CHAPTER

4

SCL Arrays

Introduction

SCL supports two types of arrays: static and dynamic. The size of a static array is set when you declare the array and cannot be changed at runtime. With dynamic arrays, you do not specify a size when you declare the array, but you can use any one of several different SCL functions to define the size of the array. With a dynamic array, you can create an array of a specfied size and resize the array as needed in your program.

The differences between ARRAY statement execution in SCL and ARRAY statement execution in the DATA step are described in Chapter 7, “Using Other SAS Software Products,” on page 77.

Declaring Arrays

You can use the DECLARE statement to declare static or dynamic arrays. Arrays that are declared with the DECLARE statement are all temporary arrays. That is, they default to the _TEMPORARY_ option. (See “Using Temporary Arrays to Conserve Memory” on page 46 for more information.) For example, the following statement declares an array named MONTH that contains five character variables that are each up to three characters in length:

```
declare char(3) month[5];
```

To declare a dynamic array, you must specify an asterisk (*) for the array dimensions:

```
declare char students[*];
```

This statement declares a one-dimensional array of type character. The DECLARE statement does not set the array bounds or create any elements. Dynamic arrays are only accessible within the scope in which they are declared.

You can use the ARRAY statement to declare indirect or non-temporary arrays. You can declare only static arrays with the ARRAY statement. You can declare temporary arrays by specifying the _TEMPORARY argument in the ARRAY statement. For example:

```
array month[5] $;
```

The ARRAY statement (but not the DECLARE statement) enables you to assign names to individual array elements. For example, the following statement assigns the names JAN, FEB, MAR, APR, and MAY to the five elements in the MONTH array.:

```
array month[5] $ jan feb mar apr may;
```

You can use these names to refer to the array elements in your SCL program.

In contrast to the ARRAY statement, you cannot use the DECLARE statement to assign names to individual array elements. The following DECLARE statement declares an array named MONTH plus five more character variables named JAN, FEB, MAR, APR, and MAY:

```
declare char month[5] jan feb mar apr may;
```

Referencing Array Elements

To reference array elements, you can use the form *array-name*[*position*], where *position* is the index position of the variable in the array. This form of array reference is called *subscripting*. Subscripting is the only way to refer to array elements that were declared with the DECLARE statement. For example, FACTOR[4] is the only way to reference the fourth element of array FACTOR if it is created with the statement

```
 declare num Factor[5];
```

This DECLARE statement also produces variables FACTOR[1] through FACTOR[5].

Because you must use the DECLARE statement to declare dynamic arrays, the only way to reference the elements of a dynamic array is with subscripting. However, you cannot reference the elements of a dynamic array until you have created the array. See “Creating and Initializing Dynamic Arrays” on page 40 for more information.

You can also use subscripting to refer to elements of an array that is declared with the ARRAY statement. For example, you can use MONTH[1] and MONTH[4] to refer to the first and fourth elements of an array that is declared with the following statement:

```
array month[5] $;
```

If the array is declared with an ARRAY statement that does not assign individual names to the array elements (as shown in this example), then you can also refer to these array elements as MONTH1 and MONTH4.

If the ARRAY statement assigns names to the individual array elements, then you can also use those names to refer to the array elements. For example, if you declare your array with the following statement, then you can refer to the elements in the array using the names JAN, FEB, and MAR:

```
array month[3] $ jan feb mar;
```

Grouping Variables That Have Sequential Names

If an application program or window has a series of variables whose names end in sequential numbers (for example, SCORE1, SCORE2, SCORE3, and so on), then you can use an array to group these variables. For example, the following ARRAY statement groups the variables SCORE1, SCORE2, SCORE3, and SCORE4 into the array SCORE:

```
array score[4];
```

Note: If the variables do not already exist as window variables, then SCL defines new, nonwindow, numeric variables with those names. △

Grouping the variables into an array is useful when your program needs to apply the same operations to all of the variables. See "Repeating an Action for Variables in an Array" on page 44 for more information.

Initializing The Elements of A Static Array

By default, all elements in a numeric array are initialized to numeric missing values if the array elements did not previously exist.

You can define initial values for the elements of a static array by listing the initial values in parentheses following the list of element names in the DECLARE or ARRAY statements. Commas are optional between variable values. For example, the following ARRAY statement creates a two-item array named COUNT, assigns the value 1 to the first element, and assigns the value 2 to the second element:

```
array count[2] (1 2);
```

You can also initialize array elements with the DECLARE statement. For example, the following program declares an array named MONTH, which contains five elements that can each contain three characters, and it assigns initial values to the array elements:

```
declare char(3) month[5]=('jan' 'feb' 'mar'
                          'apr' 'may');
INIT:
   put month;
return;
```

The example produces the following output:

```
month[1] = 'jan'
month[2] = 'feb'
month[3] = 'mar'
month[4] = 'apr'
month[5] = 'may'
```

Assigning the Same Value to Multiple Elements

You can use repetition factors to initialize static arrays. Repetition factors specify how many times the values are assigned in the array. They have the following form:

```
5 * (2 3 4)
```

In this example, **5** is the repetition factor and **(2 3 4)** is the list of initial values for the array elements. If the list consists of only a single item, then you can omit the parentheses.

For example, the following ARRAY and DECLARE statements both use repetition factors to initialize the values of the array REPEAT:

```
array repeat[17] (0,3*1,4*(2,3,4),0);
declare num repeat[17]=(0,3*1,4*(2,3,4),0);
```

This example repeats the value 1 three times and the sequence 2, 3, 4 four times. The following values are assigned to the elements of the array REPEAT:

```
0, 1, 1, 1, 2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 3, 4, 0
```

Initializing Static Multidimensional Arrays

To initialize a static multidimensional array, use the ARRAY or DECLARE statement to list values for the first row of the array, followed by values for the second row, and so on. The following examples both initialize a two-dimensional array named COUNT with two rows and three columns:

```
array count[2,3] (1 2 3 4 5 6);

dcl num count[2,3]=(1 2 3 4 5 6);
```

Figure 4.1 on page 40 shows the values of the elements of this array.

Figure 4.1 Elements of the COUNT Array

		Columns		
		column 1	column 2	column 3
Rows	row 1	1	2	3
	row 2	4	5	6

For more information about arrays, see “ARRAY” on page 254 and “DECLARE” on page 330.

Creating and Initializing Dynamic Arrays

Dynamic arrays can be created and initialized in five ways:

- with the COPYARRAY function. See “Using The COPYARRAY Function” on page 43 for more information.
- using simple assignment statements that copy the values of one array into another array. For more information, see “Using Assignment Statements” on page 42.
- by a method that returns an array. See “Returning Arrays From Methods” on page 45 for more information.
- with the REDIM function. See “Resizing Dynamic Arrays” on page 41 for more information.
- with the MAKEARRAY function.

After you have declared a dynamic array, you can create the array with the MAKEARRAY function. The MAKEARRAY function creates an array of the given size with all elements in the array initialized to missing for numerics or blank for characters. The number of dimensions must be the same as what was specified in the DECLARE statement. The low bound for all dynamic arrays is 1, and the high bound is determined at runtime by the values that you specify with the MAKEARRAY (or REDIM) function. For example, the following statements create a one-dimensional dynamic array named STUDENTS that has three elements and initializes these array elements to **Mary**, **Johnny**, and **Bobby**:

Example Code 4.1 Dynamic STUDENTS Array

```
declare char students[*];
students = MAKEARRAY(3);
students[1] = 'Mary';
students[2] = 'Johnny';
students[3] = 'Bobby';
put students;
```

The low bound for STUDENTS is 1, and the high bound 3. The output for this example is

```
students[1] = 'Mary'
students[2] = 'Johnny'
students[3] = 'Bobby'
```

Resizing Dynamic Arrays

You can use the REDIM function to change the high bound of any dimension of a dynamic array at runtime. You cannot change the number of dimensions or type of the array, only the bounds. The REDIM function will also preserve the data in the array unless you resize the array to a smaller size. If you reduce the size of an array, you will lose the data in the eliminated elements.

For example, suppose that you have declared and initialized the STUDENTS array as shown in Example Code 4.1 on page 41. To add another student, you must resize the array. The following statements increase the high bound by 1 element, add the new variable STUDENTS[4], and initialize this new element to **Alice**.

```
declare num rc;
rc = REDIM(students, DIM(students) + 1);
students[DIM(students) + 1] = 'Alice';
put students;
```

All of the existing data is preserved. The low bound for the array STUDENTS is 1, and the new high bound is 4. The output for this example would be:

```
students[1] = 'Mary'
students[2] = 'Johnny'
students[3] = 'Bobby'
students[4] = 'Alice'
```

You can also use the REDIM function to create and initialize an array that has been declared but not yet created by other means. For example, the following statements declare, create, and initialize an array of five elements to numeric missing values:

```
dcl num rc;
dcl num a[*];
```

```
rc = redim(a,5);
```

There is no limit to the number of times that you can resize an array.

Note: You can use the MAKEARRAY function to resize an array, but all the data will be lost. The MAKEARRAY function will reinitialize the elements to missing numeric values or to blank character values. △

Using Array Functions with Dynamic Arrays

You can use dynamic arrays with the other existing array functions (DIM, HBOUND, LBOUND) as long as the array has been created with MAKEARRAY or REDIM. If your program references a dynamic array before it has been created, a program halt will occur. If you pass a dynamic array to a method by reference (that is, as an input parameter), you cannot resize the array using MAKEARRAY, REDIM, or DELARRAY within the method.

Copying Elements From One Array To Another

There are two ways to copy an array:

- using an assignment statement. When assigning the values of one array to another array, the two arrays must have the same size.
- using the COPYARRAY function. When using the COPYARRAY function, the arrays do not have to be the same size.

Using Assignment Statements

You can assign values to an array from another array in an assignment statement. For example, the following code copies the values of array A into array B:

```
declare num a[3] = (1 3 5);
declare num b[3];
b = a;
put b;
```

These statements produce the following output:

```
b[1] = 1
b[2] = 3
b[3] = 5
```

When you use the assignment statement to copy an array, the two arrays must have the same type, dimensions, and size; otherwise, an error condition will occur.

You can use an assignment statement to create and initialize a dynamic array that has been declared but not yet created. If you specify a newly declared dynamic array as the array to which values are to be assigned, then SCL will create the dynamic array and copy the values of the existing array into the new array.

For example, the following statements create dynamic array B to the same size as A and then copies the values of array A into dynamic array B.

```
declare num a[3] = (1 3 5);
declare num b[*];
b = a;
```

```
put b;
```

These statements produce the following output:

```
b[1] = 1
b[2] = 3
b[3] = 5
```

Using The COPYARRAY Function

You can also use the COPYARRAY function to copy elements from one array to another. By default, the COPYARRAY function produces the same result as the assignment statement and requires that the arrays be of the same type, dimension, and size. For example, the following statements copy the array A into arrays B and C:

```
declare num a[3] = (1 3 5);
declare num b[3] c[3];
rc = COPYARRAY(a,b);
put b;
c = a;
put c;
```

The output for this code would be:

```
b[1] = 1
b[2] = 3
b[3] = 5
c[1] = 1
c[2] = 3
c[3] = 5
```

However, with the COPYARRAY function, you can copy an array to an array of a different size if you set the IGNORESIZE parameter to **Y** in the call to COPYARRAY:

```
rc = COPYARRAY(array1,array2,'Y');
```

The type and dimensions of the arrays must still match. For example, the following statements will copy array A, which has three elements, into array B, which has five elements.

```
declare num a[3] = (1 3 5);
declare num b[5];
rc = COPYARRAY(a,b,'Y');
put b;
```

This code produces the following output:

```
b[1] = 1
b[2] = 3
b[3] = 5
b[4] = .
b[5] = .
```

The COPYARRAY can also be used to create dynamic arrays, just as you can create them using assignment statements. For example, the following statements create and initialize dynamic array B:

```
declare num a[3] = (1 3 5);
declare num b[*];
```

```
rc = COPYARRAY(a,b);
put b;
```

The output for this code would be:

```
b[1] = 1
b[2] = 3
b[3] = 5
```

Note: When you use the COPYARRAY function to create a new dynamic array, it is good practice to delete the newly created array using the DELARRAY function. However, if you do not delete the array with the DELARRAY function, SCL will delete the array at the end of the routine like all other dynamic arrays. See "Deleting Dynamic Arrays" on page 46 for more information. △

Repeating an Action for Variables in an Array

To perform an action on variables in an array, you can use an iterative DO statement, using the index variable for the array subscript. A DO block is especially convenient when arrays contain many elements. For example, you could use a program like the following to sum the values of the array variables and to display the total in the SUM field:

```
array month[5] jan feb mar apr may (1,2,3,4,5);
INIT:
   do i=1 to 5;
      sum+month[i];
   end;
   put month;
   put sum=;
return;
```

The example produces the following output:

```
month[1] = 1
month[2] = 2
month[3] = 3
month[4] = 4
month[5] = 5
sum=15
```

The preceding DO block has the same effect as any one of the following assignment statements:

```
sum1=jan+feb+mar+apr+may;
sum2=sum(of month[*]);
sum3=sum(of jan--may);
put sum1= sum2= sum3= ;
```

This example produces the following output:

```
sum1=15 sum2=15 sum3=15
```

Passing Dynamic Arrays to Methods

Passing a dynamic array to a method is no different than passing a static array, but the dynamic array must have been created by MAKEARRAY, REDIM, COPYARRAY, or an assignment statement. If the dynamic array is not created before the method call, then an error condition will occur.

Dynamic arrays can be resized via a method if the method's parameter is a reference array and an output parameter. See "ARRAY" on page 254 for more information on reference arrays.

Suppose you have defined the resizeDynamicArray method as follows:

```
resizeDynamicArray:method parm1[*]:O:num;
declare num rc = redim(parm1, 5);
endmethod;
```

The parameter PARM1 is output parameter and a reference array. When you call this method, it will resize the dynamic array passed to it into an array with a low bound 1 and a high bound of 5. In the following code, resizeDynamicArray resizes an array with 3 elements into an array with 5 elements:

```
declare num a[*] = MAKEARRAY(3);
object.resizeDynamicArray(a);
put a;
```

The output for this code would be:

```
a[1] = .
a[2] = .
a[3] = .
a[4] = .
a[5] = .
```

Because PARM1 is a reference array, it is using the same memory as the dynamic array A.

You can now resize the array using MAKEARRAY, REDIM, COPYARRAY, DELARRAY, or an assignment statement.

Returning Arrays From Methods

Arrays can also be assigned values from a method that returns an array. The array to which values are being assigned must have the same type, dimensions, and size as the array returned from the method. Otherwise, an error condition will occur.

Suppose you define the getArrayValues method as follows:

```
getArrayValues:method return=num(*);
declare num a[3] = (1 3 5);
return a;
endmethod;
```

To assign the values that are returned by getArrayValues to an array, you could use the following code:

```
declare num b[3];
b = object.getArrayValues();
put b;
```

The output for this example is

```
b[1] = 1
b[2] = 3
b[3] = 5
```

Dynamic arrays (that have not yet been created by other means) can be created when an array is returned from a method call. If your program returns values into a dynamic array that has been declared but not yet created, SCL will first create the new dynamic array, then copy the returned values into the new array. For example, the following code creates dynamic array B with the same size as the returned array and copies the values of the returned array into B.

```
declare num b[*];
b = getArrayValues();
put b;
```

The output of these statements is

```
b[1] = 1
b[2] = 3
b[3] = 5
```

Deleting Dynamic Arrays

The DELARRAY function is used to delete a dynamic array that has been created using the MAKEARRAY or REDIM. The array's contents cannot be accessed after the array is deleted. If you do not delete the created dynamic array using DELARRAY, the array will be automatically deleted when exiting the routine.

Using Temporary Arrays to Conserve Memory

If you want to use an array in an SCL program but do not need to refer to array elements by name, then you can add the _TEMPORARY_ argument to your ARRAY statement:

```
array total[4] _temporary_;
```

When you use the _TEMPORARY_ argument, you must use subscripting to refer to the array elements. For example, you must use TOTAL[2] to refer to the second element in the array TOTAL, defined above. You cannot use the variable name TOTAL2 as an alternative reference for the array element TOTAL[2]. Using the _TEMPORARY_ argument conserves memory. By default, SCL allocates memory for both the name of the array and the names of the individual array elements. However, when you use the _TEMPORARY_ argument, SCL allocates memory only for the array name. For large arrays, this can result in significant memory savings.

Note: Do not use the _TEMPORARY_ option if you plan to use the SET routine to read values from a SAS table directly into array elements. You must use the GETVARN or GETVARC function to read values from a SAS table into the elements of a temporary array. △

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS® Component Language: Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999.

SAS® Component Language: Reference, Version 8

ISBN 1–58025–495–0

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999