Paper 129-2013

# Hashing in PROC FCMP to Enhance Your Productivity

Andrew Henrick, Donald Erdman, and Stacey Christian, SAS Institute Inc., Cary, NC

## ABSTRACT

Hashing is an important tool often used to improve performance of operations such as merging, filtering and searching. Hashing has been supported in the DATA step for more than a decade, and beginning with SAS® 9.3 is available to user-defined subroutines through the FCMP Procedure. Subroutines have always encapsulated and modularized code, making programs reusable, and the addition of hashing allows users to extend the scope of their programs, tackling larger problems without sacrificing simplicity. This paper compares existing hashing support in the data step with the new implementation in the FCMP Procedure. Examples are provided demonstrating how hashing in user-defined subroutines is used to improve performance and streamline an existing program.

## INTRODUCTION

Hashing in this paper refers to a search algorithm. Through the use of a hash function, an input string or number (*key*) is converted to an integer number (*hash*). This hash value is then used as an index into a *hash table*. For each key, the hash table can then store and retrieve associated data. Hashing is considered the fastest way to search a large amount of information that is referenced through keys.

While hashing is used in many places behind the scenes in SAS, it is also available to SAS users through the DATA step hash object. Several good sources of information about hash objects in DATA step programs are listed in the reference section of this paper. Version 9.3 of SAS provided an important new feature; access to hash objects from within the FCMP procedure.

The FCMP procedure enables you to create, test, and store SAS functions and CALL routines, and make these reusable blocks of code available to other SAS Procedures or DATA step programs. With the addition of the hash object to FCMP, users can now embed smart hashing techniques into a myriad of other places. Among the areas of SAS which can use FCMP function are:

- DATA step programs,
- WHERE clauses,
- custom SAS macros,
- the Output Delivery System (ODS),
- and many procedures, including CALIS, GA, GENMOD, MCMC, MODEL, NLIN, NLMIXED, NLP, PHREG, REPORT, RISK, SIMILARITY, SQL.

## HASHING FROM A WHERE STATEMENT

The WHERE statement is one of the most frequently used tools to subset data in both procedures and DATA step programs. When you want to subset data based on a list of keys, the WHERE clause can quickly get unwieldy. Use of the IN() function makes the syntax a little less clumsy, but the key list must be static, i.e. either entered in directly or provided via a macro variable. As an alternative to complicated in-line WHERE clause expressions, the FCMP procedure lets us to write a function that efficiently performs the equivalent of the IN() function. We can then reuse this function in the WHERE clause anywhere in SAS.

In the following code, we use the FCMP Procedure syntax to define the HASHIN() function:

```
function hashin(name $);
   declare hash h(dataset: “work.in”);
   rc = h.defineKey(“name”);
   rc = h.defineDone();
   rc = h.check();
   return(not rc);
endsub;
```

Dissecting the HASHIN() function, we see a variable input parameter, NAME, and a fixed data set, WORK.IN. The variable NAME is used as the key to the hash object and the data set WORK.IN is used to initialize the hash object with the list of all valid key values. The HASHIN function returns a 1 if the NAME variable passed into the function is found in WORK.IN and 0 otherwise. This new HASHIN function can now be used within any WHERE clause:

```
data subset; set testdata;
  where hashin(string);
run;
```

Note that the hash object syntax used in this function is identical to the hash object syntax in the DATA step. The following code sample demonstrates the same useage of the hash object in the DATA step:

```
data subset; set testdata;
  if _N_ = 1 then do;
     declare hash h(dataset: “work.in”);
     rc = h.defineKey(“name”);
     rc = h.defineDone();
  end;
  if h.find() = 0 then output;
run;
```

However, this hash object, defined within the data step, cannot be reused elsewere for WHERE clause processing. This is the beauty of encapsulating the code within an FCMP function!

For a complete syntax description, please see Appendices B and C of this paper.

Regarding efficiency, it is important to note that the DECLARE statement is not an executable statement in FCMP. Furthermore, after DEFINEDONE is called on the hash object and WORK.IN is read, DEFINEKEY and DEFINEDONE become non-executable statements as well, with no intervening operations. The need to wrap these calls in `“If _N_ = 1 then do; … end;”`, as is done in the DATA step version, is eliminated..

## HASH OBJECTS AS ARGUMENTS

Giving the user the ability to break down a program into reusable functional blocks is a key reason to use the FCMP procedure. Since confining a hash object entirely within a function or subroutine is limiting, hash objects are now acceptable arguments to functions created with FCMP. For this example let us look at a problem of calculating the current value of a set of customers’ stock holdings within a portfolio. One data set (called ‘stocks’) provides the ID and price for each stock in the portfolio.

A hash object and corresponding hash iterator provide the mechanism for traversing all the stock IDs and prices:

```
length id $ 5;

declare hash stockhash(dataset: “stocks”);
rc = stockhash.defineKey(“id”);
rc = stockhash.defineData(“id”);
rc = stockhash.defineData(“price”);
rc = stockhash.defineDone();
declare hiter stockiter(“stockhash”);
```

Note that we use the variable ‘id’ both as a key and data variable in this hash object. This is necessary because key variables are not updated as the hash object is traversed by an iterator. Not only do we want to ensure that each key is visited, but we also need to retrieve it as a data variable (along with the data variable ‘price’) for our calculation.

Other data sets conatins the stock ID and holdings of each stock by customer. For each customer data set we have a common operation: determine if the customer owns a stock, and if so, given his holdings and the price, calculate the value. For such tasks it is natural to have a hash object for each customer data set, which we declare as follows:

```
   declare hash c1(dataset: "cust1");
   rc = c1.defineKey("id");
   rc = c1.defineData("holding");
   rc = c1.defineDone();

   declare hash c2(dataset: "cust2");
   rc = c2.defineKey("id");
   rc = c2.defineData("holding");
   rc = c2.defineDone();
```

Each of the hashes above will return the customer's holding for a given stock ID.

Now we can define a general purpose function that can be passed any customer's hash object:

```
   proc fcmp;

      /* Use stock ID and customer hash to get customer's holding. */
      /* Return product of holding and price or                    */
      /*        0, if not found                                    */

      function get_value(custhash HASH, id $, price);
         rc = custhash.find();
         if rc eq 0 then return(holding * price);
         return(0);
      endsub;
   run;
```

Notice that the HASH keyword in the argument list of the function allows you to pass different hash objects that share a common definition into the function. This will allow us to pass different customers' hash objects for evaluation.

Finally we are ready to iterate over all the stocks in our data set and calculate the total value of each customer's holdings within the portfolio. Here is the FCMP code:

```
   proc fcmp;

      /* Include declarations of hashes and functions
         outlined above                                 */

      /* Iterate over stocks, retrieving id and price. */
      /* Accumulate customers' total portfolio value.  */

      rc = stockiter.First();
      valuec1 = 0;
      valuec2 = 0;
      do while (rc eq 0);
         valuec1 = valuec1 + get_value(c1, id, price);
         valuec2 = valuec2 + get_value(c2, id, price);
         rc = stockiter.Next();
      end;

      put valuec1= valuec2=;
   run;
```

In our example, we happened to have a common key variable across all hash objects. This is not a requirement, as long as the proper key variable is set before calling the hash's FIND method. A simple modification of the get_value function to accommodate different names would be the following:

```
   function get_value(custhash HASH, stockid $, price);
      id = stockid;
      rc = custhash.find();
      if rc eq 0 then return(holding * price);
      return(0);
   endsub;
```

Similarly, the HITER keyword can be used to pass hash iterator objects to functions. Keep in mind that it is not currently possible to pass DATA step hash and hash iterator objects to a PROC FCMP function.

## CACHING WITH HASH OBJECTS

Use of a hash object is a common way to improve performance when we have an expensive process called many times, and when many of these calls have identical input. Rather than perform redundant computations, a hash allows us to store computed results (i.e., create a cache of computed results) using the values of input parameters as hash keys. In this example, we look at a function that performs a costly integration. This function depends on three input parameters. Following is the PROC FCMP code for this 'expensive' function:

```
function expensive(num_DOF, den_DOF, nc);

    cdf = 0;
    do i = 1 to 1000;
       pdf = pdf("F", i*5/(1000-1), num_DOF, den_DOF, nc);
       cdf = cdf + pdf * (5/1000);
    end;

    return (cdf);
endsub;
```

It is not important to understand exactly what the function is doing. For this example, we only need to note that the function is quite expensive to run.

If we also expect that when we call this function we tend to repeat the input parameters, it becomes a good candidate for caching. Fortunately, we expect the first two parameters to be integer valued and of restricted range; and that the third parameter is typically zero. This implies that there is a finite and reasonably small number of combinations of the inputs. Thus we can create a cache to reduce the number of times we actually do the calculation and speed up our program. Rewriting the function to use the hash object as a cache yields:

```
function faster(num_DOF, den_DOF, nc);

    declare hash h();

    rc = h.defineKey("num_DOF", "den_DOF", "nc");
    rc = h.defineData("cdf");
    rc = h.definedone();

    /* look up first and return if found */

    notfound = h.find();
    if notfound = 0 then return(cdf);

    cdf = 0;
    do i = 1 to 1000;
       pdf = pdf("F", i*5/(1000-1), num_DOF, den_DOF, nc);
       cdf = cdf + pdf * (5/1000);
    end;

    /* save the answer so we don't have to compute again */

    rc = h.add();
    return (cdf);

endsub;
```

In this new implementation, the declaration of the hash is made with no data set specified. Next, we see if the current combination of input arguments has been seen before; and if so, the precomputed integral (cdf) is returned. If the combination of the inputs has not been seen previously, the function proceeds to compute the answer, store the value in the hash object for reuse later, and finally returns the new value. Note that the hash object is keyed with the current input arguments to the function.

A quick performance test will reveal the magnitude of any gains achieved by caching. Let us use the following FCMP code to call the EXPENSIVE function (the one without caching) repeatedly with a range of arguments:

```
proc fcmp;

do i=1 to 100000;

   /* Numerator and denominator degrees of freedom range from 1 - 100 */
   num = ceil(ranuni(12345) * 100);
   den = ceil(ranuni(12345) * 100);
   nc = round( ranuni(12345) * 3);

   ans = expensive(num, den, nc);
end;
run;
```

**Integration without hashing**

```
NOTE: PROCEDURE FCMP used (Total process time):
      real time           1:26.78
      user cpu time       1:25.52
      system cpu time     0.12 seconds
      memory              828.17k
      OS Memory           10352.00k
```

**Output 1. Output from PROC FCMP without Hash Object Caching**

Now call the cached version of the function:

```
proc fcmp;

do i=1 to 100000;

    /* Numerator and denominator degrees of freedom range from 1 - 100 */
    num = ceil(ranuni(12345) * 100);
    den = ceil(ranuni(12345) * 100);
    nc = round( ranuni(12345) * 3);

    ans = faster(num, den, nc);
end;
run;
```

**Integration with hashing**

```
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.21  seconds
      user cpu time       0.20  seconds
      system cpu time     0.08 seconds
      memory              938.76k
      OS Memory           10352.00k
```

**Output 2. Output from PROC FCMP with Hash Object Caching**

This example played to the hash object's strengths. Note the drastically improved timing but an increase in memory use.

## CONCLUSION

We have demonstrated several important uses for embedding the hash object in PROC FCMP functions and subroutines. The use of hashing allows users to extend the scope of their programs, tackling larger problems without sacrificing simplicity. We are confident that our users will come up with many more uses for hash objects in this form.

## REFERENCES

- SAS Institute Inc. 2013. *The FCMP Procedure*. Cary, NC: SAS Institute, Inc. Available at http://support.sas.com/documentation/cdl/en/proc/65145/PDF/default/proc.pdf
- Dorfman, Paul, Shajenko, Lessia,and Vyverman, Koen. 2008. "Hash Crash and Beyond", *Proceedings of the SAS Global Forum 2008 Conference - http://www2.sas.com/proceedings/forum2008/037-2008.pdf*
- SAS Institute Inc. 2013. SAS® 9.3 *Component Objects: Reference*. Cary, NC: SAS Institute, Inc. Available at http://support.sas.com/documentation/cdl/en/lecompobjref/63327/PDF/default/lecompobjref.pdf
- Loren, Judy. 2008. "*How* Do I Love Hash Tables? Let Me Count The Ways*!*", *Proceedings of the SAS Global Forum 2008 Conference*
- Secosky, Jason and Bloom, Janet. "Getting Started with the DATA Step Hash Object", Available at http://support.sas.com/rnd/base/datastep/dot/hash-getting-started.pdf

## RECOMMENDED READING

- *Base SAS® Procedures Guide*
- *SAS® Hash Object Programming Made Easy*

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Andrew Henrick
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-3362
Andrew.Henrick@sas.com
http://www.sas.com

Donald Erdman
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-7685
Donald.Erdman@sas.com
http://www.sas.com

Stacey Christian
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-4135
Stacey.Christian@sas.com
http://www.sas.com

## APPENDIX A – FULL CODE SAMPLES

### Code for WHERE Clause Example

```
data work.in;
 name = "test";
run;
proc fcmp outlib=work.foo.in;

/* returns 1 if name is work.in data set */
function hashin( name $ );

   declare hash h(dataset:"work.in");
   rc = h.defineKey( "name");
   rc = h.definedone();

   rc = h.find();
   return( not rc );
endsub;

run;

option CMPLIB=work.foo;

%let datasize =1000000;
%let subsetsize =1000;

data testdata;  /* generate sample data */

length string $ 16;

do i = 1 to &datasize;
   string = put(md5(rand("UNIFORM")), hex16.);
   output;
end;
drop i;
run;

data in; set testdata(obs=&subsetsize);
rename string=name;
run;

data subset; set testdata;
where hashin(string);
run;

data subset; set testdata;

if _N_ = 1 then do;
   declare hash h(dataset:"work.in");
   rc = h.defineKey( "name");
   rc = h.definedone();
end;

name = string;
rc = h.find();
if rc =0 then output;
run;
```

### Code for Hash Argument Example

```
data work.stocks;
input id $ price;
cards;
S0001 157.23
S0002 42.56
S0003 12.75
S0004 38.61
S0005 101.28
S0006 97.46
S0007 23.57
```

```
S0008 32.75
S0009 48.62
S0010 14.84
run;

data work.cust1;
input id $ holding;
cards;
S0003 4
S0008 10
S0010 10
run;

data work.cust2;
input id $ holding;
cards;
S0001 20
S0005 10
S0006 10
run;

proc fcmp;

   function get_value(custhash hash, id $, price);
      rc = custhash.find();
         if rc eq 0 then
            return(holding * price);
      return(0);
   endsub;

   length id $ 5;

   declare hash stockhash(dataset: "stocks");
   rc = stockhash.defineKey("id");
   rc = stockhash.defineData("id");
   rc = stockhash.defineData("price");
   rc = stockhash.defineDone();
   declare hiter stockiter("stockhash");

   declare hash c1(dataset: "cust1");
   rc = c1.defineKey("id");
   rc = c1.defineData("holding");
   rc = c1.defineDone();

   declare hash c2(dataset: "cust2");
   rc = c2.defineKey("id");
   rc = c2.defineData("holding");
   rc = c2.defineDone();

   rc = stockiter.First();
   valuec1 = 0;
   valuec2 = 0;
   do while (rc eq 0);
      valuec1 = valuec1 + get_value(c1, id, price);
      valuec2 = valuec2 + get_value(c2, id, price);
      rc = stockiter.Next();
   end;

   put valuec1=;
   put valuec2=;
run;
```

### Code for Caching Example

```
   proc fcmp outlib=work.foo.foo;

   /* Create an expensive function to compute a cdf --------------*/
   function expensive( num_degrees_of_freedom, den_degrees_of_freedom, non_centrality);

       cdf = 0;
```

```
    do i = 1 to 1000;
       pdf = pdf("F", i*5/(1000-1), num_degrees_of_freedom,
                   den_degrees_of_freedom, non_centrality);
       cdf = cdf + pdf * (5/1000);
    end;

    return (cdf);
endsub;

/* Create a faster function to compute a cdf with caching ----------*/
function faster(num_degrees_of_freedom, den_degrees_of_freedom, non_centrality);

    declare hash h();

    rc = h.defineKey( "num_degrees_of_freedom", "den_degrees_of_freedom",
                      "non_centrality");
    rc = h.defineData("cdf");
    rc = h.definedone();

    /* look up first and return if found */
    notfound = h.find();
    if notfound = 0 then return(cdf);

    cdf = 0;
    do i = 1 to 1000;
       pdf = pdf("F", i*5/(1000-1), num_degrees_of_freedom,
                    den_degrees_of_freedom, non_centrality);
       cdf = cdf + pdf * (5/1000);
    end;

    /* save the answer so we don't have to compute again */
    rc = h.add();

    return (cdf);

endsub;


run; quit;

options cmplib=work.foo fullstimer;

proc fcmp;

num =15; den=17; nc=0;
do i=1 to 100000;

   /* Numerator and denominator degrees of freedom range from 1 - 100 */
   num = ceil(ranuni(12345) * 100);
   den = ceil(ranuni(12345) * 100);
   nc = round( ranuni(12345) * 3);

   *ans = expensive( num, den, nc );
   ans = faster( num, den, nc );
end;
put ans=;
run;
```

## APPENDIX B - FCMP HASH OBJECT SYNTAX

With the SAS® 9.3 release, the following statements/methods are supported for FCMP hash objects:

- DECLARE
- DEFINEKEY
- DEFINEDATA
- DEFINEDONE
- DELETE
- FIND
- CHECK
- NUM_ITEMS
- ADD
- REMOVE

The usage and syntax of these methods is identical to the DATA step component hash object. For more information about what is available to DATA step users, please refer to the *SAS® 9.3 Component Object Reference* guide (SAS Institute Inc. 2013).

## DECLARE STATEMENT

The declare statement is used to create a new instance of a hash object and if desired, initialize data. The syntax for the declare statement is as follows:

```
DECLARE hash object-name<(argument_tag-1:value-1, … argument_tag-n: value-n)>;
```

The declare statement along with the 'hash' keyword tells PROC FCMP you wish to create a new hash object with the name provided. The arguments and values that follow the name are optional.

### Arguments and Values

*Dataset: 'dataset_name'*

Specifies the name of a SAS data set to load into the hash object. The name must be provided as a quoted string. Currently SAS data set options such as WHERE clauses and renaming data set variables are not supported.

*Hashexp: n*

The expected size of the hash object's internal table, expressed as a power of 2. The default value for n is 8, which equates to a hash table size of $2^8$ or 256.

### Differences between PROC FCMP and DATA Step

In the DATA step, the DECLARE statement is an executable statement. This means that steps must be taken to ensure that the hash object is declared only once.

```
data _null_;
    if _N_ = 1 then do;
          declare hash h(dataset:"data1", hashexp:4);
          …
          …
    end;
    set data2;
    …
    …
run;
```

Functions within PROC FCMP are meant to be reusable blocks of code. For the sake of efficiency, PROC FCMP treats the declare statement as declarative and creates the hash object at parse time, not at run time.

## DEFINEKEY METHOD

The benefit of using a hash object is data is stored and retrieved based on lookup keys. This provides a fast and flexible means to organize the data without first requiring a SAS data set be manipulated into the state desired. The DEFINEKEY method is used to set up the key variables for the hash object. Multiple calls to DEFINEKEY are permitted but will be ignored after the DEFINEDONE method is called for the hash object, even across multiple calls to the function declaring the hash object.

```
rc = object.DEFINEKEY('keyname-1'<,…'keyname-n'>);
```

### Arguments

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash object defined in the declare statement.

*'keyname'*

The name of the key variable. This can be expressed directly as a quoted string or indirectly through a character variable. The key variable itself can be a character or numeric variable. At least one key variable must be given but if desired multiple keys are supported.

### Alternate Form

```
rc = object.DEFINEKEY(ALL:'YES');
```

If the DATASET option was used in declaring the hash object, this is a convenient shortcut to define all data set variables as key variables. Keep in mind that if a shortcut is used to set the values of the key variables in another hash object method, for example:

```
rc = myhash.FIND(key:99, key:"Staff");
```

The key values provided must be given in the order of the columns in the data set.

## DEFINEDATA METHOD

After the key variables for the hash object are set, the data variables to associate with the keys needs to be defined. This is done using the DEFINEDATA method. Multiple calls to DEFINEDATA are permitted but will be ignored after the DEFINEDONE method is called for the hash object, even across multiple calls to the function declaring the hash object.

```
rc = object.DEFINEDATA('dataname-1'<,…'dataname-n'>);
```

### Arguments

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash object defined in the declare statement.

*‘dataname’*

The name of the data variable. As with keys, this can be expressed directly as a quoted string or indirectly through a character variable. The data variable itself can be a character or numeric variable. At least one data variable name must be provided.

#### Alternate Form

```
rc = object.DEFINEDATA(ALL:’YES’);
```

If the DATASET option was used in declaring the hash object, this is a convenient shortcut to define all data set variables as data variables. Keep in mind that if a shortcut is used to set the values of the data variables in another hash object method, for example:

```
rc = myhash.ADD(key:99, data:”Staff”, data:”Director”);
```

The data values provided must be given in the order of the columns in the data set.

### DEFINEDONE METHOD

Indicates that the key and data variable specification are complete, the hash object is defined, and can be initialized. If the DATASET option was used in declaring the hash object, the data set is loaded at this time. After DEFINEDONE is called for a hash object any subsequent calls to DEFINEKEY, DEFINDDATA, and DEFINEDONE will be ignored, even across multiple calls to the function declaring the hash object.

```
rc = object.DEFINEDONE();
```

#### Arguments

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash object defined in the declare statement.

### DELETE METHOD

When you are finished with a hash object, the delete method will free any resources allocated. The hash object cannot be used after delete is called, any attempt to do so will result in an error.

```
rc = object.DELETE();
```

#### Arguments

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash object defined in the declare statement.

### FIND METHOD

Searches a hash object based on the values of defined key variables. If the lookup is successful, defined data variables are updated with the corresponding data. If the lookup fails, no data variables will be touched.

```
rc = object.FIND(<KEY: keyvalue-1, … KEY: keyvalue-n>);
```

**Arguments**

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash object defined in the declare statement.

*KEY: keyvalue*

An optional shortcut method for specifying values for keys defined using the DEFINEKEY method. The values provided must match the type for each key defined. An exhaustive list is expected, in other words the number of 'KEY: keyvalue' pairs must match the number of keys defined.

## CHECK METHOD

Searches a hash object based on the values of defined key variables. Whether the lookup succeeds, data variables will not be updated.

```
rc = object.CHECK();
```

**Arguments**

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash object defined in the declare statement.

## NUM_ITEMS METHOD

Returns the number of items in a hash object.

```
variable_name = object.NUM_ITEMS();
```

**Arguments**

*variable_name*

The number of items in the hash object is returned, not a return code, so direct assignment to a local numeric variable is expected.

*object*

The name of the hash object defined in the declare statement.

## ADD METHOD

Adds data for to be associated with provided key values.

```
rc = object.ADD(<<KEY: keyvalue-1, … KEY: keyvalue-n>,<DATA: datavalue-1, … DATA:
datavalue-n>>);
```

**Arguments**

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash object defined in the declare statement.

*KEY: keyvalue, DATA:datavalue*

An optional shortcut method for specifying values for keys and data defined using the DEFINEKEY and DEFINDEDATA methods. The values provided must match the type for each key and data variable defined. An exhaustive list is expected, in other words the number of 'KEY: keyvalue' and 'DATA:datavalue' pairs must match the number of key and data variables defined.

### REMOVE METHOD

Removes data associated with provided key values from the hash object.

```
rc = object.REMOVE(<KEY: keyvalue-1, … KEY: keyvalue-n>);
```

#### Arguments

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash object defined in the declare statement.

*KEY: keyvalue*

An optional shortcut method for specifying values for keys defined using the DEFINEKEY method. The values provided must match the type for each key variable defined. An exhaustive list is expected, in other words the number of 'KEY: keyvalue' pairs must match the number of key variables defined.

## APPENDIX C - FCMP HASH ITERATOR OBJECT SYNTAX

With the SAS® 9.3 release, the following statements/methods are supported for FCMP hash iterator objects:

- DECLARE
- FIRST
- LAST
- NEXT
- PREV

The usage and syntax of these methods is identical to the DATA step component hash iterator object. For more information about what is available to DATA step users, please refer to the *SAS® 9.3 Component Object Reference* guide (SAS Institute Inc. 2013).

### DECLARE STATEMENT

The declare statement is used to create a new instance of a hash iterator object. The syntax for the declare statement is as follows:

```
DECLARE hiter object-name("hash-objname");
```

The declare statement along with the 'hiter' keyword tells PROC FCMP you wish to create a new hash iterator object with the name provided. The argument that is enclosed in parentheses is the hash object name as a quoted string. The hash object must have been declared at least to create a hash iterator object for it.

### FIRST METHOD

Updates the defined data variables with the first data items in the hash object. The hash object definition must be complete, indicated by using the DEFINEDONE method, before the hash iterator object can be used.

```
rc = object.FIRST();
```

#### Arguments

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash iterator object defined in the declare statement.

### LAST METHOD

Updates the defined data variables with the last data items in the hash object. The hash object definition must be complete, indicated by using the DEFINEDONE method, before the hash iterator object can be used.

```
rc = object.LAST();
```

**Arguments**

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash iterator object defined in the declare statement.

### NEXT METHOD

Returns the data items in the hash object in key order. If you use the next method without calling the first method, next will start with the first item in the hash object.

```
rc = object.NEXT();
```

**Arguments**

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash iterator object defined in the declare statement.

### PREV METHOD

Returns the data items in the hash object in reverse key order. If you use the PREV method without calling the last method, PREV will start with the last item in the hash object.

```
rc = object.PREV();
```

**Arguments**

*rc*

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

*object*

The name of the hash iterator object defined in the declare statement.