SAS® GLOBAL FORUM 2020 #SASGF

Paper 5178-2020

Dynamic Programming With The SAS® Hash Object

Paul Dorfman, Independent Consultant

Don Henderson, Henderson Consulting

ABSTRACT

The original impetus behind the development of the SAS hash object was to provide SAS programmers with a fast and simple-to-use memory-resident storage and retrieval medium that could be used for matching data sets via table lookup. However, the forward-looking SAS R&D team ended up with a product that goes far beyond this simple functionality by creating the first truly run-time dynamic SAS data structure. At the basic level, its dynamic nature is manifested by the ability of a hash object table to grow and shrink at run time, ridding the programmer of the need to know its size in advance. At a more advanced level, the very structure and behavior or a hash table can be affected at run time via argument tags that accept general SAS expressions. Finally, a hash object can be made store pointers to other hash objects and used to manipulate them as pieces of data. Combined, these features entail possibilities for dynamic programming unfathomed in the pre-hash SAS era. This paper is an attempt to illustrate, via a series of examples, at least some of these possibilities and hopefully sway some programmers away from hard-coding habits by showing what the SAS hash object can offer in terms of truly dynamic DATA step programming.

INTRODUCTION

The dynamic nature of the SAS hash object manifests itself in many guises. When you first use it, you notice that it is not necessary to allocate memory for it to make sure all its items will fit into it - as it would be if an array were used. Rather, every time an item is added, more memory is acquired dynamically at run time, while the attribute reflecting the number of items is adjusted automatically. As you use the hash object more, you discover that instead of hard-coding hash arguments you can code SAS expressions - for example, to define keys, data, or I/O destinations at run time. At some point, you learn that a hash table can contain data identifying different hash object instances and use it to activate, at run time, the instance your program needs to work on. Getting comfortable with these dynamic capabilities gradually weans you off the urge to assemble code at compile time, as you realize that the hash object can do most things you need at run time inside the DATA step without resorting to external constructs like macros. This paper is an attempt to illustrate this concept by illustrating the various dynamic aspects of the hash object.

GROW AND SHRINK

First, let us note that for the purposes of this paper, we are using the terms "hash object", "hash object instance", and "hash table" interchangeably. The first dynamic angle of the hash object we will consider is related to its run-time ability to acquire and release memory needed to store its items. To recap, a *hash item* represents a hash table row containing actual values of the hash variables in the key and data portions (conceptually somewhat similar to an observation in an indexed SAS data set).

After a hash object instance is first created and initialized, it contains no items. When we add an item, SAS immediately, at run time, acquires just enough memory for this item according to its size. When we remove the item, this exact amount of memory is released to the disposal of the operating system. Equally importantly, thanks to the hashing algorithm behind these operations, they occur (a) extremely rapidly and (b) in *O(1)* time. *O(1)* is just a cryptic way to denote that an operation takes the same time regardless of the number of items currently present in the table.

DATA DEDUPLICATION

The ability of the hash object to grow one item apiece at run time lends itself to a very simple and elegant approach to data deduplication. The latter falls in to two basic task categories:

- Simple: Given a file, get the records with the first occurrence of a given key-value.
- Selective: From each group of records with the same key, select one record based on the values of one or more non-key fields.

Let us look at simple deduplication first.

Simple Deduplication

Suppose we have an input file like this:

```
data have ;
  input key data ;
  cards ;
3   31
4   41
4   42
3   32
2   21
4   43
3   33
1   11
4   44
2   22
;
```

We are interested in reading the file once and outputting the records shown in boldface, as they represent the first instances of each key-value in the file. The simplest algorithm of attaining the goal follows the natural human "paper-and-pencil" method:

1. Prepare *some kind* of an empty table.
2. Read the next record (i.e. the first at the outset).
3. If the record key is not in the table, output the record and store the key in the table.
4. If the record is not the last on file, repeat from #2.
5. Otherwise, stop.

The *kind of table* that suits the purpose best should be able to:

1. Grow by 1 item *dynamically at run time* with each new key added
2. Reside in memory to make searches and insertions fast
3. Run both searches and insertions in *O(1)* time

In SAS, the only table with this combination of properties is offered by the hash object; so, let us use it and follow the plan drafted above to a tee:

```
data want ;
  if _n_ = 1 then do ;         /* Before file is read */
    dcl hash h() ;             /* Create instant of H */
    h.definekey ("key") ;     /* Define table key    */
    h.definedone () ;         /* Initialize instant  */
  end ;
  set have end=lr nobs=n ;    /* Read next record    */
  if h.check() ne 0 then do ; /* Key is not in table */
    output ;                  /* Output              */
    h.add() ;                 /* Add key to table    */
  end ;
run ;
```

Before the step terminates, the table H contains all the unique keys from the input file. We can use it to extract the information about the number of both distinct keys and duplicates by adding the following piece of code before the RUN statement:

```
  if lr ;
  n_unique = h.num_items ;
  n_dupes = n - n_unique ;
  put (n_unique n_dupes) (=) ;
```

No counting needs to be done thanks to the attribute H.NUM_ITEMS, auto-adjusted every time a hash item is added or removed. The simplicity of this deduplication method results from using the hash object as a dynamic storage medium. Without it, organizing an efficient search-insert process proves to be a substantial programming effort. For example, if our key were composite, we would only need to define its components in the key portion and keep the rest of the program intact; while basing the search-insert process on arrays would require rather intricate custom coding. Thus, by using the hash object, we get a very simple deduplication method with a number of benefits to boot, to wit:

- The records in the unduplicated file are kept in the original input order.
- No sorting is required - and no re-sorting if keeping the original order is necessary.
- The duplicate-key records can be written to a separate file by adding a single line of code.

Note that this task does not require to store any actionable data in the data portion of H. (The hash variable KEY is auto-defined in the data portion since no DEFINEDATA call is present, though no use is made of it.) This is because we only need to output the records with the first instance of any given key. In a more complex case of *selective deduplication* (considered next) we will actually need it.

Selective Deduplication

Suppose that instead of selecting the records with just the first occurrence of a given key-value, we need to select a specific record from all the records sharing the same key-value based on a certain condition. Imagine, for example, that DATA in our sample file represents a date and from all the records with the same KEY, we want to select the first record with the most recent date. In other words, we want to select and output the following (KEY,DATA) pairs from our sample file in the following (i.e. original) relative sequence:

```
3   33
1   11
4   44
2   22
```

To do this, we need to modify the program plan:

- Use H as before. However, add DATA to the data portion, so that its value could be later compared with DATA (renamed as _D) from the file. Also, define the input record number RID to the data portion to help keep the output in the input order if need be
- Read the next record, and search H for the value of KEY from the record
- If not found, add a hash item with the KEY, DATA, and RID values from the record

- Otherwise, get the DATA value from the table and compare it with _D in the record
- If _D > DATA, update H with KEY, DATA=_D, and RID as the current record number.

After all the records have been processed, the items in H contain the needed KEY and DATA values, so we can write them as records to a file. If the output file is needed in the original file order, the indexed RID can be used with BY to read it in the required sequence. Let us look at the amended plan expressed as SAS code:

```
data _null_ ;
  if _n_ = 1 then do ;
    dcl hash h() ;
    h.definekey  ("key") ;
    h.definedata ( "key", "data", "rid" ) ;
    h.definedone () ;
  end ;
  set have (rename=data=_d) end = lr curobs = _r ;
  if h.find() ne 0 or _d > data then do ;
    data = _d ;
    rid  = _r ;
  end ;
  h.replace() ;
  if lr then h.output (dataset: "want (index=(rid))") ;
run ;
```

The amended parts of the plan are shown in boldface:

- Call the DEFINEDATA method to define KEY, DATA, and RID in the hash data portion. Rename DATA=_D on input to be able to compare the DATA values retrieved from H with the input values of _D from the current record.
- Call the FIND method.
- If KEY from the current record is found in H (FIND returns 0), the PDV DATA value is auto-updated with the hash value of DATA. If _D > DATA, the corresponding H-item needs to be updated with DATA=_D.
- Otherwise, the value of KEY is new (not yet in H), so assign DATA=_D and insert it alongside with KEY into H as a new hash item.
- In either case, call the REPLACE method. If KEY is new, REPLACE adds an item with the KEY and DATA=_D. Otherwise if _D > DATA, it updates H with KEY and DATA=_D.
- After the last record was read (LR=1), H contains the (KEY,DATA) pairs needed for the output. Call the OUTPUT method to write its content to the SAS data set WANT.

Note that if instead of the records with the *maximal* values of DATA for a given key, the records with the *minimal* values of DATA are needed, it can be accommodated by simply reversing the sign in the inequality _D > DATA. Selective deduplication has two notable differences compared to simple deduplication:

- We cannot output the deduplicated records on the fly and thus have to store them in H as items before rendering the output. This is because before having read the file all the way through, we do not know which value of DATA for a given key-value is at maximum.
- The order of items in H does not follow the input sequence since it is determined by the internal hash structure. However, the indexed RID added to the output can be used to read the output in the original input record sequence using the BY statement.

An alert reader may ask how the program in its current form could accommodate many other non-key input file variables to be output in addition to KEY and DATA. It is a good question because simply following the code template to add more variables gets the more unwieldy, the more extra variables have to be handled:

- First, all the extra variable would have to be added to the data portion, which can inflate the hash memory footprint.
- Second, all of them would have to be renamed on input and reassigned just like DATA and RID, which inflates the code and makes it convoluted.

However, there is a good way out of such a situation. Instead of doing any of the above:

- Read the input file keeping only KEY and DATA and execute the deduplication logic as is.
- Instead of calling the OUTPUT method, iterate through H and use RID to get the needed records together with all the needed variables directly from HAVE via POINT=RID.
- If keeping the original record order is required, create an ordered table, R, with only RID defined as both key and data. When H is ready, reload RID from H to R; then use RID from R to read the needed recods from HAVE directly via POINT=RID.

Below, a code variant corresponding to the situation described in the third bullet is shown:

```
data want (drop = _:) ;
  if _n_ = 1 then do ;
    dcl hash h() r(ordered:"a") ;
    h.definekey  ("key") ;
    h.definedata ("data", "rid") ;
    h.definedone () ;
    r.definekey ("rid") ;
    r.definedone () ;
    dcl hiter hi("h") ri("r") ;
  end ;
  set have (keep=key data rename=data=_d) end = lr curobs = _r ;
  if h.find() ne 0 or _d > data then do ;
    data = _d ;
    rid  = _r ;
  end ;
  h.replace() ;
  if lr ;
  do while (hi.next() = 0) ;
    r.add() ;
  end ;
  do while (ri.next() = 0) ;
    set have point = rid ;
    output ;
  end ;
run ;
```

This way, the deduplication algorithm is kept intact, and there is no need to define H with any more data variables than DATA. (In fact, now we do not even need KEY in there since it comes directly from HAVE via POINT=RID.) The only extra overhead is the extra table R. However: (a) it is needed only if the input record order is to be preserved and (b) its item size at 16 bytes is the smallest possible, so it does not impact the memory much. On the plus side, reading HAVE via POINT=RID in order makes it faster by reducing paging.

Finally, note that if you need to declare more than one hash or hash iterator object, it can be done in a single DCL (DECLARE) statement, as shown on the lines 3 and 9 above.

GROW + SHRINK

Thus far, we have only seen how the hash object *grows* dynamically at run time. At times, a data structure is needed that would also shrink when an item is removed from it. Typical examples are binary trees, queues, and stacks:

- There is no need to implement a binary tree in SAS because the hash object is a collection of binary AVL trees. If the argument tag HASHEXP:0 is specified, the object has only one (2**0=1) underlying binary tree. So, by having the hash object, we already have binary trees implemented in SAS. An oblique proof is that if a hash table defined with HASHEXP:0 is loaded with unsorted keys and then its content is output, the output is always sorted regardless of whether the ORDERED:"A" argument tag is coded or not; and this is because a binary tree is always traversed in the ascending key order.

- Queues are implemented in SAS via the LAG function. Though it is a true queue and very useful, it is *static*, meaning that the number of its items is fixed at compile time. The hash object can be used to implement a dynamic queue, whose number of items can grow and shrink at run time.
- Stacks are not implemented in SAS via a specific structure of function. So, let us implement a stack using the hash object here, especially since it well exemplifies its "grow+shrink" dynamic angle.

Implementing A Stack

A *stack* is a list of items accessed and operated upon according to the LIFO principle (Last In - First Out). It encompasses two actions:

1. Push: Add an item on the top of the stack.
2. Pop: Extract the item pushed last and take it off the stack.

Of course, it can be implemented using an array. The problem is, since we do not know ahead of time how many items are going to be pushed, nor do we know how to size the array. Not so with the hash object because it does not need to be sized beforehand. The algorithm of implementing a stack is rather straightforward. First, create a hash table with an integer key SEQ to track the item number. Then:

1. To *push* an item on the stack, increase the current number of items in the table by 1 and assign the result to SEQ; then insert the item into the table.
2. To *pop*, find the item with the key-value of SEQ equal to the current number of items.
3. Remove the item just popped from the table. This way, the next time a push or pop is performed, the item pushed immediately before the item just popped is always on top.

In the program below, the numeric items 111, 222, 333 are first *pushed* onto the stack in a loop; then the next loop *pops* them one at a time until no items are left on the stack:

```
data _null_ ;
  dcl hash s() ;
  s.definekey  ("seq") ;
  s.definedata ("item") ;
  s.definedone () ;
  call missing (seq, item) ;
  put ">>> Pushed on  stack: " @ ;
  do item = 111, 222, 333 ;
    seq = s.num_items + 1 ;
    s.add() ;    /* grow by 1 item */
    put item @ ;
  end ;
  put / "<<< Popped off stack: " @ ;
  do until (seq = 0) ;
    seq = s.num_items ;
    s.find() ;
    s.remove() ; /* shrink by 1 item */
    put item @ ;
    seq = s.num_items ;
  end ;
run ;
```

The message printed in the SAS log as a result is:

```
>>> Pushed on  stack: 111 222 333
```

```
<<< Popped off stack: 333 222 111
```

Calling the REMOVE method immediately after an item is popped is what makes the table shrink dynamically. Later in the paper, we will see how the program above can be made much more palatable by utilizing the ability of the hash object to work with its arguments as general SAS expressions.

HASH ARGUMENTS AS EXPRESSIONS

In the SAS documentation related to the hash object, its arguments and argument tags are invariably exemplified as *hard-coded constants*. While simplifying coding examples, it creates the impression that this the only permissible way to parameterize hash code.

In reality, the hash object methods and operators cam always accept argument as general SAS run-time expressions. A constant is merely the simplest form of an expression. The assumption that hash code arguments are limited to constants has two negative consequences:

- Unnecessary kludges and subterfuges are used to make the code work
- The dynamic power of the hash object based on its run-time flexibility is underused

Let us look at some examples to illustrate (a) how hard-coded hash arguments can get in the way of dynamic hash programming, (b) how it can be avoided, and (c) what kind of benefits using general expressions with hash object arguments offers.

DEFINING KEYS AND DATA

Every SAS programmer first coming in touch with the hash object quickly internalizes that in order to define key and data hash variables, they need to be listed in the DEFINEKEY and DEFINEDATA method calls as comma-separated quoted names, such as:

```
data _null_ ;
  length K1-K2 8 D1-D3 $ 12 ;
  if _n_ = 1 then do ;
    dcl hash h() ;
    h.definekey  ("K1", "K2") ;
    h.definedata ("D1", "D2", "D3") ;
    h.definedone () ;
    call missing (of K:, of D:) ;
  end ;
run ;
```

When we have just a few hash variables to define, there is no problem with this method, even though even in this case it is rather easy to miss a quote or a comma. But imagine that you need to define, say, 10 K-variables in the key portion and 30 D-variables - in the data portion. It is fairly obvious that typing them all, replete with the quotes and commas, is a pretty tedious and error-prone exercise. So, a naturally lazy (in a good way) programmer might decide to create the lists programmatically in a separate step and insert them into the hash code as macro variables, for example:

```
data _null_ ;
  length klist dlist $ 256 ;
  do i = 1 to 2 ;
    klist = catx (",", klist, quote (cats ("K", i))) ;
  end ;
  do i = 1 to 3 ;
    dlist = catx (",", dlist, quote (cats ("D", i))) ;
  end ;
  call symputx ("klist", klist) ;
  call symputx ("dlist", dlist) ;
run ;
```

After that in the hash step, the methods defining the hash variables would be called as:

```
  h.definekey  (&klist) ;
  h.definedata (&dlist) ;
```

This is certainly much better than hard-coding and typing but still leaves the impression that for all its dynamic might, the hash object needs external help in a rather simple situation. Well, in actuality, it does not!

One Define Call At A Time

Even though you cannot find it anywhere in the documentation, if more than one variable is defined to the key or data portion, they need not be defined all in a single call. From the standpoint of the final result, there is no difference between:

```
  h.definekey  ("K1", "K2") ;
  h.definedata ("D1", "D2", "D3") ;
```

and the following sequence of calls:

```
  h.definekey  ("K1") ;
  h.definekey  ("K2") ;
  h.definedata ("D1") ;
  h.definedata ("D2") ;
  h.definedata ("D3") ;
```

Of course, typing a number of such separate calls is no better (in fact, worse). However, it remains true only as long as the arguments are hard-coded as constants.

Upping The Game: Using Non-Constant Expressions

But they do not have to be hard coded as constants - because they can be coded as variables! Taking the first call as an example, it can be recoded this way:

```
  length kd $ 32 ;
  kd = "K1" ;
  h.definekey (kd) ;
```

In other words, the DEFINEKEY argument does not have to be a (quoted) character literal but instead it can be a valued *variable* . Above, the variable KD plays this role - defined as $32 because it is intended to hold the name of a SAS variable. Since its value can be varied at run time, we can now define the keys and data one at a time *in a loop*, thus avoiding the hard-coded calls altogether. For example, they can be replaced this way:

```
  length kd $ 32 ;
  do i = 1 to 2 ;
    kd = cats ("K", i) ;
    h.definekey (kd) ;
  end ;
  do i = 1 to 3 ;
    kd = cats ("D", i) ;
    h.definedata (kd) ;
  end ;
```

Note that now (a) we no longer have to venture outside the DATA step to compose the key- and data-defining arguments and (b) we can accommodate any number of K-variables and D-variables by simply changing the upper bounds of the DO loops.

Compared to the hard-coded constants, using a variable is surely a leap forward. But the variables are still *simple* expressions, just a tad up in the hierarchy after the constants. The next logical question an inquisitive programmer would ask is whether we can bypass the interim variable and assign the CATS expressions directly as the DEFINEKEY and DEFINEDATA arguments. As you may have guessed, the answer is "yes". The hash object does not care what kind of *general expression* is used, as long as it is (a) of the correct data type (in this case, character) and (b) resolves to the correct value. Thus, our code can be shortened still further:

```
do i = 1 to 2 ;
  h.definekey (cats("K", i)) ;
end ;
do i = 1 to 3 ;
  h.definedata (cats("D", i)) ;
end ;
```

In this case, the needed expression is represented by a function rather than a variable. To show this picture from yet another practical angle, let us code it differently:

```
array K    8 K1-K2 (2 * . ) ;
array D $ 12 D1-D3 (3 * "") ;
do over K ;
  h.definekey (vname(K)) ;
end ;
do over D ;
  h.definedata (vname(D)) ;
end ;
```

Now the methods are valued with expressions based on a different function; but the final result is the same. Note that by using the arrays K and D, we can get rid of both the LENGTH statement and CALL MISSING since the arrays do both (a) parameter type matching (i.e. create a PDV host variable for each hash variable defined), and (b) initialize the PDV host variables as well.

The ability of the hash object to define keys and data one variable at a time using variables or more complex expressions as arguments becomes invaluable when we need to create a generic program for a certain type of data processing parameterized by control tables specifying how particular hash tables should be defined. This is because instead of hard-coding specific names for the key and data variables into the DEFINEKEY and DEFINEDATA method calls, we could read a control table whose variables (or expressions based on them), assigned as arguments to the methods, determine the content of the hash table entry. Detailed examples of this kind of parameterization (for multi-level data aggregation) can be found in Chapter 9 of (Dorfman and Henderson, 2018).

USING EXPRESSIONS FOR ARGUMENT TAGS

The DEFINEKEY and DEFINEDATA methods use arguments directly, as a function would. However, most of the time the hash object syntax uses arguments assigned to *argument tags*. It is true for both statements and methods. Here is an example of argument tags in an assignment statement that uses the _NEW_ operator to create a hash object instance:

```
h = _new_ hash (hashexp:16, ordered:"A", multidata:"Y", dataset:"stuff") ;
```

where the keywords HASHEXP, ORDERED, and MULIDATA are all argument tags; and 16, "A", and "Y" are the arguments assigned to them, respectively. The same can be used when the action of the _NEW_ operator is incorporated in the combined statement:

```
 declare hash h (hashexp:16, ordered:"A", multidata:"Y", dataset:"stuff") ;
```

Similar argument tag syntax is used with the hash object methods, for example:

```
  h.find (key:7) ;
  h.check (key:"FL") ;
  h.replace (key:1, data:"USA") ;
```

Note that in these examples all the argument tags are assigned constants. Just like with the untagged arguments, there is nothing wrong about it (and often this is exactly what it needed). However, using general expressions instead of constants offers much more programming flexibility. Let us look at some practical examples.

Choosing Input Data Set Names Conditionally

Suppose we have a programming situation whereby we want to load a hash table with data from a specific data set and in a specific order depending on a run-time condition. A naive attempt to make it happen could look like this:

```
  if foo then dcl hash h (dataset:"FOO", ordered:"A") ;
  else        dcl hash h (dataset:"BAR", ordered:"D") ;
```

However, *it fails to work*, ending in the compiler error "*Variable h already defined*". This is because the DCL statement declares a non-scalar variable H intended by SAS to hold the identifier of the active hash object instance; but the compiler can see it declared only once. When it sees the second DCL statement, it reacts negatively in the same manner (and with a similar error message) as it does when it sees an array declared more than once.

To prevent this, we can rephrase the code by assigning the needed argument values to extra variables and using these variables to value the argument tags accordingly:

```
  if foo then do ;
    dsn = "FOO" ;
    seq = "A" ;
  end ;
  else do ;
    dsn = "BAR" ;
    seq = "D" ;
  end ;
  dcl hash h (dataset:dsn, ordered:seq) ;
```

That does work. But again, a variable is the second simplest expression, while argument tags can accept general expressions. Thus, the entire piece of code above can be replaced, for example, with the following statement:

```
  dcl hash h (dataset: ifc(foo, "FOO", "BAR"), ordered: char("AD", 1 + ^foo)) ;
```

Any other pair of expressions for DATASET and ORDERED will work as well, as long as they are of the required data type and resolve to the same values. From this example alone it should be intuitively clear how dynamic and flexible hash code can be when the idea of using general expressions as hash arguments is properly internalized. By induction, the inquisitive programmer would surmise that by varying the DATASET argument tag at run time, one can control hash output data set names as well. Not only this is true, but output data sets can be created with names depending on a run-time condition.

Creating Output With Conditionally Named Data Sets

To illustrate the concept, let us look at the historically first example of using general expressions to value hash argument tags - splitting a file. Consider a shorter version of our sample file HAVE and assume that it is intrinsically sorted by KEY:

```
data have ;
  input key data ;
  cards ;
1   11
2   21
2   22
3   31
3   32
3   33
;
```

Suppose that we want to split the file into 3 files with the following names: WANT_K1, WANT_K2, WANT_K3 intended to hold the input records with KEY=1, KEY=2, KEY=3, respectively. Outside of the hash object coding domain, this task invariably requires a two-step process with two passes through the input file to: (1) collect N distinct key-values and (2) assemble conditional code to distribute the input records between the N files named in the DATA statement. However, with the hash object it can be done in a single pass using the following scheme:

- Read the input file one BY KEY group at a time.
- While reading each group, store the data in a hash table.
- After each group is read, pass the expression CATX("_","WANT",KEY) to the DATASET argument tag to the OUTPUT method. The method will open a file with the newly formed data set name, write the table content to it, and close the file.

Expressing it in the SAS language:

```
data _null_ ;
  if _n_ = 1 then do ;
    dcl hash h (ordered:"A") ;
    h.definekey ("rid") ;
    h.definedata ("key", "data") ;
    h.definedone () ;
  end ;
  do rid = 1 by 1 until (last.key) ;
    set have ;
    by key ;
    h.add() ;
  end ;
  h.output (dataset: cats ("_", "want", key)) ;
  h.clear() ;
run ;
```

The purpose of coding the hash object with the ORDERED:"A" argument tag is to ensure that the records in the output have the same relative order as in the input.

Incidentally, this exercise demonstrates yet another dynamic capability brought to SAS with the advent of the hash object. As any SAS programmer not familiar with it would readily notice, the step writes out three data sets not even named in the DATA statement. The reason is that the hash object manages I/O all by itself, and the OUTPUT method opens, writes, and closes the data set whose name is passed to its DATASET argument tag completely autonomously, all at run time.

If you turn the SAS DATA step debugger on, you will observe that as soon as the OUTPUT method is called after a BY group, the information of a new data set written out immediately appears in the SAS log. This is a stark departure from the manner SAS data sets are written when they are named in the DATA statement - for in this case, they remain open until the DATA step has finished its execution and are closed only after it is over.

Note that the DO loop with the SET statement inside it employs the construct known as the DoW loop to process the input one BY group at a time. For more information about it see (Henderson et al., 1991; Dorfman and Vyverman, 2009; Allen, 2010; Foty, 2012; Li, 2014).

Back To The Stack

By using general expressions as hash arguments, we can now neaten our earlier program that implements a stack. In the table below, its core push and pop logic (stripped of the PUT statements for clarity) is shown side by side in two variants: (a) as implemented earlier using expressions pre-assigned to the SEQ and ITEM variables and (b) using expressions assigned directly to the argument tags:

Assigned to SEQ and ITEM	Assigned Directly to Argument Tags
`do item = 111, 222, 333 ;`	`do item = 111, 222, 333 ;`
`  seq = s.num_items + 1 ;`	`  *--- not needed ---;`
`  s.add() ;`	`  s.add (key:s.num_items + 1, data:item) ;`
`end ;`	`end ;`
`do until (seq = 0) ;`	`do until (s.num_items = 0) ;`
`  seq = s.num_items ;`	`  *--- not needed ---;`
`  s.find() ;`	`  s.find (key:s.num_items) ;`
`  s.remove() ;`	`  s.remove (key:s.num_items) ;`
`  seq = s.num_items ;`	`  *--- not needed ---;`
`end ;`	`end ;`

Note how (a) the NUM_ITEMS attribute is automatically adjusted behind-the-scenes as the table grows and shrinks and (b) its value forms the argument tag KEY expressions pointing to the key-value needed at the moment.

Other Run-Time Dynamic Structures

As mentioned above, dynamic data structures other than stacks that can be implemented using the hash object. Dynamic queues, link lists, etc. can be easily implemented as well. For more information, see (Dorfman and Henderson, 2018; Haikou and Maddox, 2014; Hoyle, 2009).

HASH OF HASHES

As any SAS programmer knows, in a situation when a number of variables have to be treated similarly, a good prescription is to incorporate the variables in an array and loop through it, applying the treatment to one element per iteration. The simplest example is using an array to avoid the hard-coding of what Ian Whitlock has colloquially christened as "wall paper". Suppose, for instance, that we have 100 variables H1-H100 and want to get their running sum, i.e. add H1 to H2, then add H2 to H3, and so on. A static approach would be to hard code 100 assignment statements. A dynamic approach would be to array the H-variable and process then in a DO loop, thus reducing the 100 statements to 4. Below, the two approaches are shown side by side:

Static: "Wall Paper" Code	Dynamic: Array + DO loop

`H[2] = H[2] + H[1] ;`
`H[3] = H[3] + H[2] ;`
`. . .`
`/* the other 96 statements */`
`. . .`
`H[99] = H[99] + H[98] ;`
`H[100] = H[100] + H[99] ;`
	`array H[100] ;`
	
	`do j = 1 + lbound (H) to hbound (H) ;`
	`  H[j] = H[j] + H[j-1] ;`
	`end ;`

Writing the "wall paper" code means that it would need to be edited every time the summation formula, the number, or indeed the name of the variables may change. Using the array instead, we just tell SAS how to treat each array item, while the constant 100 defining the number of iterations essentially serves as a parameter. Now let us look at a situation where we have many hash tables that need to be treated identically and see whether the same dynamic principle can be applied.

MULTI-TABLE SUBSETTING

Let us consider a simple example to help illustrate the concept. Suppose that we want to subset the data set SASHELP.CARS by some of its variables, whose names and the values by which to subset are stored in separate tables in a permanent library KEYS. For example, imagine that we have 3 such tables:

`data keys._make ;`
`input make :$13. division :$8. ;`
`cards ;`
`Acura     Luxury`
`Chevrolet Standard`
`Honda     Standard`
`;`
	`data keys._type ;`
	`input type :$8. note :$8. ;`
	`cards ;`
	`Sedan  4-door`
	`SUV    All-type`
	`Truck  Small`
	`Wagon  5-door`
	`;`
		`data keys._cylinders ;`
		`input cylinders shape :$3. ;`
		`cards ;`
		`6  I+V`
		`8  V`
		`;`

Note that the names of the key variables match the names of the corresponding control data sets, except that the name of each data set is intentionally prefixed by an underscore. (This is done for the convenience of being able to read all the data sets in the library using a colon as a wild card. This convenience, as well as the non-key data variables DIVISION, NOTE, and SHAPE, while irrelevant to the task at hand, will be used in the sections devoted to the multi-file data augmentation later on.)

Any programmer familiar with the SAS hashing basics knows that a hash table can be used to subset a file against another file. So, it is logical to apply the process to each of the control tables using three hash tables, to wit:

- Create three separate hash tables (one per control file)
- Load them with the key-values from the respective control files
- For each record from SASHELP.CARS, search each of the hash tables
- If the respective key-values are found in each, output the record

This basic scheme is workable; the only question is how to implement it. Let us compare two approaches: (a) hard-coded and (b) dynamic.

Subsetting By Hard-Coding Multiple Hash Tables

A natural stream-of-consciousness inclination would be to just hard code each of the following:

- Three hash tables with three different names
- The loading of the key-values from the control files into them
- Searching the three hash tables, one at a time

The step below is just the above translated into SAS *verbatim*. Note that most programmers would likely name the hash tables after the respective control files. Below, they are named H1-H3 with an eye towards making the code dynamic later on.

```
data cars (drop = match) ;
  if _n_ = 1 then do ;
    dcl hash h1 (dataset:"keys._make") ;
    h1.definekey ("make") ;
    h1.definedone () ;
    dcl hash h2 (dataset:"keys._type") ;
    h2.definekey ("type") ;
    h2.definedone () ;
    dcl hash h3 (dataset:"keys._cylinders") ;
    h3.definekey ("cylinders") ;
    h3.definedone () ;
  end ;
  set sashelp.cars ;
  match = 0 ;
  match + h1.check() = 0 ;
  match + h2.check() = 0 ;
  match + h3.check() = 0 ;
  if match = 3 ;
run ;
```

As testing shows, this code works as expected. However, it is also easy to envision certain problems with this hard-coded approach, for example:

- If files are deleted from or added to KEYS, the step has to be almost totally recoded
- In production, that would mean having to deal with change control, versioning, etc.
- It leaves a bad taste on the palate of any dynamically-inclined programmer since all the actions related to the table H1-H3 (i.e. declare, define key, initialize, search) follow exactly the same pattern for each of them

So, a programmer who does not like hard-coding and/or dealing with change control red tape would likely think of making the program more dynamic. Leaving aside the idea of making it into a macro (which is of course possible but antithetical to the topic at hand), one may wonder if the tables H1-H3 could be incorporated in an array, which then could be used to loop through in a manner similar to our array example above.

This is not an unsound idea! However, a simple test shows that it cannot be done because this attempt, for example:

```
data _null_ ;
  dcl hash h1() h2() h3() ;
  array h[*] h1-h3 ;
run ;
```

results in the following warm greeting from the SAS log:

```
ERROR: Cannot create an array of objects.
ERROR: DATA STEP Component Object failure. Aborted during the COMPILATION phase.
```

The reason is that the variables H1-H3 created in the PDV by the DCL statement are not *scalar* (i.e. numeric or character). Rather, they are *non-scalar* (of the *type hash object*); and non-scalar variables cannot be part of a *regular* SAS array since its elements must be scalar. It raises the question: Is there a different kind of array that can house non-scalar variables? In general, a structure known as *associative array* can do this. But does SAS have associative arrays?

HOH (HASH-OF-HASHES): THE CONCEPT

Well, those who came in contact with the SAS hash object at the time of its inception may remember that it was first christened "associative array"; its much terser alias "hash" has been adopted by its users later. And despite the fact that the term has long fallen out of use, you still can code:

```
  dcl associativearray h() ;
```

without running into any syntactic problems. In other words, a hash object instance itself can serve as an array of the exact kind we need. It can have non-scalar variables defined in its data portion, while their non-scalar values can identify existing hash object instances (and also hash iterator object instances as well). Such hash object instance is colloquially termed as "Hash of Hashes" or, abbreviated, simply as "HoH".

HoH Mechanics

Before applying HoH to our task at hand, let us first see how it can be organized and take a look at its underlying mechanics. As an example, let us do the following:

- Create a hash object (associative array) HoH
- Define it with a numeric key HOHKEY to be valued with consecutive integers
- Add HOHKEY to the data portion of HOH as well
- In the data portion, define a non-scalar variable H of type hash
- In each iteration of a DO loop from HOHKEY=1 to HOHKEY=2:
    1. Create an instance a hash object H
    2. Define it with key HKEY in both the key and data portions and initialize it
    3. Load it with 3 hash items (instance 1) and with 6 items (instance 2)
    4. Add an item with the current HOHKEY and current PDV value of H to HoH
- Create a hash iterator IHOH linked with HOH
- Iterate through HOH one item at a time and for each iteration:
    1. Retrieve the value of H and HOHKEY from HoH into the PDV
    2. Find the number of items N_ITEMS in the instance pointed at by the PDV value of H
    3. Display HOHKEY and N_ITEMS in the SAS log

The intent of displaying HOHKEY and N_ITEMS is to see how many items the two instances of H, whose pointers are stored in HoH, contain. If the number were 3 and 6 , respectively, it would confirm that each H value stored in HOH correctly identifies the instance of H to which it is linked. Now let us express this plan *verbatim* using the SAS language:

```
data _null_ ;
  dcl hash hoh (ordered:"a") ;
  hoh.definekey  ("hohkey") ;
  hoh.definedata ("hohkey", "h") ;
  hoh.definedone () ;
  do hohkey = 1 to 2 ;
    dcl hash h() ;
    h.definekey ("hkey") ;
    h.definedone() ;
    do hkey = 1 to hohkey * 3 ;
      h.add() ;
    end ;
    hoh.add() ;
  end ;
  dcl hiter ihoh ("hoh") ;
  do while (ihoh.next() = 0) ;
    n_items = h.num_items ;
    put hohkey= n_items= ;
  end ;
run ;
```

The info printed in the SAS log shows:

```
hohkey=1 n_items=3
hohkey=2 n_items=6
```

This confirms that the pointers to the two instances of H, stored in HOH as the values of the non-scalar variable H and then retrieved into the PDV via the iterator, do indeed identify their respective instances correctly. Here are the most critical logical parts of this scheme:

- The DCL HASH H() statement plays two roles:
- At compile time, it declares the non-scalar PDV variable H of type hash object.
- At run time, its hidden behind-the-scenes execution of the _NEW_ operator:
  1. Creates a new hash object instance linked with the variable H
  2. Generates a unique non-scalar value and tags the instance with this value
  3. Stores this value in the PDV, thus making the instance *active*

Active Hash Object Instances

The notion of an *active hash instance* is the most important concept in this scheme of things. An instance of a hash object H is *active* when its non-scalar identifying value (or pointer if you will) is currently stored as the PDV value of variable H. When a hash method or an attribute is referenced by a hash object name (e.g., H.DEFINEKEY, H.ADD, H.NUM_ITEMS, etc.), the component interface needs to know to which instance of H the requested operation must be applied. In order to do that, the interface looks at the pointer value of H currently stored in the PDV, locates the hash instance tagged by the value, and applies the operation to this instance.

Therefore, if we want to tell SAS programmatically which hash instance to work on, we need to place its pointer value (generated when the instance was created) into the non-scalar PDV variable whose name is used to refer to the object. If we have but a single instance to deal with, we have nothing to do in this respect since the pointer value, stored in the PDV at the time of its creation, stays there for the duration of the step. However, if we have more than one instance, as in the program above, every time a new instance of H is created, *its* pointer value is stored in the PDV variable H, overwriting the value with which the instance created earlier is tagged.

Hence, if the previously generated pointer is not stored anywhere to be recalled later, all the instances created before the last one continue to exist (and occupy memory), yet there is no way to identify any of them and make any of them active if need be. The table HoH serves as a storage for the pointer values generated for all hash instances created, each identified in the table by the value of HOHKEY. This way, if we need to tell SAS to work on instance #N, we can retrieve its hash pointer value H from the table HoH using HOHKEY=N, update the PDV host variable H with it, and go on our merry way using the instance #N we have thus made active. In the second DO loop of the above program, the next existing instance is made active in the next iteration of the loop because the iterator retrieves its hash pointer value of H into the host variable H. This is why in each iteration, H refers to a different hash instance, from which the attribute NUM_ITEMS is extracted.

Of course, using the iterator is not the only way to retrieve the hash value of H into its PDV host variable H from HoH. It can be done with any hash object method capable of performing the *Retrieve* hash operation, notably, by calling the FIND method. For example, if instead of printing the value of H.NUM_ITEMS for each instance of H we only needed to do it for the instance #2, we could ditch the iterator and replace the entire DO WHILE loop with:

```
    hoh.find (key:2) ;
    n_items = h.num_items ;
    put hohkey= n_items= ;
```

Essentially, if in the process of learning the ropes of HoH you have the concept of the active hash instance internalized, everything falls into its proper logical bucket. Now let us return to the original task of making the subsetting of SASHELP.CARS HoH-dynamic.

DYNAMIC MULTI-TABLE SUBSETTING

Now we are armed to the teeth to un-hard-code the subsetting task in the associative array style. Since in the library KEYS the data sets names match the names of the variables by which to subset save for the leading underscore, we can key the HoH table using the library member names and also use them to define the key hash variables for the hash instances we are going to create. To make the code even more dynamic, the dictionary view SASHELP.VSTABLE can be used to extract the information about all the subsetting control tables in a loop. So, here is the plan:

- Before any records from SASHELP.CARS are read in:
- Create a HoH table keyed by the member name in the library KEYS
- Create a hash iterator IHOH linked to the HOH table
- For each member from SASHELP.VSTABLE in the library KEYS:
  1. Create an instance of a hash object H corresponding to the member name
  2. Value its argument tag DATASET dynamically with the fully qualified name of the subsetting control data set to be loaded into it
  3. Define its key as the value of the member name (minus the underscore)
  4. Initialize the instance by calling the DEFINEDONE method for this particular instance
  5. Add a new key item with the value of the instance pointer H to HoH
- Read the next record from SASHELP.CARS
- Use the iterator IHOH to loop through the instances of H, and in each iteration:
  1. If the corresponding key variable is found in the current instance, add 1 to MATCH
- If MATCH equals the number of items in HoH (i.e. the number of the members in KEYS), all the input variables have matches in the data sets from KEYS, so output the record

Translating this plan *verbatim* into the SAS Language:

```
data cars (drop = match libname memname) ;
  if _n_ = 1 then do ;
    dcl hash hoh() ;
    hoh.definekey  ("memname") ;
    hoh.definedata ("h") ;
    hoh.definedone () ;
    dcl hiter ihoh ("hoh") ;
    do until (lr) ;
      set sashelp.vstable (keep = libname memname) ;
      where libname = "KEYS" ;
      dcl hash h (dataset: catx (".", libname, memname)) ;
      h.definekey (substr (memname, 2)) ;
      h.definedone() ;
      hoh.add() ;
    end ;
  end ;
  set sashelp.cars ;
  do match = 0 by 0 while (ihoh.next() = 0) ;
    match + h.check() = 0 ;
  end ;
  if match = hoh.num_items ;
run ;
```

Note that the argument of the DEFINEKEY method is an expression intended to get rid of the leading underscore. The log notes from running this step appear as follows:

```
NOTE: There were 2 observations read from the data set KEYS.CYLINDERS.
NOTE: There were 3 observations read from the data set KEYS.MAKE.
NOTE: There were 4 observations read from the data set KEYS.TYPE.
NOTE: There were 3 observations read from the data set SASHELP.VSTABLE
      WHERE libname='KEYS';
NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.CARS has 27 observations and 15 variables.
```

Evidently, using the dynamic expression to value the DATASET argument tag, though coded but once, directs each separate instance of H to load from its own separate data set as soon as the instance is initialized by the DEFINEDONE call. Also, the match checks against each separate instance can now be run in a DO loop without the need to hard code each call. Note that with the program in this form, if the content of the library KEYS were changed to exclude some or include more of subsetting control files, nothing in the program would have to be changed.

At last, a disclaimer: Since this is the SAS System, there are at least a dozen other ways to approach the task. By demonstrating the technique shown above, we do not intend to claim its absolute superiority over any other. Rather, the goal is to illustrate the dynamic capabilities of the hash object from a specific angle and give the reader some hash food (perhaps not tasted yet) for thought.

SIMILAR HOH APPLICATIONS

File subsetting based on multiple control tables is perhaps the simplest task to which the HoH principle can be fruitfully applied. However, with a few coding changes, the same principle can be applied to a number of similar tasks. For example, here are few obvious candidates:

- Adding variables to a file from multiple files - essentially, a multi-table join
- Updating a file with data from multiple control tables
- Multi-Level data aggregation

Dynamic Multi-Table Data Augmentation

Now suppose that in addition to subsetting the input file, we need to add the data variables DIVISION, NOTE, and SHAPE from the control data sets in KEYS to the output records. The program below does just that:

```
data cars (drop = match libname memname name) ;
  if _n_ = 1 then do ;
    dcl hash hoh() ;
    hoh.definekey  ("memname") ;
    hoh.definedata ("h") ;
    hoh.definedone () ;
    dcl hiter ihoh ("hoh") ;
    do until (lr) ;
      do until (last.memname) ;
        set sashelp.vcolumn (keep = libname memname name) end = lr ;
        where libname = "KEYS" ;
        by memname ;
        if first.memname then dcl hash h (dataset: catx (".", libname, memname)) ;
        if substr (memname, 2) = upcase (name) then h.definekey (name) ;
        else h.definedata (name) ;
      end ;
      h.definedone() ;
      hoh.add() ;
    end ;
    if 0 then set keys._: ;
  end ;
  match = 0 ;
  set sashelp.cars ;
  do while (ihoh.next() = 0) ;
    match + h.find() = 0 ;
  end ;
  if match = hoh.num_items ;
run ;
```

The few changes from the pure subsetting program needed to accommodate what is essentially a multi-table inner join are shown above in boldface:

- Since we now need to add a call to DEFINEDATA, we need to loop through VCOLUMN instead of VSTABLE to get the values of NAME as the argument to DEFINEDATA
- The key is defined for the active instance of H only if NAME = SUBSTR (MEMNAME,2)
- Otherwise, NAME is used as the argument to DEFINEDATA
- Because the new data variables are not part of SASHELP.CARS, the corresponding host variables need to be defined in the PDV at compile time, which is done by IF 0 THEN SET
- Since in this case, not only we need to check whether the record keys have their matches in the corresponding control tables but also *retrieve* the data portion hash values into the PDV, the FIND method is called instead of the CHECK method

However, the overall HoH scheme, as you can readily see, remains exactly the same as with the pure subsetting: All the control tables from KEYS are handled dynamically, and if any should be deleted and/or added, there is no need to change the code at all.

Dynamic Multi-Table Update

The data augmentation example shown above suggests that updating a file record from multiple control tables in the same manner is a trivial matter of applying the same code. The only difference is that the data variables in the control tables would have the same names as the file variables to be updated. In general, the hash object is a fantastic update tool that comes in quite handy, for example, when a data warehouse fact table needs to be updated from its dimensions. A comprehensive treatment of the topic can be found in the book (Dorfman and Henderson, 2018) where the entire Chapter 7 "Supporting Data Warehouse Star Schemas" is dedicated to it.

Multi-Level Data Aggregation

A few SAS tools can compare with the power and flexibility of the hash object in terms of its ability to create any kind of data aggregates - both additive and non-additive (such as count distinct) in a single pass through the input data. As the authors have learned in the real data aggregation world the hard way, under certain circumstances using the hash object is the only way to aggregate massive input to the point of actual data delivery, even when everything else fails due to constraints imposed by the enterprise computing resources.

This topic is way too big for a mere section in a paper. In fact, the authors have devoted an entire paper to it (Dorfman and Henderson, 2015) and included two chapters on the topic in their book (Dorfman and Henderson, 2018). From the standpoint of the dynamic nature of the hash object, Chapter 9 is of special interest, as it goes in a detailed way over using the HoH approach to create a table-driven process of computing multi-level aggregates. In this section, we just scratched it surface to illustrate the general principle.

COMPUTING NLEVELS

So ... Mr. Stat walks into the cube of Mr. Hash with a stick it note in hand, shows it and says: "I have a SAS file like this on the left, and I want to get the number of distinct values for each N-variable to get a file looking like this on the right":

Have	Want
`  data have ;`	`data want ;`
`  input N1-N5 ;`	`  input Nvar :$32. Count ;`
`  cards ;`	`  cards ;`
`3 5 4 7 9`	`N1 3`
`1 6 4 2 5`	`N2 2`
`1 5 4 7 7`	`N3 1`
`8 6 4 9 3`	`N4 4`
`1 5 4 1 8`	`N5 5`
`;`	`;`

The dialog continues as follows:

– Mr. Hash: No problem. Just use proc FREQ with the NLEVELS option and output what you need via the ODS.
– Mr. Stat: I know. Problem is, this is just a small sample. In reality, I have nearly 1000 N-variables and about 1 million records. FREQ+ODS takes seemingly forever to run, plus before it can finish, it flat out runs out of memory. Can it be done with the hash object?
– Mr. Hash: Sure, if you had just 5 N-variables, you could code a separate hash for each and compute what you need easily. But 1000? Hmm ... let me think about it.

NLEVELS: HoH To The Rescue

After some reflection, Mr. Hash figures out that if W hash object instances are needed for each individual "count distinct" processed in the same manner, they could be incorporated into a HoH and the program could proceed as follows:

- Create a HoH keyed by the number of H-variable
- Create W instances of a hash object H, each keyed by a numeric variable (for example, _N_ can be used, as it is auto-droppable), and store them in W items of HoH
- For each record from the input file:
  1. Loop through HoH varying _I_=1 to W
  2. At each iteration _I_, the H-instance #_I_ becomes active
  3. If the value of H-variable #_I_ is not in the (active) instance #_I_, add it
- After the file is done, the number of items in each H-instance is the needed count

- Looping again through HoH, collect the counts and output them in the vertical format

Translated into the SAS language strictly according to the plan above, Mr. Hash's program looked as follows:

```
%let W = 5 ; * N variables ;

data want_hoh (keep = nvar count) ;
  set have end = end ;
  array nn N1-N&W ;
  if _n_ = 1 then do ;
    dcl hash hh () ;
    hh.definekey ("_i_") ;
    hh.definedata ("h") ;
    hh.definedone () ;
    do over nn ;
      dcl hash h () ;
      h.definekey ("_n_") ;
      h.definedone () ;
      hh.add() ;
    end ;
  end ;
  do over nn ;
    hh.find() ;
    h.ref (key:nn, data:nn) ;
  end ;
  if end then do over nn ;
    Nvar = put (vname (nn), $8.) ;
    hh.find() ;
    Count = h.num_items ;
    output ;
  end ;
run ;
```

To test if it is sound, Mr. Hash first ran it against the smallish file provided by Mr. Stat, and the output figures did add up. Then to test for performance, Mr. Hash concocted a much larger input sample:

```
%let W =  1000 ; * variables ;
%let R = 10000 ; * records ;

data have ;
  call streaminit (1) ;
  array nn N1-N&w ;
  do _n_ = 1 to &r ;
    do over nn ;
      nn = rand ("integer", 1000 * _i_) ;
    end ;
    output ;
  end ;
run ;
```

This input got processed in about 20 seconds on Mr. Hash's garden variety laptop with the RAM footprint of about 800 MB. Evidently, processing 1 million records would take about 30 minutes, of which Mr. Stat said he would be happy with that. However, upon some reflection, Mr. Hash realized that at R=10000 the maximum number of distinct values for each H-variable - and thus the maximum hash memory footprint - might not yet have been reached, so if the program were run against the whole file, it could run out of RAM.

To forestall the potential problem, Mr. Hash decided to resort to the time-tested divide-and-conquer approach. Testing the program at W=100 and R=1000000 revealed that it ran in 2 minutes with the RAM footprint of 350 MB. Hence, he had the step rewritten in the following fashion:

- Read the input file with first 100 variables and generate the first 100 output records
- Void all the H-instances by calling the CLEAR method in a loop through HoH
- Repeat for the next 100 variables and so forth till all the N-variables are processed

This way, Mr. Stat's entire file could be processed in 20 minutes without ever exceeding 350 MB of RAM usage, and subsequent testing proved it to be true. Needless to say, Mr. Stat was quite happy with that. We offer it to a curious and adventurous reader to put this code together as an exercise in dynamic hash programming. Note: It must be done in a single DATA step.

More Information

Above, we have only scratched the HoH surface to illustrate the concept. For more information about its ins and outs and practical uses, see (Dorfman and Henderson, 2018; Loren, DeVenezia, 2011; Hinson, 2013).

CONCLUSION

SAS programmers first coming in contact with the SAS hash object - and even some of those who have been using it for a while - tend to underutilize its run-time dynamic capabilities. This is regrettable, for understanding them and making conscientious use of them makes hash code dynamic, flexible, robust, and easy to govern in a table-driven process without having to change the code itself. In this paper, we have made an attempt to somewhat rectify the situation by calling attention to the dynamic capabilities of the SAS hash object viewed at the following angles:

- Using the hash object's ability to grow-and-shrink at run time
- Using the arguments and argument tags as run-time expressions
- Using the ability of the hash object to store non-scalar identifiers of other hash objects and scan through them to implement array-like iterative processing

The authors hope that the material presented in this paper can help the programmers new to hash object embrace its dynamic nature from the outset. As for those already using the hash object productively but maybe underutilizing its dynamic capabilities, we hope that the paper can remind them of what they are missing and help them take a look at their hash code with a new set of eyes.

REFERENCES

1. Dorfman, Paul and Henderson, Don. 2018. *Data management Solutions Using SAS Hash Table Operations. A Business Intelligence Case Study*. Cary, NC: SAS Press.

2. Henderson, Donald J., Merry G. Rabb, Jeffrey A. Polzin. 1991. The SAS System Supervisor - A Version 6 Update. *Proceedings of the 1991 SUGI Conference*. New Orleans, LA.

3. Dorfman, Paul M., Henderson, Don. 2015. Data Aggregation Using the SAS® Hash Object. *Proceedings of the SAS Global 2015 Conference*. Dallas, TX.

4. Dorfman, Paul M., Vyverman, Koen. 2009. The DOW-Loop Unrolled. *Proceedings of the SAS Global 2009 Conference*. National Harbor, MD.

5. Loren, Judy, DeVenezia, Richard A. 2011. Building Provider Panels: An Application for the Hash of Hashes. *Proceedings of the SAS Global 2011 Conference*. Las Vegas, NV.
6. Hinson, Joseph. 2013. The Hash-of-Hashes as a "Russian Doll" Structure: An Example with XML Creation. *Proceedings of the SAS Global 2011 Conference*. San Francisco, CA.
7. Bian, Haikou, Maddox, David. 2014. More Hash: Some Unusual Uses of the SAS® Hash Object. *Proceedings of the SESUG 2014 Conference*. Myrtle Beach, SC.
8. Li, Arthur. 2014. Understanding and Applying the Logic of the DOW-Loop. *Proceedings of the PharmaSUG 2014 Conference.* San Diego, CA.
9. Allen, Richard Read. 2010. Practical Uses of the DOW Loop. *Proceedings of the WUSS 2010 Conference*. San Diego, CA.
10. Foty, Fuad J. 2012. The DOW-loop: a Smarter Approach to your Existing Code. *Proceedings of the SAS Global 2012 Conference.* Orlando, FL.
11. Hoyle, Larry. 2009. Implementing Stack and Queue Data Structures with SAS® Hash Objects. *Proceedings of the SAS Global 2009 Conference*. National Harbor, MD.

ACKNOWLEDGMENTS

The authors would like to thank Scott Bass for the invitation to present at SAS Global Forum 2020.

CONTACT INFORMATION

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

Paul M. Dorfman
Independent Consultant, Proprietor
4437 Summer Walk Ct
Jacksonville, FL 32258
Phone: (904) 226-0743
Email: sashole@gmail.com

Don Henderson
Henderson Consulting Services, LLC
3470 Olney-Laytonsville Road, Suite 199
Olney, MD 20832
Work phone: (301) 570-5530
Fax: (301) 576-3781
Email: Don.Henderson@hcsbi.com
Web: http://www.hcsbi.com
Blog: http://hcsbi.blogspot.com