Paper 114-2007

Using SAS® to Process Repeated Measures Data – A Hands-On Approach

Terry Fain, RAND Corporation, Santa Monica, CA
Cyndie Gareleck, RAND Corporation, Santa Monica, CA

ABSTRACT

Data that contain multiple observations per case are called "repeated measures data." Common examples are medical records, credit card charges, and airline travel history. During this hands-on workshop, attendees will use the SAS® DATA step to process repeated measures data, in order to generate summary data for reports or for submission to analytic techniques (e.g., regression). We will address both horizontal repeated measures (using arrays) and vertical repeated measures (using RETAIN, FIRST., LAST., OUTPUT, and other SAS statements and options), as well as how to convert horizontally structured data to vertically structured data and vice versa.

INTRODUCTION

Repeated measures data generally take one of two forms: (1) one record per case, with the repeated measures in an array and different variable names for each individual observation, or (2) multiple records per case, with each record containing data for a single episode (such as a medical visit or a credit card transaction). This paper will discuss the advantages and disadvantages of data stored in these two formats, with an emphasis on how to use the SAS DATA step to process these two forms of repeated measures data. We will also see how to convert from one record per case to multiple records per case, and vice versa. This is a basic workshop that introduces the appropriate SAS tools for processing repeated measures data.

WHY PROCESS REPEATED MEASURES DATA?

By definition, repeated measures data involve multiple events. But most of the time, we are less interested in the individual events than in a summary of some or all of the events. For example, let's say you had a dataset of all your credit card transactions for the past year and you wanted to know how many times you used your card to buy groceries, and how much you spent altogether on groceries for the entire year. Or, you might be interested in knowing how many times you bought groceries during the month of April, and how much you spent in April on groceries. If we have a record of each individual transaction, we can *count* the number of occurrences and *sum* the amounts spent. We can also *bracket* or *filter* our counting and summing to include only the part of the data that we are interested in, e.g., only transactions in April. When we are finished, we want to have one output record that gives us the total number of times we charged groceries during the year, the total amount spent during the year on groceries, the total number of times we charged groceries during April, and the total amount spent during April. These four functions--counting, summing, bracketing, and filtering--are the most common methods that we use to process repeated measures data in order to generate a single output record.

Now, let's take it a step further. Suppose we had a SAS data set on credit card charges, not just for one person, but for everyone who uses a particular kind of credit card (e.g., everyone who has a Visa card issued by Bank of America). We could ask the same questions for each individual, generating one output record per individual. We could then use PROC MEANS to determine the mean number of transactions per credit card, and the mean amount spent on groceries. Or we could use PROC FREQ to see how many had more than 50 transactions during the past year, or PROC PRINT to list the amount spent on groceries by each credit card holder. In fact, repeated measures data, once reduced to one record per case, may serve as input for any SAS PROC.

In this workshop, we will use a simple dataset to illustrate various ways to process multiple measures data. The same techniques may be used, regardless of the size or complexity of your data.

PROCESSING ARRAY DATA

Continuing with our example of credit card transactions, consider the sample data shown in Figure 1 (this figure shows only part of the data set, which actually contains up to 11 transactions for each of the 4 customer IDs). To simplify our example, we have restricted our sample data set to include only transactions for buying groceries and gasoline for four credit card holders, and only two months' worth of data (March and April).[1].

id	purpose1	transdate1	amount1	purpose2	transdate2	amount2	purpose3
1234	gasoline	03/05/2004	$20.30	groceries	03/07/2004	$53.45	groceries
1235	groceries	03/03/2004	$63.45	gasoline	03/05/2004	$24.44	groceries
1236	groceries	03/02/2004	$32.30	groceries	03/04/2004	$65.45	gasoline
1237	gasoline	03/01/2004	$33.84	groceries	03/03/2004	$66.99	groceries

id	transdate3	amount3	purpose4	transdate4	amount4	purpose5	transdate5
1234	03/15/2004	$54.44	groceries	03/30/2004	$78.34	gasoline	03/30/2004
1235	03/13/2004	$88.34	gasoline	03/28/2004	$35.30	groceries	03/28/2004
1236	03/12/2004	$25.56	groceries	03/27/2004	$90.34	gasoline	03/27/2004
1237	03/11/2004	$67.98	groceries	03/26/2004	$91.88	groceries	03/26/2004

Figure 1--Part of a Repeated Measures Data Set in Array Format

Since the data already contain only a single record per case, we can use the SAS ARRAY statement to answer our questions. First we need to use a PROC CONTENTS to determine the maximum number of elements in each array we'll be looking at. The output of the PROC CONTENTS for our sample array data, shown in Figure 2, tells us that each person in our data set may have up to 11 transactions.

The CONTENTS Procedure

-----Variables Ordered by Position---

#	Variable	Type	Len	Format
1	id	Num	8	
2	purpose1	Char	11	
3	purpose2	Char	11	
4	purpose3	Char	11	
5	purpose4	Char	11	
6	purpose5	Char	11	
7	purpose6	Char	11	
8	purpose7	Char	11	
9	purpose8	Char	11	
10	purpose9	Char	11	
11	purpose10	Char	11	
12	purpose11	Char	11	
13	transdate1	Num	8	MMDDYY10.
14	transdate2	Num	8	MMDDYY10.
15	transdate3	Num	8	MMDDYY10.
16	transdate4	Num	8	MMDDYY10.
17	transdate5	Num	8	MMDDYY10.
18	transdate6	Num	8	MMDDYY10.
19	transdate7	Num	8	MMDDYY10.
20	transdate8	Num	8	MMDDYY10.
21	transdate9	Num	8	MMDDYY10.
22	transdate10	Num	8	MMDDYY10.
23	transdate11	Num	8	MMDDYY10.
24	amount1	Num	8	DOLLAR10.2
25	amount2	Num	8	DOLLAR10.2
26	amount3	Num	8	DOLLAR10.2
27	amount4	Num	8	DOLLAR10.2
28	amount5	Num	8	DOLLAR10.2
29	amount6	Num	8	DOLLAR10.2
30	amount7	Num	8	DOLLAR10.2
31	amount8	Num	8	DOLLAR10.2
32	amount9	Num	8	DOLLAR10.2
33	amount10	Num	8	DOLLAR10.2
34	amount11	Num	8	DOLLAR10.2

Figure 2--PROC CONTENTS Output for Sample Array Data Set

This allows us to produce a DATA step that will tell us the total number of grocery transactions and the amount of money spent on groceries, both for the entire period and for the month of April only. We will build up the DATA step code needed to answer these questions in separate pieces, so that we can understand each step in the process.

First, we use a DATA statement with a KEEP= data set option to keep only the summary variables that we are going to generate, plus an ID for each individual:

```
data answer (keep=id alltrans totalamt aprtrans apramt);
set vector;
```

Next, we set up arrays to simplify processing the variables that we will be using for filtering, counting, and summing. Note that since we already know that an individual has a maximum of 11 transactions, we could specify the exact

number of elements in the array. Alternatively, we can let SAS determine the number by counting the number of variables that we specify in the definition of the array, by using an asterisk for the array size. These statements illustrate the latter:

```
array purpose {*} purpose1-purpose11;
array transdate {*} transdate1-transdate11;
array amount {*} amount1-amount11;
```

Now we need to initialize the summary variables that we will use to hold our counts and totals. To add clarity, we also include a LABEL statement to associate a label with each created variable:

```
alltrans = 0;
totalamt = 0;
aprtrans = 0;
apramt = 0;

label
    alltrans = 'total number of transactions for groceries'
    totalamt = 'total amount spent on groceries'
    aprtrans = 'number of transactions for groceries in April'
    apramt   = 'amount spent on groceries in April';
```

We are now ready to write the code to use the variables in the PURPOSE array as filters, those in the AMOUNT array as counters and summands, and the dates in the TRANSDATE array to bracket the variables we are creating that pertain only to April:

```
do i = 1 to 11;
    if purpose {i} = 'groceries' then do;
           alltrans + 1;
           totalamt + amount {i};
           if month (transdate {i}) = 4 then do;
                  aprtrans + 1;
                  apramt + amount {i};
                  end;
           end;
    end;
```

This code filters the PURPOSE variables so that we are only counting purchases for groceries, ignoring the gasoline purchases that are also in the data set. Whenever we find a transaction for groceries, we increment ALLTRANS, the variable counting all transactions, and add the amount of the transaction to TOTALAMT, which is going to give us the total amount spent on groceries. Using the TRANSDATE variables, we set up a DO loop that is only used if the transaction occurred in April, and repeat a similar counting and summing process within the DO loop.
Finally, to make sure our created data looks the way we want it to, we add a FORMAT statement:

```
format totalamt apramt dollar10.2;
```

This completes the entire DATA step that we need. If we then add

```
run;
```

and submit the job, we will create a new data set that contains the variables we are interested in. Figure 3 shows a PROC PRINT output for this data set.

id	alltrans	totalamt	aprtrans	apramt
1234	7	$511.81	4	$325.58
1235	6	$503.38	3	$258.25
1236	5	$377.98	2	$189.89
1237	8	$645.43	4	$379.74

Figure 3--PROC PRINT Output for Summary Data from Example Array Data Set

As we have already noted, this new data set can now be used as input to any SAS PROC. Figure 4 shows the output from a PROC MEANS as an example.

The MEANS Procedure

Variable	Label	Mean	Median
alltrans	total number of transactions for groceries	6.50	6.50
totalamt	total amount spent on groceries	509.65	507.60
aprtrans	number of transactions for groceries in April	3.25	3.50
apramt	amount spent on groceries in April	288.37	291.92

Variable	Label	Minimum	Maximum	N
alltrans	total number of transactions for groceries	5.00	8.00	4
totalamt	total amount spent on groceries	377.98	645.43	4
aprtrans	number of transactions for groceries in April	2.00	4.00	4
apramt	amount spent on groceries in April	189.89	379.74	4

Figure 4--PROC MEANS Output for Summary Data from Example Array Data Set

PROCESSING MULTIPLE RECORD DATA

Many software programs include functions that allow you to process array data in much the same way that we have illustrated above. The particular strength of SAS is in processing repeated measures data where you have a variable number of records per individual. And in contrast to the example above, we do not have to determine the maximum number of observations per individual in order to process multiple records data.

Suppose we have the same data as above, but rather than one record per case with observations listed array-style, we have a separate record for each transaction. Figure 5 shows a portion of the multiple records data set.

	purpose	id	transdate	amount
1	gasoline	1234	03/05/2004	$20.30
2	groceries	1234	03/07/2004	$53.45
3	groceries	1234	03/15/2004	$54.44
4	groceries	1234	03/30/2004	$78.34
5	gasoline	1234	03/30/2004	$25.30
6	groceries	1234	04/06/2004	$83.34
7	groceries	1234	04/13/2004	$77.44
8	gasoline	1234	04/15/2004	$23.55
9	groceries	1234	04/22/2004	$88.45
10	groceries	1234	04/29/2004	$76.35
11	groceries	1235	03/03/2004	$63.45
12	gasoline	1235	03/05/2004	$24.44
13	groceries	1235	03/13/2004	$88.34
14	gasoline	1235	03/28/2004	$35.30
15	groceries	1235	03/28/2004	$93.34
16	gasoline	1235	04/11/2004	$33.55
17	groceries	1235	04/13/2004	$98.45
18	groceries	1235	04/20/2004	$86.35
19	groceries	1235	04/27/2004	$73.45

Figure 5--Part of a Multiple Records Data Set of Repeated Measures

Let us look at how we might use these data to generate the same summary variables that we produced with array processing. As before, we will build up the DATA step code gradually, so that each step in the process is clear. First we need to make sure that the data set is sorted by ID:

```
proc sort data=sugi.example
                out=example;
    by id;
run;
```

Now, just as before, we use a DATA statement with a KEEP= data set option to save only the summary variables and an ID for each individual. But now we introduce a new concept into the DATA step, the BY statement, which in this context works with the SET statement. We have paved the way for this usage by the PROC SORT code above, since a BY statement used in a DATA step requires that the data be sorted by the variable(s) specified in the BY statement. Using a BY statement allows us to determine the first and last observation in the "by-group," i.e., all the records with the same value in the variable specified in the BY statement (and often called the "by-variable"). When you use a BY statement in the DATA step, SAS creates two temporary variables that may be used only in that DATA step. These are FIRST.*byvariable* and LAST.*byvariable.* FIRST.*byvariable* has a value of 1 for the first observation that takes on a new value for the by-variable, 0 otherwise. Similarly, LAST.*byvariable* has a value of 1 only for the last observation in the by-group, and 0 for all other observations. We will see below that this is a powerful and extremely useful tool in process these types of data.

```
data answer (keep=id alltrans totalamt aprtrans apramt);
set example;
    by id;
```

We also need to use a RETAIN statement for the summary variables we will be creating. RETAIN tells SAS to save the value of the variable even when it goes to the next observation. This is very important in processing repeated measures data in the multiple records format. Without the RETAIN statement, each new observation will start with missing data, rather than your accumulating counts and totals, and the resulting summary variables will not be accurate.

```
retain alltrans totalamt aprtrans apramt;
```

Next, we must initialize the summary variables, as we did when processing array data. In this instance, though, since we have multiple records for the same individual, we need to initialize the summary variables only when we encounter a new ID. This is where it becomes critical to identify the first instance of a particular value in the by-variable. We do this with the use of the FIRST. ("first-dot") feature. Here, since our by-variable is ID, we will use FIRST.ID to identify the first time we encounter a new ID:

```
if first.id then do;
    alltrans = 0;
    totalamt = 0;
    aprtrans = 0;
    apramt = 0;
    end;
```

We now count the number of transactions and accumulate the total amounts spent, in much the same way we did above when the data were in an array. However, with multiple records per case, we do not need to use array processing, since we want to do the same thing for every record we encounter. The SAS code looks like this:

```
if purpose = 'groceries' then do;
    alltrans + 1;
    totalamt + amount;
    if month (transdate) = 4 then do;
           aprtrans + 1;
           apramt + amount;
           end;
    end;
```

Finally, when we have passed through all the records for the by-group, i.e., when we come to the last record for a given value of the variable ID, we are ready to write out the resulting summary variables. To do this, we use the LAST. ("last-dot") feature and an explicit OUTPUT statement. In DATA steps that do not include an explicit OUTPUT statement, SAS assumes an implicit OUTPUT statement at the end of the DATA step. The explicit OUTPUT statement tells SAS to create an output record only when we want to do so. Combining the LAST. feature and the

explicit OUTPUT statement tells SAS to create an output record each time we come to the last record in the by-group:

```
if last.id then output;
```

Although LABEL and FORMAT statements can go anywhere in a DATA step, it seems to make sense to place them after the OUTPUT statement when processing multiple records per case.

```
label
    alltrans = 'total number of transactions for groceries'
    totalamt = 'total amount spent on groceries'
    aprtrans = 'number of transactions for groceries in April'
    apramt   = 'amount spent on groceries in April';
format totalamt apramt dollar10.2;
```

A RUN statement causes the DATA step to be executed:

```
run;
```

Figure 6 shows a PROC PRINT output for the data set created by the DATA step. A comparison with Figure 3 above shows that we have produced exactly the same result using multiple record processing as we did using array processing.

id	alltrans	totalamt	aprtrans	apramt
1234	7	$511.81	4	$325.58
1235	6	$503.38	3	$258.25
1236	5	$377.98	2	$189.89
1237	8	$645.43	4	$379.74

Figure 6--PROC PRINT Output for Summary Data from Example Multiple Records Data Set

USING FUNCTIONS WITH REPEATED MEASURES DATA

What if you wanted to know the mean time between visits for a unique id or the mean time between visits for all observations in the file? One way to do this is with the DIF and LAG functions. These two queuing functions are used for accessing prior values of a variable. In simple terms, the LAG function stores values in a queue and returns values from prior calls. The queue is created when the LAG function is called and is initialized as a missing value. The DIF function returns the difference between the value of the current observation and its lag. In order to get accurate results when calling both the DIF and LAG functions, it's critical that the data be sorted in the correct order e.g., chronologically.

Let's use these functions to calculate, for each unique ID, both the average time between visits and the average difference in amount between visits. First we sort the data by ID and date, and for the sake of simplicity we will keep only data on grocery visits:

```
proc sort data=sugi.example (where=(purpose='groceries'))
          out=example;
    by id transdate;
run;
```

We now create a new data set and generate three new variables: one for the lag of amount spent on groceries, one for the difference between the amount spent at the current visit and the preceding observation, and one to store the number of days between the current visit and the preceding visit. As we did earlier, we will use a BY statement in the data set to create the temporary variable FIRST.*byvariable*, which we will use later on in the data step.

```
data lag_difg;
set example;
    by id transdate;
format amount_lag amount_dif dollar10.2;
amount_lag = lag(amount);
amount_dif = dif(amount);
days_dif = dif(transdate);
```

If we add a RUN statement and stop here, data for the first eight records in the file are shown in Figure 7.

id	purpose	transdate	amount	amount_ lag	amount_ dif	days_dif
1234	groceries	03/07/2004	$53.45	.	.	.
1234	groceries	03/15/2004	$54.44	$53.45	$0.99	8
1234	groceries	03/30/2004	$78.34	$54.44	$23.90	15
1234	groceries	04/06/2004	$83.34	$78.34	$5.00	7
1234	groceries	04/13/2004	$77.44	$83.34	$-5.90	7
1234	groceries	04/22/2004	$88.45	$77.44	$11.01	9
1234	groceries	04/29/2004	$76.35	$88.45	$-12.10	7
1235	groceries	03/03/2004	$63.45	$76.35	$-12.90	-57

Figure 7--Results of LAG and DIF Functions

But note that the calculated values for the first occurrence of ID 1235 actually reflect the data from the last occurrence of ID 1234, producing erroneous results. To correct this problem, we use the temporary variable FIRST. to set the value of each new occurrence of an ID to missing. Note that this must be done *after* using the LAG and DIF functions in the data step.

```
if  first.id then do;
    amount_lag=.;
    amount_dif=.;
    days_dif=.;
    end;
run;
```

The corrected results are shown in Figure 8.

id	purpose	transdate	amount	amount_ lag	amount_ dif	days_dif
1234	groceries	03/07/2004	$53.45	.	.	.
1234	groceries	03/15/2004	$54.44	$53.45	$0.99	8
1234	groceries	03/30/2004	$78.34	$54.44	$23.90	15
1234	groceries	04/06/2004	$83.34	$78.34	$5.00	7
1234	groceries	04/13/2004	$77.44	$83.34	$-5.90	7
1234	groceries	04/22/2004	$88.45	$77.44	$11.01	9
1234	groceries	04/29/2004	$76.35	$88.45	$-12.10	7
1235	groceries	03/03/2004	$63.45	.	.	.
1235	groceries	03/13/2004	$88.34	$63.45	$24.89	10

Figure 8--Re-initializing the First Record of a By-group After Using LAG and DIF Functions

Now let's Use PROC MEANS to calculate the average time between visits for each and the average difference in amount spent.

```
proc means data=lag_difg mean maxdec=2;
class id;
var amount_dif days_dif;
run;
```

Figure 9 shows the resulting output.

customer id	N Obs	Variable	Mean
1234	7	amount_dif	3.82
		days_dif	8.83
1235	6	amount_dif	2.00
		days_dif	11.00
1236	5	amount_dif	17.04
		days_dif	12.00
1237	8	amount_dif	3.27
		days_dif	7.57

Figure 9--Mean Difference Computed by LAG and DIF Functions

CONVERTING ARRAY DATA TO MULTIPLE RECORDS PER CASE

We have seen how we can use different approaches to achieve the same results with array data and multiple records data. From time to time, you may feel that you want to convert your data set from array format to multiple records. For example, if some cases have a large number of entries while others have only a few, the multiple record format may save a lot of storage space. We will now look at how to convert a data set with an array into a data set which has multiple records per case.

We will use the same array data set as in Figure 1 above, with an array size of 11, since a given ID may have as many as 11 transactions. As you will see, this process involves ARRAY and OUTPUT statements in a slightly different way than we have used them up to now. We begin with a DATA statement that includes a KEEP= option, since we only want to keep the four output variables we will be creating.

```
data multiple (keep=id purpose transdate amount);
set sugi.vector;
```

We now use ARRAY statements, as we did before, to simplify processing:

```
array pur {*} purpose1-purpose11;
array trans {*} transdate1-transdate11;
array amt {*} amount1-amount11;
```

Next we want to combine array processing with an explicit OUTPUT statement, so that we generate multiple records. We accomplish this by including the OUTPUT record in a DO loop. We also use an IF statement to make sure we only write a new record if the array elements are not missing. The SAS code looks like this:

```
do i = 1 to 11;
    if pur {i} ^= '' then do;
           purpose = pur {i};
           transdate = trans {i};
           amount = amt {i};
           output;
           end;
    end;
```

As before, we may add labels and specify formats:

```
label
    id = 'customer id'
    purpose = 'purpose of transaction'
    transdate = 'transaction date'
    amount = 'transaction amount';
format transdate1-transdate11 mmddyy10.;
format amount1-amount11 dollar10.2;
```

Finally, a RUN statement tells SAS to execute the DATA step.

```
run;
```

The result is a new data set which has a variable number of records per ID, and only the four variables that we created. Figure10 shows the first two records for this new data set. A comparison with Figure 5 above shows that we have successfully produced the multiple record data set from the array data set.

id	purpose	transdate	amount
1234	gasoline	16135	20.30
1234	groceries	16137	53.45
1234	groceries	16145	54.44
1234	groceries	16160	78.34
1234	gasoline	16160	25.30
1234	groceries	16167	83.34
1234	groceries	16174	77.44
1234	gasoline	16176	23.55
1234	groceries	16183	88.45
1234	groceries	16190	76.35
1235	groceries	16133	63.45
1235	gasoline	16135	24.44
1235	groceries	16143	88.34
1235	gasoline	16158	35.30
1235	groceries	16158	93.34
1235	gasoline	16172	33.55
1235	groceries	16174	98.45
1235	groceries	16181	86.35
1235	groceries	16188	73.45

Figure 10--Part of a Multiple Record Data Set Converted from Array Data

CONVERTING MULTIPLE RECORDS DATA TO ARRAYS

Now let us suppose we want to go the other way, converting from a multiple records per case format into a data set that includes only a single record per case. To do this, we must first determine the size of the arrays we will need in order to store all the data. This requires a simple DATA step, combined with a PROC FREQ, to give us the maximum number of records per case. To simplify matters, we will only keep a single variable, which gives the number of records for each ID. This will involve another use of the by-group processing and RETAIN and OUTPUT statements that we have used above. First, we make sure that the data are sorted by ID, which allows us to use by-group processing:

```
proc sort data=example;
    by id;
run ;
```

Next we define a new DATA step, complete with a BY statement to set up our by-group:

```
data count (keep=count);
set multiple;
    by id;
```

We will now define a new variable in order to count the number of records in each by-group. We want to retain its value, so we need a RETAIN statement:

```
retain count;
```

We initialize this variable when we encounter a new value in our by-group, then increment it each time we encounter a new record within the by-group, and finally write it out when we come to the last record in the by-group:

```
if first.id then count = 0;
count + 1;
if last.id then output;
run;
```

Figure 11 shows the output a PROC FREQ, using the data set we have just created.

The FREQ Procedure

count	Frequency	Percent	Cumulative Frequency	Cumulative Percent
8	1	25.00	1	25.00
9	1	25.00	2	50.00
10	1	25.00	3	75.00
11	1	25.00	4	100.00

Figure 11--Number of Records per Case in Example Data Set

This tells us that the maximum number of records within the ID by-group is 11, so we need to size our arrays at 11. We are now ready to start creating the new data set. As before, we will use a KEEP= data set option so that our new data set includes only the variables we want.

```
data vector (keep=id purpose1-purpose11 transdate1-transdate11 amount1-amount11);
set multiple;
    by id;
```

We need a LENGTH statement to make sure the variables we will be creating have enough storage space. The LENGTH statement has to come before any other statements.

```
length purpose1-purpose11 $ 11;
```

Because we need to process multiple records before we write an output record, we need to use a RETAIN statement to specify all the new variables, along with the variable (i) that we will use as the index of the arrays.

```
retain i purpose1-purpose11 transdate1-transdate11 amount1-amount11;
```

Next we define the arrays, as we have done above.

```
array pur {*} purpose1-purpose11;
array trans {*} transdate1-transdate11;
array amt {*} amount1-amount11;
```

When we first encounter a new by-group, we must re-initialize all the retained variables.

```
if first.id then do j = 1 to 11;
    pur {j} = '';
    trans {j} = .;
    amt {j} = .;
    end;
```

In addition, we need to re-initialize the indexing variable:

```
if first.id then i = 0;
```

Each time we encounter a new record within the by-group, we increment the indexing variable:

```
i + 1;
```

We are now ready to assign values within the arrays we have set up. The indexing variable indicates where in the array each value goes.

```
if purpose ^= '' then do;
    pur {i} = purpose;
    trans {i} = transdate;
    amt {i} = amount;
    end;
```

When we come to the end of the by-group, we are ready to output the data.

```
if last.id then output;
```

Finally, we add any LABEL or FORMAT statements needed, and execute the resulting DATA step.

```
format transdate1-transdate11 mmddyy10.;
format amount1-amount11 dollar10.2;
run;
```

Figure 12 shows part of a PROC PRINT output for this newly created data set. As we expected, this new data set has a single record per ID, and separate variables to store data for up to 11 transactions.

id	purpose1	transdate1	amount1	purpose2	transdate2	amount2	purpose3
1234	gasoline	03/05/2004	$20.30	groceries	03/07/2004	$53.45	groceries
1235	groceries	03/03/2004	$93.45	gasoline	03/05/2004	$24.44	groceries
1236	groceries	03/02/2004	$37.30	groceries	03/04/2004	$65.45	gasoline
1237	gasoline	03/01/2004	$33.84	groceries	03/03/2004	$66.99	groceries

id	transdate3	amount3	purpose4	transdate4	amount4	purpose5	transdate5
1234	03/15/2004	$54.44	groceries	03/30/2004	$78.34	gasoline	03/30/2004
1235	03/13/2004	$88.34	gasoline	03/28/2004	$35.30	groceries	03/28/2004
1236	03/12/2004	$25.56	groceries	03/27/2004	$90.34	gasoline	03/27/2004
1237	03/11/2004	$67.98	groceries	03/26/2004	$91.88	groceries	03/26/2004

id	amount5	purpose6	transdate6	amount6	purpose7	transdate7	amount7
1234	$25.30	groceries	04/06/2004	$83.34	groceries	04/13/2004	$77.44
1235	$93.34	gasoline	04/11/2004	$33.55	groceries	04/13/2004	$98.45
1236	$37.30	groceries	04/10/2004	$89.44	gasoline	04/12/2004	$35.55
1237	$38.84	groceries	04/02/2004	$96.88	groceries	04/09/2004	$90.98

Figure 12--Part of a PROC PRINT Output for the Array Data Set After Conversion from Multiple Records

CONCLUSION

In this workshop, we have introduced the concept of repeated measures processing, and illustrated these concepts with sample data and SAS DATA step code. We have seen how to process repeated measures data in both array and multiple records formats within a SAS DATA step, and how to convert multiple records to arrays and vice versa. As we have seen, SAS includes several powerful concepts and statements for processing repeated measures data. Using by-group processing, along with RETAIN, ARRAY, IF, DO, and OUTPUT statements, allows us to easily manipulate repeated measures data regardless of its format.

CONTACT INFORMATION

We value and encourage your comments and questions. You may contact the authors:

Terry Fain
RAND Corporation
1776 Main Street
Santa Monica CA 90401
310-393-0411x7804
fain@rand.org

Cyndie Gareleck
RAND Corporation
1776 Main Street
Santa Monica CA 90401
310-393-0411x7815
cyndie@rand.org

[1] Repeated measures data imported from other software programs are often represented as arrays because some software lack the ability to process multiple records per case.