Formats, Informats and How to Program with Them

Ian Whitlock, Independent SAS Consultant, Kennett Square, PA

Abstract

Formats tell how to display stored data and informats how to read them. In other words, they allow the separation of data from the values one works with. While most programming languages provide some sort of formatting capability, SAS® provides an extensive system that comes to dominate how one works with SAS.

This talk will review some of the supplied formats, show you how to make your own with code or data, and then supply programming examples for how to get the best use of this system.

Introduction

Formats allow a computer language to separate how data are stored from how the data are displayed. Informats allow the language to separate how data are entered from how the data are stored. In some sense formats and informats must play some role in almost all computer languages because numbers are rarely stored as they appear. The difference in SAS is the extent to which formats and informats are implemented. Thus they play a far more important role in SAS than they do in many other computer languages.

We use the term "formats" sometimes to refer to the class of both formats and informats, and sometimes to refer only to formats as distinguished from informats. It is left to the reader to distinguish, which is intended, based on context.

There are two types of formats - those automatically supplied by SAS, and those that you as a SAS programmer can create.

System Formats and Informats

There are formats to read and write numbers. Do you want them with commas? How many decimal places? Or do you prefer hexadecimal to decimal? The on-line documentation for SAS Version 9.1.3 lists 41 formats for character variables (up from 15 in version 8) and 93 for numeric (up from 93 in version 8). The corresponding list of informats, although shorter, is still extensive. Why are there so many? SAS has only two types of data values - character and real floating point. It falls to the system of informats to get data from the rich external world of many types into these two categories. Similarly the system of formats must display to the external world in many more types than just the two that we have. From this point of view it is not surprising that SAS had to have such a large system of formats and informats.

The two most basic formats are the F format for reading (or displaying) character digits and the $CHAR format for reading (or displaying) characters. Typically the $-sign is used to indicate character data.

Suppose that I have a number X somewhere between 3 and 4. How wide should the display field be? How many decimal places should there be? In general format names use an integer immediately following the name to indicate the total width (including any decimal point), followed by how many of those places should come after the decimal point. Thus F8.3 would say to display my number in 8 columns with exactly three rounded decimal places showing. The whole thing should be right justified.

So in the above case there would be exactly three leading blanks. In practice, one rarely gives the name F and only gives the 8.3. For example, I might have

```
data _null_ ;
   x = 12 / 3.7 ;
   put x 8.3 ;
run ;
```

This data step would write, " 3.243", on the log.

On the other hand, the informat F8.3 says something significantly different! Here a decimal point is usually displayed in the number. The 3 in F8.3 says if there is no decimal point shown, then assume it is 3 places from the right hand end, i.e. divide by 1000. Consequently the decimal specifier in formats is important to know how to display the number, but dangerous when

used in an informat because it is usually a mistake that will look as if it is working, until it reads an integer value. However, on some occasion you may have to read decimal data with an assumed decimal point for data coming from a legacy system.

Perhaps the most important area for formats is that of dates. How should SAS store a date? Characters would rule out calculations like what month will it be in 82 days, so numeric is the obvious choice. But even having numbers does not mean that one can do arithmetic. Some computer systems store November 5, 2007 as 20071105. This is a number, but it is useless for doing date calculations, since 20071105 + 82 is not even a date.

About 400 years ago Rene Descartes tied geometry to numbers by choosing a 0 point and then a 1 point on a line. SAS does the same thing for the time line. It uses January 1, 1960 as the zero point and 1 to indicate one day. Thus January 2, 1960 is 1 and December 31, 1959 is -1. So November 5, 20007 is 17,475 days since January 1, 1960. Who can figure out their birthday that way? Aha! We said the system was good for calculations; we did not say it was a good thing to show your clients. Remember formats and informats, that is the topic and informats should supply the solution. We need informats to read human recognizable dates into SAS and formats to display dates in human terms for our reports, but these human readable forms do not make a good way to store dates because they are hard to use in date calculations.

It should not surprise you by now to find there are 72 different formats for displaying dates listed in the on-line documentation for version 9.1.3. Here are a few examples using August 19, 2001.

DATE9. Displays army style (5Nov2007)

DAY2. Displays the day of the month (11)

Julian7. Displays Julian date (=2007309)

MMDDYY8. Displays American (11/05/07)

MONYY7. Displays (Nov2007)

WEEKDATE29. Displays (Monday, November 5, 2007)

WORDDATE18. Displays (November 5, 2007)

The lesson should be clear; data should be stored for easy calculation. Formats and informats should be able to display and read the data in any reasonable fashion. A typical beginner's mistake is to fail to store dates as SAS dates, and consequently using elaborate and often incorrect routines to do simple date calculations provided by SAS functions. A description of the SAS date handling functions is beyond the scope of this paper, but coding a SAS date seems reasonable. To assign the variable DATE using a human readable date (called a date literal), use the form indicated in the following example.

```
date = "5nov2007"d ;
```

Note there are no spaces between the double quote and the letter, d.

Problem - calculate your age in days. Let's assume you were born on September 15, 1978.

```
data _null_ ;
  AgeInDays=today()-"15sep1978"d;
  put AgeInDays comma6. ;
run ;
```

SAS also provides a similar system for date time values as the number of seconds since midnight January 1, 1960. In addition there is a system to measure relative time in seconds since midnight, and another to measure absolute time since midnight of December 31, 1959, i.e. the zero second of January 1, 1960. It is left to the reader to explore how time is stored, and then calculate his/her age in seconds.

Make Your Own Formats

With PROC FORMAT SAS provides the ability to make formats and informats. They are stored in a SAS catalog typically called FORMATS. (In general catalogs are used to hold system information created by the user. They differ from data in that the user cannot usually see inside a catalog entry.) By default the catalog is stored in the library, WORK.FORMATS. In this case the formats exist only for the life of the SAS session.

A common example is provided by storing gender information. For example, we typically store the number 1 for male and 2 for female. It would be embarrassing to hand a client a report from PROC PRINT with

ID	Gender	State
100001	1	LA
100002	2	LA
100003	.	AL

We need a format to dress up the report.

```
proc format ;
   value sex
      1 = "Male"
      2 = "Female"
      . = "Unknown"
   ;
run ;

proc print data = report ;
   var id gender state ;
   format gender sex7. ;
run ;
```

The VALUE statement provides a set of translations. On the left of the equal sign are the values stored. On the right are the character values to display. So now the report is.

ID	Gender	State
100001	Male	LA
100002	Female	LA
100003	Unknown	AL

This provides the simplest most basic use for formats.

Now let's consider an AGE variable. In a frequency report we might get any and all ages, say between 5 and 85. But we would like to get age collapsed into a few groups. A format is the answer.

```
proc format ;
   value AgeGrp
      1 - 10 = "Child"
      11 - 20 = "Teenager"
      21 - 40 = "Adult"
      40 <-< 65 = "Middle Aged"
      65 - high = "Senior"
      other = "Error"
   ;
run ;
```

The dash indicates a range of real numbers. The number, 40, is considered "Adult" not "Middle Aged" because the less than sign indicates that 40 is to be excluded from the "Middle Aged" range. Similarly exactly 65 is excluded from that range. "High" is a key word indicating the largest number. Note that key words are not quoted. Similarly there is a key word "low" for the lowest negative number (excluding missing values). Remember that on a computer there is a lowest number and a highest one. "Other" is the key word for all values not accounted for.

It is not surprising that a print would show the labels instead of ages, but what about PROC FREQ? This procedure also respects formats for counting so that all people between 21 and 40 will be counted as adults. This provides a second important use of formats - grouping. In many languages one would have to create a new variable indicating the group in a process called recoding. Formats often make recoding unnecessary in SAS. However not all procedures respect formats so one must check each case individually. In general, formats are respected if it makes sense to do so.

How does the PROC FREQ know our variable, AGE, had a format? We could have told it in a FORMAT statement just as we did in the PRINT example, but it would be nice to tell the dataset instead, so that any procedure could check with the data to see if a format is present. This can be done by placing a FORMAT statement in the DATA step that makes the dataset. What if the data were made by a procedure or somebody else?

PROC DATASETS can modify the header of SAS dataset where knowledge of the format is stored.

```
proc datasets lib = survey ;
   modify agedata ;
   format age agegrp. ;
quit ;
```

Note that only the format name is stored in the header. Remember the format itself is stored in the WORK.FORMATS catalog.

After this we can produce the collapsed counts with:

```
proc freq data = survey.agedata ;
   table age ;
run ;
```

If, for some reason, we wanted to see the ages instead of groups we could remove the format association with a format statement.

```
format age ;
```

One might even use

```
format _all_ ;
```

to remove all format associations. This is often convenient when you do not have the associated formats or they are interfering with debugging and you need to see the values. Thus we see that formats may be applied, changed, and removed without ever modifying the data values.

Permanent Formats

PROC FORMAT has a LIBRARY parameter (abbreviated LIB) to specify where formats should be stored. Thus one might have

```
libname library "c:\myplace" ;
proc format lib = library ;
   /* code to make formats */
run ;
```

In this case the formats would be stored in LIBRARY.FORMATS. We use the libref LIBRARY because SAS by default looks in two places for formats

1. Work.Formats
2. Library.Formats

Why LIBRARY? The concept of a library of formats in SAS is older than the notion of a libref and catalogs.

The system option FMTSEARCH allows you to specify which libraries and even which catalogs should be searched for formats. For example, the statement

```
options fmtsearch =
    ( mylib mylib.special ) ;
```

causes the system to look for formats in four places:

1. Work.Formats
2. Library.Formats
3. MyLib.Formats
4. MyLib.Special

in that order. The first format found is the one used when the same name appears in more than one catalog.

If the format referenced is not found in any of the search catalogs, then the system issues an error message and may stop processing depending on the setting of the system option, FMTERR. By default it is set so that processing is stopped. To continue processing ignoring the request for the format, use

```
options nofmterr ;
```

With this option, if a format is not found the system issues a NOTE that the format is not found, but continues processing.

One fear, that often prevents beginners from making permanent formats is the fear that they cannot find the code or retrieve the format specifications when the code is lost. There is a PROC FORMAT option, FMTLIB which will report formats and informats. You can use a SELECT statement to obtain a report on specific formats or EXCLUDE statement to obtain all but a few formats.

Thus there is no good reason not to make permanent formats and in fact there are good reasons to do so. Formats can standardize and document how a project looks at its data values. Formats should become part of the standard environment in which project code is written rather than a part of the code. If there are many formats, it can take a significant amount of time to create the formats and they can take up a significant portion of the code in a program. Thus it makes a great deal of sense for a project to build a library of formats, just as it should have libraries of permanent SAS data.

Using SAS Datasets to Specify Formats

Just as the FMTLIB option can make a report on stored formats, the CNTLOUT= option names a SAS dataset that holds information capable of making the formats. Again SELECT or EXCLUDE statements can control which formats are represented in the dataset.

So far we have made formats with code, but now we can go directly from data to formats. This can be a very powerful idea because formats can be data dependent. Another thing this hints at is that formats can be quite large, say with a thousand or even a hundred thousand entries. Here it would be painful to have to write out such code.

Armed with an appropriate dataset, say, FMTDATA. The code is

```
proc format cntlin = fmtdata ;
run ;
```

So what properties must FMTDATA have? At a minimum only three variables are required.

1. FMTNAME to supply the name of the format (including the $-sign for character formats).
2. START to supply the left hand side of an assignment.
3. LABEL to supply the right hand side of an assignment.

Suppose I am dealing with an educational survey and have a SAS data set with ID, an eight digit school identifier and SCHOOLNAME, a 40 byte character variable holding the name of the corresponding school. Then I might choose to store only the identifier on other data sets and use a format, $SCHLFMT. to display the names of the schools. The code to make the format might go like this.

```
data fmtdata ;
   retain fmtname "$schlfmt" ;
   set schools
     (keep = id schoolname
      rename = ( id = start
            SchoolName = Label )
     ) ;
run ;

proc format cntlin = fmtdata ;
run ;
```

Or perhaps you would like to show both the identifier and the school name. Then use

```
data fmtdata
    (keep = fmtname start label);
   retain fmtname "$schlfmt" ;
   set schools
     (keep = id schoolname
      rename = ( id = start )
     ) ;
   Label = id||", "||SchoolName
run ;

proc format cntlin = fmtdata ;
run ;
```

When you have learned the full power of DATA step character handling functions, you will find many opportunities for variations of this idea.

So, just how large can a format be? Well typical of SAS, there is no prescribed limit. A format must be held in memory during its use so there is a practical limit. I consider 100,000 entries my limit, but even this can depend on how long the labels are.

Why do we need the format name as a variable? Couldn't it be a parameter on PROC FORMAT? No, one can actually specify many formats with one dataset, so it is better the way it has been designed.

What are the optional values and what kind of values must they have? The version 9 documentation leaves much to be desired on this point. The simple answer is: make a small format with code having the features you wish to know about. Then use the CNTLOUT= option to make the corresponding dataset and study it. For example, consider the key words and ideas we introduced earlier in a format.

```
proc format cntlout=fmtdata ;
   value datachk
       low -< 0 = "Negative"
       0 = "Zero"
       0 <- high = "Positive"
       other = "Missing"
   ;
   select datachk ;
run ;
```

You should quickly find that the relevant variables appear to be:

Start, End, Label, SExcl, EExcl, and HLO

are the important variables in the table, and that "H" is for HIGH, "L" is for LOW, and "O" is for other. Later we will add "F" for specifying a nested format.

Start	End	Label	SSexcl	EExcl	HLO
Low	0	Neg	N	Y	L
0	0	Zero	N	N	
0	High	Pos	Y	N	H
Other	Other	Missing	N	N	O

Armed with this information we can add an other condition to our $SCHLFMT.

```
data fmtdata ;
   retain fmtname "$schlfmt" ;
   if eof then
   do ;
      hlo = "O" ;
      label = "Error" ;
      output ;
   end ;
   set schools
     (keep = id SchoolName
      rename = ( id = start
            SchoolName = Label )
     ) end = eof ;
   output ;
run ;

proc format cntlin = fmtdata ;
run ;
```

Although it is not clear from the above, the value of START is irrelevant when HLO = "O". The format used to learn abut HLO is interesting in its own way. The following code provides a quick check for missing and negative values.

```
proc freq data = anyset ;
   tables _numeric_ / missing ;
   format _numeric_ datachk. ;
run ;
```

Remember PROC FREQ respects the grouping indicated by a format. Similarly

```
proc format ;
   value $datachk
      " " = "Blank"
      other = "present"
   ;
run ;
```

provides a quick check for character variables.

Informats

Up to this point we have concentrated on formats. It is now time to consider informats. They are made with an INVALUE statement. They come in two varieties - those that make numeric values and those that make character values.

Suppose you are reading a four digit column of numbers, but some of the entries are "XXXX" to indicate the value is missing. If you use

```
input ... number 4. ... ;
```

then there will be an invalid data message on the log. You could use

```
input ... number ?? 4. ... ;
```

to suppress the invalid data message, but then you also miss the message when it as appropriate. The answer is to make in informat.

```
proc format ;
   invalue numchk
   "XXXX" = . ;
run ;
```

Now the value "XXXX" is quietly read and converted to missing. What about other values? When a format or informat does not specify how to treat a value then the default treatment is used. Thus

```
input ... number numchk4. ... ;
```

will turn all other values over to BEST4. for reading as numbers. If a value cannot be read as a number then there will be a message to alert you to the problem.

As an example of a character informat we might return to our gender example. This time we want to read the numbers 1 or 2, but store the words "Male" or "Female".

```
proc format ;
   invalue gender
      "1" = "Male"
      "2" = "Female"
   ;
run ;

data _null_ ;
   length sex $ 6 ;
   input sex $gender1. ;
   put sex= ;
cards ;
1
2
;
```

Note that the LENGTH statement is needed to get the proper length for the variable, SEX.

Key Idea

Informats and formats have been useful for reading from and writing to a buffer, but there is more. SAS also supplies the INPUT function to read from a character variable according to an informat and the PUT function to write to a character variable according to a format.

Suppose CHARX is a character variable holding 3 digits. Then we can convert to a numeric variable NUMX using

```
numx = input ( charx , 3. ) ;
```

Similarly we might convert in the other direction with

```
charx = put ( numx , 3. ) ;
```

Note that in both cases the format is numeric. In the first case it is numeric because the resulting value, NUMX, is numeric. In the second case it is a numeric informat because we are writing a numeric value. Beginners often get confused here and want to add a $-sign, but it is wrong. Just remember, INPUT always reads character stuff and PUT always writes character stuff.

Suppose we have a numeric id and we want a character id with leading zeros. There is a system format, Z for writing leading zeroes. Hence

```
charID = put ( numID , z8. ) ;
```

would convert to an 8-digit character identifier with leading zeros.

Often one wants to keep the same names. This gets a little tricky because it involves several ideas at once. Suppose we have a dataset W with a variable X that is character (always digits). We want to end up with a dataset W and a numeric variable X corresponding to the original X. We have to free up the name X on the input set so that we have it available for use on the output dataset. Here is the code.

```
data w ( drop = temp ) ;
   set w ( rename = (x=temp) ) ;
   x = input ( temp, best12. ) ;
run ;
```

The code is simple, but I suspect very few programmers discover and put together all the features needed for this problem. Most of them learn it from somebody who already knows the answer. It is my favorite problem for illustrating the inadequacy of SAS documentation, since every feature is mentioned, but nowhere is the problem discussed. It is left to a "faq" sheet at http://support.sas.com.

Recoding and Look-up

In considering grouping formats, I pointed out that recoding is often not needed in SAS. However, sometimes one must recode because, say, a procedure like REG requires it. Consider the AGE variable we used before. Now we make a new format

```
proc format ;
   value AgeGrp
      1 - 10 = "1 Child"
      11 - 20 = "2 Teenager"
      21 - 40 = "3 Adult"
      40 <-< 65= "4 Middle Aged"
      65 - high = "5 Senior"
      other = ". Error"
   ;
run ;
```

and use it in a DATA step

```
data recoded ( drop = age ) ;
   set agedata ;
   agegrp =
     input(put(age,agegrp1.),1.);
run ;
```

to recode the AGE variable. Programmers coming from another statistical package often ask, "Where is the recode procedure located? The answer is that formats can do the job and no procedure is needed.

Now consider a slight change in point of view and the format $SCHLFMT that we made from a SAS dataset. Here the line

```
SchoolName = put(id, $schlfmt40.);
```

can be thought of as a look-up function - given the ID value for a school look up the name of the school.

Often one solves this type of look-up problem with a sort and merge by ID where one file has the ID and needed data while the other file has ID and SchoolName. There are advantages to both.

The primary advantage of the format method is that the files do not need sorting. This can be very important when you need to look up values based on several different variables. The format method provides flexibility.

On the other hand, the merge solution is probably better when one wants to look up many different variables where the look-up information is all in one file. The merge is more efficient when the files happen to already be in their required order. As the size of the look-up file grows, so does the importance of this efficiency.

SAS arrays are numerically indexed. Sometimes it would be very convenient if one could have an array indexed by character values. Suppose we wanted to count how many times certain words were used in some text. For a list of say 100 words we might set up an array with

```
array count (100) ;
```

The problem is which word goes with which array element. In this case it doesn't matter as long as it is fixed in some specific order. We might use an informat, SPECIAL, to recode the special words into the numbers from 1 to 100. We could then send all other words to 0. Now for any value of WORD we could use the following code to subset with the WANTED format.

```
x = input ( word , special3. ) ;
if 0 < x <= 100 then
    count [ x ] + 1 ;
```

Eliminating IF's

All of the problems discussed in the previous section have lot in common, and they could have been solved with IF statements. IF statements provide a powerful method for making programs flexible. However they should be used with great care because they also make a program more difficult to follow. A reasonable goal is to eliminate as many IF's as possible.

Consider the recoding of age. We could have written

```
data recoded ( drop = age ) ;
   set agedata ;
   if 1 <= age <= 10 then
      AgeGrp = 1 ;
   else
   if 11 <= age <= 20 then
      AgeGrp = 2 ;
   else
   if 21 <= age <= 40 then
      AgeGrp = 3 ;
   else
   if 40 < age < 65 then
      AgeGrp = 4 ;
   else
      AgeGroup = 5 ;
run ;
```

Note that this step is longer and more tedious with poorer documentation. After comparing it with our original recoding, it is clear that what the format did was to give us the ability to lift the tedious IF/ELSE chain out of the DATA step and move it to a more appropriate place. But then if it were a stored format it would not even be in the code at all. In any case it allows us to treat the IF/ELSE chain as a black box.

For the example the IF/ELSE chain was not too bad because it was relatively short, but the school name look-up in the second example might have gone on for thousands of lines. In the school name case, remember we actually started with a dataset and made the format from data. It should now be clear that we have three different methods of storing information:

1. Store it in a dataset
2. Store it in a format
3. Store it in SAS code

So formats provide another method of storing and using information. In considering the three methods, code is usually the hardest to modify, the most tedious to change, and the most error prone. On the other hand, PROC FORMAT will display an error messaage when overlapping ranges are specified, but the compiler will find nothing wrong with the corresponding IF/ELSE chain and yet it is likely that the code is wrong.

I call the above DATA step "wall paper" code. It all looks the same with a simple pattern to it. In general it is a good idea to remove wall paper code, and formats provide one important tool for doing this. Arrays provide another most important tool in this area. The third tool, SAS macro, provides the most sophisticated tool for this purpose. However, it is a good idea for the beginner to get a sound grasp of the principles involved with formats and arrays before turning to macro code.

Picture Formats

The VALUE statement provides you with the ability to specify a single display value for a collection of stored values. Sometimes one would like to describe a pattern telling how to insert some extra symbols in the number. For example, display the phone number, 1234567890, as (123)456-7890. Note that this cannot be done in general with a VALUE statement without listing every phone number. Or perhaps you would like to display money values in thousands of dollars, e.g. 143,265 dollars would display as $143K. For this type of problem we need a new statement, the PICTURE statement. The idea originally came from COBOL which used picture formats.

For the telephone example, one might try

```
proc format ;
   picture phone
       0 - 9999999999 =
          "(999)999-9999"
   ;
run ;
```

However, this does not work; 1234567890 is shown as

```
123)456-7890
```

with the leading left parenthesis missing. When the leading characters are not numbers one has to specify them in a special option prefix.

```
proc format ;
   picture phone
       0 - 9999999999 =
          "*999)999-9999"
          (prefix="(")
   ;
run ;
```

Note that in the picture itself I placed an asterisk first. It doesn't matter what this symbol is, but space must be reserved for the prefix symbols. This makes for what I call the "Mother May I" type of syntax after the children's game "giant steps". It is a case where the developer took short -cuts at the expense of the user.

For the money in thousands example, we could use

```
proc format ;
    picture money
        500 - 999499 = "*009K"
            (prefix="$" mult=.001)
        other = "error" ;
run ;
```

In this example we added another picture option, MULTIPLIER= (or MULT=). This tells SAS to divide the number by 1000 before placing into the picture. Again there is a funny thing. If we consider the number, 1899, it is displayed as

```
$1K
```

instead of the expected, $2K. In general, formats will round numbers to the indicated number of decimal places; however picture formats truncate instead of rounding. Fortunately there is a ROUND option, but it is a general format option instead of one specific to the PICTURE statement. Hence, it cannot be placed in the parentheses with the PREFIX= option as one might expect. The correct form of the statement is

```
    picture money (round)
        500 - 999499 = "*009K"
            (prefix="$" mult=.001)
        other = "error" ;
```

Finally suppose we want to display money values in thousands, but allow one decimal place. Then one might think that

```
    picture money (round)
        500 - 999499 = "*009.0K"
            (prefix="$" mult=.001)
        other = "error" ;
```

would work. However, the decimal point in the picture indicates that SAS is to divide by 10 so that the correct statement is

```
    picture money (round)
        50 - 999949 = "*009.0K"
            (prefix="$" mult=.01)
        other = "error" ;
```

Note that I have extended the range of valid values because we can now display a wider range of values because of the extra decimal place. Of course we do not have to make all other values and error. We could have left them to display in an unformatted form or we could have added more range pictures to include more values.

Why are picture formats so funny to work with? I suspect it is the COBOL heritage showing in SAS.

Nested Formats

Formats and informats can be nested. That is you can use a format in specifying a range for another format. For example, suppose we want to display months for this year using the MONYY5. format, but all other dates should display just the year because we want more detail for the present and less as time recedes. Then the code might be

```
proc format ;
    value thisyr
        "1jan2007"d - "31dec2007"d
            = [monyy.]
        other = [year.]
    ;
run ;
```

Note that the specified format in the label does not and cannot appear in quotes. On some computer systems you have to use the combination "(|" for "[" and "|)" for "]".

The above example used system formats, but the same principle can be handy with user formats. Suppose you have project format PROJECT, but the codes for 1 and 2 need to be changed. Then instead of rewriting the whole format you could use

```
proc format ;
    value myproj
        1 = "First change"
        2 = "Second change"
        other = [project16.]
    ;
run ;
```

Although the first example format was called THISYR, the name is really only good for 2007. How could one write the code so that the format would always be appropriate to this year? For this simple example one could use some macro code, but let's solve it with simpler DATA step techniques already discussed. Remember we can use a SAS dataset to specify a format. In this case HLO is used to also specify format instructions.

```
data fmtdata ;
    fmtname = "thisyr" ;
    start = intnx ( "year",
                    today(),
                    0 ) ;
    end = intnx ( "year" ,
                  today(),
                  0,
                  "end" ) ;
   label = "date9." ;
    hlo = "F " ;
    output ;
    label = "year4." ;
    hlo = "FO" ;
    output ;
run ;

proc format cntlin = fmtdata ;
run ;
```

Note that there are no brackets around the name of the nested format. The variable HLO tells SAS that the label is a format name instead of the brackets used with format code.

Multi-label Formats

Version 8 introduced a new option for formats, the MULTILABEL option. In general, it doesn't make much sense to have one value display in more than one way. However, remember that PROC SUMMARY and PROC TABULATE respect formats in

grouping statistics. For these two procedures it does make sense to allow one value to go to several labels. In this case we want several groups to include the corresponding statistics. The same reasoning might apply to PROC FREQ but the multilabel option has not been implemented for it.

For example, suppose we have a variable RATE with values 1 to 10. Perhaps 1 - 5 are considered low and 6 - 10 high. Now we would like to add an overlapping middle group 4 - 7.

```
proc format ;
    value rate (multilabel)
        1-5 = "Low"
        6-10 = "High"
        4 - 7 = "Middle"
    ;
run ;

proc tabulate data = look ;
    class rate / mlf ;
    format rate rate. ;
    table rate * n ;
run ;
```

Note that the CLASS statement uses the option MLF to stand for multilabel. In the "Mother May I" fashion that SAS has been developing since the change to a C based language, you cannot spell it out the CLASS statement, while you must spell it out in the FORMAT statement.

Conclusion

We have not discussed all the options and things one can do with formats and informats. However, I hope that we have covered enough for you to see that it is an intrinsic part of SAS that you must know if you are going to be a SAS programmer.

Contact Information

The author may be contacted via mail using the address

Ian Whitlock
Independent SAS Consultant
29 Lonsdale Lane
Kennett Square, PA

or perhaps better via e-mail at

Ian.Whitlock@comcast.net

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration