PharmaSUG 2024 – Paper SD-444

Five Reasons To Swipe Right on PROC FCMP, the SAS® Function Compiler for Building Modular, Maintainable, Readable, Reusable, Flexible, Configurable User-Defined Functions and Subroutines

Troy Martin Hughes

ABSTRACT

The FCMP procedure (aka, the SAS® function compiler) empowers SAS practitioners to build our own user-defined functions and subroutines—callable software modules that containerize discrete functionality, and which effectively extend the Base SAS programming language. This introduction explores five high-level problem sets that user-defined functions can solve. Learn how to hide a hash object (and its complexity) inside a function, how to manipulate SAS arrays, how to design a format (or informat) that calls a function, how to run a DATA step (or SAS procedure) inside of a DATA step (aka, DPDD), and how to avoid unnecessary usage of the SAS macro language. Interwoven throughout the discussion are the specific software quality characteristics—such as modularity, maintainability, readability, reusability, and configurability—that are achieved through the design and implementation of FCMP user-defined functions. For additional information, context, and examples, please consult that author's 2024 SAS Press book: *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler.* (Hughes, 2024)

INTRODUCTION

SAS *user-defined functions* and subroutines (referenced collectively in this text as "functions") are defined inside the FCMP procedure, and compiled when FCMP executes. Thereafter, the named function can be called in the same manner as SAS *built-in functions*—those provided as part of out-of-the-box Base SAS functionality. Thus, each new user-defined function that is developed, tested, and released into production represents a new building block that can be implemented by diverse users, and which supports diverse use cases in future software products and projects.

This concise text in no way intends to introduce FCMP syntax. Rather, it demonstrates several use cases solved through FCMP user-defined functions. To this end, examples compare functionally equivalent solutions—first without user-defined functions, and subsequently with user-defined functions. The conspicuous benefits of user-defined functions are showcased, including the specific software quality characteristics that user-defined functions imbue.

Software modularity describes the extent to which software is cleaved into bite-sized chunks, the aim of which is to enable one software module to be modified without adversely (or inadvertently) affecting any other software modules. Modular software is contrasted with monolithic software, in which extensive code is (unfortunately) commingled within one program file. User-defined functions support modularity because a function's functionality is encapsulated within the FCMP procedure.

Software maintainability describes the extent to which software can be modified readily when needed—whether to support scheduled maintenance that alters functionality or improves performance, or emergency maintenance that corrects software defects, or which restores functionality after compromised software availability. User-defined functions support maintainability because function definitions (encapsulated within the FCMP procedure) can be modified independently of the programs calling these functions.

Software readability describes the extent to which software can be readily understood, including both its syntax and accompanying internal comments. User-defined functions improve the readability of *functions themselves* because functionality is discrete and containerized. Moreover, user-defined functions improve the readability of the *programs calling functions* because any functionality contained inside a function can be inherently removed from the program calling that function.

Software reusability describes the extent to which software can be reused in future software products or projects. Reuse can benefit the original function developer or—where the function is shared with teammates, within an organization, or disseminated through publication—reuse can benefit others. User-defined functions support reusability because their modularity, conciseness, flexibility, and singular

functionality collectively yield a level of abstraction that can be pliably implemented by various users to support various use cases and various data.

Software configurability describes the extent to which software can be configured by end users—that is, the extent to which users can supply varied input (arguments) to effect varied output (return values, return codes, or other outcomes). End users will often describe a highly *configurable* function as being highly *flexible* because it can be used to support various software products or projects—and their varied data. User-defined functions support configurability because they can declare character and numeric scalar parameters, as well as non-scalar parameters such as SAS arrays. And in this regard, reusable FCMP functions can often far surpass the functionality of SAS macros engineered for a similar purpose.

With even this scant introduction to software quality characteristics, the high-level benefits of user-defined functions begin to become apparent. Thus, although this text enumerates five low-level problem sets solved by FCMP user-defined functions, in each case, higher-level aspects of software quality are commensurately and inarguably improved. For a more comprehensive view of software quality, the author's text provides a 600-page introduction: *SAS® Data Analytic Development: Dimensions of Software Quality*. (Hughes, 2016)

1. HIDE YOUR HASH!

Consider the requirement to validate categorical data—for example, to determine whether a U.S. state abbreviation contained within a transactional data set is valid. This lookup operation effectively evaluates *membership*—whether one value is a member of a set of master values.

The following master data represent the enumeration of all 50 valid state abbreviations.

```
data state_abbrev;
   infile datalines;
   length st $2;
   input st $ @@;
   datalines;
AK AL AR AZ CA CO CT DE FL GA HI IA ID IL IN KS KY LA MA MD ME MI MN MO MS MT NC ND
NE NH NJ NM NV NY OH OK OR PA RI SC SD TN TX UT VA VT WA WI WV WY
;
```

The following transactional data represent state abbreviations that need to be validated, with the District of Columbia and the Virgin Islands representing invalid data.

```
data possible_states;
   infile datalines;
   length state_abbr $2;
   input state_abbr;
   datalines;
CA
MS
CA
AL
AL
DC
VI
;
```

The following DATA step leverages the hash object and the hash CHECK method to evaluate membership.

```
data validated (drop=st);
   set possible_states;
```

```
   length st $2;
   if _n_=1 then do;
      declare hash h(dataset: 'state_abbrev');
      rc = h.definekey('st');
      call missing(st);
      rc = h.definedone();
      end;
   rc = h.check(key: state_abbr);
run;
```

The CHECK hash method returns a 0 when the state abbreviation is found in the State_abbrev master table, and a non-zero value (e.g., 160038) when the state abbreviation is not found—that is, when the value is invalid. The Validated data set denotes that neither DC nor VI are valid state abbreviations (per the master table).

	state_abbr	rc
1	CA	0
2	MS	0
3	CA	0
4	AL	0
5	AL	0
6	DC	160038
7	VI	160038

But what if this lookup functionality is required elsewhere—for example, in another program being developed by another SAS practitioner to analyze different data? In this scenario, the entire hash object would need to be recreated in the new DATA step.

A more modular, reusable solution would instead declare and evaluate the hash object inside a user-defined function. Thereafter, an associated function call—a single line of code—could be used to provide equivalent functionality to the preceding (and more convoluted) DATA step.

The following FCMP procedure defines the VALIDATE_STATE function, which returns 0 for valid state abbreviations and a non-zero value for invalid state abbreviations.

```
proc fcmp outlib=work.funcs.lookup;
   function validate_state(st $);
      length st $2;
      declare hash h(dataset: 'state_abbrev');
      rc = h.definekey('st');
      rc = h.definedone();
      return(h.check());
      endfunc;
quit;
```

Because the hash complexity is encapsulated inside the function's definition, the VALIDATE_STATE user-defined function can now be called using a single line of code.

```
options cmplib=work.funcs;
data validated_fcmp;
   set possible_states;
```

```
   length rc 8;
   rc=validate_state(state_abbr);
run;
```

As demonstrated previously, the Validated_FCMP data set demonstrates that the first five observations contain valid state abbreviations, whereas the last two observations (VI and DC) are invalid. Moreover, this modular user-defined function is more readable and reusable than the functionally equivalent use of the hash object directly in the DATA step. The author's text provides more comprehensive usage of the hash object to validate, clean, and standardize data: *SAS® Data-Driven Development: From Abstract Design to Dynamic Functionality*. (Hughes, 2023)

2. MANIPULATE ARRAYS

Consider the requirement to manipulate an array within a DATA step. For example, given an array that contains a list of superhero superpowers, how would you add a new superpower to the first empty array element? In stack theory, this operation is known as *pushing a stack*, and it can be accomplished directly (in the DATA step) or indirectly (facilitated by a user-defined function).

The following DATA step creates the Superpowers data set.

```
data superpowers;
   infile datalines dsd delimiter=',' truncover;
   length hero $32 power1 power2 power3 power4 power5 $20;
   input hero $ power1 $ power2 $ power3 $ power4 $ power5 $;
   datalines;
Thor,electricity,flight,longevity,weather manipulation
Doctor Strange,levitation,telekinesis,teleportation,time manipulation,magic
Thanos,telekinesis,gravity manipulation,teleportation,time manipulation
Wanda Maximoff,telepathy,mental manipulation
;
```

The final observation indicates that Wanda has two superpowers—telepathy and mental manipulation. However, "healing" could be added as a third superpower, and this scenario describes the need to push "healing" to Wanda's stack of superpowers. And SAS arrays can operationalize this functionality.

The following DATA step declares the Powers array, which references five variables—Power1 through Power5. For Wanda, only the first two array elements contain data, so the WHICHC function evaluates that the third array element is the first empty element. Thereafter, "healing" is added to the third array element (i.e., Power5).

```
data push;
   set superpowers (where=(hero='Wanda Maximoff'));
   array powers power1-power5;
   put powers[*]=;
   * calculate first empty position;
   emp = whichc('', of powers[*]);
   * push new superpower to stack;
   if emp ^= 0 then do;
      powers[emp] = 'healing';
      put powers[*]=;
      end;
run;
```

The log indicates that “healing” has been added to the Powers array.

```
power1=telepathy power2=mental manipulation power3=  power4=  power5=
power1=telepathy power2=mental manipulation power3=healing power4=  power5=
NOTE: There were 1 observations read from the data set WORK.SUPERPOWERS.
      WHERE hero='Wanda Maximoff';
```

However, a more concise, reusable solution would instead define a user-defined function to effect the same functionality.

At first glance, the following FCMP function would appear to replicate the push functionality.

```
proc fcmp outlib=work.funcs.stacks;
   function push_stack_char(arr[*] $, char_val $);
      outargs arr;
      emp = whichc('', of arr[*]);
      if emp ^= 0 then do;
         arr[emp] = char_val;
         return(1);
         end;
      else return(0);
      endfunc;
quit;
```

However, as described in a previous text by the author, the OF operator does not function in the FCMP procedure, and the log demonstrates this failure. (Hughes, 2023)

```
728  proc fcmp outlib=work.funcs.stacks;
729     function push_stack_char(arr[*] $, char_val $);
730        outargs arr;
731        emp = whichc('', of arr[*]);
ERROR: The OF operator isn't allowed on an ARRAY with a dynamic size.
732        if emp ^= 0 then do;
733           arr[emp] = char_val;
734           return(1);
735           end;
736        else return(0);
737        endfunc;
738  quit;

NOTE: The SAS System stopped processing this step because of errors.
```

Thus, the OF operator and the WHICHC function cannot be utilized so a DO loop instead identifies the first empty array element. Note that the return value is initialized to 1 when the function succeeds (i.e., the CHAR_VAL parameter can be added to the array), and it is initialized to 0 when the function fails (i.e., the array is already full so the CHAR_VAL parameter cannot be added).

```
proc fcmp outlib=work.funcs.stacks;
   function push_stack_char(arr[*] $, char_val $);
      outargs arr;
      do emp=1 to dim(arr);
         if arr[emp]='' then leave;
```

```
         end;
      if emp <= dim(arr) then do;
         arr[emp] = char_val;
         return(1);
         end;
      else return(0);
      endfunc;
quit;
```

And the following DATA step calls the PUSH_STACK_CHAR user-defined function to add “healing” to Wanda’s list of superpowers—specifically, to the third array element, Power3.

```
options cmplib=work.funcs;
data push_fcmp;
   set superpowers (where=(hero='Wanda Maximoff'));
   array powers power1-power5;
   put powers[*]=;
   rc = push_stack_char(powers, 'healing');
   put powers[*]=;
run;
```

Thus, the user-defined function delivers identical functionality but it can now be called to add *any* character value to *any* character array—far more flexible functionality than the original version that did not employ FCMP. Moreover, the implementation of a return code now signifies when an array is full and it cannot be pushed. For example, because Doctor Strange already has five superpowers, his Powers array is full, so calling PUSH_STACK_CHAR on Doctor Strange will yield a return code of 0 that indicates no new powers can be added to his stack.

3. APPLY A FORMAT/INFORMAT THAT CALLS A FUNCTION

The FCMP procedure has yet another trick up its sleeve—the ability to define a user-defined function that is subsequently called by a user-defined format. That is, a format (or informat) can be defined that calls a user-defined function to deliver the function’s functionality when the format is applied. A benefit to this indirect function call is the ability to apply functionality anywhere a format can be applied—including inside SAS procedures like PRINT, MEANS, and FREQ. This methodology is fully described in the author’s text. (Hughes, 2023)

Consider the requirement not only to validate state abbreviations but also to transform valid state abbreviations into their associated state names. The State_abbrev_lookup data set includes a snippet of the lookup table that can operationalize this task.

```
data state_abbrev_lookup;
   infile datalines;
   length st $2 state $20;
   input st $ state $;
   datalines;
AK Alaska
AL Alabama
AR Arkansas
AZ Arizona
CA California
MS Mississippi
;
```

The TRANSFORM_STATE function transforms valid state abbreviations into state names, and returns a missing value when an invalid state abbreviation is encountered.

```
proc fcmp outlib=work.funcs.lookup;
   function transform_state(st $) $;
      length st $2 state $20;
      declare hash h(dataset: 'state_abbrev_lookup');
      rc = h.definekey('st');
      rc = h.definedata('state');
      rc = h.definedone();
      rc = h.find();
      return(state);
      endfunc;
quit;
```

The following DATA step calls the TRANSFORM_STATE function directly.

```
options cmplib=work.funcs;
data transformed_fcmp;
   set possible_states;
   length state_name $20;
   state_name=transform_state(state_abbr);
   put state_abbr state_name;
run;
```

The log demonstrates that the first five values are valid, and shows their transformed values.

```
CA California
MS Mississippi
CA California
AL Alabama
AL Alabama
DC
VI
NOTE: There were 7 observations read from the data set WORK.POSSIBLE_STATES.
NOTE: The data set WORK.TRANSFORMED_FCMP has 7 observations and 2 variables.
```

The following FORMAT procedure defines the TRANSFORM_STATE_FMT user-defined character format, which indirectly calls the TRANSFORM_STATE user-defined function (via the OTHER option) when the format is applied. Thereafter, the PRINT procedure applies the TRANSFORM_STATE_FMT format to transform state abbreviations temporarily to state names.

```
proc format;
   value $ transform_state_fmt other=[transform_state()];
run;

proc print data=possible_states;
   var state_abbr;
   format state_abbr $transform_state_fmt.;
run;
```

Thus, the output demonstrates how the user-defined function can be applied (indirectly) inside the PRINT procedure.

Obs	state_abbr
1	California
2	Mississippi
3	California
4	Alabama
5	Alabama
6	
7	

The benefit of applying user-defined functions indirectly (via user-defined formats) is that formats can transform data temporarily using the FORMAT statement, and can do so inside many SAS procedures. And because user-defined functions can encapsulate complex business rules and conditional logic, user-defined formats can in turn transform data using methods far more intelligent than the hash object lookup that has been demonstrated.

4. DEEP PROC AND DEEP DATA (DPDD)

The FCMP procedure enables a DATA step to be executed inside of a DATA step or a SAS procedure to be executed inside of a DATA step. The built-in RUN_MACRO function supports this seeming black magic by creating a "side session" of SAS in which the child DATA step or SAS procedure executes. A separate text by the author fully introduces this usage of RUN_MACRO (and its kissing cousin RUN_SASFILE) inside of the FCMP procedure: *Undo SAS® Fetters with Getters and Setters: Supplanting Macro Variables with More Flexible, Robust PROC FCMP User-Defined Functions That Perform In-Memory Lookup and Initialization Operations.* (Hughes, 2023)

Consider the requirement to subdivide a data set programmatically. For example, from the preceding Superpowers data set, four derivative data sets could be created—one each for Thor, Doctor Strange, Thanos, and Wanda Maximoff. Many SAS practitioners would accomplish this feat using the EXECUTE subroutine (aka, CALL EXECUTE). Another option, however, relies on the FCMP procedure, which can leverage RUN_MACRO to create the derivative data sets through DPDD.

The following FCMP procedure defines the CREATE_DATASETS user-defined function, which calls the RUN_MACRO function, which calls the CREATE_DATASETS_MACRO macro, which executes the DATA step that creates each derivative data set.

```
%macro create_datasets_macro;
%let superhero=%sysfunc(dequote(&superhero));
%let superhero_file=%sysfunc(compress(&superhero));
data &superhero_file;
   set superpowers;
   where hero="&superhero";
run;
%mend;
```

```
proc fcmp outlib=work.funcs.reports;
   function create_datasets(superhero $);
      rc = run_macro('create_datasets_macro', superhero);
      return(.);
      endfunc;
quit;
```

Thus, the following DATA step creates four data sets—Thor, Doctorstrange, Thanos, and Wandamaximoff. And each data set contains only the data associated with one superhero.

```
data _null_;
   set superpowers;
   rc=create_datasets(hero);
run;
```

The full capabilities of DPDD methods lie outside the scope of this text; however, data-driven programming can be exemplified when data maintained in a data set (like Superpowers) can be leveraged to drive dynamic processing.

5. REPLACE UNNECESSARY MACRO PROGRAMMING

The need for modular, maintainable, readable, reusable, configurable code is nothing new, and long predates release of the FCMP procedure. Thus, prior to FCMP, SAS practitioners instead turned to the SAS macro language to build so-called “macro functions” that resolve to dynamic output given dynamic input. The SAS macro language was not a bad solution; however, with the introduction of the FCMP procedure burgeoned better methods to create modular, reusable functionality in Base SAS.

Consider the requirement to determine how much stuff can be crammed into a sphere, whose volume is determined by the formula $4/3\ \pi\ r^3$. For example, assuming a perfect sphere, how much stuff fits in Uranus? NASA has previously probed Uranus, and records its radius to be 15,881.5 miles. (NASA, 2024) Thus, assuming a perfect sphere, the volume of Uranus can be calculated to be approximately 16,778,887,651,557 cubic miles.

And because the formula that calculates spherical volume is not a built-in SAS function, and because SAS practitioners would rather not type and retype its complexity—a practice that would be both inefficient and error prone—a reusable software module is the preferred method for this calculation, in which a single argument represents the radius of the sphere.

A user-defined FCMP function is preferred, but first, the VOLUME_SPHERE macro calculates the volume of Uranus using the &URANUS_RAD macro variable.

```
%macro volume_sphere(radius);
%let vol=%sysevalf(4/3 * %sysfunc(constant(pi)) * &radius**3);
&vol
%mend;

%let uranus_rad=15881.5;
%let uranus_vol=%volume_sphere(&uranus_rad);
%put &uranus_vol;
```

The log demonstrates the computed volume of Uranus:

```
1131  %put &uranus_vol;
16778887651557
```

Despite its functionality, use of the SAS macro language here is unnecessary and, arguably, unwarranted. That is, the FCMP procedure can effect cleaner functionality that does not require the SAS Macro Facility to parse any macro statements. Thus, the VOLUME_SPHERE user-defined function calculates the volume of any sphere given its radius.

```
proc fcmp outlib=work.funcs.formulas;
   function volume_sphere(radius);
      return(4/3 * constant('pi') * radius**3);
      endfunc;
quit;
```

And the following DATA step similarly calculates the volume of Uranus by calling VOLUNE_SPHERE.

```
options cmplib=work.funcs;
data _null_;
   uranus_rad=15881.5;
   uranus_vol=volume_sphere(uranus_rad);
   format uranus_vol comma20.;
   put uranus_vol=;
run;
```

The log demonstrates the identical calculation without having to rely unnecessarily on the SAS macro language. Moreover, the %SYSFUNC macro function can be used to call user-defined functions.

```
%let uranus_rad=15881.5;
%let uranus_vol=%sysfunc(volume_sphere(&uranus_rad));
%put &uranus_vol;
```

The SAS macro language is a powerful advocate whose reputation should not be sullied; however, it does have limitations, such as all macro variables being character (as opposed to numeric), or the common requirement that data be masked in the SAS macro language when they contain special characters such as quotes, the ampersand symbol, or the percent sign. Rather, FCMP user-defined functions natively support both character and numeric scalar variables, in addition to character and numeric arrays. Moreover, because user-defined functions do not require data to be translated into macro variables (excepting where RUN_MACRO and RUN_SASFILE are utilized), tedious data masking of arguments is also unnecessary when calling user defined FCMP functions, as opposed to so-called macro functions.

As with the other user-defined functions that have been demonstrated, VOLUME_SPHERE increases several aspects of software quality. It is more *modular* because its functionality is encapsulated within the FCMP procedure; it is more *readable* because the function has been removed from the DATA step; it is more *reusable* and *configurable* because the volume of any sphere can be calculated; and it is *maintainable* because the function can be modified (if necessary) without altering the DATA step that calls it.

CONCLUSION

This introduction has only scratched the surface of the FCMP procedure and the powerful user-defined functions and subroutines that can be engineered within it. Some high-level FCMP functionality was demonstrated, although FCMP syntax lies outside the scope of this text. When considering how and whether to implement FCMP user-defined functions—possibly to refactor existent SAS programs—the need to increase software quality, inasmuch as the productivity of developers (who can develop user-defined functions *once* yet use them *forever*), should drive the decision to master the FCMP procedure and to implement its vast functionality.

REFERENCES

Hughes, T. M. (2016). *SAS® Data Analytic Development: Dimensions of Software Quality.* Hoboken, NJ: John Wiley and Sons.

Hughes, T. M. (2023). Make You Holla' Tikka Masala: Creating User-Defined Informats Using the PROC FORMAT OTHER Option To Call User-Defined FCMP Functions That Facilitate Data Ingestion Data Quality. *PharmaSUG.* San Francisco, CA. Retrieved from https://www.lexjansen.com/pharmasug/2023/AP/PharmaSUG-2023-AP-291.pdf

Hughes, T. M. (2023). *SAS® Data-Driven Development: From Abstract Design to Dynamic Functionality, Second Edition.* San Diego: Kindle Direct Publishing.

Hughes, T. M. (2023). Sorting a Bajillion Variables—When SORTC and SORTN Subroutines Have Stopped Satisfying, User-Defined PROC FCMP Subroutines Can Leverage the Hash Object to Reorder Limitless Arrays. *PharmaSUG.* San Francisco, CA. Retrieved from https://www.lexjansen.com/pharmasug/2023/AP/PharmaSUG-2023-AP-094.pdf

Hughes, T. M. (2023). Undo SAS® Fetters with Getters and Setters: Supplanting Macro Variables with More Flexible, Robust PROC FCMP User-Defined Functions That Perform In-Memory Lookup and Initialization Operations. *PharmaSUG.* San Francisco, CA. Retrieved from https://www.lexjansen.com/pharmasug/2023/HT/PharmaSUG-2023-HT-093.pdf

Hughes, T. M. (2024). *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler.* Cary, NC: SAS Press.

NASA. (2024, March 24). *Uranus Facts*. Retrieved from https://science.nasa.gov/uranus/facts/

CONTACT INFORMATION

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

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