INTERMEDIATE SAS

Manitoba Centre for Health Policy and Evaluation

- DRAFT -

January 30, 1995
word\sas\manuals\intermed
Murray Werbeniuk

I. ARRAY STATEMENT

Purpose

Arrays are often used in conjunction with DO loops when performing actions for a series of variables. The following example illustrates the same action being performed on two separate diagnostic field variables. The study diagnosis of 820.0 can occur in either of these fields, and the statements are identical except for the name of the diagnostic field. The intent of the following statements is to flag all occurrences of the study diagnosis by creating a new variable - "HIPFRAC" - where '1' indicates the presence of the desired diagnosis.

```
If '82000'<=DX01<='82009'  then HIPFRAC='1';
If '82000'<=DX02<='82009'  then HIPFRAC='1';
```

Sixteen diagnostic fields (DX01-DX16), however, would require 16 lines of code. Array processing can make the program more efficient by streamlining the code required to accomplish the task (depending on the situation, if-then/else statements can be faster; however, they are also more error-prone). A specified series of variables is associated with a collective name of your choice; for example, the diagnostic fields DX01 through DX16 could be associated with the name "DIAG", which will then operate similarly to variables in data step manipulations.

Syntax

Arrays are set up using an ARRAY statement. It can appear anywhere in the DATA step as long as it occurs prior to any reference to it. The variables that make up the array are called elements. Individual elements are identified by subscripts: numbers that identify an element's position in the array.

ARRAY array-name {number of variables} variable-n...variable-n;

Array-name is a name you choose to represent the group of variables.

Number-of-variables tells SAS how many variables are being grouped; it is represented by subscripts that are enclosed in braces.

<u>Variable-n...variable-n</u> lists the names of the variables (the variable list does not have to begin at 1 - e.g., DX5-DX16)

Example

ARRAY diag{16} $ dx01-dx16;

This statement tells SAS to:

a) create a group or array name DIAG for the duration of the DATA step.

b) have DIAG represent 16 variables: diagnostic fields DX01 through DX16.

Note that DX01-DX16 are character variables and thus must be preceded by a "$".

You can refer to the entire array or just one of its elements when performing logical comparisons or arithmetic calculations. All variables listed in the ARRAY statement are assigned extra names with the form array-name{position}, where position is the position of the variable in the list (1,2,3,...,16 in the example). The additional name is called an array reference, and the position is often called the subscript.

In the above ARRAY statement, DX01 is assigned the array reference DIAG{1}; DX02 the array reference DIAG{2}; etc. From that point in the data step, you can refer to the variable by either its original name or by its array reference; for example, the names DX01 and DIAG{1} are equivalent.

Caution

An array is simply a convenient way of temporarily identifying a group of variables; it exists only for the duration of the DATA step. Arrays are not variables.

January 30, 1995
Murray Werbeniuk

II. DO LOOP

Purpose

The iterative DO statement is used to execute repeatedly a set of statements occurring between the DO statement and the END statement (the DO loop). It is often used in conjunction with ARRAY statements so that the repeated actions occur within the loop for each of a specified series of variables.

Syntax

A DO loop begins with an iterative DO statement, followed by other SAS statement(s), and completed with an END statement. This loop iterates (is processed repeatedly) according to the directions in the DO statement. Its basic form is:

```
DO <index-variable=1> TO <upper bound of array>;
       [or DO <index-variable=1> TO <upper bound of array> UNTIL
        <specified condition>]
       [or DO <index-variable=1> TO <upper bound of array> WHILE
        <specified condition]
  <SAS statements>
END;
```

Index-variable is a name you choose (e.g., "I"). Its value changes with each iteration of the loop, or each time the loop is processed. By default, the value of the index-variable (I) is increased by 1 before each new iteration of the loop, consecutively representing the values 1 to n (number of variables in array).

Number-of-variables-in-array: If used in conjunction with an array, the loop will execute as many times as there are variables in the array. If there are 16 variables, the loop will execute the statements on each of the 16 variables (i.e., DO I=1 to 16). The processing stops when the value of the index variable becomes greater than number-of-variables-in-array.

Three examples follow, 1) the basic DO loop, 2) DO UNTIL, and 3) DO WHILE. The most efficient method is the last - DO WHILE - and is highly recommended over the others.

Example 1

The SAS statements in an iterative DO loop often contain references to an array. In the following example, the array name is "diag" and the number of variables represented in its subscript is 16. With each iteration of the loop, the value of the subscript is replaced with the current value of the index variable ("i"). Successive iterations of the loop process the statements on consecutive variables in the array. The intent of this example is to search all 16 diagnostic fields to flag any occurrence of a hip diagnosis of ICD-9-CM 820.

```
ARRAY diag{16} $ dx01 - dx16;                              (1)
hipdiag=0;                                                 (2)
DO i=1 to 16;                                              (3)
  IF '820  '<=diag{i}<='82099' THEN hipdiag='1';           (4)
END;                                                       (5)
```

(1) Create an array called "diag" to represent the group of diagnostic fields dx01-dx16 for the duration of the data step.

(2) Create a new variable named "hipdiag" and set it equal to zero. It will remain at 0 if none of the 16 diagnostic fields for that record have a hip diagnosis present.

(3) Perform the actions in the loop sixteen times for each record in the data set. When the value of "**i**" is 1, SAS reads the array reference as DIAG{1} and processes the following statements on DIAG{1}, that is, DX01. In each iteration of the loop, the subscript associated with DIAG is replaced with the index variable's (i) current value.

(4) When a diagnosis within the specified range is encountered, the variable "hipdiag" will be assigned a value of '1'.

(5) All iterative DO loops must end with an END statement.

Example 1: Output

```
OBS  DX01 DX02  DX03  DX04 DX05  DX06 DX07 DX08...DX16 HIPDIAG

   1  650  V270                                          0
   2  71783                                              0
   3  4549                                               0
   4  V664 82021 E8809 4538 3569 36250 7213 2859         1
   5  V301 7746                                          0
   6  8208 E888  4019  3310                              1
   7  4111 4140  4280  4011 42731 586  5990 V668         0
   8  486                                                0
   9  V72                                                0
  10  650  V270                                          0
```

Observation 4 has hipdiag set to '1' because DX02 has a value of '82021', thus falling within the ICD-9-CM range 820-820.99. Observation 6 similarly has hipdiag set to '1' because DX01 has a value of '8208 '.

Conditional Loops

The DO loop in the above example is not very efficient because, for each observation, it will process all 16 diagnostic fields regardless of whether a hip diagnosis is found. Even though the condition hipdiag=1 is found within dx02 in observation 4, for example, the DO loop will continue to process the remaining diagnostic fields.

Conditional loops - using **DO WHILE** and **DO UNTIL** statements - increase the efficiency of the DO loop processing. **DO UNTIL** in Example 2 performs at least one iteration before it evaluates the condition. **DO WHILE** in Example 3 tests the condition before the first iteration is made (so it is possible no iteration will be performed), and processes one less loop than DO UNTIL.

Using DO WHILE, the statement in the DO loop for observation 4 in the above example will be executed twice (i=2) because the condition hipdiag='1' was met with the second diagnostic field (dx02). Using DO UNTIL, however, the statement is processed three times (i=3) in order to meet the condition hipdiag^='0' after setting hipdiag to '1' with dx02.

Example 2: DO UNTIL statement

```
ARRAY diag{16} $ dx01 - dx16;
hipdiag=0;
DO i=1 to 16 UNTIL (diag{i}=' ' OR hipdiag='1');   (1)
 IF '820  '<=diag{i}<='82099' THEN hipdiag='1';     (2)
END;
```

(1) Perform the actions within the DO LOOP up to a maximum of 16 times. The loop will continue to process the statements for each consecutive diagnostic field until it encounters one that is missing (DIAG{i}=' ') **OR** until a field containing hip diagnoses is encountered (hipdiag^=0). If either of these conditions are met, hipdiag will be assigned a value of '1'.

(2) When a diagnosis within the specified range is encountered for the record, hipdiag will be assigned a value of '1' and the remaining diagnostic fields will not be processed. For example, if hipdiag is set to "1" because dx01 falls within the specified range, then the DO loop would only be processed once.

Example 3: DO WHILE statement

```
ARRAY diag{16} $ dx01 - dx16;
hipdiag=0;
DO i=1 to 16 WHILE (diag{i}^=' ' AND hipdiag='0');    (1)
 IF '820  '<=diag{i}<='82099' THEN hipdiag='1';        (2)
END;
```

(1) Perform the actions in the DO LOOP up to a maximum of 16 times. The loop will continue to process the statements for each consecutive diagnostic field only as long as it is not missing (DIAG{i}^=' ') **AND** there are no hip diagnoses (hipdiag=0).

(2) Same as Example 2.

Caution

1. When coding many DO loops in a DATA step or when enclosing DO loops within other DO loops, take care that each DO has a corresponding END. It is very easy to lose an END in a long program!

2. If you want to flag the position of each ICD-9-CM code, use DO UNTIL to get accurate positions (DO WHILE begins the position count at "2" rather than "1").

January 23, 1995
Murray Werbeniuk

III. BY-GROUP PROCESSING (FIRST./LAST.)

Purpose

By-group processing refers to the use of a BY statement in a DATA step, which permits identification of the first- and last-occurring record for each of the specified BY variables. Two dichotomous (1/0) variables are automatically created for each variable specified in the BY statement when using SET, MERGE, or UPDATE: FIRST.varname and LAST.varname, where varname is the name of the BY variable(s). By creating these variables, a number of various calculations are possible, such as obtaining a count of records for each unique identifier.

Syntax

BY varname1 varname2...;

For the first record in a BY group, the value of the FIRST.varname1 is set to 1, with all other records in the BY group set to 0. For the last record in a BY group, the value of the LAST.varname1 is set to 1, with all other records set to 0. If the data are sorted by more than one BY variable, the FIRST.varname for each variable is set to 1 at the first occurrence of a new value for the variable. FIRST. and LAST. variables are not kept in the newly created data set since they are usually just used for subsequent processing. Their values can be assigned, however, to other variables. Following are examples of single and multiple by-group processing.

Example 1: Single BY Variable

```
PROC SORT DATA=hosp;
BY phin;
RUN;

DATA dup;
SET hosp;
BY phin;                    (1)
  firstfl=FIRST.phin;       (2)
  lastfl=LAST.phin; (3)
RUN;
```

(1) Set the data by PHIN (already previously sorted by this variable) in order to create FIRST.PHIN and LAST.PHIN.

(2) Create a new variable called FIRSTFL and assign it a value of 1 for every FIRST.PHIN=1 encountered.

(3) Create a new variable called LASTFL and assign it a value of 1 for every LAST.PHIN=1 encountered.

Example 1: Output

```
OBS        PHIN          FIRSTFL     LASTFL

  1       562737            1           1
  2       563850            1           1
  3       563961            1           1
  4       565858            1           1
  5       566739            1           1
  6       568729            1           0
  7       568729            0           0
  8       568729            0           1
  9       569961            1           1
 10       660861            1           1
```

In the above example, the person with PHIN 568729 has 3 records (observations 6-8). For the first record (#6), FIRSTFL is set to 1, indicating that it is the first record for that person, and all other records for that PHIN show FIRSTFL values set to 0. For the third and last record, LASTFL is set to 1, indicating that it is the last record for that person, and all other records show LASTFL values set to 0.

<u>Example 2: Multiple BY Variables</u>

```
PROC SORT DATA=hosp;
BY phin dateadm;
RUN;

DATA dup;
SET hosp;
BY phin dateadm;           (1)
  firstph=FIRST.phin;      (2)
  lastph=LAST.phin;        (3)
  firstdt=FIRST.dateadm;   (4)
  lastdt=LAST.dateadm;     (5)
RUN;
```

(1) Set the data by PHIN and DATEADM (already previously sorted by these variables) in order to create FIRST.PHIN and LAST.PHIN and FIRST.DATEADM and LAST.DATEADM.

(2) Create a new variable called FIRSTPH and assign it a value of 1 for every FIRST.PHIN encountered.

(3) Create a new variable called LASTPH and assign it a value of 1 for every LAST.PHIN encountered.

(4) Create a new variable called FIRSTDT and assign it a value of 1 for every FIRST.DATEADM encountered.

(5) Create a new variable called LASTDT and assign it a value of 1 for every LAST.DATEADM encountered.

Example 2 Output

```
OBS      PHIN          DATEADM   FIRSTPH LASTPH FIRSTDT LASTDT

 1       599748        901003       1       1      1      1
 2       690876        910109       1       1      1      1
 3       695729        900607       1       0      1      1
 4       695729        900829       0       0      1      1
 5       695729        901004       0       1      1      1
 6       794727        910304       1       1      1      1
 7       794872        910326       1       1      1      1
 8       795729        910112       1       0      1      0
 9       795729        910112       0       1      0      1
10       795745        901211       1       1      1      1
```

When sorted and set by PHIN and DATEADM the FIRST.PHIN and LAST.PHIN variables will have different values if there are multiple records per PHIN. If there is more than one record with the same date of admission, the FIRST.DATEADM and LAST.DATEADM variables will also have different values. If DATEADM is not the same, however, then the first record for that date is the last record for that date and both FIRST.DATEADM and LAST.DATEADM will be flagged as 1.

In the above example, the person with PHIN 795729 has 2 records. FIRSTPH is set to 1 for the first record (#8) and 0 for the second, or last, record, with LASTPH set to 1 for the last record (#9) and 0 for the first record. DATEADM is identical as well on both records, so FIRSTDT is set to 1 on the first record and 0 for the other, with LASTDT set to 1 for the last record and 0 for the first record.

The person with PHIN 695729 has 3 records. For the first record (#3), FIRSTPH is set to 1, representing the first record for that person, and all other records have FIRSTPH set to 0. For the last record (#8), LASTPH is set to 1, representing the last record for that person, and all other records show this variable set to 0. Unlike PHIN 795729, however, DATEADM is unique for each record, and FIRSTDT and LASTDT are each set to 1 for each record.

Caution

When conducting BY-group processing, DO NOT do any data exclusions; data manipulation is ok. Data exclusions can be done in a subsequent data step.

January 23, 1995
Murray Werbeniuk

IV. RETAIN STATEMENT

Purpose

The RETAIN statement is used to keep a specified value (assigned by an INPUT or assignment statement) from the current iteration of the DATA step to the next. Otherwise, SAS automatically sets such values to missing before each iteration of the DATA step. The RETAIN statement allows values to be kept across observations; for example, computing a running total of values, counting the number of occurrences of a variable's value, setting indicators within a BY-group, and so on.

Syntax

The RETAIN statement can be used to specify initial values for variable(s) or elements of an array. All elements or variables will be initialized to the specified value.

RETAIN <varlist> [initial-value(s)];

Varlist: specifies the names of the variables, lists or arrays whose values you wish to retain.

Initial-value(s): the initial value(s) can be numeric or character (e.g.,'y') and is assigned to all listed variables. If the initial value is not specified, it is set to missing.

The following shows four variables specified in each retain statement.

RETAIN var1-var4 1; sets initial values of var1, var2, var3, var4 to 1.
RETAIN var1-var4 (1); only var1 is set to 1; var2-4 are set to missing.
RETAIN var1-var4 (1 2 3 4); OR
RETAIN var1-var4 (1,2,3,4); var1 is set to 1, var2 to 2, var3 to 3, var4 to 4.

For example, the statement ***RETAIN pop 1;*** within a DATA step will assign a value of 1 to each observation for the variable POP.

Example

The RETAIN statement is often used to carry over values of a specified variable from one observation to the next. In the following example, a count is accumulated of the number of admissions for each person (COUNT) and a flag is created to denote whether the person has been panelled for a personal care home (PANELLED).

```
PROC SORT DATA=check;
 BY phin dateadm datesep;
RUN;
                    /*Data must be sorted for subsequent by-group processing*/
DATA check2;                       (1)
 SET check;                        (2)
 BY phin dateadm datesep;                  (3)
 RETAIN count panelled 0;                  (4)
 IF FIRST.phin THEN DO;                    (5)
   count=0;
   panelled=0;
 END;
 count = count + 1;                (6)
 IF SUBSTR(primpsvc,3,2) = '99'
   THEN panelled = panelled + 1;           (7)
RUN;
```

(1) Begin a DATA step, creating a SAS data set called "check2".

(2) Read data from the existing SAS data set "check", which has been sorted by PHIN DATEADM DATESEP.

(3) Set the data set "CHECK2" by PHIN DATEADM DATESEP in order to create the variable FIRST.PHIN. (Each PHIN has also been ordered by date - from earliest-occurring dates of admission and separation to most recent admission and separation.)

(4) Keep a value of 0 for the new variables, COUNT and PANELLED, for each observation.

(5) Assign a value of 0 to both COUNT and PANELLED to the first record encountered for each person.

(6) For each person, cumulate the value of COUNT, increasing it by 1 for each subsequent admission.

(7) For each person, cumulate the value of PANELLED, flagging the first-occurring record with "1", and increasing by 1 each subsequent record that satisfies the definition of PANELLED.

Example Output

```
OBS  PHIN   DATEADM   DATESEP   PRIMPSVC   COUNT  PANELLED

1    3928   920413    920413      1000       1         0
2    3928   921207    930311      1099       2         1

3    4988   920407    920715      1099       1         1

4    0005   911223    920625      1099       1         1
5    0005   930107    930107      3000       2         1
6    0005   930310    930325      7394       3         1

7    1009   920221    920516      7397       1         0
8    1009   920610    920615      1000       2         0
9    1009   920722    921102      7399       3         1
10   1009   921203    930128      7399       4         2
```

The last PHIN, 1009, shows four records (four hospital stays). For the first record, COUNT is flagged as "1" and, for each subsequent record, it increases by 1. The last record is the fourth record in this case, so COUNT=4 for that record.

The last PHIN also shows that PRIMPSVC ends with "99" for the third record (observation #9), and it is the first time this code is encountered for that person so PANELLED is set to "1". The code is encountered again with the next record, so PANELLED increases by 1 to "2". PHIN 0005, on the other hand, has its first record set to "1" for PANELLED, and there are no other subsequent records with this code, so PANELLED remains at "1".

Keeping only the last record per person would provide a summary record of how many admissions each person had and how many times each was panelled for a personal care home.

V. DATES

Dates can be represented in a number of different ways within a data set; they can be character or numeric; for example:

Character:	7/26/89
	26JUL89
	26JUL1989
	July 26, 1989
Character or numeric:	072689
	892607

Note that the only way to tell whether values such as 892607 in a SAS data set have been input as numeric or character is to do a PROC CONTENTS on the data set. 892607 can refer to July 26, 1989 or it can be handled as the number 892,607.

Converting any of the above representations of dates to SAS date values is recommended if chronological sorting or mathematical calculations are required. In addition, SAS date values permit more flexibility in displaying date output as well as the use of a number of different SAS date functions.

A SAS date value is calculated as the number of days between January 1, 1960 and any specified date (dates prior to January 1, 1960 are represented with negative numbers). The date August 8, 1990, for example, would be converted to the number 11178 as a SAS date value. This number indicates that the day it represents is the 11,178th day after January 1, 1960.

Date Manipulation

Several examples follow of how to convert character and numeric dates to SAS dates and back again, as well as of how to select data by date. In the examples, missing values are represented by '000000' for character values, 0 for numeric values, and . for SAS date values.

Example of manipulation

```
        DATA check;
        SET hosp;
                                                  * DATE CONVERSIONS *;

         IF datesep>'000000' THEN
          sdtsep=INPUT(datesep,yymmdd6.);         * Character to SAS date;
         IF sdate>0 THEN
          sdtsrg=INPUT(PUT(sdate,z6.),yymmdd6.);       * Numeric to SAS date ;
         IF sdtsep>. THEN
          datesepx=PUT(sdtsep,yymmdd6.);          * SAS date to character date ;

                                                  * SELECTING DATA BY DATE
*;

         IF '01APR90'd <= sdtsep <= '31MAR91'd;   * SAS date ;
         IF '900401' <= datesep <= '910331';      * Character date;

                                                  * DATE CALCULATIONS *;
         IF deathdt>'000000' THEN
           sdthdt=INPUT(deathdt,yymmdd6.);             * Character to SAS date
         days2dth=(sdthdt-sdtsep);                     * Calculate days to death
using SAS
                                                   dates;
        RUN;
```

When converting character dates that are 6-digits long and comprise 2-digit year, 2-digit month, and 2-digit day to SAS dates, an example of an appropriate format is "yymmdd6."; it indicates that each of the 3 fields is 2 digits wide in order of year, month, and day, and the "6" indicates that the output should be 6 digits long, e.g., 930401 (yymmdd8. would add slashes between the 2-digit components, i.e., 93/04/01).

A SAS date value, for example, 11059, can be referred to as '12APR90'd in your program. It is a SAS date constant, and "d" tells SAS to convert the calendar date to a SAS date value.

Displaying SAS Dates

In addition to using informats to store dates as SAS dates, SAS date formats are available to transform SAS date values into recognizable calendar dates for the output. Each format name ends with a period, and contains a width specification to indicate how many columns to use when displaying the date value. Several examples are presented below.

Example of display

DATA check; * Convert character date to 4 different SAS dates ;
SET hosp;
ndate1=**INPUT**(dateadm,**yymmdd6.**);
ndate2=**INPUT**(dateadm,**yymmdd6**.);
ndate3=**INPUT**(dateadm,**yymmdd6.**);
ndate4=**INPUT**(dateadm,**yymmdd6.**);
RUN;

PROC PRINT DATA=check (**OBS=**8); * 4 ways of displaying date values;
VAR dateadm ndate1 ndate2 ndate3 ndate4;
FORMAT ndate1 **date7.**; * e.g., 09AUG90;
FORMAT ndate2 **worddate18.**; * e.g., August 9, 1990 ;
FORMAT ndate3 **weekdate29.**; * e.g., Thursday, August 9, 1990;
FORMAT ndate4 **mmddyy8.**; * e.g., 08/09/90;
run;

```
  DATEADM    NDATE1         NDATE2                      NDATE3                 NDATE4

1 900809    09AUG90     August 9,1990   Thursday,August 9, 1990   08/09/90
2 910203    03FEB91   February 3,1991   Sunday,February 3, 1991   02/03/91
3 901120    20NOV90  November 20,1990  Tuesday,November 20,1990   11/20/90
4 900528    28MAY90       May 28,1990        Monday,May 28,1990   05/28/90
5 901204    04DEC90   December 4,1990   Tuesday,December 4,1990   12/04/90
6 910302    02MAR91      March 2,1991     Saturday,March 2,1991   03/02/91
7 891204    04DEC89   December 4,1989    Monday,December 4,1989   12/04/89
8 901211    11DEC90  December 11,1990  Tuesday,December 11,1990   12/11/90
```

In the above output, the variable DATEADM is displayed in 5 different ways - the original character date value and the 4 formatted values (formatted using the SAS date variables).

Warnings

* Missing date values can be represented in several ways, e.g., a blank space, a period, 000000. Check how they are represented before performing data manipulation.

* Check for illegal values which may cause SAS dates to return missing values. Also, watch the century of your date values, e.g., 18xx and 20xx.

* Watch for dates stored as numeric values; make sure that they are converted to SAS dates before using them in calculations because SAS will not provide warnings that your formulas are inappropriate/inaccurate.

VI. ACKNOWLEDGEMENTS

All four modules were prepared by Murray Werbeniuk, with some additional development and editing by Ruth Bond and Charles Burchill. Input was also provided by Shelley Derksen, Pat Nicol, Andre Wajda, and Randy Walld.

VII. REFERENCES

DiIorio, Frank C: SAS Applications Programming: A Gentle Introduction, PWS-Kent Publishing Company., 1991. 684 pp. (arrays, do loops, by-group processing, retain)

SAS Institute Inc: SAS Language and Procedures: Usage, Version 6, First Edition, Cary. NC: SAS Institute Inc., 1989. 638 pp. (arrays, do loops, by-group processing, dates)

SAS Institute Inc: SAS Language: Reference, Version 6, First edition Cary, NC: SAS Institute Inc., 1990. 1042 pp. (arrays)

SAS Institute Inc: SAS Programming Tips: A Guide to Efficient SAS Processing, Cary, NC: SAS Institute Inc., 1990, 155 pp. (dates)

Spector, Paul E., SAS Programming For Researchers and Social Scientists. Sage Publications Ltd. 166 pp. (do loops, retain)