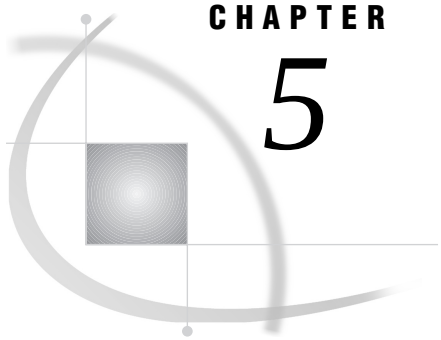

CHAPTER 5

SCL Lists

Introduction

SCL supports data structures and functions for manipulating data in SCL lists. SCL lists, like arrays, are ordered collections of data. However, lists are more flexible than arrays in many ways. For example, SCL lists are dynamic. Therefore, a program can

create a list only when and if it is needed. Lists grow and shrink to accommodate the number of items or the size of items that you assign to them. Also, an SCL list can contain items of differing data types.

Chapter 15, "SCL Elements by Category," on page 235 lists the SCL list functions. Each function and the tasks it can perform are described in Chapter 16, "SAS Component Language Dictionary," on page 249.

Creating Data Dynamically

SCL lists are dynamic rather than static. That is, SCL programs create these lists at run time. This means that list sizes are computed at run time rather than before the list is created. Further, unlike arrays, which have a fixed size, a list's length can grow or shrink to accommodate the amount of data you want to maintain.

SCL lists can contain items of mixed types, whereas SCL arrays are fixed in type. (Depending on its declaration, an array contains either numeric data or character data, but not both). One item in an SCL list can be a number and the next item can be a character string, while a third might be another list. Further, you have the freedom to replace a numeric value with a character value in a list, and vice versa. Although you can make lists that are of fixed type, you have the freedom to allow multiple types in the same list.

Note: Character values that are stored in SCL lists can be up to 32,766 characters per item. △

Identifying SCL Lists

You access lists with a list identifier, a unique value assigned to each list that you create. You must store the list identifier in an SCL variable and then reference that variable in each operation that you perform on the list. All the SCL functions that manipulate lists use the list identifier as an argument.

Note: Assigning a list identifier to another variable does not copy the list. The two variables simply refer to the same list. To copy the contents of a list to an existing or new list, use the COPYLIST function. △

Creating New Lists

SCL lists can be declared using the LIST data type, which is a reference type that stores the identifier assigned when you create the list. You assign the LIST type with the DECLARE statement. For example:

```
declare list carlist;
```

After you declare a list, you actually create it with the MAKELIST or MAKENLIST function. You can then insert numbers, characters, other lists, or objects into the list with the INSERTN, INSERTC, INSERTL, or INSERTO function, respectively. You can also specify the default number of items in the initial list by supplying an argument to the MAKELIST function. For example, the following statement makes a list that contains *n* items:

```
carlist=makelist(n);
```

Each of the *n* items is initialized to a numeric missing value. Note that *n* can be any nonnegative SCL numeric expression that is computed at run time, or it can be a simple nonnegative numeric constant such as 12, if you want to create a list with a known initial number of items. No matter how you create the list, you are free to expand or shrink it to contain as many items as you need, from 0 to as many items as your computer has memory to hold. To determine the length of a list, use the LISTLEN function.

Note: It is recommended that you declare lists with the LIST keyword to avoid problems in calling overloaded methods. See "Overloading and List, Object, and Numeric Types" on page 111 △

Example: Creating an SCL List

This section shows an SCL program that creates an SCL list, along with the output that the program produces. The program reads the columns in DICTIONARY.TABLES, a special read-only SQL view that stores information about all the SAS tables and SAS views that are allocated in the current SAS session. The columns in this view are the attributes (items of information) that are available for SAS tables. The example program stores the column values in an SCL list that is sorted in table order (the order of the columns in the ATTRIBUTES view), name order, and length order.

To create the view that lists the SAS tables, submit the following SQL procedure from the PROGRAM EDITOR window:

```
   /* Create the PROC SQL view ATTRIBUTES   */
   /* which contains information about all  */
   /* the members of type DATA in the SAS   */
   /* data libraries that are associated    */
   /* with the SAS session.                 */
proc sql noprint;
create view attributes as
   select *
      from dictionary.tables
      where memtype="DATA";
quit;
```

The SCL program creates and displays an SCL list whose values come from the view ATTRIBUTES.

```
   /* Declare COLUMNS as a list */
declare list columns;
INIT:
       /* Open the view ATTRIBUTES for reading */
   attrs=open('attributes', 'I');
   if (attrs > 0) then do;
        /* Make a list containing the same   */
        /* number of items as the number of  */
        /* columns in the view ATTRS.        */
      numcols=attrn(attrs, 'NVARS');
      columns=makelist(numcols);
      do i=1 to numcols;
            /* Set item I in list COLUMNS to  */
            /* the length of the column. The  */
            /* SETITEMN call is similar to    */
            /* the array assignment:          */
            /*   array{i} = colLen;           */
```

```
         colLen=varlen(attrs, i);
         rc=setitemn(columns, colLen, i);
             /* NAMEITEM gives item I the name */
             /* of the Ith column              */
         colName=varname(attrs, i);
         itemName=nameitem(columns, i, colName);
      end;
      sysrc=close(attrs);
         /* Print the column names in their   */
         /* order in the SAS table.  Sort by  */
         /* name and print. Then sort by      */
         /* length and print.                 */
      rc=putlist(columns,'SAS Table Order:',0);
      columns=sortlist(columns, 'NAME ASCENDING');
      rc=putlist(columns, 'Name Order:', 0);
      vars=sortlist(columns,'value');
      rc=putlist(columns, 'Length Order:', 0);
         /* Cleanup: delete the list  */
      rc=dellist(columns);
   end;
   else
      _msg_=sysmsg();
return;
```

This program produces the following output:

```
SAS Table Order:(LIBNAME=8
                MEMNAME=32
                MEMTYPE=8
                MEMLABEL=256
                TYPEMEM=8
                CRDATE=8
                MODATE=8
                NOBS=8
                OBSLEN=8
                NVAR=8
                PROTECT=3
                COMPRESS=8
                REUSE=3
                BUFSIZE=8
                DELOBS=8
                INDXTYPE=9
                )[5]
Name Order:(BUFSIZE=8
            COMPRESS=8
            CRDATE=8
            DELOBS=8
            INDXTYPE=9
            LIBNAME=8
            MEMLABEL=256
            MEMNAME=32
            MEMTYPE=8
            MODATE=8
            NOBS=8
            NVAR=8
```

```
              OBSLEN=8
              PROTECT=3
              REUSE=3
              TYPEMEM=8
              )[5]
Length Order:(PROTECT=3
               REUSE=3
               BUFSIZE=8
               COMPRESS=8
               CRDATE=8
               DELOBS=8
               LIBNAME=8
               MEMTYPE=8
               MODATE=8
               NOBS=8
               NVAR=8
               OBSLEN=8
               TYPEMEM=8
               INDXTYPE=9
               MEMNAME=32
               MEMLABEL=256
               )[5]
```

Note: [5] is the list identifier that was assigned when this example was run and may be different each time the example is run. △

Initializing the Values in a List

You can initialize an SCL list

- in a DCL statement. For example, to create a list with the constants 1, 2, 'a', 3, 'b', and 'c', you can declare the list as follows:

  ```
  DCL list mylist={1,2,'a',3,'b','c'};
  ```

  Your list may also contain sublists. For example:

  ```
  DCL list mylist={1,2,'a',mysub={'A','B','C',
                   3,'b','c'}};
  ```

- in an assignment statement after you have declared the list. For example, the following assignment statement initializes the employee list with an employee ID, name, and office location. The location is a sublist.

  ```
  DCL list employee;

  emp = {id=9999, name='Thomas',
        locate={bldg='R', room='4321'}};
  ```

- by specifying the InitialValue attribute when you create a class. In the following example, the class InitVal initializes three list attributes, which also contain sublists.

  ```
  class work.a.InitVal.class;
  public list list1 / (InitialValue=
    {COPY={
  ```

```
      POPMENUTEXT='Copy here',
      ENABLED='Yes',
      METHOD='_drop'
      },
    MOVE={
      POPMENUTEXT='Move here',
      ENABLED='Yes',
      METHOD='_drop'
      }
  }
);
public list list2 / (initialValue=
  {1,2,3,{'abc','def',{1,2,'abc'},3},'def'});
public list list3 / (initialValue=
  {id=888,name=Rob,
   answers={mchoice={'a','c','b','e'},
   math={1,4,8,9,}}
  }
);
```

For more information about creating classes, see "CLASS" on page 277.

Note: Even if you initialize a list with a DCL or assignment statement or with the **initialValue** attribute (rather than using one of the INSERT functions), you must still explicitly delete the list as described in "Deleting Lists and List Items" on page 54. △

Manipulating SCL Lists

You can create new lists and then insert numbers, character strings, objects, and even other lists into them. You can replace or delete list items, and you can move them around by reversing, rotating, or sorting a list. You can also assign names to the items in a list, and you can refer to items by their names rather than by their index (position) in the list. Thus, you can use a list to implement data structures and to access and assign values to list items by their names. Using this feature, you can add new fields to the list data structure or change the order of the list's items without modifying your SCL program.

SCL lists are maintained entirely in memory. Keep this in mind as you develop your applications. If your data is more appropriately maintained in a SAS table, you will probably want to design your application in that manner instead of trying to read the entire SAS table into a list. However, if you know your SAS table will not contain a large number of rows and many columns, and if you do not need to maintain data sharing, then you may find it convenient to read the SAS table into a list. That is, you can use SCL lists for data that you would have liked to put into an array but could not because of the restrictions imposed by arrays.

Determining the Type of a List Item

In general, SCL list functions that process data values are suffixed with either N, C, L, or O to denote the item types of numeric, character, list, or object, respectively. You can use the ITEMTYPE function to determine the type of a list element and then use a condition statement to determine which functions are used.

Passing Lists as Arguments for Methods

Lists that are not declared as LIST type are treated by the compiler as numeric types in order to maintain compatibility with Version 6. However, the more accurate specification of LIST should be used in Version 7 programs, particularly those that use lists in conjunction with method overloading. For example, suppose you use the list MYLIST as an argument for a method that has one version that takes a numeric argument and another that takes a list argument. If MYLIST is not declared as LIST type, then it is treated as a numeric type and the wrong method is called: the one that takes the numeric argument, instead of the one that takes the list argument.

When a list with type LIST is passed as an argument to a method, SCL seeks a method that accepts a LIST argument. If no exact type match is found, the list is passed to a method that accepts an numeric argument. For example, if MYLIST is declared as LIST type and is passed as an argument to method MYMETHOD, SCL will first search for a MYMETHOD that accepts lists as arguments. If none is found, SCL will pass MYLIST to a MYMETHOD that accepts numeric arguments.

Inserting and Replacing Items in Lists

To insert and replace items in a list, use the SETITEMN, SETNITEMN, SETITEMC, SETNITEMC, SETITEML, SETNITEML, SETITEMO, or SETNITEMO function. These functions can assign values to existing items or they can add new items.

With arrays, you use

```
A{i}=x;
```

but with SCL lists, you use

```
rc=setitemn(listid,x,i);
```

To add a new item to a list without replacing the existing items, use the INSERTC, INSERTL, INSERTN, or INSERTO function.

See also “Assigning Names to List Items” on page 59.

Retrieving Values from Lists

To retrieve the value of an item in a list, use the GETITEMN, GETNITEMN, GETITEMC, GETNITEMC, GETITEML, GETNITEML, GETITEMO, or GETNITEMO function.

With arrays, you use

```
x=A{i};
```

but with SCL lists, you use

```
x=getitemn(listid,i);
```

See also “Assigning Names to List Items” on page 59.

Deleting Lists and List Items

You can delete items from SCL lists by specifying the position, or index, of the item to delete; by clearing all of the values from a list; or by deleting the entire list. You can also pop items from lists, which enables you to create queues or stacks. See "Using Lists as Stacks and Queues" on page 61.

- To delete a single list item, use the DELITEM or DELNITEM function, specifying either the index or the name of the item to delete.
- To clear all the values from a list, use the CLEARLIST function, which leaves the list with a length of 0.
- To delete an entire list, use the DELLIST function. This function returns to the system the memory that was required for maintaining the list and its items.

Note: When you delete a list that has sublists, you should delete the list recursively if you do not need to use the information in the sublists. When you do not delete a list, the memory occupied by the list is not available for other tasks. To delete a list recursively, specify **Y** as the value of the *recursively* argument in the DELLIST function. For example:

```
rc=dellist(mylist,'Y');
```

△

For more information, see "Assigning Names to List Items" on page 59 and "DELLIST" on page 336.

Referencing List Items by Index Number

List indexing is similar to array indexing. An index I specifies the position of an item in the list. The first item is at index I=1, and the last item is at index I=LISTLEN(mylistid), which is the length of the list. Thus, you can use DO loops to process all items in a list, as shown in the following example:

```
do i=1 to listlen(mylistid);
   t=itemtype(mylistid,i);
   put 'Item ' i ' is type ' t;
end;
```

Accessing Items Relative to the End of a List

It is also useful for you to be able to access items at or relative to the end of an SCL list. You can use negative indices to index an item from the end of the list. Counting from the end of a list, the last item is at index –1 and the first item is at position $-n$, where n is the length of the list. Thus, you do not need to subtract indices from n to access items relative to the end of the list. All of the SCL list functions recognize negative indices.

Indexing Errors

Indexing errors occur when you supply an invalid index to an SCL list function, just as it is an error to use an invalid array index. Valid values for list indexes depend on

the function. Some functions do not accept 0 as the index, whereas other functions do. Refer to the *index* or *start-index* arguments in the dictionary entries for the SCL list functions.

Implementing Sublists and Nested Structures

SCL allows you to put one list of items inside another SCL list, thereby making a sublist. For example, you can read the columns of a SAS table row into a list. You could then insert each row into another list, and repeat this process for a range of rows in the SAS table. You then have a list of lists, where the "outer" list contains an element for each row in the SAS table, and the "inner" sublists contain each row. These lists are called nested lists.

To illustrate, consider the SAS table WORK.EMPLOYEES, created with the following DATA step program:

```
data employees;
   input fname $ 1-9 lname $ 10-18
         position $ 19-28 salary 29-34;
 datalines;
 Walter   Bluerock Developer 36000
 Jennifer Godfrey  Manager   42000
 Kevin    Blake    Janitor   19000
 Ronald   Tweety   Publicist 29000
 ;
```

The following example reads the WORK.EMPLOYEES table into an SCL list. The outer list is the list in the variable OUTERLIST. Each time through the loop, a new inner list is created. Its identifier is stored in the variable INNERLIST, and INNERLIST is inserted at the end of OUTERLIST.

```
INIT:
      /* Open the EMPLOYEES table and   */
      /* create the SCL list OUTERLIST  */
   dsid=open('employees');
   outerList=makelist();
      /* Read the first table row and    */
      /* find the number of its columns  */
   rc=fetch(dsid);
   numcols=attrn(dsid,'NVARS');
       /* For each row, make a new INNERLIST  */
       /* and create and insert the sublists  */
   do while (rc=0);
      innerList=makelist();
         /* For each column, return the name  */
         /* and type.  Insert a list item of  */
         /* that name and type into the       */
         /* row's INNERLIST.                  */
      do i=1 to numcols;
         name=varname(dsid,i);
         type=vartype(dsid,i);
         if type='N' then
            rc=insertn(innerList,(getvarn
                       (dsid,i)),-1,name);
         else
```

```
            rc=insertc(innerList,(getvarc
                       (dsid,i)),-1,name);
      end;
         /* Insert each INNERLIST as an item  */
         /* into OUTERLIST and read the next  */
         /* row of the EMPLOYEES table        */
      outerList=insertl(outerList,innerList,-1);
      rc=fetch(dsid);
   end;
      /* Close the EMPLOYEES table.  Print and  */
      /* then delete OUTERLIST and its sublists. */
   sysrc=close(dsid);
   call putlist(outerList,'Nested Lists',2);
   rc=dellist(outerList,'y');
   return;
```

This program produces the following output:

```
Nested Lists( ( FNAME='Walter'
                LNAME='Bluerock'
                POSITION='Developer'
                SALARY=36000
               )[7]   ❶
              ( FNAME='Jennifer'
                LNAME='Godfrey'
                POSITION='Manager'
                SALARY=42000
               )[9]   ❶
              ( FNAME='Kevin'
                LNAME='Blake'
                POSITION='Janitor'
                SALARY=19000
               )[11]  ❶
              ( FNAME='Ronald'
                LNAME='Tweety'
                POSITION='Publicist'
                SALARY=29000
               )[13]  ❶
              )[5] ❶    ❷
```

1 [5], [7], [9], [11], and [13] are the list identifiers that were assigned when this example was run. These values may be different each time the example runs.

2 List identifier 5 identifies the "outer" list. Each row is an inner or nested list (list identifiers 7, 9, 11, and 13).

Limitless Levels of Nesting

Nested lists are highly useful for creating collections of records or data structures. There is no limit to the amount of nesting or to the number of sublists that can be placed in a list, other than the amount of memory available to your SAS application. Further, you can create *recursive* list structures, where the list A can contain other lists that contain A either directly or indirectly. The list A can even contain itself as a list item.

Simulating Multidimensional Arrays with Nested Lists

You can declare multidimensional arrays in SCL, but all lists are one-dimensional. That is, to access an item in a list, you specify only one index. However, you can use nested lists to simulate multidimensional arrays. For example, to create a list structure that mimics a 2 by 3 array, you can use the following example:

```
array a[2,3] 8 _temporary_;
init:
   listid = makelist(2);
   lista = setiteml(listid, makelist(3), 1);
   listb = setiteml(listid, makelist(3), 2);
   call putlist(listid);
   do i = 1 to dim(a,1);
      list=getiteml(listid,i);
      do j = 1 to dim(a,2);
         a[i, j] = 10*i + j;
         put a[i,j]=;
         rc = setitemn(list,a[i,j], j);
      end;
   end;
   call putlist(listid);
return;
```

This example produces the following output:

```
((. . . )[7] (. . . )[9] )[5]
a[ 1 , 1 ]=11
a[ 1 , 2 ]=12
a[ 1 , 3 ]=13
a[ 2 , 1 ]=21
a[ 2 , 2 ]=22
a[ 2 , 3 ]=23
((11 12 13 )[7] (21 22 23 )[9] )[5]
```

Note: Not all of the program is shown here. You would need to delete these lists before ending the program. [7], [9], and [5] are the list identifiers that were assigned when this example was run and may be different each time the example is run. △

Saving Nested Lists to SCL Entries

When you save a list that contains sublists, both the list and its sublists are saved in the same SLIST entry. Thus, if you create list data structures that are highly recursive and have many cycles, you should be careful about saving your lists.

For example, suppose list A contains list B. When you save list A, you also save list B; you do not need to save list B separately, because list B is already stored in list A. In fact, if you store the lists in two separate SLIST entries and then try to read them back, you do not get the same list structure that you stored originally.

The following example creates two lists, A and B, (with text values in them to identify their contents) and inserts list B into list A. It then saves each list in separate SLIST entries, A.SLIST and B.SLIST. Then, the program creates two more lists, APRIME and BPRIME, reads the two saved SLIST entries into those two lists, and then prints all the list identifiers and list values.

```
INIT:
      /* Make lists A and B and insert an item */
      /* of text into each list.  Then, insert */
      /* list B into list A.                   */
   a = makelist();
   a = insertc(a, 'This is list A');
   b = makelist();
   b = insertc(b, 'This is list B');
   a = insertl(a, b);
      /* Save lists A and B into separate  */
      /* SLIST entries.                    */
   rc=savelist
      ('CATALOG','SASUSER.LISTS.A.SLIST', A);
   rc=savelist
      ('CATALOG','SASUSER.LISTS.B.SLIST', B);

      /* Make lists APRIME and BPRIME.  Fill  */
      /* APRIME with the contents of A.SLIST  */
      /* and BPRIME with B.SLIST              */
   aPrime=makelist();
   bPrime=makelist();
   rc=fillist
      ('CATALOG','SASUSER.LISTS.A.SLIST', aPrime);
   rc=fillist
      ('CATALOG','SASUSER.LISTS.B.SLIST', bPrime);
      /* Store list APRIME into list BINA */
   bInA = getiteml(aPrime);
   put a= b= aPrime= bPrime= bInA= ;
   call putlist(a, 'List A:',0);
   call putlist(b, 'List B:',0);
   call putlist(aPrime, "List aPrime:",0);
   call putlist(bPrime, "List bPrime:",0);
      /* Delete list A and its sublist B  */
      /* Delete lists APRIME, BPRIME, and BINA */
   rc=dellist(a,'y');
   rc=dellist(aPrime);
   rc=dellist(bPrime);
return;
```

Here is the output:

```
a=5 b=7 aPrime=9 bPrime=11 bIna=13
List A:(('This is list B
         )[7]
        'This is list A
        )[5]
List B:('This is list B
        )[7]
List aPrime:(('This is list B
          )[13]
         'This is list A
         )[9]
List bPrime:('This is list B
         )[11]
```

Note that the sublist B (13) that was read from A.SLIST is not the same as the sublist BPRIME (11) that was read from B.SLIST. That is, A contains B, but B does not contain BPRIME. Therefore, changes made to B are inherently reflected in A, whereas changes to BPRIME are not reflected in APRIME.

Also note that the structures of list A and list APRIME are the same, but the list identifiers are different and do not match any of the list identifiers that were read from B.SLIST.

Note: [5], [7], [9], [11], and [13] are the list identifiers that were assigned when this example was run and may be different each time the example runs. △

Advantages of SAVELIST Recursiveness

There is an advantage to the recursive nature of the SAVELIST function. For example, if list A contains sublists B and C, SAVELIST saves all three lists when you save A to an SLIST entry. Your application can take advantage of this if you have several unrelated lists that you want to save. By creating a new list and inserting the lists that you want saved into the new list, you can save them all in one SLIST entry with one SAVELIST call, instead of saving each sublist in a separate SLIST entry with separate SAVELIST calls.

Assigning Names to List Items

SCL supports a feature called named lists, which enable you to assign a name to each item in a list, or only to some list items. The name can be any SCL character string, not just character strings that are valid SAS column names, unless the list has the SASNAMES attribute. As with SAS names, list item names can contain mixed-case characters—for example, EmployeeLocation.

If you search a list for which the HONORCASE attribute has not been set, then SCL will uppercase the item names for the search operation only. The item names are not permanently changed to uppercase.

You can use the GETNITEMC, GETNITEMN, GETNITEML, and GETNITEMO functions to access named list items by their name rather than by their position. This feature enables you to vary the contents of the list according to your application needs without having to keep track of where a particular item is located in a list. To assign or replace values that are associated with a name, use the SETNITEMC, SETNITEMN, SETNITEML, or SETNITEMO function. To delete an item by its name, use the DELNITEM function.

Item names in a list do not have to be unique unless the NODUPNAMES attribute has been assigned to the list. Item names are stored as they are entered. If the list has the HONORCASE attribute (the default), then 'abc' and 'Abc' are two different item names. Otherwise, if the list has the IGNORECASE attribute, these names are duplicate names.

To search for an item by its name, you use the NAMEDITEM function. If the list has the HONORCASE attribute, this function searches for item names that match the case specified for the search unless you use the FORCE-UP attribute for NAMEDITEM. This attribute overrides the HONORCASE attribute and converts the item name to upper case for the search. However, the case of the item name is converted only for the search; the name continues to be stored as you entered it. The function ignores trailing blanks when searching for a matching name. If a list contains duplicate names, the search function finds the first occurrence of the name unless you have specified a different occurrence of the item for the search. By inserting a new item at the beginning of the list, you can "hide" a previous value because a named search will find your new item first by default. To restore the previous value, simply delete the new item from the list.

You can freely mix named items with unnamed items in a list. You can also use both kinds of indexing (by name or by index) in any list, regardless of how the list was created or whether all, some, or no items have names.

Indexing a Named Item by its Position

To find the index of a named item in a list, use the NAMEDITEM function. This enables you to access an item later by its index in the list, which is a faster search. However, searching by index is not safe if the index of the item might change between the time you find the index and the time you use the index.

The following statement replaces the value associated with the first occurrence of the item named **ACME** in the list NUMBERS with the value **(201) 555-2263**. These statements do not modify the list if the name **ACME** is not found:

```
i=nameditem(numbers,'Acme');
if i>0 then
   rc=setitemc(numbers,'(201) 555-2263',i);
```

Determining or Replacing an Item's Name

To replace the name of an item, use the NAMEITEM function. You can also use NAMEITEM when you want to find out the name of an item but you do not want to change the item's name.

Finding an Occurrence of a Name

In general, the functions that enable you to access a list item by its name operate on the first occurrence of the name by default. However, you can combine the optional arguments *occurrence*, *start-index*, and *ignore-case* to refer to items other than the first occurrence. *Occurrence* enables you to specify the number of the occurrence of a named item that you want to find. For example, a value of three references the third occurrence, and a value of ten references the tenth occurrence. The following example demonstrates how to find the indexes of the first and third item named **SCL**:

```
   /* default occurrence is 1   */
first=nameditem(listid,'SCL');
   /* Find the third occurrence */
third=nameditem(listid,'SCL',3);
```

Specifying Where the Search for an Item Starts

The *start-index* argument specifies the position in the list in which to begin the search for a named item. The default is 1, which starts the search at the first item in the list. If the value for *start-index* is negative, then the search starts at position ABS(*start-index*) from the end of the list and searches toward the front of the list. For example, a *start-index* of –1 references the list's last item, whereas a *start-index* of –2 references the list's second-to-last item. Thus, to change the value of the last occurrence of a list item named **X** to the value *y*, you can use a statement like the following:

```
listid=setnitemn(listid,y,'X',1,-1);
```

Using Shared Data Environments

SCL lists support shared data environments. (Without a shared data environment, if you wanted an entry to pass data to many other entries, you had to pass the data explicitly in each CALL DISPLAY statement, or else you had to put the values in macro variables. However, macro variables are limited in the amount of data they can contain (only scalar values), and their names must be valid SAS names.) By placing data in a shared data environment, other programs and even other SAS applications can retrieve the data via a name. These names can be any valid SCL string, and the value associated with a name can be a numeric value, a character value, or an entire list.

The two kinds of shared data environments are implemented with local SCL lists and global SCL lists.

Local Data Environment

Each SAS software application (such as an FSEDIT application, or a SAS/AF application started with the AF command) maintains its own application environment in a local environment list. You can store information that is local to the application, but which you want to be shared among all of an application's entries, in this local environment list. The function ENVLIST('L') returns the list identifier of the environment list for the current application. Other applications' lists are maintained in the memory of each application, and even though two lists in different applications may have the same list identifier, the lists are actually different. This is analogous to the same SAS table identifier being used by different SAS applications: the identifier actually refers to different SAS tables that are opened at different times.

Global Data Environment

There is also a global environment list that stores data that can be shared across all SAS applications started in the same SAS session or process. For example, one SAS application may place some data in the global environment list and then close. Another application may then open and read the data that was created by the first application. To access the global environment list, use the list identifier returned by ENVLIST('G').

Using Lists as Stacks and Queues

You can create lists that function as stacks (first in, last out lists) or queues (first in, first out lists).

Using a List as a Stack

To use a list as a stack, use the INSERTC, INSERTN, INSERTL, or INSERTO function to insert items into a list. The default insertion position for these functions is the beginning of the list, so you need only specify the list identifier and the data to be inserted.

To pop (or delete) an item from a stack, use the POPN, POPC, POPL, or POPO function. You can use the ITEMTYPE function to determine the type of the item at the top of the stack if your application does not know the type. If your application always puts the same data type onto your stack (for example, if the stack is a stack of character

strings and you use only INSERTC to put items into the list), then you do not need to use ITEMTYPE to check the type of the item at the top of the stack before popping.

If you do not want to keep the top value, use the DELITEM or DELNITEM function to delete the top item in the stack.

To replace the top item, use the SETITEMN, SETITEMC, SETITEML, or SETITEMO function.

You should not attempt to pop or delete an item unless you are sure the list contains at least one item. You can use the LISTLEN function to return the length of the list before you use a function to pop or delete an item.

Using a List as a Queue

When you use a list as a queue, you also use the INSERTN, INSERTC, INSERTL, or INSERTO function to put items in the list. However, you use an item index of –1 to insert an item at the end of the list.

To remove an item from a queue, use the POPN, POPC, POPL, or POPO function. As with stacks, you should use the ITEMTYPE and LISTLEN functions to verify the item's type and the list's length before popping an item from the list. Here is an example:

```
INIT:
   listid=makelist();
   rc=insertc(listid,'1st',-1);
   rc=insertc(listid,'2nd',-1);
   rc=insertc(listid,'3rd',-1);
   rc=insertc(listid,'4th',-1);
   rc=insertc(listid,'5th',-1);
   put 'Test of first in, first out queue:';
   do i=1 to listlen(listid);
      cval=popc(listid);
      put 'Popping item' i cval=;
   end;
   rc=dellist(listid);
return;
```

This program produces the following output:

```
Test of first in, first out queue:
Popping item 1 cval=1st
Popping item 2 cval=2nd
Popping item 3 cval=3rd
Popping item 4 cval=4th
Popping item 5 cval=5th
```

Assigning List and Item Attributes

You can assign attributes to lists or to items in a list. Attributes are useful for controlling the use and modification of lists. For example, you can specify that a list is not available for update, which means that other programs called by your program (for example, via CALL DISPLAY) cannot change the data in the list or cannot add or delete items from the list. You can also assign attributes such as NOUPDATE or NODELETE to individual items in a list.

Because it is easy to change the type of any item in a list simply by replacing the value with a new value, it would be quite easy for one application to accidentally

change a list in a way that you did not intend. To prevent this possibility, you may want to specify that a list or items in a list have a fixed type. When you assign the proper attributes to the lists and items that you create, you do not need to worry about other parts of the application corrupting your data, and you can avoid adding data validation statements to your programs.

Assigning list and item attributes is not required. However, doing so can facilitate application development, because an attempt to violate an attribute, which indicates a bug in the application, causes the application to stop with a fatal error.

To set the attributes of a list or item, use the SETLATTR function. The GETLATTR function returns a string that describes the current attributes. The HASATTR function returns 1 if the list or item has the specified attribute and 0 if it does not.

Using File Interfaces

Two SCL list functions enable you to store lists in SAS catalog entries or in external files and to read lists from these files. The SAVELIST function stores a list, and the FILLIST function reads data from a catalog entry or external file and fills a list with the text from the file.

Debugging List Problems

SCL provides a List Diagnostic Utility (or list analyzer), which reports any SCL lists that are not freed at the appropriate time in a program. SCL lists that are not deleted when they are no longer needed can waste significant amounts of memory. The list analyzer highlights every statement in an SCL program that creates an SCL list that is not deleted directly or indirectly by the program.

To use the list analyzer, issue the command **SCLPROF LIST ON** from any SAS window to start the data collection phase. Then invoke the window associated with the program that you want to test. When you return to the window from which you issued the SCLPROF LIST ON command, issue the command **SCLPROF LIST OFF** to end the data collection phase. The data collected during this phase is stored in the SAS table WORK.SCLTRAC1. If you end the task from which you started the data collection phase, the data collection phase ends.

Note: To avoid collecting lists that are not deleted until the end of the task or application, begin the data collection phase on the second invocation of the window that you are testing. △

As soon as the data collection phase ends, the interactive data presentation phase begins. From the data presentation phase, you can save the data by selecting **Save As** from the File menu. To view the stored data, issue the command **SCLPROF LIST DATA=*analysis-data-set*.** The interactive presentation phase opens two windows:

- The SUMMARY window displays summary statistics of the list analysis.
- The List Diagnostic Utility window lists the catalog entries containing SCL programs that created lists that were not deleted during the analysis.

If warnings were generated during the analysis, a third window opens to display the warning messages.

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