Paper 166-2023

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

Troy Martin Hughes

ABSTRACT

Getters and setters are common in some object-oriented programming (OOP) languages such as C++ and Java, where “getter” functions retrieve values and “setter” functions initialize (or modify) variables. In Java, for example, getters and setters are constructed as conduits to private classes, and facilitate data encapsulation by restricting variable access. Conversely, the SAS® language lacks classes, so SAS global macro variables are typically utilized to maintain and access data across multiple DATA steps and procedures. Unlike an OOP program that can categorize variables across multiple user-defined classes, however, SAS maintains only one global symbol table in which global macro variables can be maintained. Additionally, maintaining and accessing macro variables can be difficult when quotation marks, ampersands, percentage signs, and other special characters exist in the data. This text introduces user-defined getter functions and setter subroutines designed using the FCMP procedure, which enable data lookup and initialization operations to be performed within DATA steps. Among other benefits, user-defined getters and setters can facilitate the evaluation of complex Boolean logic expressions that leverage data stored across multiple data sets—all concisely performed in a single SAS statement! Getters and setters are thoroughly demonstrated in the author’s text: *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler*. (Hughes, 2023)

CALCULATING CALORIC CONTENT: FROM PSEUDOCODE TO POOR SOLUTIONS

The full functionality and context of getters and setters lie far beyond the scope of this text. In a nutshell, however, getters retrieve data values from encapsulated variables, and setters initialize or modify variables. This functionality can be mimicked in SAS by storing key-value pairs within SAS data sets, by retrieving values by calling user-defined *functions*, and by modifying values by calling user-defined *subroutines*. In all cases, user-defined functions and subroutines are created using the FCMP procedure.

For example, consider the need to calculate the number of calories in some broccoli that you want to fry up. You *massuse* (i.e., calculate the mass of) the ingredients—raw broccoli, extra virgin olive oil, and garlic cloves—and subsequently determine the number of calories per gram in each ingredient. This formula for total caloric content can be represented in pseudocode:

```
total cal = [100 * broccoli(cal)] + [30g * olive_oil(cal)] + [20 * garlic(cal)]
```

In translating this pseudocode into SAS code, many SAS practitioners instinctively reach for tried and true macro variables to represent formulae such as these. Global macro variables are especially useful because they are retained across DATA steps. For example, the following three %LET statements initialize three global macro variables that represent the number of calories per gram of each food:

```
%let cal_broc=.34;
%let cal_oil=7.89167;
%let cal_garlic=1.49;
```

Thereafter, the macro variables can be used to calculate the number of calories consumed when frying up some broccoli—which turns out to be 300.55 calories:

```
data _null_;
   total_cal = (100*&cal_broc) + (30*&cal_oil) + (20*&cal_garlic);
   put total_cal=;
run;
```

And why do SAS practitioners (lazily) turn to macro variables for this type of calculation? Well, because it is often simpler than equivalently storing the data within SAS data sets, and using the DATA step to perform the calculation.

For example, the following three DATA steps initialize some nutritional data for broccoli, olive oil, and garlic, respectively:

```
data broccoli;
   length metric $20 value 8;
   metric='cal'; value=.34; output;
   metric='protein'; value=.0282; output;
   metric='carbs'; value=.0664; output;
   metric='fiber'; value=.026; output;
run;

data olive_oil;
   length metric $20 value 8;
   metric='cal'; value=125/15*.947; output;
   metric='protein'; value=0; output;
   metric='carbs'; value=0; output;
   metric='fiber'; value=0; output;
   metric='fat'; value=15/15*.947; output;
run;

data garlic;
   length metric $20 value 8;
   metric='cal'; value=1.49; output;
   metric='protein'; value=.0636; output;
   metric='fat'; value=.005; output;
   metric='carbs'; value=.331; output;
   metric='fiber'; value=.021; output;
run;
```

Caloric content for one gram of each food item is maintained in the observation for which the Metric variable is “cal”; thus, the following DATA step computes the equivalent caloric total when frying up some broccoli:

```
data calculate;
   set broccoli (in=a where=(metric='cal')) olive_oil (in=b where=(metric='cal'))
      garlic (in=c where=(metric='cal')) end=eof;
   retain total_cal 0;
   if a then total_cal = total_cal + (100 * value);
   else if b then total_cal = total_cal + (30 * value);
   else if c then total_cal = total_cal + (20 * value);
   if eof then put total_cal=;
run;
```

The identical result is achieved—that the recipe contains 300.55 calories. The complexity results from maintaining caloric values across multiple data sets and across multiple observations therein, which requires either a multi-data SET statement, a MERGE statement, or a SQL join. However, even if

maintained within a single data set, the nutritional metrics would still reside in separate observations, and thus require a RETAIN statement or other operation to facilitate computation across multiple observations.

For example, the following DATA step now creates the Food data set that contains nutritional information for broccoli, olive oil, and garlic:

```
data food;
   length item $30;
   set broccoli (in=a) olive_oil (in=b)
      garlic (in=c);
   if a then item='broccoli';
   else if b then item='olive oil';
   else if c then item='garlic';
run;
```

The Food data set is demonstrated in Table 1, in which Value represents the quantity of each metric in one gram of each food.

	item	metric	value
1	broccoli	cal	0.34
2	broccoli	protein	0.0282
3	broccoli	carbs	0.0664
4	broccoli	fiber	0.026
5	olive oil	cal	7.8916666667
6	olive oil	protein	0
7	olive oil	carbs	0
8	olive oil	fiber	0
9	olive oil	fat	0.947
10	garlic	cal	1.49
11	garlic	protein	0.0636
12	garlic	fat	0.005
13	garlic	carbs	0.331
14	garlic	fiber	0.021

Table 1. Food Data Set Containing Nutritional Data

However, even with all three caloric values maintained within a single data set, the calculation is still complex (as compared to the original single-line pseudocode) because multiple observations must be evaluated to sum the data:

```
data calculate;
   set food end=eof;
   retain total_cal 0;
   if item='broccoli' and metric='cal' then total_cal = total_cal + (100 * value);
   else if item='olive oil' and metric='cal' then total_cal = total_cal
      + (30 * value);
   else if item='garlic' and metric='cal' then total_cal = total_cal
      + (20 * value);
   if eof then put total_cal=;
run;
```

It is because of this DATA step complexity that SAS practitioners often reach instead for macro variables to express and compute formulae. However, the use of macro variables can be compromised by unruly data that contain quotes, ampersands, percent signs, and other special characters. Moreover, the solitary global symbol table requires that all global macro variables be lumped together, without care or categorization, into a morass of macro sludge! Given these pitfalls, the remainder of this text demonstrates the equivalent use of getters and setters to evaluate and manipulate values simply yet without the unnecessary scourge of macro variables.

GETTER FUNCTIONS

The FCMP procedure empowers SAS practitioners to build user-defined functions and subroutines. A primary benefit of functions is the *reusability* they promote—that is, the ability to design a function once yet use it repeatedly, including across various software projects and products, and by various users. A second benefit of functions is *encapsulation*—the hiding of complex functionality within a function's definition, which improves the readability of the program calling the function.

For example, the following FCMP procedure defines the GET_BROCCOLI function, which relies on a hash object to return the specific nutritional metric that is requested:

```
proc fcmp outlib=work.funcs.food;
   function get_broccoli(metric $);
      length value 8;
      declare hash h(dataset: 'broccoli');
      rc=h.defineKey('metric');
      rc=h.defineData('value');
      rc=h.defineDone();
      rc=h.find();
      return(value);
      endfunc;
quit;
```

The DATASET argument within the hash declaration specifies Broccoli as the data set to load into the hash object. The DEFINEKEY method subsequently declares the Metric variable as the key, and the DEFINEDATA method declares the Value variable as the value—that is, when the function is called, the key is passed (as an argument), and the corresponding value (in the key-value pair) is returned to the calling program.

For example, the following DATA step retrieves the number of carbohydrates in 100 grams of broccoli:

```
options cmplib=work.funcs;

data _null_;
   tot_carbs=get_broccoli('carbs')* 100;
   put tot_carbs=;
run;
```

Note that the CMPLIB option must specify the library and data set in which the function is saved. Thus, the OUTLIB option specified in the GET_BROCCOLI definition must correspond to the CMPLIB option.

The DATA step executes and demonstrates the number of carbohydrates in 100 grams of broccoli:

```
tot_carbs=6.64
```

Also note that initialization of the Tot_carbs variable was completed in a single statement; this is a benefit of relying on key-value pairs for lookup operations—that the external data set (Broccoli) is accessed by a function rather than a SET statement, MERGE statement, or other complex logic.

Two additional functions—GET_OLIVE_OIL and GET_GARLIC—are next defined:

```
proc fcmp outlib=work.funcs.food;
   function get_olive_oil(metric $);
      length value 8;
      declare hash h(dataset: 'olive_oil');
      rc=h.defineKey('metric');
      rc=h.defineData('value');
      rc=h.defineDone();
      rc=h.find();
      return(value);
      endfunc;
quit;

proc fcmp outlib=work.funcs.food;
   function get_garlic(metric $);
      length value 8;
      declare hash h(dataset: 'garlic');
      rc=h.defineKey('metric');
      rc=h.defineData('value');
      rc=h.defineDone();
      rc=h.find();
      return(value);
      endfunc;
quit;
```

Having defined these three functions, the number of calories in fried broccoli can again be calculated—now using a single SAS statement that calls each function, and without reliance on global macro variables:

```
data _null_;
   tot_cal=(100 * get_broccoli('cal')) + (30 * get_olive_oil('cal'))
      + (20 * get_garlic('cal'));
   put tot_cal=;
run;
```

Separate getter functions are typically required for separate classes of data—again, the SAS language does not support true classes, but SAS data sets can be used to mimic some class functionality. For example, the Broccoli, Olive_oil, and Garlic data sets could be said to be three instances of one class of data—that is, each data set contains the Metric character variable and the Value numeric variable. And because of this sameness, the three data sets could be aggregated into one Food data set, as previously demonstrated.

Thus, the updated GET_FOOD function now replaces the separate GET_BROCCOLI, GET_OLIVE_OIL, and GET_GARLIC functions. In this sense, the Food data set represents a class of data, and the Item variable is used to define one or more objects subordinate to that class—the broccoli object, the olive oil object, and the garlic object. For this reason, two parameters—ITEM and METRIC—are now declared in the FUNCTION statement, and two keys are declared in the DEFINEKEY statement—Item and Metric:

```
proc fcmp outlib=work.funcs.food;
   function get_food(item $, metric $);
      length value 8;
```

```
        declare hash h(dataset: 'food');
        rc=h.defineKey('item','metric');
        rc=h.defineData('value');
        rc=h.defineDone();
        rc=h.find();
        return(value);
        endfunc;
quit;
```

Now, rather than having to maintain three separate (but similar) functions, as well as three separate data sets, the single GET_FOOD function is maintained and called. Also note that when GET_FOOD is called, two arguments now must be passed, corresponding to the object (i.e., food name, like *broccoli*) and attribute (i.e., nutritional metric, like *calories*):

```
data _null_;
   tot_cal=(100 * get_food('broccoli','cal')) + (30 * get_food('olive oil','cal'))
      + (20 * get_food('garlic','cal'));
   put tot_cal=;
run;
```

The result (300.55 calories) is identical to the previous methods, but now succinctly relies on one function (GET_FOOD) and one data set (Food) to deliver equivalent functionality.

SETTER SUBROUTINES

Setter subroutines complement getter functions, and initialize or update values. After a setter subroutine has modified a variable, a getter function accessing the same variable will retrieve the updated value, and in this sense, getters and setters work in tandem. The setter subroutines demonstrated throughout this text only update existing variables; that is, by design, they cannot create a new variable. This practice eliminates the risk of creating an anomalous variable (using a setter subroutine) when a variable name is mistyped. However, for other purposes, setter subroutines can be designed that initialize new variables not previously declared or referenced.

Consider the need to update the nutritional metrics associated with garlic; for example, you might be cooking with some tawdry canned garlic that contains preservatives or other adulterants, and need to make the following alterations in the Food data set:

- Increase the amount of fat in 1g of garlic from 0.331 to 0.69.
- Increase the number of carbs in 1g of garlic from 0.005 to 0.02.

In pseudocode, each of these actions requires only one line of code to initialize each new value:

- fat_garlic = 0.69
- carbs_garlic = 0.02

And, if utilizing macro variables to maintain nutritional metrics, the following %LET statements equivalently modify the &FAT_GARLIC and &CARBS_GARLIC macro variables:

```
%let fat_garlic = 0.69;
%let carbs_garlic = 0.02;
```

However, as already discussed, reliance on macro variables for this type of formula evaluation would require declaring each of the 14 metrics listed in Table 1 as a separate macro variable—a solution that lacks scalability. Moreover, the underlying data structure of the Foods data set, which differentiates values by not only food type but also nutritional metric, is unfortunately dissolved when all values must be dumped into the single global macro symbol table.

Thus, as the pre-existing garlic nutritional metrics are already stored in the Food data set, a DATA step can modify the garlic values for fat and carbs:

```
data new_food;
   set food;
   if item='garlic' and metric='fat' then value=.69;
   if item='garlic' and metric='carbs' then value=.02;
run;
```

The solution is functional, and the New_food data set now contains the updated garlic values; however, the DATA step requires reading the Food data set—and ONLY Food. That is, an entire DATA step is required to make the modifications, and if a separate data set additionally requires modifications, a separate DATA step would be required. This is where setter subroutines can save the day.

User-defined functions and subroutines (created using the FCMP procedure) cannot directly include a DATA step; however, by leveraging the RUN_MACRO built-in function (which is available only inside the FCMP procedure), a user-defined function can call a user-defined macro, and that macro can include a DATA step. This is one methodology through which setter functionality can be operationalized in SAS.

For example, the following SET_FOOD subroutine calls the SET_FOOD macro; the subroutine and macro do not need to be identically named, but this facilitates identification of their symbiosis and dependency on each other:

```
%macro set_food();
%let item=%sysfunc(dequote(&item));
%let metric=%sysfunc(dequote(&metric));
data food;
   set food;
   if item="&item" and metric="&metric" then value=&value;
run;
%mend;

proc fcmp outlib=work.funcs.food;
   subroutine set_food(item $, metric $, value);
      rc=run_macro('set_food', item, metric, value);
      endsub;
quit;
```

The following DATA step calls the SET_FOOD subroutine twice—to reinitialize the Value variable for the Garlic-Fat observation (#9) and the Garlic-Carbs observation (#13) in the Food data set:

```
data _null_;
   call set_food('garlic', 'fat', .69);
   call set_food('garlic', 'carbs', .02);
run;
```

Each of the two subroutine calls, which requires the CALL statement, overwrites a single value in the Food data set because the SET_FOOD subroutine executes the DATA step inside the SET_FOOD macro. That is, when SET_FOOD is invoked, a separate (unseen) SAS session briefly executes the DATA step in real-time that modifies Value only when the Item and Metric conditions are met. Thus, the RUN_MACRO function enables a DATA step to be executed from inside a separate DATA step. Moreover, if separate getter or setter functions needed to retrieve or modify values held in separate lookup tables, these getters and setters could be executed inside the same DATA step, despite accessing different data sets.

CONCLUSION

Getter and setter functions can be simulated in Base SAS through user-defined functions and subroutines that are designed in the FCMP procedure. Although object-oriented classes and objects are not supported by Base SAS, data sets can nevertheless be used to simulate objects, and FCMP functions and subroutines can interact with and modify these data sets to provide some getter and setter functionality. These getter functions and setter subroutines, in many instances, can replace the reliance on SAS macros and the global symbol table to maintain data across multiple procedures and DATA steps, and in so doing, can create a more organized, reliable method to access and modify persistent data.

REFERENCES

Hughes, T. M. (2023). *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler.* 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