily found and corrected. Of course this is a simple illustrative example, and real life situations are not this easy to deal with. The moral is abundantly clear. When you see messages like this in the SAS log, *do not ignore them*. The system is trying to tell you something.

By the way, if you had been unlucky enough to omit two values somewhere in the data stream above, the system would never figure it out. Although the two omissions would seem to cancel each other out, all intervening data values in the SAS data set would be wrong. As is true with most SAS System tools, double trailing @'s are very powerful. But you must use them with care.

Cody_55126Chap1 6/22/99 4:11 PM Page 34

Example 12

Suppressing Error Messages

FEATURES: **? and ?? (Single and Double Question Marks)**

There may be situations when your numeric data will intentionally contain character values, such as the characters NA, meaning “Not Applicable”, or other meaningful abbreviations. Although this will have meaning to you, the SAS System will complain about seeing these “invalid” character strings when it is expecting only pure numeric data. When this happens, the SAS log will ordinarily notify you that it has encountered invalid data. If this occurs often enough as a SAS data set is being built, the system log will eventually stop issuing the error messages and notify you that it has stopped, its error message limit having been exceeded. Here is a short program with some invalid data and the accompanying SAS log:

Example 12.1

```
DATA ERRORS;
   INPUT X 1-2
         Y 4-5;
DATALINES;
11 23
23 NA
NA 47
55 66
;
```

Here is the SAS log:

```
1    DATA ERRORS;
2    INPUT X 1-2
3          Y 4-5;
4    DATALINES;

NOTE: Invalid data for Y in line 5 4-5.
RULE: ----+----1----+----2----+----3----+----4----+----5----
5     23 NA
X=23 Y=. _ERROR_=1 _N_=2
NOTE: Invalid data for X in line 7 1-2.
7     NA 47
X=. Y=47 _ERROR_=1 _N_=3
```

```
NOTE: The data set WORK.ERRORS has 4 observations and 2
      variables.
NOTE: The DATA statement used 0.06 CPU seconds and 2385K.

9    ;
```

When the system encounters an invalid value for a variable, it does a number of things: first, it informs you of its discovery and displays the offensive value. Next it assigns a missing value, . , to the offending variable for the current observation. If you know beforehand that your raw data contain values that the system will consider invalid, such as NA for a numeric variable, you may want to avoid the possibly numerous pages of SAS System error messages. You have two choices:

1. You can save paper by placing a single question mark (?) following a variable name to instruct the system to suppress this type of error message while continuing to print the offending line of data to the log.
2. You can save even more paper by using a double question mark (??) to suppress the printing of all error messages as well as the offending data lines. Here is the example with double ??'s being used, along with the resulting SAS log.

The program:

Example 12.2

```
DATA ERRORS;
   INPUT X ?? 1-2
         Y ?? 4-5;
DATALINES;
11 23
23 NA
NA 47
55 66
;
```

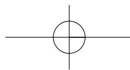
And the SAS log:

```
1    DATA ERRORS;
2       INPUT X ?? 1-2
3             Y ?? 4-5;
4    DATALINES;

NOTE: The data set WORK.ERRORS has 4 observations and 2
      variables.
NOTE: The DATA statement used 0.05 CPU seconds and 2361K.

9     ;
```

Cody_55126Chap1 6/22/99 4:11 PM Page 36



It goes without saying (but as is our wont, we'll say it anyway), that the single and double question marks should be used with caution.

Example 13

Reading Data from External Files

FEATURES: **INFILE and FILENAME Statements**

All of the examples so far have had the raw data included along with the SAS code. You therefore used the DATALINES statement to signal the start of the data. Much of the time your raw data values are stored in a separate file, external to the code. The only changes you need to make to your SAS program are to include an INFILE statement that identifies the location (file specification) where the data values are stored, and omit the DATALINES statement (obviously the data lines themselves are also omitted).

The file specification can take one of two forms when the data reside external to the file containing the SAS code.

Method 1 - Identifying the Filename Directly with the INFILE Statement

With this method, you identify the external data source directly in the INFILE statement by simply enclosing its name in single quotation marks. Suppose the data for Example 13.1 is stored in a file called 'C:\MYDATA\HTWT'. You indicate this in the INFILE statement as follows:

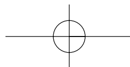
Example 13.1

```
DATA EXTERNAL;
   INFILE  'C:\MYDATA\HTWT';
   INPUT    ID HEIGHT WEIGHT GENDER $ AGE;
RUN;
```

The SAS System then reads the data from the C:\MYDATA\HTWT file, one data line at a time, and applies the INPUT statement to each record as if it had been read instream.

Method 2 - Using a Separate FILENAME Statement to Identify an External File

The other method of identifying the location of the raw data lines is to create a fileref (file reference) by means of a FILENAME statement and then to refer to this fileref in the INFILE statement. Filerefs are not enclosed in quotation marks as are external filenames. Some platforms supply alternate methods for specifying filerefs, such as DD cards in MVS or filedefs in CMS. Suppose, for example, you set up a fileref called OSCAR to refer to the file called



Cody_55126Chap1 6/22/99 4:11 PM Page 37

C:\MYDATA\HTWT. The previous code could be rewritten as follows, making use of the fileref OSCAR:

Example 13.2

```
FILENAME OSCAR 'C:\MYDATA\HTWT';

DATA EXTERNAL;
   INFILE  OSCAR;
   INPUT   ID HEIGHT WEIGHT GENDER $ AGE;
RUN;
```

Why Filerefs are Useful

Older versions of the SAS System required the use of a fileref; external files could not be directly referenced in the INFILE statement. Although this is not the case with Version 6 of the SAS System, there are times when the use of filerefs can greatly enhance your programming in terms of convenience, efficiency, and power. Long filenames (fully qualified mainframe data set names can get quite lengthy) can be abbreviated with short descriptive “handles.” There are also techniques that can be used to access different data sources dynamically without the need for manual recoding (see Example 15.2 in this chapter for an example).

Using an INFILE Statement with Instream Data to Specify Options

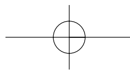
As you will recall in Example 2 (if you don’t, go take a look -- it’s been a while), you used the special DATALINES file specification in the INFILE statement to tell the SAS System that the data lines were included instream with the code. This was only necessary because you wanted to use the DLM= and DSD options of the INFILE statement on your instream data. In order to use an INFILE statement option, you must have an INFILE statement, and every INFILE statement must have a file specification. We now show some examples of other options that give you control over reading data from external files.

Example 14

Reading in Parts of Raw Data Files

FEATURES: **INFILE and OPTIONS Statements, (OBS=, FIRSTOBS=, PAD, and MISSOVER options)**

Up till now you have always read the entire raw data file as input to your program. That is usually what is desired, but there are cases in which only part of the raw data file needs to be read. You can read a specified number of records from the beginning of the file, from the end of the file, or from the middle of the file. When developing a program, it is always a good idea to



work with a small subset of the data until the code is working exactly as desired. The easiest thing to do in the developmental stage is to read only the first *n* records of a file by using the OBS=*n* option in the INFILE statement. The value of *n* is actually the record number of the last record to be read, but if you are starting from the beginning of the file, then *n* will also give you *n* records.

Reading the First 100 Records

Suppose you want to read only the first 100 records of a file called BIGDATA. The following INFILE statement accomplishes this handily:

```
INFILE 'BIGDATA' OBS=100;
```

This could also have been accomplished globally by using an OPTIONS statement of the form:

```
OPTIONS OBS=100;
```

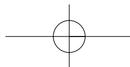
However, the use of an OBS= option in an INFILE statement allows you to control how many records should be processed from individual raw data files if more than one is being read (we'll get to how to read multiple data files in the next example). There are many processing options that can be set via the OPTIONS statement; we touch upon only some of them throughout this book. The OPTIONS statement can appear before or after any DATA step or procedure. The included options take effect from that point onward and remain in effect until they are changed by subsequent OPTIONS statements or you exit out of the SAS System. Be forewarned, don't forget to turn off the OBS= option when you no longer want or need it. To set the observation limit back to the default value, use:

```
OPTIONS OBS=MAX;
```

Skipping the First *n* Records of a File

The INPUT statement begins reading from the first data line encountered in the raw data file by default. This does not have to be the case. You can instruct the SAS System which record to read first by use of the FIRSTOBS= option in the INFILE statement. From that point on, records are read sequentially. Suppose your file were constructed in such a way that the first 100 records were preliminary and not part of the real data set. After using these first 100 records for development as above, you could then start your real analysis with the next record as follows:

```
INFILE 'BIGFILE' FIRSTOBS=101;
```



Cody_55126Chap1 6/22/99 4:11 PM Page 39

Reading a Subset from the Middle of a File

You can also deal with subsets of records from the middle of a raw data file. If, for example, your data set was constructed in such a fashion that you wanted to analyze subsets of 100 records sequentially, you could accomplish this by picking out 100 records to deal with at a time. The FIRSTOBS= option denotes the first record to read, and the OBS= option denotes the last record to read. The thing to remember is that OBS= does not refer to the *number* of records to read, but rather it refers to the *last* record to read. So, to read the second 100 records in the raw data file, code the INFILE statement as follows:

```
INFILE ‘BIGFILE’ FIRSTOBS=101 OBS=200;
```

A Special Caution with Short Records in an External File

If you have missing values at the end of a record of an external file, you have to be very careful. Suppose you have a raw data file called HT_WT that contains values for an ID number, HEIGHT, and WEIGHT. Suppose further, that you are missing either weights, or both heights and weights, for some of the subjects. A typical file might look something like this:

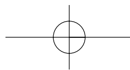
```
File HT_WT

001 68 155
002 64
003
004 72 220
```

Notice that you are missing a value for WEIGHT for observation 2 and both a HEIGHT and WEIGHT for observation 3. If these lines do not contain blanks to indicate that you are missing values for these observations, you need to use an option to indicate this. By the way, just looking at the file on your computer screen may not tell you that there are blanks in these missing positions. You need an editor that shows you where the records end to see if there are blanks or not. To be on the safe side, you should use the PAD or MISSING option in the INFILE statement to be sure that external files with short records are read correctly. The MISSOVER option is used with list input. It instructs the program not to go to a new record if all the variables have not been assigned values when the end of the current record is reached. Instead, variables that have not received values are set to missing. The PAD option is used with fixed record layouts and pads all short records to the length specified by the logical record length (see Example 16).

For example, to read data from the HT_WT file, you use the following INFILE and INPUT statements:

```
INFILE 'HT_WT' PAD;
INPUT ID 1-3 HEIGHT 5-6 WEIGHT 8-10;
```



or, using list input:

```
INFILE 'HT_WT' MISSOVER;
INPUT  ID HEIGHT WEIGHT;
```

Example 15

Reading Data from Multiple External Files

FEATURES: **INFILE, DO UNTIL and DO WHILE Statements, END= and FILEVAR= Options**

Suppose you have two separate files, FILE1 and FILE2, each containing raw data for the same set of variables. You want to read all the data from both files and create a SAS data set from the aggregate. Here is one way to code it.

Example 15.1

```
DATA TWOFILES;
   IF NOT LASTREC1 THEN INFILE 'FILE1' END=LASTREC1;
                   ELSE INFILE 'FILE2';
   INPUT ID AGE WEIGHT;
RUN;
```

The key to this program is the END= option in the first INFILE statement. It works like this: a temporary variable defined by the user is created and given the name specified after the = sign, in this case, LASTREC1. This variable is given a value of `0` each time a record is read from FILE1 except when the last record is read; it is then set to `1`. Each time the DATA step iterates, it checks the value of LASTREC1. The phrase `IF NOT LASTREC1` is equivalent to `IF LASTREC1 NE 1`. When this is true, (this will be true after each record is read from FILE1 except the last one), the first INFILE statement is executed. After the last record from FILE1 is read, LASTREC1 is set to `1`. From then on, `IF NOT LASTREC1` is false, and the second INFILE statement is executed (the ELSE condition).

A More Flexible Approach: Using a Variable to Indicate the External Filename

The previous approach works fine, but there is another way—a little complicated, but pretty neat. You can include the names of the external files you wish to read as values in instream

data lines. You accomplish this by using a FILEVAR= option in the INFILE statement. Here is the previous code rewritten using this alternate method:

Example 15.2

```
DATA TWOFILES;
   INPUT EXTNAME $; ❶
   INFILE DUMMY FILEVAR=EXTNAME END=LASTREC; ❷
   DO UNTIL (LASTREC=1);  ❸
      INPUT ID AGE WEIGHT;
      OUTPUT; ❹
   END;
DATALINES;
FILE1
FILE2
;
```

Here's how this program works. The first INPUT statement ❶ reads a value for the variable EXTNAME from the data line following the DATALINES statement. The first value read is `FILE1`. This is fed to the INFILE statement ❷ via the FILEVAR= option. The system now knows to read data from FILE1. The DO loop ❸ then processes each data line from the current source, FILE1. When the last record is read, the value of LASTREC is set to `1`. The DO loop then stops processing, and the DATA step starts over again, reading a new value (`FILE2`) in to EXTNAME. LASTREC is reinitialized to `0`, and the DO loop starts up again, this time reading from FILE2. When LASTREC is once again set to `1`, the loop stops. Since there are no more raw data lines instream to read, the DATA step ends.

Now, what about that DUMMY file specification and the OUTPUT statement? ❷ The DUMMY is just that. The source of raw data for the INFILE statement is actually the value of the FILEVAR variable (`EXTNAME` in this example.) Since an INFILE statement needs a file specification, we supply a dummy, and call it DUMMY. Pretty clever, huh?

The OUTPUT statement ❹ is necessary because data are only written to the SAS data set being built at the end of each iteration of a DATA step unless an OUTPUT statement is encountered. You need the OUTPUT statement here because this DATA step iterates only twice -- after the last record is read from each raw data file. Without the OUTPUT statement, this DATA step would only write two records to the data set you are building.

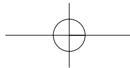
Here is the general syntax for the DO UNTIL statement:

```
DO UNTIL (condition);

   SAS statements

END;
```

The SAS statements in the DO UNTIL block are repeatedly executed until the condition (always placed in parentheses) is true. The DO UNTIL loop always executes at least once, regardless of the logical value of the condition.



Another useful looping structure is DO WHILE. The syntax is identical to DO UNTIL. The statements in the DO UNTIL loop only execute when the condition is true. If the condition is false, the DO WHILE loop does not execute at all.

An easy way to remember the difference between these two looping statements is that the DO UNTIL statement tests the condition at the bottom of the loop while the DO WHILE statement tests at the top.

Example 16

Reading Long Records from an External File

FEATURE: **INFILE Option LRECL**

When reading in external raw data files, you can, for the most part, rely on default values for describing architectural features of the files. You can override defaults when necessary, to describe such features as the record format, block size, or logical record length of the file. These features are described by the INFILE statement options RECFM=, BLKSIZE=, and LRECL= respectively. We only concern ourselves here with the last one, LRECL=. (By the way, this option is named after that famous Spanish bullfighter, "El Rec-el.") You use this option to indicate the maximum record length of the input file and only need it when the record length exceeds your system dependent default. Suppose you have a file, LONGFILE, with 256-byte records and a system default logical record length of 80. You can account for the long record length by coding the INFILE statement as follows:

```
INFILE 'LONGFILE' LRECL=256;
```

You can also use the LRECL= option when you want to read only part of very long records.

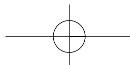
You may code an LRECL= value larger or smaller than the actual record length in your input file and it will work just fine.

Conclusion

Well, we've finally come to the end of this seemingly endless chapter. From here on in, it's relatively easy going, at least in terms of chapter lengths.

You have seen how to read a variety of data arrangements from both instream data lines and external files. You can read data values separated by delimiters such as blanks or commas or data arranged in columns. You can read data from external files and can control which values are to be read and how to use an informat to read data values such as dates. You can read multiple records to create one observation or create multiple observations from one record. Using INFILE options, you can read selected portions of an external file.

As long as we are at the end of the chapter, why not try a few simple exercises to test what you've learned? "Exercise" sounds a bit like work, and that's no fun, so let's call them problems. Actually, that's not much better, but problems usually have solutions, and that's not necessarily so with exercises. Remember, there are no absolutely "correct" solutions, but we do give you



some that will work. If yours are different, test them. They may be just fine. Our solutions to all problems in this book will be found in the Appendix, “Problem Solutions.” Here goes.

Problems

1-1. Write a program to read the raw data stored in an external file called VITAL (shown below), and create a SAS System data set called VSIGNS. VITAL contains the following variables, in the order listed: ID, HR (heart rate), SBP (systolic blood pressure), and DBP (diastolic blood pressure).

```
external file VITAL

A1 68 130 80
B3 101 148 86
C2 . . 72
D1 72 140 88
```

1-2. Redo problem 1-1 to read the comma-delimited raw data file VITALC shown below:

```
external file VITALC

A1,68,130,80
B3,101, 148,86
C2,.,.,72
D1, 72, 140 , 88
```

1-3. Given the raw data lines below, write a program to read these data and create a SAS data set called COLLEGE. The values are separated by one or more spaces, and they represent NAME, TITLE, TENURE (Y or N), and NUMBER (number of classes taught). Notice that some of the names are more than eight characters long.

```
Sample data for Problem 1-3

Stevenson Ph.D. Y 2
Smith Ph.D. N 3
Goldstein M.D. Y 1
```

1-4. This example is similar to Problem 1-3 except the values for NAME may include a single blank. The name is separated from the other values by at least two blanks. Modify the program to take this into account. Some sample data are shown below:

```
Sample data for Problem 1-4

George Stevenson Ph.D. Y 2
Fred Smith Ph.D. N 3
Alissa Goldstein M.D. Y 1
```

Cody_55126Chap1 6/22/99 4:11 PM Page 44

1-5. Given the raw data file description below, write a program to read this data file. Use starting and ending columns. Create a SAS data set called RESPOND. Use an INFORMAT statement if needed. A sample set of data is shown for you to use to test your program.

```
              File FIRE

Variable     Starting  Ending
 Name         Column   Column  Format    Description
---------------------------------------------------------
CALL_NO         1         3    Numeric   Call number
DATE            5        12    MM/DD/YY  Date of service
TRUCKS          14       15    Numeric   Number of trucks
ALARM           17       17    Numeric   Number of alarms

Sample data (in file FIRE)

001 10/21/94 03 2
002 10/23/94 01 1
003 11/01/94 11 3
```

1-6. Given the raw data file description below and the same sample data as in Problem 1-5, write a SAS System DATA step to read this data file, and create a SAS data set called RESPOND. Use formatted input; do not use ending columns.

```
              File FIRE

Variable     Starting
 Name         Column   Length  Format    Description
---------------------------------------------------------
CALL_NO         1         3    Numeric   Call number
DATE            5         8    MM/DD/YY  Date of service
TRUCKS          14        2    Numeric   Number of trucks
ALARM           17        1    Numeric   Number of alarms
```

Cody_55126Chap1 6/22/99 4:11 PM Page 45

1-7. A data file contains a seven-character ID, a quantity, and a price. The first two characters of the ID represent a factory number, and the last two characters represent state abbreviations. The column specifications and some sample data values are shown. Write a program to read these data instream and produce a SAS data set called FACTORY.

```
                     File Inventory

Variable      Starting
 Name          Column     Length   Format
--------------------------------------------------------
    ID             1        7      Character
 QUANTITY          8        2      Numeric
  PRICE           10        7      Numeric (contains
                                   dollar sign and comma)

13AB2NY44   $123
22XXXCT88 $1,033
37123TX11$22,999
```

Hint: Use the DOLLAR7. informat to read PRICE.

1-8. You have a Social Security number in columns 1-11 of a file, followed by one space, followed by ten 3-digit scores. Using a variable list and an informat list, write a SAS program to read the sample data below, and create a SAS data set called SCORES. Name the 10 scores SCORE1 to SCORE10.

```
           Sample data for Problem 1-8

123-45-6789 100 98 96 95 92 88 95 98100 90
344-56-7234  69 79 82 65 88 78 78 92 66 77
898-23-1234  80 80 82 86 92 78 88 84 85 83
```

Cody_55126Chap1 6/22/99 4:11 PM Page 46

1-9. You have collected four systolic and four diastolic blood pressure readings (the higher and lower numbers in a reading such as 120/80, respectively) over a period of four months. They are recorded as follows:

```
Variable     Start-End
Name         Column
----------------------
  SBP1         1-3
  DBP1         4-6
  SBP2         7-9
  DBP2        10-12
  SBP3        13-15
  DBP3        16-18
  SBP4        19-21
  DBP4        22-24
```

Write a program that uses absolute and relative pointers to read the sample data below. Create a SAS data set called PRESSURE.
Hint: Read all the SBP's first, then go back and read the DBP's.

```
Sample data for Problem 1-9

120 80122 84128 90130 92
140102138 96136 92128 84
122 80122 80124 82122 78
```

1-10. You are given a raw data file called MIXED_UP that contains two types of records. If column 12 is a 1, the record layout is:

```
          File MIXED_UP
(Record Layout Where Column 12 = 1)

    Variable      Start-End
      Name          Column
-----------------------------------
     EMP_ID          1-3
     HEIGHT          4-5
     WEIGHT          6-8
```

If column 12 is a 2, the record layout is:

```
          File MIXED_UP
(Record Layout Where Column 12 = 2)

     Variable      Start-End
      Name          Column
-------------------------------------
     EMP_ID          1-3
     HEIGHT          5-6
     WEIGHT          8-10
```

Write a SAS program to create a SAS data set called HTWT. Use column input and do not include the data instream.

```
Sample data (in file MIXED_UP)

00168155 1
002 70 200 2
00362102 1
04 74 180 2
```

1-11. Write a program to read in pairs of values from the sample data below, representing make of car and gas mileage. Have the program create a SAS data set called MILEAGE. Notice that there are several pairs of values per line. Note also that some of the car makes are more than eight characters long.

```
Sample Data

Taurus 20 Civic 29 Cutlass 20 Cadillac 17
Celica 24 Corvette 17
```

1-12. You have names and test scores in two files, FILE_ONE and FILE_TWO. NAME is in columns 1-10 and SCORE is in columns 11-13. Write a SAS program to read data from both files and create a SAS data set called SCORES. Some sample data are shown below:

```
    FILE_ONE
---------------
 CODY      100
 PASS       98
 BAGGETT    96
 JOYNER     45

    FILE_TWO
---------------
 FRANKS     66
 BEANS      68
```

Cody_55126Chap1 6/22/99 4:11 PM Page 48