(8) DATES

SAS has numerous informats for reading dates and formats for displaying dates. Dates can be read with either numeric, character, or date informats. If you read a date as a character variable, you will not be able to use it in any calculations, e.g. to calculate the number of days that separate two given dates. If you read a date with a numeric format, SAS will store the date as a number, but not as a numeric value that can be used to calculate time differences. However, if you use one of the SAS date informats, the date is stored as a numeric variable that can be used in calculations.

No matter how a date might appear in a raw data file, there probably is a SAS informat that will allow it to be read and stored as a numeric variable. The following shows a series of SAS informats that can be used to read a specified date (March 15, 1994) stored in a number of different forms...

RAW DATA	INFORMAT
031594	mmddyy6.
940315	yymmdd6.
03/15/94	mmddyy8.
94/03/15	yymmdd8.
03 15 94	mmddyy8.
94 03 15	yymmdd8.
03151994	mmddyy8.
15mar94	date7.
03/15/1994	mmddyy10.
15mar1994	date9.

Once a date is read with a date informat, SAS stores the date as a number, i.e. the number of days between the date and January 1, 1960 (3/15/1994 is stored as 12492). This permits the use of dates in calculations. For example, imagine you were given a data file that contained a date admission to a hospital and a date of discharge. You might be interested in computing the length stay.

...Example 8.1...
```
data patients;
informat admit disch mmddyy8.;                                             1
format admit disch date9.;                                                 2
input admit disch;                                                         3
los = disch - admit;                                                       4
label
los = 'LENGTH OF STAY'
admit = 'ADMIT DATE'
disch = 'DISCHARGE DATE'
;
datalines;
03/21/90 04/01/90
05/15/90 05/16/90
;
run;
```

```
           ADMIT    DISCHARGE     LENGTH
Obs         DATE      DATE       OF STAY
 1     21MAR1990    01APR1990       11                                     5
 2     15MAY1990    16MAY1990        1
```

1 An INFORMAT statement tells SAS to read the variables ADMIT and DISCH with a DATE informat.
2 A FORMAT statement tells SAS to display the two variables as dates, not simply as numeric data.
3 LIST input is used to read the two dates.
4 A new variable is created, LOS, that is the difference in days between DISCH and ADMIT.
5 The output from PROC PRINT shows that the variables DISCH and ADMIT were treated as dates and that a difference in days was calculated correctly.

In addition to using an INFORMAT to tell SAS to treat the variables ADMIT and DISCH as dates, a FORMAT is used to control how the dates will be displayed. Without a format, the dates would be displayed as numbers...

```
proc print data=patients label;
* temporarily remove formats from variables;
format _all_;
run;
```

```
       ADMIT    DISCHARGE     LENGTH
Obs     DATE       DATE      OF STAY
 1     11037      11048         11
 2     11092      11093          1
```

The numbers shown under ADMIT and DISCHARGE are the number of days each date is from 1/1/1960. The calculation of LOS is still correct.

Just as SAS has a many INFORMATS for reading dates, it also has a wide selection of FORMATS for displaying dates, e.g. March 15, 1994 can be displayed as follows...

```
FORMAT                        DISPLAY
------                        -------
mmddyy.                    03/15/94
mmddyy6.                   031594
mmddyy8.                   03/15/94
yymmdd.                    94-03-15
yymmdd6.                   940315
yymmdd8.                   94-03-15
date7.                     15MAR94
date9.                     15MAR1994
weekdate.                        TUESDAY, MARCH 15, 1994
weekdatx.                        TUESDAY,  15 MARCH 1994
worddate.                       MARCH 15, 1994
```

The last three formats will display the leading blanks that are shown, e.g. using WORDDATE. results in five leading blanks. Example 8.1 used the DATE9. format to control the display of the variables DISCH and ADMIT.

In addition to the informats and formats that SAS supplies for reading and writing dates, there are also a number of SAS functions that can be used to work with dates.

...Example 8.2....

```
proc format;                                                                                      1
value dayofwk
1 = "SUNDAY"      2 = "MONDAY"      3 = "TUESDAY"
4 = "WEDNESDAY"   5 = "THURSDAY"    6 = "FRIDAY"      7 = "SATURDAY";
run;

data function;
format dob future now mmddyy10. dob_dow dayofwk.;                                                 2

dob      = '15jul75'd;                                                                            3
dob_dow  = weekday(dob);                                                                          4
future   = mdy(12,31,2010);                                                                       5
now      = today();                                                                               6
agethen  = (future - dob) / 365.25;                                                               7
agenow   = (now - dob) / 365.25;
age_then = yrdif(dob,future,'actual');                                                            8
age_now  = yrdif(dob,now,'actual');

label
dob      = "BIRTHDATE"
dob_dow  = "DAY OF WEEK/BIRTH DATE"
future   = "DAY IN/THE FUTURE"
```

```
now       = "TODAY"
agethen   = "AGE IN THE FUTURE (OLD WAY)"
agenow    = "AGE NOW (OLD WAY)"
age_then = "AGE IN THE FUTURE (YRDIF)"
age_now   = "AGE NOW (YRDIF)"
;
run;

proc print data=function label;
run;
                                                          AGE IN THE    AGE NOW    AGE IN THE
              DAY IN THE                    DAY OF WEEK     FUTURE        (OLD       FUTURE      AGE NOW
 BIRTHDATE      FUTURE          TODAY     (BIRTH DATE)   (OLD WAY)       WAY)       (YRDIF)     (YRDIF)

07/15/1975    12/31/2010    04/02/2001       TUESDAY        35.4634     25.7166      35.4630     25.7151
```

1 PROC format is used to create a format that will allow the printing of a literal day of the week given a number in the range 1 to 7.
2 FORMATS are assigned to several variables that are dates.
3 A date variable can be created if the date is expressed as shown (day/month/year, with a 3-character literal month) and the date is in quotes followed by the letter d.
4 The day of the week of any given day can be determined via the WEEKDAY function - the function returns a number in the range of 1 to 7 (Sunday-to-Saturday). The number is a numeric variable.
5 A date variable can also be created via the MDY function if the month, day, and year are supplied.
6 The current date can be assigned to a variable using the TODAY() function.
7 New variables (AGETHEN, AGENOW) are created. These are ages in years
8 A SAS function, YRDIF, is used to create new variables similar to those created in step #7. The YRDIF function requires three arguments: the starting date of an interval; the ending date of an interval; the method to be used to calculate the interval. To get a true difference in years, use the word ACTUAL in quotes as shown in the example.

The output from PROC PRINT shows all the variables in the data set. The variables created with the YRDIF function are not too different from the old method of calculating the age in years.

You can use the capability of expressing a date referred to in step #3 in example 8.2 to group observations by date ranges with a user-written format. The next example demonstrates this capability, together with some SAS-supplied formats that can be used to group observations.

...Example 8.3...

```
proc format;                                                                                    1
value interval
'01jan1997'd - '30jun1997'd = '1ST HALF 1997'
'01jul1997'd - '31dec1997'd = '2ND HALF 1997'
'01jan1998'd - '30jun1998'd = '1ST HALF 1998'
'01jul1998'd - '31dec1998'd = '2ND HALF 1998'
;
run;

data admits;
input admit1 :  mmddyy10. @@;                                                                   2
admit2 = admit1; admit3 = admit1; admit4 = admit1;
admit5 = admit1; admit6 = admit1; admit7 = admit1;
label
admit1 = 'USER-WRITTEN FORMAT'
admit2 = 'YEAR FORMAT'
admit3 = 'MONTH FORMAT'
admit4 = 'MONYY7 FORMAT'
admit5 = 'QTR (QUARTER) FORMAT'
admit6 = 'MONNAME (MONTH NAME) FORMAT'
admit7 = 'DOWNAME (DAY OF WEEK NAME) FORMAT'
;
```

```
datalines;
01181998 02111998 02281998 02161998 02271998 03291998 04181998 05081998
05071998 05101998 06031998 08021998 08131998 07241998 08151998 10011998
01081997 01251997 02041997 02171997 03071997 02181997
03161997 03281997 03301997 03271997 04031997 04271997
05311997 06071997 06131997 05311997 06181997 06161997
06201997 07141997 08071997 07131997 08071997 09051997
;
run;

proc freq data=admits;                                                                                  3
table admit1-admit7;
format
admit1 interval.
admit2 year.
admit3 month.
admit4 monyy7.
admit5 qtr.
admit6 monname.
admit7 downame.
;
run;
```

```
                        USER-WRITTEN FORMAT
                                          Cumulative    Cumulative
       admit1    Frequency     Percent     Frequency      Percent
-------------------------------------------------------------------
1ST HALF 1997          19       47.50            19        47.50
2ND HALF 1997           5       12.50            24        60.00
1ST HALF 1998          11       27.50            35        87.50
2ND HALF 1998           5       12.50            40       100.00

                        YEAR FORMAT
                                   Cumulative    Cumulative
admit2    Frequency     Percent     Frequency      Percent
------------------------------------------------------------
  1997          24       60.00            24        60.00
  1998          16       40.00            40       100.00

                        MONTH FORMAT
                                   Cumulative    Cumulative
admit3    Frequency     Percent     Frequency      Percent
------------------------------------------------------------
     1           3        7.50             3         7.50
     2           7       17.50            10        25.00
     3           6       15.00            16        40.00
     4           3        7.50            19        47.50
     5           5       12.50            24        60.00
     6           6       15.00            30        75.00
     7           3        7.50            33        82.50
     8           5       12.50            38        95.00
     9           1        2.50            39        97.50
    10           1        2.50            40       100.00
```

```
                        MONYY7 FORMAT
                                    Cumulative    Cumulative
 admit4    Frequency     Percent     Frequency      Percent
------------------------------------------------------------
JAN1997           2        5.00             2         5.00
FEB1997           3        7.50             5        12.50
MAR1997           5       12.50            10        25.00
APR1997           2        5.00            12        30.00
MAY1997           2        5.00            14        35.00
JUN1997           5       12.50            19        47.50
JUL1997           2        5.00            21        52.50
AUG1997           2        5.00            23        57.50
SEP1997           1        2.50            24        60.00
JAN1998           1        2.50            25        62.50
FEB1998           4       10.00            29        72.50
MAR1998           1        2.50            30        75.00
APR1998           1        2.50            31        77.50
MAY1998           3        7.50            34        85.00
JUN1998           1        2.50            35        87.50
JUL1998           1        2.50            36        90.00
AUG1998           3        7.50            39        97.50
OCT1998           1        2.50            40       100.00

                    QTR (QUARTER) FORMAT
                                   Cumulative    Cumulative
admit5    Frequency     Percent     Frequency      Percent
-----------------------------------------------------------
     1          16       40.00            16        40.00
     2          14       35.00            30        75.00
     3           9       22.50            39        97.50
     4           1        2.50            40       100.00

                  MONNAME (MONTH NAME) FORMAT
                                      Cumulative    Cumulative
   admit6    Frequency     Percent     Frequency      Percent
--------------------------------------------------------------
  January           3        7.50             3         7.50
 February           7       17.50            10        25.00
    March           6       15.00            16        40.00
    April           3        7.50            19        47.50
      May           5       12.50            24        60.00
     June           6       15.00            30        75.00
     July           3        7.50            33        82.50
   August           5       12.50            38        95.00
September           1        2.50            39        97.50
  October           1        2.50            40       100.00

               DOWNAME (DAY OF WEEK NAME) FORMAT
                                      Cumulative    Cumulative
   admit7    Frequency     Percent     Frequency      Percent
--------------------------------------------------------------
Wednesday           4       10.00             4        10.00
 Saturday           7       17.50            11        27.50
  Tuesday           2        5.00            13        32.50
   Monday           4       10.00            17        42.50
   Friday           8       20.00            25        62.50
   Sunday           8       20.00            33        82.50
 Thursday           7       17.50            40       100.00
```

1 A format is created that groups dates into four intervals
2 The data set ADMITS is created with seven variables, admit1-admit7, all having the same value.
3 Various formats are used to group observations based on the value of a variable that contains a date.

Two functions can be used to work with date intervals, INTCK and INTNX. The INTCK function computes the number of intervals between any two given dates, with the interval being either day, week, month, qtr, or year. The INTNX allows you to specify the time interval (same choices as with INTCK), a starting date, and the number of intervals you would like to cross. SAS then returns a date. These functions are not discussed here in any detail.

...YEARCUTOFF
When a year is expressed with only two digits, how does SAS know what century to use when it creates a numeric date value? There is a SAS system option called YEARCUTOFF and its value determines how two digit dates are evaluated. The default value of this option in version 8 is 1920. Any two digit date is assumed to occur AFTER 1920. If that value is changed via an OPTIONS statement, SAS will use the new value.

...Example 8.4...
```
options yearcutoff=1900;                                          1
data test;
format dt weekdate.;
dt = mdy(01,12,10);                                               2
label dt = 'YEARCUTOFF 1900';
run;

proc print data=test label noobs;
run;

options yearcutoff=1920;                                          3
data test;
format dt weekdate.;
dt = mdy(01,12,10);
label dt = 'YEARCUTOFF 1920';
run;

proc print data=test label noobs;
run;
```

```
          YEARCUTOFF 1900
    Wednesday, January 12, 1910

         YEARCUTOFF 1920
    Tuesday, January 12, 2010
```

1 The YEARCUTOFF option for two-digit dates is set to 1900. Any two-digit date is considered as occurring on or after 1900.
2 The MDY function is used to create a date variable.
3 The YEARCUTOFF option is changed to 1920. Any two-digit date is considered as occurring on or after 1920.

You can see the difference in how SAS treats the variable DT depending on the value of YEARCUTOFF.

...LONGITUDINAL DATA

There is a feature of a sorted data set that allows you to find the first and/or last observation in a sequence using a data step. You already know that you read a SAS data set with a SET statement. You also know about BY-GROUP processing from working with grouped data. A combination of a SET statement and a BY statement within a data step gives you access to two SAS-created variables.

...Example 8.5...

```
data manyids;
input id : $2. visit : mmddyy.;                                         1
format visit mmddyy8.;
datalines;
01  01/05/89
01  05/18/90
01  11/11/90
01  02/18/91
02  12/25/91
03  01/01/90
03  02/02/91
04  05/15/91
04  08/20/91
04  03/23/92
04  07/05/92
;
run;

proc sort data=manyids;                                                 2
by id visit;
run;

data oneid;
set manyids;                                                            3
by id;
if first.id then output;                                                4
run;

proc print data=oneid;
run;

OBS    ID       VISIT                                                   5
 1     01    01/05/89
 2     02    12/25/91
 3     03    01/01/90
 4     04    05/15/91
```

1 A data set is created containing two variables, ID and VISIT (a date variable).
2 The data set is sorted by ID and by VISIT (date) within each ID.
3 A new data set is created. The date are read with a SET/BY combination. Using BY ID; creates a new TEMPORARY variable name FIRST.ID that can be used within the data step.
4 The new data set contains only the observation with the first VISIT within each ID.

A new, SAS-created, temporary variable in example 8.5 is FIRST.ID. When you use a SET statement in combination with a BY statement, SAS will create a new variable for each by-variable. There is only one by-variable (ID), so SAS created FIRST.ID. That is not the entire story of SET/BY since SAS also creates a LAST.ID (a two-for-one deal), i.e. for every variable in the BY statement, SAS will create a FIRST.<by-variable> and a LAST.<by-variable>. The variables only exist for the duration of the data step and do not become part of any SAS dataset.

FIRST. and LAST. variables take on only two values, 1 or zero. The following are the values of FIRST.ID and LAST.ID when the sorted version of the dataset MANYIDS is used in the data step shown in example 8.5...

```
                FIRST.ID   LAST.ID
01   01/05/89       1         0
01   05/18/90       0         0
01   11/11/90       0         0
01   02/18/91       0         1
02   12/25/91       1         1
03   01/01/90       1         0
03   02/02/91       0         1
04   05/15/91       1         0
04   08/20/91       0         0
04   03/23/92       0         0
04   07/05/92       0         1
```

You can see that the first observation in a given sequence results in a FIRST. value of 1, while the last observation results in a LAST. value of 1. If an observation is both the first and last observation in a sequence (i.e. the only one as with ID 02), both the FIRST. and LAST. values are 1. If an observation is not the first or the last in a sequence, both the FIRST. and LAST. values are 0. When you use a statement such as that in example 8.5...

```
if first.id then output;
```

you are asking SAS to evaluate whether he value of the FIRST. variable is 1 or zero. If it is 1, then SAS performs the task. You could write the above statement in a number of different ways. All would result in the same data set being created....

```
if first.id eq 1 then output;

if first.id;

if not first.id then delete;
```

How could you change example 8.5 to create a data set with the last VISIT rather than the first?

...Example 8.6...

```
proc sort data=manyids;
by id descending visit;
run;

data oneid;
set manyids;
by id;
if first.id then output;
run;

proc print data=oneid;
run;

OBS    ID       VISIT
 1     01    02/18/91
 2     02    12/25/91
 3     03    02/02/91
 4     04    07/05/92
```

Since the data are sorted in descending date order within each ID, the first observation within each ID group is the last VISIT. You could also modify the data step instead of the sort.

...Example 8.7...
```
proc sort data=manyids;
by id visit;
run;

data oneid;
set manyids;
by id;
if last.id then output;
run;

proc print data=oneid;
run;

OBS    ID       VISIT
 1     01    02/18/91
 2     02    12/25/91
 3     03    02/02/91
 4     04    07/05/92
```

What if your task was to determine how long any individual in the dataset MANYIDS had been part of your study, i.e. what is the difference in days between the first and last visits? This can be done in a number different ways. One involves match-merging data sets (that's for later in the semester). The following only requires one sort, one data step, plus the use of RETAIN and DO-END statements.

...Example 8.8...
```
proc sort data=manyids;
by id visit;
run;

data duration;
retain firstvis;;
set manyids;
by id;
if first.id then firstvis=visit;
if last.id then do;
   diffdays = visit - firstvis;
   output;
end;
run;

proc print data=duration;
var id firstvis visit diffdays;
format firstvis visit mmddyy6.;
run;

OBS    ID    FIRSTVIS       VISIT    DIFFDAYS
 1     01    01/05/89    02/18/91       774
 2     02    12/25/91    12/25/91         0
 3     03    01/01/90    02/02/91       397
 4     04    05/15/91    07/05/92       417
```

The IF FIRST.ID THEN... statement tells SAS to store the value of the variable VISIT as variable FIRSTVIS when the first observation within a given ID is encountered. The RETAIN statement tells SAS to hold onto the value assigned to FIRSTVIS rather than set it back to missing each time SAS cycles back to the top of the data step. Remember, the default behavior of SAS within a data step is to set the value of MOST (not all) variables to missing each time SAS reaches the top of the data step. The RETAIN statement can selectively alter that behavior. The DO-END statement allows you to perform multiple actions. Since the DO-END statement is embedded in an IF-THEN statement, SAS will perform multiple actions if the IF-THEN statement is TRUE. When the observation within an ID group is found, the date of the first visit is subtracted from the date of last visit and an observation is writtem to the data set by using an OUTPUT statement.

What if in addition to the VISIT data, you also had another variable that measured some characteristic of an individual at each visit, e.g. cholesterol, that you hoped was changing over time as the result of some study intervention. How could you modify the data step in example 8.8 to determine both the difference in days and cholesterol values between first and last visits?

...Example 8.9...

```
data manyids;
input id : $2. visit : mmddyy8. chol;
format visit mmddyy8.;
datalines;
01  01/05/89   400
01  05/18/90   350
01  11/11/90   305
01  02/18/91   260
02  12/25/91   200
03  01/01/90   387
03  02/02/91   380
04  05/15/91   380
04  08/20/91   370
04  03/23/92   355
04  07/05/92   261
;
run;

proc sort data=manyids;
by id visit;
run;

data twodiffs;
retain firstvis firstcho;
format firstvis mmddyy8.;
set manyids;
by id;
if first.id then do;
   firstvis=visit;
   firstcho=chol;
end;
if last.id then do;
   diffdays = visit - firstvis;
   diffchol = chol  - firstcho;
   output;
end;
run;

proc print data=twodiffs;
var id firstvis visit diffdays firstcho chol diffchol;
format firstvis mmddyy8.;
run;
```

```
OBS    ID    FIRSTVIS       VISIT    DIFFDAYS    FIRSTCHO    CHOL    DIFFCHOL
 1     01    01/05/89    02/18/91       774          400      260       -140
 2     02    12/25/91    12/25/91         0          200      200          0
 3     03    01/01/90    02/02/91       397          387      380         -7
 4     04    05/15/91    07/05/92       417          380      261       -119
```

In example 8.8, the value of the variable VISIT was stored when the first observation within an ID group was read. In example 8.9, both the date (the value of the variable VISIT) and the initial cholesterol reading are stored. When the last observation in an ID group is read, the difference in days between the first and last visit, and the difference in cholesterol can be calculated.

If you read a data set with a combination of SET and BY and use more than one by variable, you create more than one pair of first.<var> and last.<var> variables. Each variable listed in the BY statement results in a pair of first.<var> and last.<var> variables.

...Example 8.10...

```
data test;
input x y z;
datalines;                                                                  1
1 1 100
1 1 110
1 1 120
2 1 10
3 1 99
1 2 200
1 2 210
3 2 199
1 3 300
1 3 310
;
run;
proc sort data=test;                                                        2
by x y;
run;

data first_last;
set test;                                                                   3
by x y;
firstx = first.x;
firsty = first.y;                                                           4
lastx  = last.x;
lasty  = last.y;
run;

proc print data=first_last;
var x y z firstx lastx firsty lasty;
run;

Obs    x    y     z    firstx    lastx    firsty    lasty                   5
  1    1    1   100       1        0         1        0
  2    1    1   110       0        0         0        0
  3    1    1   120       0        0         0        1
  4    1    2   200       0        0         1        0
  5    1    2   210       0        0         0        1
  6    1    3   300       0        0         1        0
  7    1    3   310       0        1         0        1
  8    2    1    10       1        1         1        1
  9    3    1    99       1        0         1        1
 10    3    2   199       0        1         1        1
```

1 A data set is created that will be used to show first.<var> and last.<var> values for two by variables, X and Y.
2 SET with a BY statement requires that data be sorted according to the BY variable(s).
3 A SET statement in combination with a BY statement is used to read the data set. There are two BY variables.
4 Since first.<var> and last.<var> variables are temporary (i.e. they only exist during the execution of the data step), new variables are created to hold the their values for subsequent printing.
5 The output from PROC PRINT shows the values of all the first.<var> and last.<var> variables.

If you look at the values of the variable X, you can see that the values of the first.X and last.X are what you have already seen in the previous examples. However, notice that the value of first.Y and last.Y cycle within each value of X.

(9) FUNCTIONS

There are a large number of SAS functions that allow you to work with data WITHIN a single observation. Remember that SAS PROCs work WITHIN variables/ACROSS observations. SAS functions work ACROSS variables/WITHIN observations. A number of SAS functions have already been introduced. In the section on DATES, several functions were shown that could be used to work with date (numeric) variables, e.g. MDY, WEEKDAY, TODAY, YRDFIF. One feature common to all SAS functions is that they are all followed by a set of parentheses that contain zero or more arguments. The arguments (sometimes referred to as parameters) are information needed by the function to produce a result. The TODAY function requires no argument, just the parentheses, to return the value of the current date. The WEEKDAY function requires one argument, a SAS date. The YEARDIF function requires three arguments. Just as SAS variables can be classed as either NUMERIC or CHARACTER, SAS functions can be divided into those that are used with numeric variables and those that are used with character variables. There are only a few functions that can be used with either type of data.

...NUMERIC FUNCTIONS

SAS numeric functions can be further divided into categories that describe the action of the function. The SAS language manual divides numeric functions into the following categories: arithmetic, date and time (dates are numeric variables), financial, mathematical (or 'more complicated arithmetic'), probability, quantile, random number, simple statistic, special, trigonometric and hyperbolic (no exaggeration), truncation. The following example uses grades for 29 students who took a series of exams.

...Example 9.1...

```
data midterm (drop=g1-g29);                                                          1
input type : $7. g1-g29;
mingr  = min(of g1-g29);                                                             2
maxgr  = max(of g1-g29);
meangr = mean(of g1-g29);
stdgr  = std(of g1-g29);
vargr  = var(of g1-g29);
missgr = nmiss(of g1-g29);
nmbgr  = n(of g1-g29);
meangr = round(meangr,.1);                                                           3
stdgr  = round(stdgr,.1);
vargr  = round(vargr,.1);
label
type   = 'TEST'
mingr  = 'MINIMUM'
maxgr  = 'MAXIMUM'
meangr = 'MEAN'
stdgr  = 'STANDARD DEVIATION'
vargr  = 'VARIANCE'
missgr = '# MISSING'
nmbgr  = '# NON-MISSING'
;
datalines;
QUIZ1    6  8  5 10 10  9 10  8  9  9  7 10 10 10 10  8 10  6 10 10 10 10 . 10  8 10 10 10 10
QUIZ2   10 10  2 10 10  . 10 10 10 10 10  2 10 10 10 10 10 10 10 10 10 10 . 10 10 10 10 10 10
MIDTERM  9  2  0  9  7 10 10 10  8  9  5  3  9  8  9 10  2 10  9  9  8 10 .  8  8 10  9 10  8
QUIZ3   10  2  2 18 19 12 20  . 10  8  .  2  . 10 10 18 12 10 15 10  5 15 .  5 12 20 20 20 20
QUIZ4   23  5  5 20 25 15 25 22 25 18 18  2 18 22 20 22 15 18 23 22 20 25 . 20 18 25 25 20 22
FINAL   25 20 15 25 25 22 25 25 22 18 21 15 22 20 18  . 20 15 23 24 22 20 . 15 24 22 25 20 20
;
run;

proc print data=midterm label noobs;
var type nmbgr missgr meangr stdgr vargr mingr maxgr;
run;
```

```
             #
           NON-       #              STANDARD
 TEST    MISSING  MISSING  MEAN  DEVIATION  VARIANCE  MINIMUM  MAXIMUM
QUIZ1       28        1     9.0     1.5         2.2        5       10
QUIZ2       27        2     9.4     2.1         4.6        2       10
MIDTERM     28        1     7.8     2.8         7.8        0       10
QUIZ3       25        4    12.2     6.1        37.3        2       20
QUIZ4       28        1    19.2     6.1        37.6        2       25
FINAL       27        2    21.0     3.3        11.2       15       25
```

1 Since only the new values computed in the data step are to be placed in the data set, the values of the individual grades are dropped.
2 Several arithmetic and statistical functions are used to compute the values of new variables.
3 The ROUND function is used to round the value of three variables to one decimal place

Naming the grade variables G1 through G29 made it very simple to place the values of the grades as arguments in the various functions. It might not be obvious, but each of the functions that used grades 1 through 29 ignored all grades with a missing value, just as would have been done if the data had been rearranged and analyzed with PROC MEANS. Both SAS functions and SAS procedures will ignore missing values when computing arithmetic or statistical values. The N (and NMISS) function allow you to determine how many values were used to compute various values. The output from PROC PRINT shows the mean, standard deviation, and variance of the grades for each question rounded to one decimal place. These values were stored in the data set in lieu of keeping values with many decimal places. If the ROUND function had not been used, but the following format placed in the data step or PROC PRINT...

```
format meangr stdgr vargr 6.1;
```

the same values would have been printed, but the stored values of the variables would still have many decimal places. The ROUND function actually changes the stored value while a FORMAT would only affect the appearance of the variable value.

There are several other functions that will alter the stored value of a numeric variable. They differ in the values that are returned for positive versus negative numbers.

...Example 9.2...

```
data alter;
input x;
ceil_x  = ceil(x);
floor_x = floor(x);
int_x   = int(x);
round_x = round(x,1.);
y = x;
format y 6.;
label
x       = 'ORIGINAL VALUE X'
y       = 'FORMATTED VALUE OF X'
ceil_x  = 'CEILING'
floor_x = 'FLOOR'
int_x   = 'INTEGER'
round_x = 'ROUND'
;
datalines;
7.5
-7.5
8.5
-8.5
;
run;
```

```
proc print data=alter noobs label;
var x y int_x round_x ceil_x floor_x;
run;

ORIGINAL    FORMATTED
 VALUE X    VALUE OF X    INTEGER    ROUND    CEILING    FLOOR
   7.5            8          7          8        8          7
  -7.5           -8         -7         -8       -7         -8
   8.5            9          8          9        9          8
  -8.5           -9         -8         -9       -8         -9
```

The results of the ceiling and floor functions depend on the whether the value of a variable is positive or negative. The integer and round functions work the same regardless of sign. The formatted value of the variable Y is the same as the rounded value, but Y is still stored as 7.5, -7.5, etc. since it is stored with the same values as X (remember, Y=X in the data step).

There are a number of ways of expressing the arguments required by functions that require the values of a number of numeric variables. In example 9.1, the convention of naming variables <name>1 through <name>N made it easy to place a large number of variable values as a function argument.

...Example 9.3...

```
data test;
input gradeone gradetwo gradethr;
mean_one = mean (of gradeone gradetwo gradethr);
mean_two = mean (of gradeone--gradethr);
datalines;
80 80 95
;
run;

proc print data=test noobs;
run;

gradeone    gradetwo    gradethr    mean_one    mean_two
   80          80          95          85          85
```

The name of each variable can be placed in the argument list. If you know the order of the variables in the data set, the convention of specifying a list with two variable names separated by two dashes (- -) can be used. Remember that using <start var>- -<end var> depends on the order of variables within the data set. If you are not sure of the order, you can see what is by using PROC CONTENTS and looking at the first column (showing the position of the variable in the data set.

As stated earlier, a function will ignore missing values. What if you use a function and the value of each variable in the argument list is missing? The value returned by the function will also be missing. To avoid this, you can specify the following...

```
sum_one = sum (of gradeone--gradethr, 0);
```

Adding a zero to the argument list ensures that the value returned by the function will never be missing, even if the value of each variable in the argument list is missing. Whether you want the function to result in missing or zero is up to you.

...CHARACTER FUNCTIONS
There are a large number of SAS functions that can be used with character data. They are especially useful if you are working with a SAS data set that contains character variables that must be 'cleaned up', examined, split apart, combined - in a word, manipulated. What if you are give a data set with a first name, last name, street number, street name, city, state, and zip code, each stored as a separate variable. You want to combine the names into one variable and the address into another.

...Example 9.4...
```
data manyvars;
input f_name $  1-8   l_name  $  9-20
      st_num   21-25  st_name $ 26-47
      city   $ 48-59  state   $ 60-61  zip $ 62-66;
datalines;
MIKE    ZDEB        59   LENOX AVE             ALBANY      NY12203
THE     PRESIDENT        PENNSYLVANIA AVENUE   WASHINGTON  DC00001
;
run;

data nameaddr (keep=name addr);
set manyvars;
name = trim(f_name) || ' '  || l_name;                                                       1
addr = trim(st_num) || ' '  || trim(st_name) || ', '  ||                                     2
       trim(city)   || ', ' || trim(state)   || '   ' || zip;
run;

proc print data=nameaddr noobs;
run;

  name                             addr
MIKE ZDEB        59 LENOX AVE, ALBANY, NY   12203
THE PRESIDENT     . PENNSYLVANIA AVENUE, WASHINGTON, DC   00001
```

1 Just as a plus sign (+) is used to add numeric variables, the operator **||** is used to add character variables to on another. The name CONCATENATION is often used when referring to combining character variables with the **||** operator. The text in both variables is combined to form another variable. In addition to text, a character constant (a space) is added between the first name and last name. The TRIM function is used to remove trailing blank characters from a variable prior to concatenation.
2 The same process is used to form the new variable address. Notice that, in addition to blanks, commas are added to the address.

The TRIM function removes trailing blanks. The operator '**||**' performs concatenation (again, jargon for joining). Since all trailing blanks are removed with TRIM, blank spaces (and commas) are added in various spots of the address. Notice that there is a period as the street number in the second observation. The variable ST_NUM was read as NUMERIC. The log file that resulted from the data step showed..

```
NOTE: Numeric values have been converted to character
      values at the places given by: (Line):(Column).
```

Since ST_NUM was used in an operation that required character variables, SAS converted the numeric value to character as stated in the log. Missing numeric data are displayed as a period. If ST_NUM had been read as a character variable, there would be no period, just a blank space indicating a missing character value. How could you get rid of the extra space if the street number was missing?

The results of PROC contents run for the two datasets are...

```
dataset MANYVARS...
#    Variable    Type    Len
----------------------------
5    city        Char     12
1    f_name      Char      8
2    l_name      Char     12
4    st_name     Char     22
3    st_num      Num       8
6    state       Char      2
7    zip         Char      5



dataset NAMEADDR...
#    Variable    Type    Len
----------------------------
2    addr        Char     61
1    name        Char     21
```

When a new variable is created by concatenating the values of other variables, the resulting variable is the length of all variables being combined, plus any additional characters such as commas or spaces. The new variable NAME has a length of 21, the combined length of variables F_NAME (8), L_NAME (12) , plus the space added between the names. The length of ADDR is 61, It is a combination of...

ST_NUM (12) + space (1) +
ST_NAME (22) + comma/space (2) +
CITY (12) + comma/space (2) +
STATE (2) + three spaces (3) +
ZIP (5)

Where did the 12 characters come from in the street number (ST_NUM)? Remember that ST_NUM was a numeric variable and that SAS converted it to character data when forming the address. When SAS does such a conversion, the result is always a character variable with a length of 12. Confusing? That's OK. It's not that important to remember exactly what occurred. The important concept is that if you involve numeric variables in the creation of new character variables, you might get some unexpected results.

Just as there are character functions (and operators) that allow you combine variable values, there are also functions that allow you to separate the value of a variable into a number of new variables.

...Example 9.5...

```
data bignames;
input name $25. addr $40.;                                                  1
datalines;
MIKE ZDEB                59 LENOX AVENUE, ALBANY, NEW YORK 12203
;
run;

data manyvars (drop=name addr);
set bignames;
fname = scan(name,1);                                                       2
lname = scan(name,2);
street= scan(addr,1,',');                                                   3
city  = scan(addr,2,',');
stzip = scan(addr,3,',');
run;

proc print data=manyvars;
run;
```

```
proc contents data=manyvars noobs;
run;

fname    lname        street          city         stzip

MIKE     ZDEB     59 LENOX AVENUE    ALBANY    NEW YORK 12203

#    Variable    Type    Len
----------------------------
4    city        Char    200
1    fname       Char    200
2    lname       Char    200
3    street      Char    200
5    stzip       Char    200
```

1 A data set is created with variables that we want to take apart, i.e. use them to create more character variables.
2 The SCAN function is used to make new variables from the variable NAME.
3 Once again, the SCAN function is used. But this time it only looks for commas when breaking the old variable ADDR into words.

The SCAN function looks at the value of a character variable and separates it into WORDS using a number of different characters as word boundaries. If the word boundaries are not explicitly stated as the third argument in the function, SCAN uses the following characters to search for words...

```
blank .  < ( + & ! $ * ) ; ^ - / , % |
```

In example 9.6, no third argument was used when breaking up the variable NAME. SAS found BLANKS (SPACES) in the name and used them as word boundaries. When the SCAN function was used with the variable ADDR, SAS was told to break the variable into words base on the presence of only COMMAS.

A new feature of the SCAN function is the ability to scan text backwards using negative numbers. The following example shows a use of this feature. The data set contains a variable address in which the last word is always the zip code. The zip code can be extracted from the address even though the address have different contents.

...Example 9.6...

```
data addr;
input address $50.;
datalines;
PO BOX 666, UTICA, NY  13502
127 LAKE AVENUE NORTH, ST LOUIS, MO   23456
ANDOVER, MA   01010
;
run;

data addr;
length zip $5;
set addr;
zip = scan(addr,-1);
run;

proc print data=addr noobs;
run;

 zip     address
13502    PO BOX 666, UTICA, NY  13502
23456    127 LAKE AVENUE NORTH, ST LOUIS, MO   23456
01010    ANDOVER, MA   01010
```

An important feature of the SCAN function is shown in the output from PROC CONTENTS. The SCAN function always results in variables with a length of 200. A LENGTH statement should be used to explicitly set the length of all variables created via the SCAN function. Otherwise, you will end up with some very large data sets.

What if we have two data sets and we want to compare the values of character variables in the two data. If the variables are names, it is possible that the names in one of the files contained some lower case letters while all names were upper case in the other. One file might have embedded characters other than letters in the names, e.g. O'Henry in one versus OHenry in the other, SMITH JR. in one versus SMITH JR in the other, Jones in one JONES in the other, etc. The COMPRESS function will allow us to get rid of unwanted characters (ah, if life were so simple), while the UPCASE function will convert lowercase letters to uppercase.

...Example 9.7...
```
data up_comp;
input oldname $char13.;
newname1 = compress(oldname);                                                      1
newname2 = compress(oldname,"'.");                                                 2
newname3 = compress(oldname,"'. ");                                                3
newname4 = upcase(newname1);                                                       4
newname5 = upcase(newname2);
newname6 = upcase(newname3);
datalines;
Mc Donald
Jones
O'Henry
SMITH JR.
;
run;

proc print data=up_comp noob;
run;

oldname     newname1   newname2   newname3   newname4   newname5   newname6
Mc Donald   McDonald   Mc Donald  McDonald   MCDONALD   MC DONALD  MCDONALD
Jones       Jones      Jones      Jones      JONES      JONES      JONES
O'Henry     O'Henry    OHenry     OHenry     O'HENRY    OHENRY     OHENRY
SMITH JR.   SMITHJR.   SMITH JR   SMITHJR    SMITHJR.   SMITH JR   SMITHJR
```

1 The COMPRESS function is used to make a new variable. Without a second argument, the function will only remove spaces from a variable.
2 A second argument is added to the COMPRESS function, i.e. list of characters to be removed from the variable. However, once a second argument is used, COMPRESS no longer automatically removes spaces.
3 A space is added to the second argument used with the COMPRESS function.

There is another function that performs a task similar to that of the COMPRESS function. COMPBL removes blanks just as the COMPRESS function does. However, it reduces multiple consecutive blanks to one blank rather than removing all blanks.

...Example 9.8...
```
data mult_spaces;
input name $char50.;
name_compress = compress(name);                                                    1
name_compbl = compbl(name);                                                        2
datalines;
MIKE      ZDEB
   Mike                    Zdeb
;
run;
```

```
proc print data=mult_spaces;
run;

                                    name_        name_
name                               compress      compbl
MIKE      ZDEB                     MIKEZDEB    MIKE ZDEB                                   3
   Mike                   Zdeb     MikeZdeb     Mike Zdeb

#    Variable         Type    Len
---------------------------------
1    name             Char     50
3    name_compbl      Char    200                                                          4
2    name_compress    Char     50
```

1 The COMPRESS functions is used to remove all spaces from a character variable.
2 The COMBL function is used to convert multiple spaces to a single space in a character varaible.
3 Results of the two functions are show that COMPBL is not always the answer to fixing character data. There are still spaces at the beginning of the value of the variable name in observation two.
4 While the COMPRESS function resulted in a variable with the same length as the character variable used with the function, the COMBL function produced a variable with a length of 200. Just as when the SCAN function is used, a LENGTH statement should be used whenever the COMPBL function is used.

While the COMPBL has reduced multiple spaces between the first and last names within the variable name, it has left a space at the beginning of the name in the second observation. There are two functions that can reposition text within a character variable, LEFT and RIGHT.

...Example 9.9...

```
data left_right;
length name_left name_right $30.;                                                          1
input name $char50.;
name_left  = compbl(name);
name_left  = left(name_left);                                                              2
name_right = compbl(name);
name_right = right(name_right);                                                            3
datalines;
MIKE      ZDEB
   Mike                   Zdeb
;
run;

proc print data=left_right;
format name_right $30.;                                                                    4
run;

name_left    name_right                      name
MIKE ZDEB                         MIKE ZDEB    MIKE      ZDEB
Mike Zdeb                         Mike Zdeb       Mike                   Zdeb
```

1 A LENGTH statement is used for the new variables that are created with the COMBL function.
2 The LEFT function left justifies the value of the variable name_left.
3 The RIGHT function right justifies the value of the variable name_right.
4 A FORMAT statement is used in PROC PRINT to show the entire value of the variable name_right. What would the results of PROC PRINT look like with this format statement?

Another commonly used character function is SUBSTR (or substring). The SUBSTR function allows you to extract portions of a variable. The SCAN function performs a similar task, but it depends on the presence of delimiters in a variable to make decisions as to how to extract portions of an old variable to create a new variable. The SUBSTR function requires an explicit statement as to what portion of an old variable to extract.

...Example 9.10...

```
data names;
input name $20.;                                                              1
datalines;
MICHAEL ZDEB
PUFF    DADDY
GEORGE  WASHINGTON
;
run;

data names;
length first2 $8 last2 $12;                                                   2
set names;
first1 = substr(name,1,8);                                                    3
first2 = substr(name,1,8);
last1  = substr(name,9,12);
last2  = substr(name,9,12);
run;

proc print data=names noobs;
var name first1 first2 last1 last2;
run;

proc contents data=names;
run;
```

```
   name               first1     first2     last1         last2
MICHAEL ZDEB          MICHAEL    MICHAEL    ZDEB          ZDEB
PUFF    DADDY         PUFF       PUFF       DADDY         DADDY
GEORGE  WASHINGTON    GEORGE     GEORGE     WASHINGTON    WASHINGTON

#    Variable    Type    Len
----------------------------
4    first1      Char     20
1    first2      Char      8
5    last1       Char     20
2    last2       Char     12
3    name        Char     20
```

1 A data set is created with a variable (NAME) that has different information (first and last names) in the same location across observations.
2 Lengths are assigned to two new variables.
3 New variables are created using the SUBSTR function. The function requires three arguments: a variable name; a starting position within the variable for extraction of text; an ending position within the variable to stop text extraction.

In the output from PROC PRINT, variable FIRST1 looks the same as FIRST2, and LAST1 looks the same as LAST2. However, look at the lengths of the variables shown in the results of PROC CONTENTS. The two new variables FIRST1 and LAST1 have the same length as the variable NAME. This is the default behavior of the SUBSTR function, i.e. any variable created using the SUBSTR function has the same length as the variable used as the first argument. A LENGTH statement was used to assign lengths to variables FIRST2 and LAST2.

The SUBSTR function can also occur on the left side of the equals sign in a SAS statement. When used on the left side of the ='s, it allows you to replace characters in selected positions within the value of a variable. One such use of the SUBSTR function is shown in the next example. You have a data file that contains a first and last name for each subject in a study and all the names have been entered in upper case. You want to write a report using the names, but want only the first letter of each name to be upper case, the rest lower case.

...Example 9.11...
```
data study;
length firstn $10 lastn $20;
input firstn lastn;                                                          1
datalines;
MICHAEL ZDEB
BRITNEY SPEARS
PUFF DADDY
;
run;

data study;
set study;
firstn = lowcase(firstn);                                                    2
lastn  = lowcase(lastn);
substr(firstn,1,1) = upcase(substr(firstn,1,1));                             3
substr(lastn,1,1)  = upcase(substr(lastn,1,1));
name = trim(lastn) || ', ' || firstn;                                        4
run;

proc print data=study noobs;
run;

firstn     lastn          name
Michael    Zdeb      Zdeb, Michael
Britney    Spears    Spears, Britney
Puff       Daddy     Daddy, Puff
```

1 A data set is created in which all variable values are in UPPER CASE.
2 The LOWCASE function is used to change all characters in variables to lower case.
3 The SUBSTR function is used on the left side of the ='s to specify what characters in the variables should be altered by the functions on the right side of the ='s. On the right side, the SUBSTR function is used to extract the first letter from the variables. The UPCASE function is used to convert the case of the first letter. The SUBSTR function on the left side of the ='s results in the uppercase character being substituted in the first and last names.
4 A new variable is created by concatenating old variables.

Four character functions can be used to search for the presence of specified characters or groups of characters. The INDEXC function looks for the occurrence of a single character within a given character variable while the INDEX function looks for the occurrence of the entire string of characters. The INDEXW function is similar to the INDEX function. However, the string of characters must be found as a separate word, not embedded within the text being searched. The VERIFY function performs a task similar (but opposite) to that of the INDEXC function in that it looks for characters that are not present within a given list of characters. The next

...Example 9.12...
```
data test;
input
@01 address   $char20.
@21 zip       $char5.
;
address_index  = index(address,'ST');                                        1
address_indexw = indexw(address,'ST');                                       2
zip_indexc     = indexc(zip,'0123456789');                                   3
zip_verify     = verify(zip,'0123456789');                                   4
datalines;
125 FIRST AVENUE    10095
1 SECOND ST         1oo82
;
run;

proc print data=test noobs;
run;
```

```
                            address_    address_     zip_      zip_
    address          zip      index      indexw     indexc    verify
125 FIRST AVENUE    10095       8           0          1         0
1 SECOND ST         1oo82      10          10          1         2
```

1 The INDEX function searches for the character string ST at any location with the value of the variable address.
2 The INDEXW function also searches for the character string ST, but as a separate word (not embedded within other text, i.e. not part of a word).
3 The INDEXC function searches for the first occurrence of any of the characters in the given list of numbers within the variable zip.
4 The VERIFY function searchs for characters that are not in the given list of numbers within the variable zip.

In the first observation, the INDEX function found the character string ST within the address at position 8 (the eight character of the address). The INDEXW did not identify ST as a separate word and returned a value of zero, or not found. The first character in the zip code matched a character in the list searched by the INDEXC function and the function returned a value of 1. All the characters in the zip in the first observation matched the list used in the VERIFY function. Therefore, it returned a value of zero indicating that no unlisted characters were found.

Using the same functions with data from the second observation produces different results. Once again, the INDEX function finds the string ST in the address, but the INDEXW function also identifies the string as a separate word and returns a value as to the position where the word is found in the address. The INDEXC function once again finds a one as the first character of the address. However, the second and third characters of the zip are not zeroes, but o's (OHs). The VERIFY function returns a value of 2, the position where the first unlisted character was found.

Two character functions allow you to alter the contents of a character variable. The SUBSTR function has already been used (example 9.10) to replace text at specified positions within a character variable, The TRANSLATE and TRANWRD can search for and replace single characters or strings of characters. The SUBSTR function requires a specific position to be identified when altering the content of a character variable. Both TRANSLATE and TRANWRD find the position of the text to be replaced rather than having it specified.

...Example 9.13...
```
data fix_zips;
input @01 zip  $5.;
fixed_zip = translate(zip,'OO11',' olL');                                                    1
datalines;
12201
122 1                                                                                        2
122o1
l220L
1220
;
run;

proc print data=fix_zips noobs;
run;

         fixed_
 zip      zip
12201    12201
122 1    12201
122o1    12201
l220L    12201
1220     12200
```

1 The TRANSLATE function is used to convert text within a character variable.
2 The observations contain a number of errors that are to be corrected with the TRANSLATE function. There are OHs and ELs in some locations rather than zeroes and ones. There re also spaces rather than zeroes.

The TRANSLATE function requires three arguments. The first is the variable whose contents will be altered. The second and third are the 'to' and 'from' strings. The TRANSLATE function looks for characters in the 'from' string and replaces them with characters in the 'to' string. Characters are matched one-to-one, i.e. the first 'to' character replaces the first 'from' character, etc.

The TRANSLATE function replaces individual characters. If you want to search for and replace an entire string of characters, you must use the TRANWRD function.

...Example 9.14...
```
data test;
input oldaddr $char40.;
address = upcase(oldaddr);                                                         1
address = compress(address,'.');                                                   2
address = tranwrd(address,'BOULEVARD','BLVD');                                     3
address = tranwrd(address,'STREET','ST');
address = tranwrd(address,'AVENUE','AVE');
address = tranwrd(address,' AV ',' AVE ');
address = tranwrd(address,'WEST','W');
datalines;
59 Lenox Avenue
100 MADISON AVE.
88 RAVEN LN
2 Manning Boulevard
15 West 155th Street
3 WESTERN AVENUE
;
run;

proc print data=test noobs;
run;
```

```
OLDADDR                 ADDRESS
59 Lenox Avenue         59 LENOX AVE
100 MADISON AVE.        100 MADISON AVE
88 RAVEN LN             88 RAVEN LN
2 Manning Boulevard     2 MANNING BLVD
15 West 155th Street    15 W 155TH ST
3 WESTERN AVENUE        3 WERN AVE
```

1 A new variable, ADDRESS, is created from the old address, OLDADDR, using the uppercase function.
2 Periods are removed from the address using the COMPRESS function.
3 A series of changes is made to the new variable, searching for and replacing text strings.

The TRANWRD function searches for and replaces entire character strings. Notice that the second and third arguments of the TRANWRD function are 'from' then 'to' strings, the opposite of the TRANSLATE function. Also notice that you must be careful when specifying the 'from' string. In the example, 'AV' was surrounded by blanks since it could occur as part of a real word. Without the surrounding blanks, the instruction to convert 'AV' to 'AVE' would have resulted in occurrences of the string 'RAVEEN' in the third observation. The string 'WEST' was NOT surrounded by blanks. The instruction to convert 'WEST' to 'W'' was adequate when modifying the fifth observation, but caused an unwanted result for the sixth, resulting in ''WESTERN' being converted to 'WERN'.

The last three character functions to be discussed are LENGTH, REVERSE, and REPEAT. The LENGTH function returns the length of a character variable, specifically the position of the last non-blank character. The REVERSE function reverses the positions of text within a character string, while the REPEAT function can be used to create a character variable by repeating one or more characters a given number of times.

...Example 9.15...

```
data last_three;
input name $50.;                                                                  1
datalines;
MICHAEL ZDEB
BRITNEY SPEARS
PUFF DADDY
;
run;

data new;
set last_three;
reversed_name = reverse(name);                                                    2
first_name    = scan(name,1);                                                     3
last_name     = scan(name,2);
many_char     = repeat('*',3);                                                    4
many_chars    = repeat('+-',3);
many_firsts   = repeat(trim(first_name),3);
length_first  = length(first_name);                                               5
length_last   = length(last_name);
length_char   = length(many_char);
length_chars  = length(many_chars);
length_firsts = length(many_firsts);
run;

proc contents data=new;
run;

proc print data=new;
run;

proc means data=new maxdec=0 max;
var length_:;
run;
```

```
Variable         Type    Len
----------------------------
first_name       Char    200                                                      6
last_name        Char    200
length_char      Num       8
length_chars     Num       8
length_first     Num       8
length_firsts    Num       8
length_last      Num       8
many_char        Char    200
many_chars       Char    200
many_firsts      Char    200
name             Char     50
reversed_name    Char     50

                                  first_     last_
     name         reversed_name     name        name
MICHAEL ZDEB        BEDZ LEAHCIM    MICHAEL    ZDEB
BRITNEY SPEARS    SRAEPS YENTIRB    BRITNEY    SPEARS
PUFF DADDY            YDDAD FFUP    PUFF       DADDY
```

```
many_     many_                                       length_    length_    length_    length_    length_
char      chars      many_firsts                        first       last       char      chars     firsts
****     +-+-+-+-    MICHAELMICHAELMICHAELMICHAEL          7          4          4          8         28
****     +-+-+-+-    BRITNEYBRITNEYBRITNEYBRITNEY          7          6          4          8         28
****     +-+-+-+-    PUFFPUFFPUFFPUFF                      4          5          4          8         16
```

```
Variable              Maximum
-----------------------------
length_first                7
length_last                 6
length_char                 4
length_chars                8
length_firsts              28
-----------------------------
```

1 A data set is created with one variable, NAME, a character variable with a length of 50.
2 The REVERSE function is used to create a new variable from the variable name.
3 The first and last names are extracted and placed in separate variables using the SCAN function.
4 Three new variables are created using the REPEAT function. The first repeats one character. The second repeats a string of two characters, while the third repeats the value of the variable first_name. Why was the TRIM function used? What would happen if it was not used?
5 The LENGTH function is used to find the position of the last non-blank character in a number of variables.
6 The results of PROC CONTENTS shows the lengths of the new variables. Both the SCAN and REPEAT functions result in variables with a length of 200. The output from PROC PRINT shows that new varaibles are much shorter than 200 characters. Finally, PROC MEANS shows values that could be used in a length statement to change the lengths of some of the character variables.

...OTHER FUNCTIONS
If a data step is executing and a character variable is used as a numeric variable (or vice-versa), SAS performs a type conversion and allows the data step to continue. The LOG for the data step will contain a message telling you that SAS has performed such a conversion. There is no such conversion done in procedures. If you attempt to use character variables in a procedure that requires numeric variables, no conversion is done and the LOG will contain and error message, not a note about type conversion.

There are SAS functions for that can be used to convert variable types. You must convert variable types when you have character variables and need numeric variables to use a procedure such as PROC MEANS. Even in those instances when SAS will convert data types for you in a data step, it is more efficient to use the SAS functions for type conversion.

...Example 9.16...
```
data testa;
input x $ y street  $ &  zip;                                                                   1
tot1 = x + y;
tot2 = sum(of x y);
st_zip = street || '   ' || zip;
datalines;
1 2  59 LENOX AVE  12203
;
run;

data testb (drop=oldx oldzip);
set testa (rename=(x=oldx zip=oldzip));                                                          2
x      = input(oldx,2.);                                                                         3
zip    = put(oldzip,5.);                                                                         4
tot1   = x + y;
tot2   = sum(of x y);
st_zip = street || '   ' || zip;
run;
```

```
proc contents data=testa;
run;

proc contents data=testb;
run;

THE LOG...                                                                                5

1750  data testa;
1751  input x $ y street  $ &  zip;
1752  tot1 = x + y;
1753  tot2 = sum(of x y);
1754  st_zip = street || '   ' || zip;
1755  datalines;

NOTE: Character values have been converted to numeric values at the places given by:
      (Line):(Column).      1752:8    1753:15
NOTE: Numeric values have been converted to character values at the places given by:
      (Line):(Column).      1754:29
NOTE: The data set WORK.TESTA has 1 observations and 7 variables.
NOTE: DATA statement used:       real time           0.00 seconds

1757  ;
1758  run;
1759
1760  data testb (drop=oldx oldzip);
1761  set testa (rename=(x=oldx zip=oldzip));
1762  x      = input(oldx,2.);
1763  zip    = put(oldzip,5.);
1764  tot1   = x + y;
1765  tot2   = sum(of x y);
1766  st_zip = street || '   ' || zip;
1767  run;

NOTE: There were 1 observations read from the data set WORK.TESTA.
NOTE: The data set WORK.TESTB has 1 observations and 7 variables.
NOTE: DATA statement used:       real time           0.00 seconds

(contents of data set TESTA)...

Variable    Type    Len
-----------------------
st_zip      Char     15
street      Char      8
tot1        Num       8
tot2        Num       8
x           Char      8
y           Num       8
zip         Num       8

(contents of data set TESTB)...

Variable    Type    Len
-----------------------
st_zip      Char     15
street      Char      8
tot1        Num       8
tot2        Num       8
x           Num       8
y           Num       8
zip         Char      5
```

1. The INPUT statement reads X as a character variable and ZIP as a numeric variable, even though X is used later in two calculations and ZIP is later treated as a character variable.
2. The data set TESTA is read with a set statement and a RENAME option is used to change the names of two variables.
3. The INPUT function converts a character variable to a numeric variable.

4 The PUT function converts a numeric variable to a character variable.
5 The LOG from the first data step shows that variable type conversion was done in the first data step, when the character variable X was used as a numeric variable and when the numeric variable ZIP was used as a character variable. However, there are no notes in the LOG file when the second data step executes since functions were used to change variable types prior to using the varaibles.

When computing both TOT1 and TOT2, the variable X was used as a NUMERIC variable even though it is a CHARACTER variable. In creating the new variable ST_ZIP by concatenating STREET and ZIP, the variable ZIP was used as a CHARACTER variable even though it is a NUMERIC variable. The LOG indicates that SAS was tolerant of the misuse of the variables and was able to complete the data step with no errors, just NOTES. Though SAS will correct such use of variables, there is a price to pay in that each of these type conversions takes a bit of CPU time. With a few observations and passes through the data step, the price is cheap. With millions of passes through a data step, it is good idea to learn how to convert variable types explicitly.

The original data set TESTA contains a CHARACTER variable X and a NUMERIC variable ZIP. A rename data set option is used in the second data step to give variables X and ZIP new names, OLDX and OLDZIP respectively. An INPUT function is used for CHARACTER-to-NUMERIC conversions, creating NUMERIC variable X. A PUT function is used for NUMERIC-to-CHARACTER conversion, creating CHARACTER variable ZIP. The same arithmetic and concatenation operations are performed as in the previous example, but there are no NOTES in the LOG. PROC CONTENTS shows that the revised data set TEST contains a NUMERIC X and CHARACTER ZIP. Notice the length of the variables. Variable X has a length of X (remember, all NUMERIC variables have a default length of 8). The second argument in the INPUT function should be large enough to accommodate reading the widest value of X. For example, if X had been 1000, the second argument in the INPUT function would have to been at least 4. The INPUT function reads variable values just as if they were being read from a file of raw data, but it really is reading a value stored in the computers memory using the informat specified as the second argument.

Variable ZIP takes its length from the second argument in the PUT function, in this case 5. Instead of READING data as it does with the INPUT function, SAS is WRITING data with the PUT function. Thus, the second argument in the PUT function is a FORMAT, not an INFORMAT as with the INPUT function. The second argument in the PUT function should be wide enough to display the value of the first argument. Also, since you are WRITING a NUMERIC value, you must use a NUMERIC format.

The PUT function can also be used with character variables, in effect performing a CHARACTER-to-CHARACTER conversion. The SUBSTR function was used earlier to convert a 5-digit diagnosis code to a 3-digit code...

```
pdx3=substr(pdx5,1,3);
```

The SUBSTR function would have given PDX3 a length of 5 (the same as variable PDX5). If the PUT function had been used to perform the same task, PDX3 would have a length of 3...

```
pdx3=put(pdx5,$3.);
```

The final function to be discussed is the LAG function. The LAG function allows you to look backward when a data step is running and use the values of variables from observations previous to that being processed. When working with time series data, one common task is to compute a moving average.

...Example 9.17...

```
data test;
input year : $4. num den;
rate1 = 1000 * num / den;                                                              1
num_one = lag1(num);                                                                   2
num_two = lag2(num);                                                                   3
den_one = lag1(den);
den_two = lag2(den);
rate3 = 1000 * (num + lag1(num) + lag2(num)) / (den + lag1(den) + lag2(den));          4
datalines;
1990 10 1234
1991 11 1000
1992 13 1500
1993 15 1999
1994  9 1100
1995  8 1000
;
run;

proc print data=test;
run;

from the LOG...

NOTE: Missing values were generated as a result of performing an operation on missing values.
Each place is given by: (Number of times) at (Line):(Column).
2 at 1826:14   1 at 1826:21   2 at 1826:33   2 at 1826:46   1 at 1826:53   2 at 1826:65

the OUTPUT from PROC PRINT...

Obs    year    num    den     rate1     num_one    num_two    den_one    den_two     rate3
 1     1990     10    1234     8.1037       .          .          .          .        .
 2     1991     11    1000    11.0000      10          .        1234         .        .
 3     1992     13    1500     8.6667      11         10        1000       1234     9.10552
 4     1993     15    1999     7.5038      13         11        1500       1000     8.66859
 5     1994      9    1100     8.1818      15         13        1999       1500     8.04523
 6     1995      8    1000     8.0000       9         15        1100       1999     7.80678
```

1 A rate is calculated using data from one year.
2 The LAG function is used to bring back the value of the variable NUM from the last time the LAG function was executed in this statement.
3 The LAG function is used to bring back the value of the variable NUM not from the last time the LAG function was executed in this statement, but from two times ago.
4 The current and lagged values are used to compute a three year rate.

The LAG function uses a SUFFIX to indicate 'how far back' the data step goes when looking for the values of variables in previous observations. Each time the LAG function is used with a variable, it stores a value of that variable for use in a later cycle of the data step. During the first pass through the data step in this example, the LAG function has not been executed with either variable NUM or DEN. Therefore the values of LAG1(var) and LAG2(var) are missing. When the missing values are used to calculate RATE3, the result is missing. During the next pass through the data step, there is one LAG value for both NUM and DEN, i.e. LAG1(var), but not LAG2(var) values, resulting in a missing value for RATE3. The excerpt from the LOG tells you the story about the missing values. Once the third observation is reached, there are LAG1 and LAG2 values and RATE3 is calculated as a three-year moving average. The value of RATE3 for 1992 is calculated using data from 1990, 1991, and 1992, RATE3 for 1993 uses data from 1991, 1992, and 1993.

There is another function that works with lagged data, the DIF function. The DIF function returns the difference between the current value of the argument and its value the last time the DIF function was executed. Using the same data as were used in example 9.17, the next example shows how the diff function works.

...Example 9.18...

```
data differences;
set  test;
change_num = dif(num);                                                    1
change_den = dif(den);
run;

proc print data=differences noobs;
var num change_num den change_den;
run;

       change_            change_
num      num       den      den
 10        .      1234        .                                           2
 11        1      1000     -234
 13        2      1500      500
 15        2      1999      499
  9       -6      1100     -899
  8       -1      1000     -100
```

1 The DIF function is used to compute differences in the variables NUM and DEN from one observation to the next.

2 The ouput from PROC PRINT shows the values of NUM and DEN, plus the changes from observtion-to-observation. Notice that the first value for each change is missing since there's no preceding observation available to use to calculate a difference.

(10) REARRANGING DATA

If you want to conduct an analysis across observations in a data set, you can use SAS procedures. If you want to conduct an analysis within observations, you can use SAS functions. However, there are occasions when neither a procedure nor a function will suffice. Your data may be arranged in such a way that the only way to complete a given task is to first rearrange the data set. Rearranging in this sense means converting variables into observations or observations into variables. Rather than discuss methods of rearranging and showing the situations where they are applicable, two different problems are presented and methods for solving the problems are shown.

...ONE-TO-MANY

The first problem is one in which you have observations that contain multiple occurrences of a medical diagnosis. Each observation represents one person and your task is to create a table that shows how often each diagnosis occurs within the data set. The following creates the data set to be used...

```
data many_dx;
infile datalines missover;
length id $2 dx1-dx5 $3;
input id dx1-dx5;
datalines;
01 647 038
02 428 518 416
03 642 674 431 648
04 415 280 997 427 453
05 641 668 648 666 641
06 670 041 728 427 518
07 864 866
08 648 801
09 666 666 286 286
10 659 560 348 507
;
run;
```

One approach is to use PROC FREQ on each of the five diagnoses and add the results of the five tables...

```
proc freq data=many_dx;
tables dx1-dx5;
run;
```

Another approach is to rearrange the data so a new data set to be analyzed contains only one variable, diag, with one observation for each diagnosis in the original data set, i.e. create many observations from one observation (one-2-many).

...Example 10.1...

```
data all_dx (keep=diag);                                                     1
set many_dx;
diag = dx1; if diag ne ' ' then output;                                      2
diag = dx2; if diag ne ' ' then output;
diag = dx3; if diag ne ' ' then output;
diag = dx4; if diag ne ' ' then output;
diag = dx5; if diag ne ' ' then output;
run;

proc freq data=all_dx;
table diag;
run;
```

```
                                        Cumulative    Cumulative
diag    Frequency     Percent     Frequency      Percent
-----------------------------------------------------------
038            1        2.78             1         2.78
041            1        2.78             2         5.56
280            1        2.78             3         8.33
286            2        5.56             5        13.89
348            1        2.78             6        16.67
415            1        2.78             7        19.44
416            1        2.78             8        22.22
427            2        5.56            10        27.78
428            1        2.78            11        30.56
431            1        2.78            12        33.33
453            1        2.78            13        36.11
507            1        2.78            14        38.89
518            2        5.56            16        44.44
560            1        2.78            17        47.22
641            2        5.56            19        52.78
642            1        2.78            20        55.56
647            1        2.78            21        58.33
648            3        8.33            24        66.67
659            1        2.78            25        69.44
666            3        8.33            28        77.78
668            1        2.78            29        80.56
670            1        2.78            30        83.33
674            1        2.78            31        86.11
728            1        2.78            32        88.89
801            1        2.78            33        91.67
864            1        2.78            34        94.44
866            1        2.78            35        97.22
997            1        2.78            36       100.00
```

1 A new data set is created that contains only one variable, DIAG.
2 The value for each diagnosis (dx1 through dx5) in the original data set is placed in the variable diag. If the value is not missing, an observation is output to the new data set.

Using the new, rearranged data, there is only one table needed to count all the diagnosis in the original data set. The method used in example 10.1 is fine when there are a small number of diagnoses to be rearranged into a single variable. If there were many more diagnosis within each record in the original data set, this method would still work. There would just be a lot more SAS code since you have to process each diagnosis. Another solution involves the use of an array.

...Example 10.2...

```
data diags (keep=diag);
set many_dx;
array dx(5) dx1-dx5;                                                          1
do j=1 to 5;                                                                  2
   diag=dx(j);                                                                3
   if diag ne ' ' then output;
end;                                                                          4
run;

proc freq data=diags;
table diag;
run;
```

1 An array is used to store values of the variables dx1 through dx5.
2 A DO loop is used to look at each element of the array dx (the same as looking at the values of variables dx1 through dx5).
3 The same type of logic is used as in example 10.1 except an array element is used instead of the original variable name.
4 An END statement closes the DO loop.

Arrays are merely alternative ways of referring to variables. That is a simple definition and not all inclusive. However, it describes the use of an array in example 10.2. The use of arrays is tied to the use of DO loops. Loops allow you to execute one or more SAS instructions a given number of times. The loop starts with a DO statement and ends with an END statement. There are many types of loops. The one in example 10.2 uses a variable (J) as a counter. The value of 'J' starts at one and ends at five. All statements between the DO and END statements are executed five times. Within the loop, the variable DIAG is set equal to array elements. When J=1 (the first time through the loop), DIAG is set equal to DX(1), i.e. variable DX1. The last time through the loop, J=5 and DIAG is set equal to DX(5), i.e. variable DX5.

The advantage of this method is that no matter how many diagnoses are present in the original data, the data step would not be any longer. Only the size of the array and number of passes through the do loop would change.

...Example 10.3...
```
data many_dx;
infile datalines missover;
length id $2 dx1-dx15 $3;
input id dx1-dx15;                                                                  1
datalines;
01 647 038 320 348 V66
02 428 518 416 710 710 424 666 443
03 642 674 431 648 584 427 518 427 V46 038 648 663 666 648 535
04 415 280 997 427 453 626 414 250 218 458
05 641 668 648 666 641 668 668 669 669 672 666 665 664 648 622
;
run;

data diags (keep=diag);
set many_dx;
array dx(15);                                                                       2
do j=1 to 15;                                                                       3
   diag=dx(j);
   if diag ne ' ' then output;
end;
run;

proc freq data=diags;
table diag;
run;
```

1 The original data set now contains fifteen diagnoses per observation.
2 An array is used with 15 elements. If no list of variables follows the array name, the variables are assumed to have the same name as the array - in this case dx1-dx15.
3 The upper limit on the loop is now 15.

The data step that creates the new data set is no longer than in example 10.2 where there were only five diagnoses. The LOG file from execution of the data step looks as follows...

```
NOTE: There were 5 observations read from the data set WORK.MANY_DX.
NOTE: The data set WORK.DIAGS has 48 observations and 1 variables.
```

Five observations were read from the original data set and forty-eight observations were written to the new data set, one observation for each diagnosis found in the original data.

There is yet another method for rearranging data, PROC TRANSPOSE. This procedure can be used to turn variables into observations, and to turn observations into variables. In each example shown thus far, variables (diagnoses) within one observation have been turned into one variable (diag) occurring across many observations.

...Example 10.4...

```
data many_dx;
infile datalines missover;
length id $2 dx1-dx5 $3;
input id dx1-dx5;
datalines;
01 647 038
02 428 518 416 710
03 642
04 415 280
05 641 668 648 666 641
;
run;

proc transpose data=many_dx out=diags;                                            1
var dx1-dx5;                                                                      2
run;

proc print data=diags;
run;

Obs    _NAME_    COL1    COL2    COL3    COL4    COL5                             3
 1      dx1      647     428     642     415     641
 2      dx2      038     518             280     668
 3      dx3              416                     648
 4      dx4              710                     666
 5      dx5                                      641
```

1 PROC TRANSPOSE is used to rearrange the data in data set many_dx, resulting in data set diags.
2 A VAR statement specifies the variables to be transposed (rearranged).
3 The data set created by PROC TRANSPOSE has data in observations that were in variables in the original data set.

Though the data have been rearranged, the result is not quite what was desired, a data set with one variable (diag). Using a by statement in PROC TRANSPOSE will make the resulting data set look more like those produced with data steps and arrays in previous examples.

...Example 10.5...

```
proc transpose data=many_dx out=diags;
var dx1-dx5;
by id;                                                                            1
run;

proc print data=diags;
run;

Obs    id    _NAME_    COL1                                                       2
  1    01     dx1      647
  2    01     dx2      038
  3    01     dx3
  4    01     dx4
  5    01     dx5
  6    02     dx1      428
  7    02     dx2      518
  8    02     dx3      416
  9    02     dx4      710
 10    02     dx5
 11    03     dx1      642
 12    03     dx2
 13    03     dx3
 14    03     dx4
 15    03     dx5
 16    04     dx1      415
 17    04     dx2      280
 18    04     dx3
 19    04     dx4
 20    04     dx5
 21    05     dx1      641
 22    05     dx2      668
 23    05     dx3      648
 24    05     dx4      666
 25    05     dx5      641
```

1 A BY statement is added to PROC TRANSPOSE specifying that the variables specified in the VAR statement should be transposed within each value of the ID variable.
2 The resulting data set is much closer to the desired data set.

Notice that in both examples 10.4 and 10.5, PROC TRANSPOSE created its own names for the variables in the output data set, COL followed by a numeric suffix. In example 10.4, the variable named COL1 is the one that should be named DIAG. The procedure adds a new variable to the data set, _NAME_, and its value across the observations is the names of the variables occurring in the VAR statement. In example 10.5, the BY variable is also added to the data set, and there are the same number of observations within each different value of by variable, even if some have missing data in the variable COL1. More changes to PROC TRANSPOSE can produce a data set exactly like that created in examples 10.1 and 10,2.

...Example 10.6...

```
proc transpose
data=many_dx
out=diags (keep=col1 rename=(col1=diag) where=(diag ne ' ')) ;                          1
var dx1-dx5;
by id;
run;

proc print data=diags;
run;

Obs    diag                                                                             2
  1    647
  2    038
  3    428
  4    518
  5    416
  6    710
  7    642
  8    415
  9    280
 10    641
 11    668
 12    648
 13    666
 14    641
```

1 Several data set options are used on the output data set. First, a KEEP option specifies that only the variable COL1 is to be added to the data set DIAGS. Next, a RENAME option changes the variable name COL1 to DIAG. Finally, a WHERE option tells PROC TRANSPOSE to only output observations in which the value of the variable DIAG is not missing.
2 The data set is exactly what is desired, containing only one variable named DIAG with no observations having missing data..

...MANY-TO-ONE

The second problem is one in which the there is a data set containing the information on the number of births to mothers who took drugs during their pregnancy. There are three variables in the data set: the year the birth occurred (YEAR); whether the mother took illegal drugs, therapeutic or prescribed drugs, or no drugs (DRUGS); the number of births in the observation (BIRTHS). The task is to put all the data for a given year into one observation and then compute the percentage of births to mothers who took either illegal and/or therapeutic+prescribed drugs. In the data set, the variable DRUGS has the following values: 1, no drugs taken; 2, therapeutic+prescribed drugs taken; 3, illegal drugs taken As in the first ONE-TO-MANY example, a data step can be used to rearrange the data.

...Example 10.7...
```
data births1;                                                                              1
length year $4 drugs $1;
input year drugs births @@;
datalines;
1993   1   269155  1993   2     8430 1993   3     4792  1994   1   261746
1994   2    12132  1994   3     4067 1995   1   250486  1995   2    16812
1995   3     3744  1996   1   240699 1996   2    18964  1996   3     3948
1997   1   231441  1997   2    21679 1997   3     3856  1998   1   230911
1998   2    23218  1998   3     3619 1999   1   227689  1999   2    24066
1999   3     3392  2000   1   225870 2000   2    25141  2000   3     3219
;
run;

data births2;
retain d1-d3;                                                                              2
set births1;                                                                               3
by year;
if drugs eq '1' then d1 = births;                                                          4
else
if drugs eq '2' then d2 = births;
else
if drugs eq '3' then d3 = births;
if last.year then output;                                                                  5
run;

data births2 (keep=year total pct1 pct2);                                                  6
set births2;
total = sum (of d1-d3);
pct1 = 100 * d2 / total;
pct2 = 100 * d3 / total;
label
year  = 'YEAR OF BIRTH'
total = 'NUMBER OF BIRTHS'
pct1  = '% THERAPEUTIC OR PRESCRIBED'
pct2  = '% ILLEGAL'
;
run;

proc print data=births2 noobS label;
var year total pct1 pct2;
format total comma8. pct1 pct2 6.1;
run;
```

```
                                  %
             NUMBER    THERAPEUTIC
YEAR OF          OF        OR           %
 BIRTH       BIRTHS    PRESCRIBED     ILLEGAL                                              7
 1993       282,377         3.0          1.7
 1994       277,945         4.4          1.5
 1995       271,042         6.2          1.4
 1996       263,611         7.2          1.5
 1997       256,976         8.4          1.5
 1998       257,748         9.0          1.4
 1999       255,147         9.4          1.3
 2000       254,230         9.9          1.3
```

1 A data set is created with one observation for every year-drug combination.
2 The retain statement is used to prevent values of the variables D0, D1, and D2 from being set to missing during execution of the data step.
3 The original data are read with a SET statement plus a BY statement.
4 The value of the variable DRUGS is checked and the value of the variable births is assigned to the appropriate variable.
5 An observation is written to a new data set when the last observation is by group is read.

6 Another data step is used to compute percentages.
7 The output from PROC PRINT shows the percentage of births to mothers who took the two different classes of drugs within each year.

One feature of the original data set should be noted. There are three observations within each year, one observation for no, therapeutic+prescribed, and illegal drugs. If any of the years did not have all three such observations, the data step that creates data set BIRTHS2 would have to be modified as follow.

```
data births2;
retain d1-d3;
set births1;
by year;
if first.year then do;
   d1 = 0; d2 = 0; d3 = 0;
end;
if drugs eq '1' then d1 = births;
else
if drugs eq '2' then d2 = births;
else
if drugs eq '3' then d3 = births;
if last.year then output;
run;
```

The new statements that begin with IF FIRST.YEAR.... ensure that the values of D0, D1, and D2 are set to zero when each new year is encountered in the data set BIRTHS1 and that data assigned to any given year actually belong to that year. In other words, no data ends up in a a new year that have been retained from the previous year. The data step that creates data set BIRTHS2 could be modified to use an array.

...Example 10.8...

```
data births2;
array d(3);                                                              1
retain d;                                                                2
set births1;
by year;
if first.year then do j=1 to 3;                                          3
   d(j)=0;
end;
d(drugs) = births;                                                       4
if last.year then output;
drop j;                                                                  5
run;

data births2 (keep=year total pct1 pct2);
set births2;
total = sum (of d1-d3);
pct1 = 100 * d2 / total;
pct2 = 100 * d3 / total;
label
year  = 'YEAR OF BIRTH'
total = 'NUMBER OF BIRTHS'
pct1  = '% THERAPEUTIC OR PRESCRIBED'
pct2  = '% ILLEGAL'
;
run;

proc print data=births2 noobS label;
var year total pct1 pct2;
format total comma8. pct1 pct2 6.1;
run;
```

```
                                 %
                NUMBER     THERAPEUTIC
YEAR OF             OF          OR            %
 BIRTH          BIRTHS     PRESCRIBED     ILLEGAL
 1993          282,377         3.0           1.7                                         6
 1994          277,945         4.4           1.5
 1995          271,042         6.2           1.4
 1996          263,611         7.2           1.5
 1997          256,976         8.4           1.5
 1998          257,748         9.0           1.4
 1999          255,147         9.4           1.3
 2000          254,230         9.9           1.3
```

1 An array statement is used.
2 The values of the variables in the array are retained.
3 A do loop is used with the array to set values of the variables D1, D2, and D3 to 0 when each new year is encountered.
4 The value of variable DRUGS is used to assign the value of the variable BIRTHS to the appropriate variable. Notice that DRUGS is a character variable and the index of an array must be numeric. The LOG file shows that SAS does a character-to-numeric conversion when D(DRUGS)=BIRTHS is encountered.
5 The index variable from the do loop is dropped.
6 The output from PROC PRINT is identical to that from the data step without the array.

As in the ONE-TO-MANY example, the advantage of using the array is that the data step that rearranges tha data would not change very much if there were more categories for the variable DRUGS. In example 10.8, there were three categories of drug use. If there had been ten categories, the data step would look as follows.

```
data births2
array d(10);
retain d;
set births1;
by year;
if first.year then do j=1 to 10;;
   d(j)=0;
end;
d(drugs) = births;
if last.year then output
drop j;
run;
```

The data step is no longer than example 10.8 where there were three categories of drug use. PROC TRANSPOSE can also be used to rearrange the data in data set BIRTHS1.

...Example 10.9...

```
proc transpose data=births1 out=births2;                                                1
var births;                                                                             2
id drugs;                                                                               3
by year;                                                                                4
run;

proc print data=births2;
run;

Obs    year    _NAME_       _1        _2      _3
 1     1993    births    269155      8430    4792                                       5
 2     1994    births    261746     12132    4067
 3     1995    births    250486     16812    3744
 4     1996    births    240699     18964    3948
 5     1997    births    231441     21679    3856
 6     1998    births    230911     23218    3619
 7     1999    births    227689    24066    3392
 8     2000    births    225870    25141    3219
```

1 PROC TRANSPOSE is used to rearrange the data in data set BIRTHS1 to create data set BRTHS2.
2 The VAR statement specifies the variable whose values will appear in the new data set.
3 The ID statement specifies the variable that controls the variable that will contain the value of the variable in the VAR statement.
4 The BY variable specifies the variable that controls the observation into which variables are placed.
5 The output from PROC PRINT shows that the data set BIRTHS2 needs new variable names, and that a new variable _NAME_ was added to the data set.

By default, PROC TRANSPOSE uses an underscore plus the values of the ID variable as a suffix to create the variable names in the output data set. A PROC TRANSPOSE option and a data set option can be to alter the content of data set BIRTHS2.

...Example 10.10...

```
proc transpose data=births1 out=births2 (drop=_name_) prefix=d;                  1
var births;
id drugs;
by year;
run;

proc print data=births2;
run;

Obs    year      d1        d2      d3
 1     1993    269155     8430    4792                                            2
 2     1994    261746    12132    4067
 3     1995    250486    16812    3744
 4     1996    240699    18964    3948
 5     1997    231441    21679    3856
 6     1998    230911    23218    3619
 7     1999    227689    24066    3392
 8     2000    225870    25141    3219
```

1 A data set option is used to drop the variable _NAME_ from the data set BIRTHS2. The PREFIX= option replaces the default prefix on an underscore with a D.
2 The data set is exactly what is desired, correct variable names and no extra variables.