Arrays are an integral part of data step programming, allowing for processing of multiple variables in one step. Declaring an array is as straightforward as this:

```plaintext
Array nm(20) nm1-nm20;
Array chr(20) $ 10 chr1-chr20;
```

Then the data can be manipulated in a similar easy step:

```plaintext
do i = 1 to 20;
  if nm(i) = . then do;
    nm(i) = 30 + i;
    chr(i) = 'test';
  end;
end;
```

When creating an AF data entry application, one screen may allow for input of multiple variables. Could arrays in screen control language (SCL) be as easy to use as arrays in data step code? Well, yes and no.

It is important to remember that there is a major difference between dataset variables and screen control variables, how they are used and how they can be manipulated. When coding a data step process, only dataset variables need be manipulated; both dataset and screen variables need appropriate reference within SCL code.

Assume an AF data entry system has been created where one screen allows for input of 20 numeric variables, nm1 through nm20, each with a screen length of 4. The display screen has been created and the attributes of each screen variable have been edited to change TYPE to NUM. (Although this is an AF display screen, the SCL code may be applicable to FSP).

As in any good data entry system, the command line has been turned off (to keep the user out of trouble!). Since system error messages can no longer be sent to the screen with the command line off, an error message variable &errmsg1 has been created to send messages to the user.

It would be an easy yet cumbersome task to code the validation checks for each of the 20 variables. Rather, using arrays will prove much more efficient. First, an array statement would be coded. (Since ARRAY statements are evaluated at compile time rather than during execution, these statements normally precede the INIT section of the program).

```plaintext
Array nm(20) nm1-nm20;
```

This array will be used for checking the value of the variable and for resetting this value if necessary. For example, to reset a null value to _blank_ so that the display will be cleaner:

```plaintext
do i = 1 to 20;
  if nm(i) = . then nm(i) = _blank_;
end;
```

This array cannot serve all the needs of the program; it cannot be used to test error conditions on the screen variable or set flags on that field. In other words, it cannot deal with the screen variable names, only their values. Attempting to code this operation as:

```plaintext
cursor(nm(i));
```

will cause a compile error since the value of the variable is not being reset to cursor!

An additional array must be declared:

```plaintext
Array nmfld(20) $ ('nm1' 'nm2' 'nm3' 'nm4' ...);
```

This array refers to the variable names (not values). Therefore, the elements are character values referring to the screen variable names.

Validation of these variables could be coded as:

```
&errmsg1
```

---

**Enter 0-9999 for each field:**

<table>
<thead>
<tr>
<th>&amp;nma</th>
<th>&amp;nmb</th>
<th>&amp;nmc</th>
<th>&amp;nmd</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;nme</td>
<td>&amp;nrf</td>
<td>&amp;nmh</td>
<td>&amp;nmj</td>
</tr>
<tr>
<td>&amp;nmi</td>
<td>&amp;nmk</td>
<td>&amp;nml</td>
<td>&amp;nmn</td>
</tr>
<tr>
<td>&amp;nmf</td>
<td>&amp;nmg</td>
<td>&amp;nm2</td>
<td>&amp;nm3</td>
</tr>
</tbody>
</table>

---

**NESUG '93 Proceedings**
do i = 1 to 20;
    if nm(i) ^= blank and (nm(i) < 0 and nm(i) > 9999) then do;
        alarm;
        errmsg1 = 'Enter 0 to 9999';
        nm(i) ^= blank;
        rc = field('cursor',nmfld(i));
    end;
end;

In this code the FIELD function with the 'CURSOR' argument is used to reset the cursor to the screen variable (not value).

The FIELD function may be of use in many situations, including when formatted variables such as dates are to be validated and manipulated. Assume a screen contained 5 date variables which were formatted and informatted as MMDDYYYY. Where it is easy to write a statement to check the nonformatted numeric variables as existing in a range from 0 to 9999, it would be difficult to validate the formatted date values in the same manner. Therefore the code will use system error checking of the attributes informat. Remember that the command line has been turned off and system messages cannot appear on the screen. To handle this problem, the FIELD function with the 'ERROR' argument will be used to check for system errors including type, informat and format attributes. (The CONTROL ERROR statement is added to override the system error checking and provide for customized error checking modules.)

To redefine the needs of a particular project, a screen has 5 numeric variable and 5 date variables for input and validation. A portion of the code for the array validation would be:

*Arrays to check variable values;
array nms(5) nms1 - nms5;
array dates(5) dates1-dates5;

*Arrays to check variable names;
array numsfld(5) $ ('nms1' 'nms2' 'nms3' 'nms4' 'nms5');
array datefld(5) $ ('dates1' 'dates2' 'dates3' 'dates4' 'dates5');

init:
    *Allow for customized error checking;
    control error;
    return;

main:
*Check numeric fields:
do i = 1 to 5;

    *This IF statement shows validation totally within the SCL code. The system validation by the variable's attributes is not used--by checking to see if all entries range between 0 and 9999, no non-numeric values are accepted;
    if nms(i) ^= blank and (nms(i) < 0 and nms(i) > 9999) then do;
        alarm;
        errmsg1 = 'Enter 0 to 9999';
        nms(i) ^= blank;
        rc = field('cursor',nmsfld(i));
    end;

    *This IF statement shows validation using system validation using defined variable attributes. A date field may be in error if the value is non-numeric and/or if the value is not properly informatted;
    if field('error',datefld(i)) then do;
        alarm;
        errmsg1 = 'Enter MMDDYYYY';
        dates(i) ^= blank;
        rc = field('cursor',datefld(i));
    end;

end;

return;

At first this is incredibly cumbersome and not too easy to understand. It does get easier with practice and it definitely is more efficient both programmatically and programmer-wise.

SAS™ is a registered trademark of SAS Institute, Inc., Cary, NC.

References:
SAS Screen Control Language, Reference, Version 6, SAS Institute, Inc., Cary, NC.
SAS Screen Control Language, Usage, Version 6, SAS Institute, Inc., Cary, NC.

NESUG '93 Proceedings
THREE-WAY CROSSTABULATION WITH PROC FREQ AND THE OUTPCT OPTION

Adeline Wilcox
Human Nutrition Information Service, U.S. Department of Agriculture

The OUTPCT option for the TABLES statement in PROC FREQ was introduced with SAS® Release 6.07. Used with the OUT= option on the TABLES statement, the OUTPCT option outputs row, column and table percentages from PROC FREQ. I show how a three-way crosstabulation can be output and, in a single subsequent data step, rearranged using a two-dimensional ARRAY.

These data are read by the program at right.

The variable VISIT identifies which clinic visit the record represents. A value of 1 for the variable SHOWEDUP indicates the patient attended clinic while a value of zero indicates he failed to keep his appointment. The variable GROUP identifies the treatment assignment to which the patient was randomized.

The three-way crosstabulation produces the data set shown in Box 1. The variables PCT_ROW and PCT_TABL are not shown because they will be dropped in the next data step. Notice there are 12 different combinations of values for VISIT, SHOWEDUP, and GROUP. I want to reshape these data into a data set with one observation for each value of VISIT. Rather than process the data twice, once to change observations to variables for GROUP, and once again to change observations to variables for

DATA A; INPUT VISIT SHOWEDUP GROUP;
DATALINES;
RUN;

PROC FREQ DATA=A;
TABLES VISIT*SHOWEDUP*GROUP/SPARSE
OUT=B OUTPCT;
RUN;

DATA C;
ARRAY VISITS{0:1,1:2} MISSED1 MISSED2
COMPLTD1 COMPLTD2;
ARRAY PCTS{0:1,1:2} PMISSD1 PMISSD2
PCMPLTD1 PCMPLTD2;
DO SHOWEDUP=0 TO 1;
  DO GROUP=1 TO 2 UNTIL (LAST.VISIT);
    SET B(DROP=PCT_ROW PCT_TABL PCT_COL);
    BY VISIT SHOWEDUP GROUP;
    VISITS{SHOWEDUP,GROUP}=COUNT;
    PCTS{SHOWEDUP,GROUP}=PCT_COL;
    END;
    IF LAST.VISIT THEN OUTPUT;
  END;
  DROP SHOWEDUP GROUP COUNT PCT_COL;
RUN;

DATA _NULL_; SET C;
FILE PRINT HEADER=HEADING;
PUT @7 VISIT @12 COMPLTD1 +1 PCMPLTD1 5.1
@27 MISSED1 +1 PMISSD1 5.1 @42 COMPLTD2 +1
PCMPLTD2 5.1 @57 MISSED2 +1 PMISSD2 5.1 /;
RETURN;
HEADING:
PUT /// @20 'Group 1' @50 'Group 2' //
@12 'Completed' @27 'Missed' @42 'Completed' @57
'Missed' // @5 'Visit' @12 'N' +5 '%' @27 'N' +5 '%' @42 'N' +5 '%' @57 'N' +5 '%' /;
RUN;

The three-way crosstabulation produces the data set shown in Box 1. The variables PCT_ROW and PCT_TABL are not shown because they will be dropped in the next data step. Notice there are 12 different combinations of values for VISIT, SHOWEDUP, and GROUP. I want to reshape these data into a data set with one observation for each value of VISIT. Rather than process the data twice, once to change observations to variables for GROUP, and once again to change observations to variables for
SHOWEDUP, I used a two-dimensional ARRAY. This creates a data set with one observation for each value of VISIT, the first variable named in the TABLES statement.

If there are any zero cells the SPARSE and OUTPCT options cannot be used together in Version 6.07 without producing the error message "Floating Point Zero Divide". In Version 6.08, these options can be used together.

DATA step C illustrates use of coding techniques for reshaping observations to variables. These techniques are described in the SAS Applications Guide (1) on page 81 and SAS Communications (2) on pages 37-38.

Two ARRAYs are named, one for the COUNT variable and one for the PCT_COL variable. Notice that the SET statement is inside the nested iterative DO groups. Since each DO group iterates twice, four observations from DATA step B are read into DATA step C each time DATA step C executes. DATA step C executes three times, outputting the three observations shown in Box 2. The ARRAYs create the new variables from the COUNT and PCT_COL variables. The statement "IF LAST.VISIT THEN OUTPUT;" cannot be shortened to "IF LAST.VISIT;". Doing so will drop the values for the variables created during the first iteration of the first DO loop.


Adeline Wilcox
Food Consumption Research Branch
Human Nutrition Information Service
U.S. Department of Agriculture
6505 Belcrest Road
Hyattsville, MD 20782