PharmaSUG 2026 – Paper TT-417

A Fantasy in Three with SAS® PROC FCMP: Memoization of Resource-Intensive Calculations, in-Memory Hash Object Storage and Retrieval Operations, and Disk-Based Persistent Data Set Modification and Preservation

Troy Martin Hughes

ABSTRACT

The SAS® Function Compiler procedure (i.e., PROC FCMP) empowers SAS practitioners to design user-defined functions and subroutines that make software more maintainable, flexible, configurable, readable, and reusable. The SAS hash object, a built-in data structure and type of associative array, stores values in key-value pairs, and facilitates fast and efficient in-memory lookup operations that store and retrieve one or more values. Combining hash horsepower and FCMP finesse yields a function that is both powerful and pretty while abstracting (and hiding) complexity inside the function definition. Because of their key-value structure, hash objects commonly operationalize *memoization*—the key-based caching of resource-intensive results so they do not need to be recalculated in the future. Even superheroes have flaws, and two documented limitations of FCMP hash functionality are its inability to store SAS arrays and its inability to save hash objects natively to SAS data sets for persistent storage and retrieval. This text introduces novel (i.e., brilliant) approaches that enable hash objects to represent character and numeric arrays, and to export their records to SAS data sets when a DATA step terminates. This combination of memoization, in-memory lookup operations, and disk-based data set preservation maximizes speed and efficiency, and is adapted from the author's groundbreaking textbook: *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler, Second Edition.* (Hughes, 2026)

INTRODUCTION

The FCMP procedure can leverage three built-in data structures endemic to the SAS language—the ubiquitous SAS data set, the methods-based hash object, and the ever-elusive dictionary object—as well as one built-in *quasi* data structure—the SAS array, which SAS documentation goes to great lengths to caveat is *not* a true data structure. This text illustrates the functionality of (and relationships among) data sets, the hash object, and SAS arrays, and demonstrates their use with respect to the SAS function compiler procedure (i.e., PROC FCMP), hereinafter referenced as the *FCMP procedure*.

Memoization describes key-based caching that aims to store the results of resource-intensive computations so they can be retrieved from memory (rather than recalculated) in the future. In the SAS language, memoization is commonly facilitated through hash object key-value pairs. In a programming-language-agnostic sense, the value(s) returned via key lookup can contain various data types or data structures, such as scalars (e.g., integers, strings), composites (e.g., lists, arrays, sets), and more complex tabular, matrixed, or hierarchical data. However, memoization operationalized through the SAS hash object is inherently limited to the values that a hash object can maintain, which unfortunately includes only numeric scalars and character scalars. Thus, by definition, a hash object is unable to maintain numeric or character arrays. Boo!

Consider the seemingly straightforward use case that spurred this text—to calculate and memoize the prime factors in composite numbers and prime numbers—that quickly spiraled beyond the bounds of SAS built-in functionality. For example, a prime has exactly one prime factor—itself—so 7 is the prime factor of 7:

```
7 = 7 x 1
```

Every composite has at least two prime factors. For example, the prime factors of 6 are 2 and 3:

```
6 = 2 x 3
```

And most composites have more than two prime factors, including prime factors that are repeated. For example, the prime factors of 12 are 2, 2, and 3:

```
12 = 2 x 2 x 3
```

Thus, in mapping a series of prime factors (i.e., values) to an integer (i.e., key), and in further memoizing these key-value pairs that are calculated (to ensure that they are calculated only once), the stated use case requires the following functionality:

- The key-based data structure selected to maintain an integer and its prime factors must be able to scale to store a series of data (i.e., the prime factors).
- Efficiency constraints require that the data structure be stored *in memory* during lookup operations (to eliminate input/output overhead that would be required for repeated disk-based lookup).
- Any prime factors previously calculated in prior SAS sessions should be loaded into memory (from disk) only once, which requires both the ability to save prime factors to disk and to retrieve prime factors therefrom.

Memoization inherently requires key-based indexing and caching, and the only two key-based, built-in SAS data structures are the hash object and the dictionary object. Setting aside the dictionary, which has been observed in the wild on fewer occasions than a Thylacine, adoption of a hash-based solution is immediately met with several functional limitations of the SAS language:

- A hash object can maintain a fixed number of scalar values but cannot inherently maintain a scalable, composite value (such as a SAS array) that is required to hold a series of prime factors.
- Although a hash object can be initialized to the values maintained in a SAS data set, no hash method (or equivalent built-in functionality) exists to write a hash object to disk, which is required to preserve prime factors calculated during the *current* SAS session for *future* SAS sessions.
- A hash object instantiated inside a user-defined function can be accessed only via a call to that specific function, so conceptualizing various hash-based operations (e.g., reading prime factors from disk, evaluating if a number has been factored previously, calculating prime factors, and saving prime factors to disk) requires each operation to be defined inside the same function.

This text overcomes these limitations, and effectively extends the functionality of the hash object with respect to the FCMP procedure. Techniques and workarounds are introduced independently, after which a combined solution demonstrates the prime factorization functionality that is sought.

TRICKING A SAS HASH OBJECT INTO MAINTAINING A NUMERIC ARRAY

Whether instantiated inside the DATA step or the FCMP procedure, a hash object can maintain only numeric or character keys and values—by design and by definition. For example, if tasked only with maintaining (not calculating) the prime factors of 12, the Factor_12 function struggles because independent variables (Factor1, Factor2, and Factor3) must be separately defined to maintain each factor:

```
proc fcmp outlib=work.funcs.maths;
   function factor_12();
      length num factor1-factor3 8;
      declare hash h();
      rc = h.defineKey('num');
      rc = h.defineData('factor1', 'factor2', 'factor2');
      rc = h.defineDone();
      num = 12;
      factor1 = 2;
      factor2 = 2;
      factor3 = 3;
      rc = h.add();
      return(.);
      endfunc;
quit;
```

That is, whereas the shortcut `factor1-factor3` can be used in the LENGTH statement to declare multiple variables (without the need to specify each one individually), no similar shortcut exists to define associated keys or values using the DEFINEKEY and DEFINEDATA hash methods, respectively. Note that the ADD hash method adds the Num key (12) to the hash object, as well as its associated values (2, 2, and 3). RETURN returns only a missing value, as this inchoate function is not intended to be called.

The first step toward redemption is to recognize that both hash keys and hash values can be declared *incrementally*. That is, a subsequent DEFINEDATA hash method call does not overwrite any hash values previously declared through prior DEFINEDATA calls, but rather appends one or more values to the list of current hash values. Thus, the following functionally equivalent function iteratively (and dynamically) declares Factor1, Factor2, and Factor 3 by referencing the Cnt variable:

```
proc fcmp outlib=work.funcs.maths;
   function factor_12();
      length num factor1-factor3 8;
      declare hash h();
      rc = h.defineKey('num');
      do cnt=1 to 3;
         rc = h.defineData('factor' || strip(put(cnt,8.)));
         end;
      rc = h.defineDone();
      num = 12;
      factor1 = 2;
      factor2 = 2;
      factor3 = 3;
      rc = h.add();
      return(.);
      endfunc;
quit;
```

The next step is to recognize that rather than declaring Factor variables (e.g., Factor1, Factor2, and Factor3) using the LENGTH statement, the ARRAY statement can declare these variables while also containerizing the variable names for added convenience, scalability, and readability. For example, the following functionally equivalent function now incrementally declares hash values using the Factor array:

```
proc fcmp outlib=work.funcs.maths;
   function factor_12();
      length num 8;
      array factor[10];
      declare hash h();
      rc = h.defineKey('num');
      do cnt=1 to dim(factor);
         rc = h.defineData('factor' || strip(put(cnt,8.)));
         end;
      rc = h.defineDone();
      num = 12;
      factor1 = 2;
      factor2 = 2;
      factor3 = 3;
      rc = h.add();
      return(.);
      endfunc;
quit;
```

To be clear, DEFINEDATA does not directly maintain the Factor array; however, by collectively referencing the static “factor” character literal and dynamic Cnt variable, the loop effectively adds the variables that correspond to the Factor array (e.g., Factor1 to Factor10, now expanded to support greater scalability) to the hash object declaration. Thus, the hash object is tricked into maintaining the Factor numeric array, and equivalent functionality (not shown) supports tricking hash objects into maintaining character arrays.

The benefits of representing hash values as an array are myriad, and are better exposed when the hash object is instantiated more dynamically from disk. Also exposed, unfortunately, is an undocumented defect at the nexus of FCMP hash objects and arrays. Thus, although the prior two Factor_12 functions will each compile and execute without runtime error or failure, a sinister bug lurks within that must be squashed.

To begin, a one-observation data set (Factors) is created that holds an integer (Num), and its prime factors in the variables Factor1, Factor2, and Factor3:

```
data factors;
   length num factor1-factor10 8;
   call missing(of factor:);
```

```
   num = 12;
   factor1 = 2;
   factor2 = 2;
   factor3 = 3;
run;
```

The Factors data set simulates integers and their associated prime factors that will be calculated by a user-defined function (defined subsequently), and saved for posterity within a SAS data set.

The Factorize user-defined function is defined, which still does not calculate prime factors, but rather retrieves previously calculated prime factors from the Factors data set by leveraging the DATASET argument of the DECLARE HASH statement:

```
proc fcmp outlib=work.funcs.maths;
   function factorize();
      file log;
      length num 8;
      array factor[10];
      declare hash h(dataset: 'factors');
      rc = h.defineKey('num');
      do cnt=1 to dim(factor);
         rc = h.defineData('factor' || strip(put(cnt,8.)));
         end;
      rc = h.defineDone();
      num=12;
      rc_find = h.find();
      put factor=;
      return(.);
      endfunc;
quit;
```

After the DEFINEDONE hash method completes the instantiation (and initialization) of the hash object (H), Num is initialized to 12, and the FIND hash method searches the hash object for a key value of 12. Because this key exists in the hash object (as ingested from the Factors data set), the FIND method *should* return a value of 0, indicating that the key was identified in the hash object.

The CMPLIB options statement instructs SAS where to find the compiled Factorize function, and the DATA step calls Factorize:

```
options cmplib=work.funcs;

data _null_;
   null = factorize();
run;
```

The DATA step fails, however, and the log erroneously spouts lies about Factor1 not being defined in the program:

```
ERROR: Variable 'factor1' used in the definekey/definedata method of HASH object 'h' is not
       defined in the program.
ERROR: Error reported in function 'DEFINEDATA_HASH_' in statement number 5 at line 11 column
11.
       The statement was:
    0     (11:11)    rc = DEFINEDATA_HASH_( "factor" || STRIP( PUT( cnt=1, 8.(26841600) ) ) )
factor[1]=. factor[2]=. factor[3]=. factor[4]=. factor[5]=. factor[6]=. factor[7]=. factor[8]=.
factor[9]=. factor[10]=.
```

At issue, a pernicious defect in the FCMP procedure causes variables declared by the ARRAY statement to be available only after at least one of the array variables has been referenced by variable name or by array index. Thus, although the Factor1 variable is referenced dynamically (in which the Cnt counter is appended to the “factor” variable name base), because neither “Factor1” (the array variable name) nor “Factor[1]” (the array index) is ever referenced in the code, execution disastrously fails.

Even if the “Factor1” variable is specified statically by the DEFINEDATA method, because the variable name is referenced as a character variable (rather than a variable name), the defect persists:

```
if cnt=1 then rc = h.defineData('factor1');
```

Thus, the following function successfully compiles but fails with an identical runtime error during execution due to the same SAS defect—despite the three variable names being statically referenced (albeit in quotes):

```
proc fcmp outlib=work.funcs.maths;
   function factorize();
      file log;
      length num 8;
      array factor[10];
      declare hash h(dataset: 'factors');
      rc = h.defineKey('num');
      do cnt=1 to 3;
         if cnt=1 then rc = h.defineData('factor1');
         else if cnt=2 then rc = h.defineData('factor2');
         else if cnt=3 then rc = h.defineData('factor3');
         end;
      rc = h.defineDone();
      num = 12;
      rc_find = h.find();
      put factor=;
      return(.);
      endfunc;
quit;

data _null_;
   null = factorize();
run;
```

Changing one line of code to reference the Factor1 variable name (rather than the Factor array) does the trick, and replacing the prior PUT statement with the following statement results in success:

```
put factor1=;
```

And although the runtime error message is referencing a failure to recognize the Factor1 variable, in truth, any of the array variables can be referenced in the code to overcome this defect, so any of the following individual statements will equivalently cause the DATA step to succeed:

```
put factor1=;
put factor[1]=;
put factor2=
put factor[2]=;
```

Oftentimes, SAS practitioners will not have reason to write comments to the SAS log in production software, so a conditional `IF 0` can preface the PUT statement so it never executes:

```
if 0 then put factor1=;
```

The revised function definition and function invocation follow:

```
proc fcmp outlib=work.funcs.maths;
   function factorize();
      file log;
      length num 8;
      array factor[10];
      declare hash h(dataset: 'factors');
      rc = h.defineKey('num');
      do cnt=1 to dim(factor);
         rc = h.defineData('factor' || strip(put(cnt,8.)));
         end;
      rc = h.defineDone();
      num = 12;
      rc_find = h.find();
      if 0 then put factor1=;
      put factor=;
      return(.);
      endfunc;
```

```
quit;

data _null_;
   null = factorize();
run;
```

The conditional PUT statement does not execute, but the second PUT statement prints the Factor array, and the log demonstrates the three prime factors of 12:

```
factor[1]=2 factor[2]=2 factor[3]=3 factor[4]=. factor[5]=. factor[6]=. factor[7]=.
factor[8]=. factor[9]=. factor[10]=.
```

This defect affects only the FCMP procedure and note the DATA step, as demonstrated by the successful execution of the following DATA step without any static reference to the Factor array variable names or its indexed variables:

```
data _null_;
   length num 8;
   array factor[10];
   if _n_=1 then do;
      declare hash h(dataset: 'factors');
      rc = h.defineKey('num');
      do cnt=1 to dim(factor);
         rc = h.defineData('factor' || strip(put(cnt,8.)));
         end;
      rc = h.defineDone();
      end;
   num = 12;
   rc_find = h.find();
   put rc_find=;
run;
```

The log demonstrates that the FIND hash method returns 0, correctly evaluating that 12 appears as a key in the hash object (H):

```
rc_find=0
```

Faulty SAS syntax aside, the benefits of being able to reference hash values as an array are many, and it allows individual values to be iterated separately or treated as a containerized whole. For example, in the next section, the WRITE_ARRAY built-in function is leveraged to write a numeric array to a data set, and this workaround overcomes the oversight in which SAS Institute provided no built-in methods that export hash objects to data sets from the FCMP procedure.

TRICKING WRITE_ARRAY INTO SAVING A NUMERIC HASH OBJECT TO A DATA SET

The WRITE_ARRAY built-in function is supported only by the FCMP procedure and writes a two-dimensional numeric array to a SAS data set; character arrays, inexplicably and indefensibly, are not supported by WRITE_ARRAY. This is why we can't have nice things!

The first parameter of WRITE_ARRAY specifies the data set name (in quotes); the second parameter specifies the array name; and subsequent (optional) parameters define variable names in the output data set. For example, the following call to WRITE_ARRAY writes the Arr_final array to the Factors data set.

```
 rc_write = write_array('factors', arr_final,
    'num', 'factor1', 'factor2', 'factor3', 'factor4', 'factor5',
    'factor6', 'factor7', 'factor8', 'factor9', 'factor10');
```

As demonstrated previously, a hash object can be tricked into containing a numeric array. Thus, all that is yet required is a process that exports a hash object (containing all numeric data) into a two-dimensional numeric array, in which the hash key occupies the first position in the array and the hash values occupy the subsequent array positions, with each subsequent hash record maintained in a new array row.

The following code snippet iteratively traverses the hash object and exports one record at a time into a dynamic (i.e., resizable) array:

```
hash_size = h.num_items();
array arr_final[1] / nosymbols;
call dynamic_array(arr_final, hash_size, 11);
declare hiter iter('h');
do while(iter.next()=0);
   hash_row+1;
   arr_final[hash_row, 1] = num;
   do cnt=1 to 10;
      arr_final[hash_row, cnt+1] = factor[cnt];
      end;
   end;
rc_write = write_array('factors', arr_final,
   'num', 'factor1', 'factor2', 'factor3', 'factor4', 'factor5',
   'factor6', 'factor7', 'factor8', 'factor9', 'factor10');
```

The NOSYMBOLS option is supported only by the FCMP procedure, and declares a dynamic array that can be subsequently resized; for this reason, the initial size of the array can be defined as 1 because additional array elements—and, in fact, array dimensions—can be added later, as needed:

```
array arr_final[1] / nosymbols;
call dynamic_array(arr_final, hash_size, 11);
```

The DECLARE HITER statement declares a hash iterator object, which facilitates the manual traversal of the hash object (H). The ITER hash iterator method evaluates to 0 until the last hash record is reached, at which point the DO WHILE loop terminates:

```
declare hiter iter('h');
do while(iter.next()=0);
```

Within the DO WHILE loop, the Hash_row variable specifies the hash object record number that is used to initialize array values. Note that the first array element in each array row is initialized to the Num variable—the integer being factored:

```
   hash_row+1;
   arr_final[hash_row, 1] = num;
```

Thereafter, the next ten elements of the array row are initialized to the ten hash object values in the current record:

```
   do cnt=1 to 10;
   arr_final[hash_row, cnt+1] = factor[cnt];
   end;
end;
```

The final step, as demonstrated previously, is the WRITE_ARRAY statement that exports the Arr_final array and saves it as the Factors data set after the DO WHILE loop terminates. At this point, a hash object has been effectively exported from a user-defined function into a data set—functionally that is saliently absent from the catalog of tools that SAS provides its developers.

TRICKING A HASH OBJECT INTO COLLECTIVE USE BY MULTIPLE FUNCTIONS

The hash object is an inarguably invaluable in-memory data structure whose reputation should not be sullied. All hail the sages, Paul Dorfman and Don Henderson, who elucidated the mysteries of SAS hash to the masses in their timeless tome: *Data Management Solutions Using SAS® Hash Table Operations: A Business Intelligence Case Study*. (Paul Dorfman, 2018) My signed, worn, well-loved copy abuts my bed and I consult it daily.

Praise aside, SAS hash does have several limitations, one of which is evinced when a hash object is instantiated inside a user-defined function (or subroutine)—only that specific function can access the in-memory hash object. Thus, even if two identically named hash objects are instantiated by separate functions and those functions are called within the same DATA step, those identically named hash objects will nevertheless reference two independent in-memory data structures. This limitation does restrict function design, but is not insurmountable, and *wrapper functions* can save the day—functions that operate at an

interstitial level of procedural abstraction and which collectively call the same primary (core) function to perform disparate functionality based on the wrapper function that was called.

For example, at least three functional objectives are conceptualized in this use case—the ability to read an existing data set that contains integers and their prime factors; the ability to compute prime factors for integers not yet encountered; and the ability to save integers and their prime factors to an ever-growing data set. This variety of functionality could be placed inside a single function (in violation of the principle that functions should be functionally discrete and do *one and only one thing*), although the inability of SAS user-defined functions to declare optional parameters would require that every function call pass every parameter irrespective of the requested functionality. Meh! Function wrappers instead provide a simpler alternative by calling a core function indirectly.

The PRIME_FACTOR user-defined subroutine is a core function intended to be called only from wrapper functions. It declares an ACTION parameter that denotes the functionality being requested by the wrapper function. When ACTION is `init`, the hash object is initialized by ingesting the Factors data set. When ACTION is `factor`, the hash object is searched for the integer being evaluated—if the integer is found, the precalculated prime factors are retrieved from the hash object, and if the integer is not found, the prime factors are calculated. When ACTION is `save`, the hash object is exported to a data set:

```
proc fcmp outlib=work.funcs.maths;
   subroutine prime_factor(action $, num, arr_div[*]);
      outargs arr_div;
      *initialize data set and hash object;
      if action = 'init' then do;
         length var $15;
         array factor[10];
         declare hash h(dataset: 'factors');
         rc = h.defineKey('num');
         rc = h.defineData('num');
         do cnt=1 to dim(arr_div);
            rc = h.defineData(catx('', 'factor', strip(put(cnt,3.))));
            end;
         rc = h.defineDone();
         end;

      * calculate prime factors;
      else if action = 'factor' then do;
         mod_num = num;
         pos = 0;
         div = 1;
         rc_find = h.find();
         if rc_find=0 then put 'hash: ' @8 num= @20 factor;
         else do;
            do while(div < mod_num);
               div+1;
               if mod(mod_num, div) = 0 then do;
                  pos+1;
                  factor[pos] = div;
                  mod_num = mod_num/div;
                  div = div-1;
                  end;
               end;
            rc = h.add();
            put 'calc: ' @8 num= @20 factor;
            end;
         do cnt=1 to dim(arr_div);
            arr_div[cnt] = factor[cnt];
            end;
         end;

      * write hash object to data set;
      else if action = 'save' then do;
         hash_size = h.num_items();
         array arr_final[1] / nosymbols;
         call dynamic_array(arr_final, hash_size, 11);
```

```
         declare hiter iter('h');
         do while(iter.next()=0);
            hash_row+1;
            arr_final[hash_row, 1] = num;
            do cnt=1 to 10;
               arr_final[hash_row, cnt+1] = factor[cnt];
               end;
            end;
         rc_write = write_array('factors', arr_final,
            'num', 'factor1', 'factor2', 'factor3', 'factor4', 'factor5',
            'factor6', 'factor7', 'factor8', 'factor9', 'factor10');
         end;
      endsub;
quit;
```

The hash object initialization functionality has already been described as well as the hash object export functionality. The remaining factorization functionality first leverages the FIND hash method to evaluate whether the integer already appears in the hash object, and prints “hash” to the log when the memoized value is retrieved rather than calculated. When FIND does not locate the integer, the DO WHILE loop iteratively calculates all prime factors of the integer, after which the ADD hash method appends the new key-value pair to the hash object and prints “calc” to the log.

PRIME_FACTOR is not designed to be called directly (e.g., from a DATA step) but rather from three function wrappers—the PRIME_FACTOR_INIT function, the PRIME_FACTOR_CALC subroutine, and the PRIME_FACTOR_SAVE subroutine—that prepare input before passing arguments to PRIME_FACTOR. Only in the case of PRIME_FACTOR_CALC, the list of prime factors is also made available to the calling program via the OUTARGS statement, which specifies that the array parameter is passed *by reference*.

The PRIME_FACTOR_INIT function creates the Factors data set if it does not exist, after which it calls PRIME_FACTOR to instantiate the hash object, an operation that occurs only once per DATA step. If the Factors data set does not exist, the RUN_MACRO function calls the PRIME_FACTOR_INIT macro in which the data set is created:

```
%macro prime_factor_init();
data factors;
   length num 8 factor1-factor10 8;
   stop;
run;
%mend;

/*
function returns:
 - 0 if data set is created (not found)
 - 1 if data set already exists
*/
proc fcmp outlib=work.funcs.maths;
   function prime_factor_init();
      if exist('factors') = 1 then rc_init = 1;
      else do;
         rc_init=0;
         rc = run_macro('prime_factor_init');
         end;
      action = 'init';
      num = .;
      array arr_div[10];
      call prime_factor(action, num, arr_div);
      return(rc_init);
      endfunc;
quit;
```

Because the PRIME_FACTOR subroutine declares three parameters, all three parameters always must be passed by each of the wrapper functions—even if that wrapper function does not require the parameter. Thus, the initialization of the Factors data set (if necessary) and subsequent instantiation of the hash object (H) do not require the NUM or ARR_DIV parameters; however, dummy arguments nevertheless must be

passed to PRIME_FACTOR. For this reason, Num is initialized to a numeric missing value, and the Arr_div array is initialized to a ten-element numeric array that contains no values. Ten elements are required because the dimensionality of this array (as evaluated by the DIM function inside PRIME_FACTOR) determines the number of hash values that are declared using the DEFINEDATA hash method.

PRIME_FACTOR_CALC is a wrapper subroutine that calculates the prime factors of an integer, or which retrieves the prime factors from memory if they have been previously calculated. However, in truth, this functionality is performed inside the PRIME_FACTOR subroutine, and PRIME_FACTOR_CALC is only the pass-through that sends arguments to PRIME_FACTOR:

```
proc fcmp outlib=work.funcs.maths;
   subroutine prime_factor_calc(num, arr_div[*]);
      outargs arr_div;
      action = 'factor';
      call prime_factor(action, num, arr_div);
      endsub;
quit;
```

PRIME_FACTOR_SAVE is the final wrapper subroutine, which calls PRIME_FACTOR to export the contents of the hash object (H) to a data set:

```
proc fcmp outlib=work.funcs.maths;
   subroutine prime_factor_save();
      action = 'save';
      num = .;
      array arr_div[1];
      call prime_factor(action, num, arr_div);
      endsub;
quit;
```

CALCULATING, MEMOIZING, AND SAVING PRIME FACTORS

A primary benefit of this modular software design paradigm is the elimination of unnecessary arguments in function calls. For example, the PRIME_FACTOR_INIT function requires no parameters to initialize the Factors data set and instantiate the hash object, so its function call is simplified:

```
rc = prime_factor_init();
```

Without this added level of abstraction (e.g., if PRIME_FACTOR were called directly from the DATA step), meaningless placeholder values would have had to be passed in the DATA step function call. Instead, although more wrapper functions are now required, each of their calls is simplified.

The following DATA step creates sample integers to be factored:

```
data some_nums;
   do num = 7, 12, 1083, 14079, 14079;
      output;
      end;
run;
```

Note that 14,079 is listed twice, so the first time the number is encountered, its prime factors will be calculated; however, during subsequent encounters, the prime factors will be retrieved from the hash object, and this retrieval of cached values can result in dramatic efficiency and performance gains where calculations are redundant or resource intensive or both.

The following DATA step calls PRIME_FACTOR_INIT to create the Factors data set and instantiate the hash object; it calls PRIME_FACTOR_CALC to factor all integers; and it calls PRIME_FACTOR_SAVE once all observations have been read to export the hash object to the Factors data set:

```
data primes;
   set some_nums end=eof;
   array factor [10];
   if _n_=1 then rc=prime_factor_init();
   call prime_factor_calc(num, factor);
   put @9 num= @21 factor[*];
```

```
   if eof then call prime_factor_save();
run;
```

The log demonstrates that the first four integers were factored, whereas the prime factors for the final observation were instead retrieved from the hash object (demonstrating *in-memory* memoization):

```
calc:   num=7       7 . . . . . . . . .
        num=7       7 . . . . . . . . .
calc:   num=12      2 2 3 . . . . . . .
        num=12      2 2 3 . . . . . . .
calc:   num=1083    3 19 19 . . . . . . .
        num=1083    3 19 19 . . . . . . .
calc:   num=14079   3 13 19 19 . . . . . .
        num=14079   3 13 19 19 . . . . . .
hash:   num=14079   3 13 19 19 . . . . . .
        num=14079   3 13 19 19 . . . . . .
NOTE: There were 5 observations read from the data set WORK.SOME_NUMS.
```

And the real magic occurs when the DATA step is run a second time. Because the Factors data set has now been created, which contains the prime factors of all four unique integers, each of the prime factors is retrieved from memory (rather than recalculated), as shown in the log (demonstrating *disk-based* memoization for *all* observations):

```
hash:   num=7       7 . . . . . . . . .
        num=7       7 . . . . . . . . .
hash:   num=12      2 2 3 . . . . . . .
        num=12      2 2 3 . . . . . . .
hash:   num=1083    3 19 19 . . . . . . .
        num=1083    3 19 19 . . . . . . .
hash:   num=14079   3 13 19 19 . . . . . .
        num=14079   3 13 19 19 . . . . . .
hash:   num=14079   3 13 19 19 . . . . . .
        num=14079   3 13 19 19 . . . . . .
NOTE: There were 5 observations read from the data set WORK.SOME_NUMS.
```

With this advancement, the hash object has now been effectively persisted from one DATA step to another, owing to the hash object having been exported to a data set (Factors) and subsequently imported from that data set when prime factors again were needed. These data preservation mechanics, never before demonstrated by an FCMP hash object, now ensure that incremental calculations made toward complex or resource-intensive operations are not lost and can be preserved for continuation of the composite work.

CONCLUSION

Despite its power and its necessity in the toolbelt of any serious SAS practitioner, the SAS hash object nevertheless suffers several inadequacies. SAS hash cannot contain arrays. SAS hash cannot be exported from user-defined functions. And SAS hash cannot be shared among different user-defined functions and subroutines. This text demonstrated workarounds that overcome each of these limitations, and which collectively facilitate fancy hash with fuller functionality.

REFERENCES

Hughes, T. M. (2026). *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler, Second Edition.* San Diego, CA.

Paul Dorfman, D. H. (2018). *Data Management Solutions Using SAS® Hash Table Operations: A Business Intelligence Case Study.* Cary, NC: SAS Press.

CONTACT INFORMATION

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

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com