SAS4336-2020

Minimizing Environmental Disturbances: Applying Leave No Trace Principles to SAS® Programming

Carl Sommer, SAS Institute Inc.

ABSTRACT

It is essential that all components behave appropriately when integrating multiple programs and macros. Although they might have worked fine in their original context, issues can come up in a reuse scenario as the result of either expectations of certain state conditions in the SAS® environment that are suddenly not in effect, or because some other program has altered the state of the SAS environment. The "Leave No Trace Seven Principles", which focus on outdoor ethics, behavior, and cleanliness, can also be applied to SAS programming. These principles help establish a coding standard that contributes positively to the business continuum, to a business process modernization, or to a migration strategy within a customer environment. This paper presents this rubric, as well as a macro that can be used to create and compare environmental snapshots, in order to help you quickly identify, isolate, and understand the before and after state of your environment.

INTRODUCTION

Good programming practice, regardless of the programming language involved, relies on the concept of modularity. Modularity provides benefits such as testability, defined interfaces, and reusability. In SAS programming, modularity is commonly provided through several different manners, some of the most common being the following:

1. Macro definitions
2. In-lined code, through %INCLUDE
3. Code executed through the AUTOEXEC
4. Code produced by nodes in a process flow in SAS Enterprise Guide® or SAS Data Integration Studio®

Modules function as building blocks that are developed and tested in isolation. Once verified, they are used to construct larger programs. As such, each module possesses an implicit sense of appropriate "state", i.e., expectations needed for successful execution. Most commonly, this is the setting of SAS options, but could also include the state, existence, or structure of data, as well as the values of SAS macro variables.

As requirements change and complexity grows in these larger programs, modules are changed, possibly unwittingly disturbing the expected state between modules. Hopefully, this is caught during a testing phase. But while it may be easy to determine that there is a problem, determining the root cause of what changed can be far more difficult.

THE SEVEN PRINCIPLES OF LEAVE NO TRACE

The Leave No Trace™ Center for Outdoor Ethics (https://lnt.org/) protects the outdoors by teaching and inspiring people to enjoy it responsibly. The Center accomplishes this mission by delivering cutting-edge education and research to millions of people every year. At the center of the Leave No Trace educational message are the following seven principles:

1. Plan Ahead and Prepare
2. Travel and Camp on Durable Surfaces
3. Dispose of Waste Properly
4. Leave What You Find
5. Minimize Campfire Impacts
6. Respect Wildlife
7. Be Considerate of Other Visitors

The underlying theme to these principles is "Leave No Trace". Obviously, the outdoors is more enjoyable and sustainable if everyone makes the effort to minimize their impacts.

APPLYING THE SEVEN PRINCIPLES TO PROGRAMMING

While we hopefully don't have literal campfires, durable surfaces, or wildlife in our SAS programs, when we look at the principles, we can easily map the concepts to applicable practices, many of which have been explored by other SAS Global Forum papers mentioned in the references section. While those papers have pointed out various concerns and best practices, the use of a principled approach enables programmers to evaluate their code and improve it without a heavy set of rules.

PLAN AHEAD AND PREPARE

What macros are you using? Do you know what side effects they might have? Are you aware of when and how these macros get changed, or will you only find out in production that a piece of code you rely upon has changed? This is where good practices in source code management, dev/test/prod testing, and change management can protect you from a call from IT in the middle of the night.

TRAVEL AND CAMP ON DURABLE SURFACES

If your environment is like mine (and I suspect it is) there are multiple pieces of code that do the same thing, or nearly the same thing. An effort to cull through this morass of redundancy will enable an organization to establish a set of best-of-breed macros. These should be the "durable surface" that you use for utility tasks, instead of blazing a new trail.

While each new programming project is to satisfy a specific need, often it can be constructed of existing portions of code, with proper parameterization. Don't reinvent the wheel; inflate and lubricate it instead!

DISPOSE OF WASTE PROPERLY

Nobody wants to go on a picnic, only to find the trash from someone else's lunch at their site. Likewise, while your code may create temporary data sets or create global macro variables, if they are not part of the documented outputs of your code, they should be cleaned up upon exit.

It's bad for bears to pick through picnic trash, as they become dependent upon this as a food source (inappropriate to their nutritional needs) instead of foraging for food as bears should.

Likewise, each module should clean up any temporary data sets or macro variables not explicitly intended for later modules to consume. I've seen cases where later modules unwisely depended upon these artifacts, assuming they would always be present, and even situations where the presence of these artifacts caused problems with another module because the module didn't expect them to exist.

LEAVE WHAT YOU FIND

If you didn't create it, you ought not delete it. That includes data sets or macro variable values. They aren't yours per se. Likewise, if you need to set an option, be sure to set it back. That includes making sure that every exit path out of your code resets options.

If your code sets global macro variable values, it needs to be with the understanding that this is expected by other modules in the larger body of code. If the job can be done with a local macro variable, then use it!

MINIMIZE CAMPFIRE IMPACTS

Probably the quickest way to have a ranger kick you out of a campground is to start a fire and not use the fire ring provided at your site. Campfires are a cherished part of an outdoor experience, but intense campfires can damage the soil. Thus, campsites have fire rings so only one location is affected.

From a programming perspective, a campfire could be thought of as "what happens when things go wrong with my program?" If your program aborts, does it properly clean up its resources?

Another perspective would be to consider the CPU, memory, and I/O resource impacts of your code. Does your code scale? Is it possible to re-work it, perhaps with a hash, to be less resource-intensive? Your code needs to not starve other workloads in the execution environment.

RESPECT WILDLIFE

The symbol of the US National Park Service is the American Bison (commonly called the buffalo). This is a majestic creature, but is best enjoyed from a distance, due to its immense size, power, and unpredictable behavior. Though fuzzy and endearing, bison are not a pet, so the best advice is to respect this creature by giving it plenty of space.

Likewise, while wildflowers are beautiful, if they are all picked there's nothing left for anyone else to enjoy, including the creatures that are dependent upon them.

What's the wildlife in your SAS code? Perhaps it's needlessly assigning librefs in an autoexec, thus consuming connections to a database that none of your code will be using. Or perhaps that same autoexec creates a plethora of global macro variables "just in case" any of the programs need them. Maybe that is OK, maybe it isn't; that's why this is a principle, not an edict.

In the previously mentioned example of the autoexec that defines numerous global macro variables with static values, you can protectively prevent other code from meddling with them by declaring with the READONLY option and assigning the value as part of the declaration, as shown below:

```
%global / readonly myRootPath=%str(c:\carl\lnt\);
```

The READONLY option is applicable to both the %GLOBAL and %LOCAL macro statements. For more details, see https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=mcrolref&docsetTarget=p1lhhti7fjxgb1n1fuiubqk11h4d.htm&locale=en#n0iw343os6vy7ln1xpxk682930dc.

BE CONSIDERATE OF OTHER VISITORS

Nothing ruins a good hike like trash on the trail or music so loud you cannot carry on a conversation or appreciate the scenery. Likewise, nothing can waste the time and patience of a SAS programmer quite like setting another module that sets SAS options which then cause your program to fail. Did you know there are over 300 SAS options? While many of them are largely innocuous to most of your code, others (such as OBS= or NONOTES) can cause great headaches if you are not aware they have been set. You should set an option only if your code requires it, and then you should set it back to what it was before you set it when you are done. This is the number one way to avoid environmental disturbances.

NECESSITY IS THE MOTHER OF INVENTION

Have you ever spent time trying to debug code that seemed to work perfectly fine in isolation? Or perhaps to trace down what piece of code was changing or modifying a table? At one point in my career I was working with a large body of code, much of which was generated by other programs, and I had a couple of problems that I couldn't find when I looked at the individual pieces.

Perhaps had I been a smarter programmer I would have found the root cause. However, what I needed was something to help me figure out at what point things went astray, as well as to help me understand what was different between different executing environments. What I needed was a tool to take snapshots of the environment (options settings, data set structures, column definitions, and macro variable values) that I could then compare to determine what assumption was not being satisfied.

A tool is a tool because it does something useful, not necessarily because it's for sale in a hardware store. Since I could not find such a tool, I wrote one myself and then proceeded to instrument my code with calls to it to create the environmental snapshots that I could compare. Ultimately, this enabled me to isolate the problem (which turned out to be a compiled secured macro I was using to set an option that foiled me). I had totally overlooked it because it generated no messages whatsoever to the SAS log. Debugging lesson learned: question all assumptions, and trust nothing!

THE %SNAPSHOT MACRO

The tool I wrote is a macro called %SNAPSHOT, which is available on the SAS Global Forum 2020 Github repository. You are welcome to use and enhance this macro for your needs, as per the license in the repository:

https://github.com/sascommunities/sas-global-forum-2020/blob/master/LICENSE

This macro is embarrassingly simple: it captures named snapshots (case-insensitive) of state information and maintains a table of these snapshots, which it can run comparisons between. This information is stored in a library that the user allocates and provides in the macro invocation. The user can perform four very simple actions:

1. Clear all snapshots (ACTION=CLEAR)
2. Capture a snapshot (ACTION=CAPTURE)
3. List all snapshots (ACTION=LIST)
4. Compare two snapshots (ACTION=COMPARE)

This macro keeps track of the current state of SAS OPTIONS, macro variable settings, and data set structure (but not individual records; again, if you want to enhance the macro to do that, feel free to do so).

COMMENT BLOCK DOCUMENTATION FROM THE MACRO

```
%macro SNAPSHOT(snaplib  = %str(SNAPLIB),
                maxsnaps = %str(100),
                action   = %str(LIST),
                name     = %str(),
                base     = %str(),
                comp     = %str(),
                libs     = %str(_NONE_));
/* Keyword Parameters:
|  SNAPLIB=  library where snapshots are recorded (Default is SNAPLIB)
|
|  MAXSNAPS= max number of snapshots to keep(Default is 100)
|            Only used on initial capture to the snaplib.
|
|  ACTION=   What type of action to perform.  One of the following:
|               CAPTURE - Take a snapshot
|               COMPARE - Compare two snapshots
|               LIST    - List the snapshots taken  (Default)
|               CLEAR   - Erase all snapshots
|
|  NAME=     Name to assign to the snapshot.  Only used (and required)
|            when specifying ACTION=CAPTURE.   This parameter value is
|            case-insensitive.
|
|  BASE=     Base snapshot to use when comparing two snapshots.  Must
|            be a valid name specified for an earlier ACTION=CAPTURE.
|            This parameter value is case-insensitive.
|
|  COMP=     Second snapshot to use when comparing two snapshots.  Must
|            be a valid name specified for an earlier ACTION=CAPTURE.
|            This parameter value is case-insensitive.
|
|  LIBS=     list of SAS librefs to monitor for changes.  Default is _NONE_
|            Blank means _NONE_.   Can also specify _ALL_, or a
|            space-delimited list. Note that SASHELP is included in _ALL_
|
|  Output: For ACTION=CAPTURE, confirmation of the snapshot being taken
|          For ACTION=LIST, a list of the snapshots that have been taken
|          For ACTION=COMPARE, text in the log indicating if any
|          differences have been noted, and PROC COMPARE output of such
*/
```

EXAMPLE USAGE OF %SNAPSHOT

The following code demonstrates the use of the %snapshot macro, with a macro variable, option, and data set change occurring between the COPY and TAX snapshots:

```
libname snaplib 'c:\temp';
%snapshot(action=CLEAR)

%snapshot(action=CAPTURE, name=INIT, libs=WORK);

proc copy in=sashelp out=work; select cars; run;
%snapshot(action=CAPTURE, name=COPY, libs=WORK)

options linesize = 120;  /* option difference         */
%let taxrate = .025;     /* macro variable difference */
data work.cars;          /* data set difference       */
  set work.cars;
  format tax dollar8.;
  tax=msrp*&taxrate;
run;

%snapshot(action=CAPTURE, name=TAX, libs=WORK)

%snapshot(action=LIST)
%snapshot(action=COMPARE, base=COPY, comp=TAX)
```

The SAS log shows the following:

```
2638  libname snaplib 'c:\temp';
NOTE: Libref SNAPLIB was successfully assigned as follows:
      Engine:        V9
      Physical Name: c:\temp
2639  %snapshot(action=clear)
**** Snapshots cleared from SNAPLIB ****
2640
2641  %snapshot(action=CAPTURE, name=INIT, libs=WORK);
**** Snapshot INIT Capture Completed ****
2642
2643  proc copy in=sashelp out=work; select cars; run;

NOTE: Copying SASHELP.CARS to WORK.CARS (memtype=DATA).
NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.CARS has 428 observations and 15 variables.
NOTE: PROCEDURE COPY used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds


2644  %snapshot(action=CAPTURE, name=COPY, libs=WORK)
**** Snapshot COPY Capture Completed ****
2645
2646  options linesize=120;
2647  %let taxrate = .025;
2648  data work.cars;
2649    set work.cars;
2650    format tax dollar8.;
2651    tax=msrp*&taxrate;
2652  run;
```

```
NOTE: There were 428 observations read from the data set WORK.CARS.
NOTE: The data set WORK.CARS has 428 observations and 16 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


2653
2654  %snapshot(action=CAPTURE, name=TAX, libs=WORK)
**** Snapshot TAX Capture Completed ****
2655
2656  %snapshot(action=LIST)
********* SNAPSHOT REPORT - 17FEB2020:23:30:34.55  *********
Name               When Taken                    Generation
INIT               17FEB2020:23:30:34.22         1
COPY               17FEB2020:23:30:34.35         2
TAX                17FEB2020:23:30:34.47         3
************************************************************
2657  %snapshot(action=COMPARE, base=COPY, comp=TAX)
********* Snapshot Comparison Started  *********
********* Base Snapshot: COPY
********* Comp Snapshot: TAX

--  MACRO Variable differences found.  See PROC COMPARE output
--  OPTION differences found.  See PROC COMPARE output
--  DATASET differences found.  See PROC COMPARE output
--  COLUMN differences found.  See PROC COMPARE output
******** Snapshot Comparison Completed ********
```

Output 1 SAS Log from the Example Code

I'll be the first to admit that the PROC COMPARE output (not shown) isn't perhaps the most obvious thing to read to discern what the differences are, but for a tool to solve the problem of finding the problem, it's good enough for my toolbox. Note also that if no differences are found for a given subject area, an appropriate message is issued in the log.

IMPLEMENTATION DETAILS AND NUANCES

The better you understand how a tool works, the better you can use it. There are a few key elements to what made creating %SNAPSHOT relatively straightforward:

1. PROC SQL Dictionary Tables
2. SAS Generational Data Sets
3. Emulating the OBS=0 Input Data Set Option in PROC SQL
4. An Interesting Use of the CASE Structure in PROC SQL

PROC SQL DICTIONARY TABLES

The information from each snapshot is provided through the PROC SQL dictionary tables. These tables, which have analogous views in SASHELP, provide a wealth of information about the current state of the SAS environment. While there are numerous dictionary tables, covering numerous aspects of the environment, I only used information from the following tables:

Dictionary Table	What information is gathered from this table?
DICTIONARY.MACROS	Contains information about currently defined macro variables
DICTIONARY.OPTIONS	Contains information about SAS system options
DICTIONARY.TABLES	Contains information about tables in all established librefs
DICTIONARY.COLUMNS	Contains information about all columns in all tables in all established librefs

Table 1 PROC SQL DICTIONARY TABLES USED

A special note is in order about the DICTIONARY.OPTIONS table. The setting for some options can be a lengthy string; as such, this table "chunks" the setting value into records for each 1024 bytes of setting information. For most options (such as NOTES) there is just a single record, but for lengthier options such as PATH there are multiple records. The OFFSET field is used to assemble these records.

There have been many SAS Global Forum papers written about using DICTIONARY tables, so if you are not familiar with them, I'd encourage you to read the documentation and a couple of the papers. I think you will find them incredibly useful.

SAS GENERATIONAL DATA SETS

Each time that a snapshot is recorded in the snapshot library, a new table is created to hold the current information for each type of data (options, macro variables, data sets, and columns). Because this tool strives to stay simple, I was looking for a way to keep track of these snapshots and not have to manage a bunch of table names. I also knew that allowing the name of the snapshot to be used to name the table would make this even harder.

I was exuberant (really!) to find that SAS has for a long time now supported "generational" data sets. A generational data set allows a base data set name to be used, with a defined maximum number of generations. Each time that a data set is overwritten, a new generation is instead created. The following code shows an example of this:

```
/* first time so define GENMAX */
data work.genExample(genmax=2);
  a = 1;
run; /* generation 1 created */

data work.genexample;
 b = 2;
run; /* generation 2 created */

data work.genexample;
 c = 3;
run; /* generation 1 is now gone */
data _null_;
  set work.genexample;
  put _all_;  /* displaces C=3 */
run;
```

When the maximum number of generations are reached, the oldest table in the generation data group is deleted. Generations can be referenced by absolute generation number or relative generation number.

EMULATING THE OBS=0 INPUT DATA SET OPTION IN PROC SQL

Because I was trying to fix a real problem and not make a career out of creating or maintaining a tool for personal use, I had to strike a balance between which features I felt were necessary and what I was willing to live with. When the number of libraries that I was attempting to track became unwieldy, I opted to support an _ALL_ value for the LIBS= parameter. When I was only interested in tracking options and macro variables, I found it helpful to also support a value of _NONE_. But what I still wanted was an empty table for tracking data sets and columns so that my generation tracking would be simple. To state that another way, I wanted a snapshot to map to a generation number, even if no libraries were being tracked by the LIBS= parameter.

I was unhappy to learn that while the OBS=0 input data set option can be used in PROC SQL, it does not work when applied to one of the DICTIONARY tables! This required a bit more rudimentary creativity, whereby I essentially guaranteed no rows satisfied the WHERE clause:

```
create table &snaplib..COLUMNS(&_genClause label='Column Snapshots')
      select libname, memname, name, type, label, format, informat, sortedby
      from dictionary.columns
      where memtype eq 'DATA'
      %if &_libs eq %str(_NONE_) or &_libs eq %str() %then %do;
            %* cheezy way to emulate obs=0;
            and 0
      %end;
      %else %if &_libs ne %str(_ALL_) %then %do;
            and libname in
            ( %_buildInClauseList(string=&_libs) )
      %end;
order by libname, memname, name ;
```

AN INTERESTING USE OF THE CASE STRUCTURE IN PROC SQL

The danger of writing a paper about good programming practices, and then providing code, is that the code better darn well measure up to the suggested standard. As such, early in the macro, specific options are queried for their current settings, and then set to appropriate values to minimize noise in the SAS log:

```
%************************* ;
%* Preserve user settings * ;
%************************* ;
%let _notes   = %SYSFUNC(getoption(notes));
%let _source  = %SYSFUNC(getoption(source));
%let _source2 = %SYSFUNC(getoption(source2));
%let _mprint  = %SYSFUNC(getoption(mprint));
%let _mlogic  = %SYSFUNC(getoption(mlogic));
options nonotes nosource nosource2 nomprint nomlogic;
```

At the end of the macro, these settings are restored (and a macro that was defined in-stream is also deleted, in accordance with good leave-no-trace practice):

```
%sysmacdelete _buildInClauseList / nowarn;
options &_notes &_source &_source2 &_mprint &_mlogic;
```

Thus far this all is in line with the practices being espoused. But in the midst of the macro is where things got interesting. When the current OPTIONS settings are recorded in the newest generation of the OPTIONS data set, it is essential that the options we just preserved are reflected. One might naively attempt to capture the options and then update the necessary ones in a separate PROC SQL UPDATE statement or a DATA step. However, doing so would create yet another generation of the OPTIONS table, and that just quickly becomes a mess.

Instead, this is a time to show off the CASE statement in PROC SQL. I have found that most of the SAS programmers I've met aren't even aware of this, and even fewer have used it in the fashion shown below, which accomplishes both the capture of data needed, while also setting specific values for certain rows:

```
%************************************************************************ ;
%* Capture the OPTIONS - handle special case of options set by this macro * ;
%* Also include OFFSET, as option values are chunked by 1024-characters   * ;
%************************************************************************ ;
create table &snaplib..OPTIONS(&_genClause label='Option Snapshots') as
      select optname,
            case optname
                  when 'NOTES'   then "&_notes"
                  when 'SOURCE'  then "&_source"
                  when 'SOURCE2' then "&_source2"
                  when 'MPRINT'  then "&_mprint"
                  when 'MLOGIC'  then "&_mlogic"
                  else setting
            end as setting, level, offset
      from dictionary.options
      order by level, optname, offset ;
```

If you have not used the CASE statement in PROC SQL, I would encourage you to give it a shot. You can never have too many good tools in your toolbox.

CONCLUSION

As code complexity and interaction increase, the ability to quickly pinpoint what changed when a problem is detected is critical to business continuity, productivity, and competitiveness. Adopting a "leave no trace" coding practice allows detrimental side affects to be reduced. In addition, by leveraging environmental state information and tooling to manage snapshots, you can gain the upper hand in finding code that has created a downstream problem.

REFERENCES

Rosenbloom, Mary F. O. and Kirk Paul Lafler. 2012. "Best Practices: Clean House to Avoid Hangovers" *Proceedings of the SAS Global Forum 2012 Conference.* Cary, NC: SAS Institute Inc. http://support.sas.com/resources/papers/proceedings12/049-2012.pdf.

Madduri, Kavitha. 2011. "Macros to Help You Clean Up!" *Proceedings of the PharmaSug 2011 Conference.* Nashville, TN. https://www.pharmasug.org/proceedings/2011/CC/PharmaSUG-2011-CC05.pdf.

Bininger, Chuck. 2009. "Proper Housekeeping – Developing the Perfect "Maid" to Clean Your SAS® Environment" *Proceedings of the SAS Global Forum 2009 Conference.* Cary, NC: SAS Institute Inc. http://support.sas.com/resources/papers/proceedings09/082-2009.pdf.

Redner, Ginger, Liping Zhang, and Carl Herremans. 2007. "Techniques for Developing Quality SAS Macros" *Proceedings of the PharmaSUG 2007 Conference*, Denver, CO. https://www.lexjansen.com/pharmasug/2007/tt/TT10.pdf.

ACKNOWLEDGMENTS

The author would like to thank Juan Carlos Ortega and Angela Hall of SAS Institute for their support and encouragement to write this paper, and John West of SAS Institute for an incredible job of editing.

RECOMMENDED READING

- The %SNAPSHOT macro discussed in this paper is available for download at https://github.com/sascommunities/sas-global-forum-2020/tree/master/papers/4336-2020-Sommer
- *SAS® 9.4 SQL Procedure User's Guide*

  Accessing SAS Information By Using DICTIONARY Tables
  Available at
  https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=sqlproc&docsetTarget=n02s19q65mw08gn140bwfdh7spx7.htm&locale=en#n0ts7s3wzdpu1gn18faqdg9d6s5e
- *SAS® 9.4 Language Reference: Concepts*

  Understanding Generation Data Sets

  Available at
  https://go.documentation.sas.com/?docsetId=lrcon&docsetTarget=p0apy93gsj2bzmn1awyqfe7kklkp.htm&docsetVersion=9.4&locale=en
- For a fuller discussion of the Seven Principles of Leave No Trace, see https://lnt.org/why/7-principles/

CONTACT INFORMATION

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

Carl Sommer
SAS Institute Inc
Carl.Sommer@sas.com