

The 7 Habits of Highly Effective SAS-ers

David L. Cassell, Design Pathways, Corvallis, OR

ABSTRACT

SAS® programmers develop habits over time. Some of those habits lead toward efficient, effective coding. Other habits lead toward annoying headaches and a constant feeling that SAS can't do anything without a lot of struggle. In this paper, we will look at seven general habits which lead toward effective SAS coding, as well as toward efficient SAS programs. We will discuss: ways in which understanding your problem can help improve your code; how to find and use the right tool; how to improve efficiency by looking for ways to reduce the number of steps; how to structure data sets so that SAS programs can use your data more effectively; how to avoid the more expensive operations in programs; and how to learn more about the aforementioned subjects.

INTRODUCTION

In several decades of SAS programming, and several years of participation on SAS-L, I have seen a lot of different programming styles. A variety of programming 'movements' have come since I first started programming back around 1970, and many of them are long-gone. Some of these programming movements have useful contributions for SAS programmers.

A lot of people program everything as if they were still coding in C, or Java, or Visual Basic, or R. Basic computer science knowledge is generally transferable across programming languages. But some programming concepts which fit in the model of a C programmer or an Eiffel programmer may not be as effective as concepts which are geared for the SAS System.

As an aside, lots of businesses have programming guidelines that their employees must use. These often cover details like the directory structures the employees must use, the software revision systems that are required, the internal macro programs that programmers are to use, and standard LIBNAME statements that have to be in every program. Programming aspects like this change from place to place, and the typical programmer has no say in these decisions. So details like these are outside the scope of this paper.

What this paper will address is some basic concepts that every SAS programmer can use. All of these are general enough to serve as guides without also turning into straitjackets.

THE SEVEN HABITS

The seven general concepts we will cover in this paper are all aimed at SAS programmers. Some of them are specific to features of the SAS System. But they all have utility in designing code, writing code, and evaluating code. They all can help you move toward more effective SAS programming. The "Seven Habits" are:

- ✓ Know Your Problem
- ✓ Use the Right Tool
- ✓ Fewer Steps Get You Farther
- ✓ Stay Tall and Thin
- ✓ Too Much of a Good Thing Is Bad
- ✓ Skip the Expensive Stuff
- ✓ Sharpen the Saw

But these names don't tell us what we need to know. So let's look at each of these habits and see what we can glean for our own uses.

KNOW YOUR PROBLEM

In order to figure out how best to approach a problem, we often need to know more than the basics. A typical problem that might be posted in SAS-L is something like:

"I need to get data from over here and put it together with data from over there. It didn't work."

Clearly, no one except The Amazing Kreskin could solve this coding problem! In the real world, we have more information than that. But do we know enough to make effective decisions?

In order to make the right decisions, we often need to know a lot more about the data. We have to ask questions. What is the scope of the problem? How big are the data sets, and how many records do we have, and how many variables? Where do the data come from, and how does that affect how we design our processes? What can we learn from the meta-data? What is the purpose of this task?

When we look at accessing a data set, we may need to know a lot more than just its LIBNAME and the name of the data set. We may need to consider what proportion of the data we will need to read. We may need to know whether the data are local, or located across a network, or part of an external Relational Database Management System (RDBMS) like Oracle or DB2. We may even need to know what the consequences are if the process takes a really long time to execute: if there are no consequences, then we hardly need to work at improving things; if the process takes ten times longer than our business case allows, then we need to fix it instantly, if not sooner.

If we are performing a complex data manipulation with many data sources and huge data files, these sorts of questions should be obvious. Programmers have grappled with these kinds of questions since the previous millenium. But we may need these kinds of questions in everyday uses of SAS.

One of the simplest things we do in SAS is descriptive statistics. So let's take our data set and compute a median. That's trivial, right? We could use PROC UNIVARIATE or PROC MEANS or PROC SUMMARY or PROC TABULATE or... Well, SAS can compute medians in lots of different procedures. So it doesn't matter which procedure we choose, or what our data are like, right? A median is a median is a median, to paraphrase Gertrude Stein.

Unfortunately for some of us, that's wrong. Computation of a median requires either: a method to read in all the data points, order them, and find the 'middle' value from the ordered list; or else a sophisticated statistical approximation. If your data set is tens of millions of records long, PROC REPORT or PROC TABULATE may fail when trying to compute the median of all values of some variable. On the other hand, PROC MEANS and PROC SUMMARY can still succeed. They have access to that 'sophisticated statistical approximation' I mentioned. That means that you can still use PROC TABULATE or PROC REPORT, but you will have to pre-process your data first to get the summary statistics. So knowing our data is important, even for what should be the simple stuff, like putting a median in your output table.

The classic example of "Know Your Problem" is the merge or join. Suppose we have two data sets, one big and one small, that we need to merge on some key variable(s), so we can add satellite information from the small file to the big file. Now we really need to know a lot about our problem in order to decide what is a good choice of method. How big are the two data sets? Are they both SAS data sets? How many key variables do we have to deal with? Are the data sets already sorted on the key variable? Are the data sets too large to sort, or on tape, or living in some external RDBMS, or otherwise impossible to sort? How many satellite variables do we need to merge into the larger data file?

If we are told that the two data sets are in SAS, and are not sorted on the sole key variable, and that the data sets could be sorted if we really had to, we still do not know what is going to be best for our purposes. It depends on knowing more about our problem.

If the large data set is 600 records, and the small data set is 6 records, and there is only one satellite variable, then our approach just doesn't matter. Nothing we do will take an appreciable amount of time. We could write the small data set out by hand and use IF-THEN-ELSE statements or a SELECT statement if we wanted to. We could create a user-defined format using the smaller data set, and then apply the format to the larger data set. We could do the old standby: sort both data sets and use the MERGE statement in a DATA step. We could do the same thing using a join in PROC SQL. We could use key-indexing. We could build a hash. If the values of the key variable are numeric and in a reasonable range, we could even build an array to hold the values of the satellite variable, and then use that array as a look-up table.

On the other hand, if the large data set is sixty million records and the smaller data set is six thousand records, then some of the above options no longer look appealing. We certainly would not want to write six thousand IF-THEN-ELSE statements, or a 6002-line SELECT statement. (I would not want to write a macro to generate such a thing, either. Think about how long it would take the resulting SAS code to execute, when it would need to walk through the logic of 6000 lines of IF-THEN-ELSE for your records!) And a sixty-million-record data set is likely to be too time-consuming to sort, assuming you even have the disk space for that sort.

For 6000 records in the smaller data set and one or two satellite variables, the use of the CNTLIN= option of PROC FORMAT to build a user-defined format still looks reasonable. However, if we have 6000 records in the smaller data set and 20 or 30 satellite variables, we might look for another option. At 6000 records, the data set is likely to be small enough that PROC SQL would perform the join by internally hashing the smaller data set and then joining the satellite data to the larger file in a single pass of the large data set.

Then let's suppose that the smaller data set jumps up to, say, 600,000 records. Now PROC SQL may not be able to optimize this with a hash because it does not have enough buffer space. But we can still use a hash - which is nothing more than a look-up table that allows for really fast searching. In SAS 9, the DATA step hash object gives us the ability to perform hashing on some fairly large data files, with really fast look-ups. And the SAS 9 hash does not have to fit in RAM, so it can grow to considerable size if you need it to do so.

Let's look at a simple example of a DATA step hash. We'll use one key variable, KEYVAR. It will allow us to link our satellite data that resides in the data set SATELLITEINFO. The satellite variables will be called SATVAR1 and SATVAR2. We want to add those data values to the data set BIG_MAIN_FILE .

The basic premise of the DATA step hash is straightforward: we create the hash with a DECLARE statement; we define the hash with <HASH>.DEFINEKEY and <HASH>.DEFINEDATA statements; we fill the hash, usually with a do-loop to read the data in; and then we use the hash however we want. Here, we'll use two DOW-loops (Whitlock do-loops that let us step through the data set explicitly instead of using the implicit loop of the DATA step) within one DATA step.

```
data merge_by_hash(drop=rc);

    /* create the hash H */
    declare hash h ();

    /* define the contents of the hash H */
    length keyvar satvar1 satvar2 8;
    rc = h.DefineKey ( "keyvar" );
    rc = h.DefineData ("satvar1","satvar2");
    rc = h.DefineDone ();

    /* load the hash with the satellite info, using a Whitlock do-loop */
    do until(eof1);
        set satelliteinfo end=eof1;
        rc = h.add (); *add key-data pairs into the hash;
    end;

    /* use the hash to load data into the main file, using a Whitlock do-loop */
    do until(eof2);
        set big_main_file end=eof2;
        satvar1=. ; satvar2=. ; *set to missing, in case there is no match;
        rc = h.find (); *if we have a match, retrieve the satellite variables;
        output;
    end;

run;
```

If you have never seen a Whitlock do-loop (or DOW-loop) before, look under "Use The Right Tool" for more explanation and more ways to use it. It is a simple, valuable way to walk through a SAS data set without using the implicit loop of the DATA step. That gives us more flexibility than we normally have with the DATA step.

Prior to SAS 9, you can still use a hash for this, but you would have to code it yourself: Paul Dorfman has written several SUGI papers explaining how to do this, so there is already-developed code that you can adapt. Check the references at the end of this paper for URLs to these papers. And, if it turns out that you want a DATA step hash, but it is still a little too slow, moving to hand-crafted hashing code borrowed from Paul Dorfman will speed things up some - with the disadvantage that your code becomes a lot less general, and only fits your specific situation.

So we have a host of possible tools, and our selection should depend on our knowledge of the data and meta-data. In many cases, we need more information than the size and scope of the data, along with the meta-data. We may need information on the data sources and other data facets, as well.

But we probably also need to know about the overall problem, and the business case or project that drives the effort. Consider a simple statistical question as an example of this idea. Someone comes to us and says:

I have a big data set and I want to estimate Y from all my regressors. Is PROC REG the right choice for my data?

Unfortunately, there is no way to tell whether PROC REG is the right procedure for our questioner - not from the given information. But it's likely to be the wrong procedure. Why? Because there are a lot of assumptions underlying the statistical analyses of PROC REG. In the real world, we often have to start with one procedure before we can analyze our results and decide on a better procedure. So let's look at some common cases, and what we could consider using in those cases:

Outliers or leverage points: PROC ROBUSTREG or PROC LOESS or
Non-normal, but continuous Y: PROC TRANSREG or
Repeated measures data: PROC GLM or PROC MIXED or PROC GENMOD or
Mixed models: PROC MIXED or PROC NLMIXED
Discrete Y: PROC FREQ or PROC CATMOD or PROC LOGISTIC or PROC GENMOD or
Nonlinear fits: PROC TRANSREG or PROC NLIN or PROC NLMIXED or
Time series data: PROC ARIMA or PROC AUTOREG or PROC MODEL or
Survival analysis data: PROC LIFEREG or PROC PHREG or
Data from survey samples: PROC SURVEYREG or PROC SURVEYLOGISTIC
Data from a QC response surface: PROC RSREG
Ill-conditioned data due to high multi-collinearity: PROC ORTHOREG or
Data with measurement errors: PROC CALIS or

And there are more. The right tool might be PROC GAM or PROC PLS or PROC GLMSELECT. Or, upon hearing the details of the problem, we might even conclude that regression is really the wrong thing to be doing. But we can't decide that until we **Know the Problem**.

USE THE RIGHT TOOL

Now that we are busy learning everything we can about our data and our problem, we can't overlook the need to learn more about SAS. SAS is a very 'rich' environment with a host of arcane procedures and a wide variety of available functions. We can't know everything there is to know about SAS. But the more we know, the more tools we have in our toolbox.

Let's consider a question that crops up on SAS-L in one form or another with a fair amount of regularity. *"I need all possible combinations of 4 of our 6 cases: X, Y, Z, W, T, and U."*

A SAS beginner might look at his bag of tools and decide that the way to tackle this is to write a macro. Or perhaps he would first try writing this as a DATA step problem:

```
data comb4of6;
  array c{6} $ ('x' 'y' 'z' 'w' 't' 'u') _temporary_ ;
  do i = 1 to 3;
    do j = i+1 to 4;
      do k = j+1 to 5;
        do l = k+1 to 6;
          val1=c{i};
```

```

        val2=c{j};
        val3=c{k};
        val4=c{l};
        output;
      end;
    end;
  end;
run;

```

Okay, this works. But it is not flexible. It requires some thought about the proper indices for the do-loop variables. It will be miserable to extend to a more complex case, like combinations of 30 cases, 22 at a time. All those problems can be solved with a sufficiently flexible macro. But that macro will be time-consuming to build properly, and to test completely. Consider what that macro will have to do. It will have to: create a correct ARRAY statement; build K nested do-loops with the correct lower and upper bounds; create the correctly-named variables; add on an OUTPUT statement and the right number of END statements; and clean up after itself. Okay, that's all manageable. But is it necessary?

So what would you do? Well, if your bag of tools were sufficiently large, you would be able to reach in and find PROC PLAN. PROC PLAN is a procedure in the SAS/STAT module, and it helps in building a variety of experimental designs. Where it really becomes helpful is when you discover that it will let you choose the sets of levels using permutation selection or *combination* selection. So here is a solution to this problem using PROC PLAN:

```

proc plan;
  factors block=15 treat=4 of 6 comb;
  output out=MyCombs cvals=('x' 'y' 'z' 'w' 't' 'u');
run;

```

The FACTORS statement lets us specify the number of blocks, and we can compute that $6C_4$, the number of possible combinations, is $COMB(6,4)=15$ so we know there are only 15 distinct combinations of 4 out of 6 cases. (We can also use the COMB() function to find out that 30 cases, taken 22 at a time, would be 5,852,925 different combinations to handle. Ick.) The TREAT= option (that's 'TREAT' for 'treatment') lets us specify the form of the combinations we want, and the COMB option lets us get the combination selection. The CVALS= option in the OUTPUT statement lets us label the values in the combinations with something other than the numbers 1 through 6. So there is a very straightforward, very efficient, very extensible approach. If we choose the right tool first.

Or consider another example. Suppose we need to create XML output that is the display of three variables from our data set PHARM.FETCHED. We could sort on our key variables and then use a DATA_NULL_ step to create the XML file:

```

data _null_;
  set SortedData end=eof;
  by keyvar1 keyvar2;
  file "../MyThreeVars.xml";

  length row1 row2 data $12;
  row1 = trim(left(put(keyvar1,best12.)));
  row2 = trim(left(put(keyvar2,best12.)));
  data = trim(left(htmlencode(info)));

  if _n_ = 1 then put @1 "<all needed XML tags go here>" / @1 "<Table>";
  if first.keyvar1 then put @3 "<keyvar1 row="" row1 +(-1) "">";
  if first.keyvar2 then put @5 "<keyvar2 row="" row2 +(-1) "">" @;
  put "data=" "" data +(-1) "" @;
  if last.keyvar2 then put "</keyvar2>";
  if last.keyvar1 then put @3 "</keyvar1>";
  if eof then put @1 "</Table>";
run;

```

But this takes a lot of coding. It takes a lot of hardcoding of variables and XML structures. It will be awful to alter if we have to change the style of XML output to match a different format. We could make this more flexible by writing it all as a complex macro that generates the statements, or by writing an SCL application. Still...

Surely this would be easier if we just used the right tool. Since SAS now provides an XML destination as part of the Output Delivery System (ODS), we don't have to write the XML tags ourselves. All we have to do is use the ODS MARKUP statement to write to an XML file, and then print off the variables in the data set.

```
ods markup file='../MyThreeVars.xml';

proc print noobs data=SortedData;
  var keyvar1 keyvar2 info;
run;

ods markup close;
```

Since SAS provides a wide variety of tagsets for XML output, this approach seems easier, and simpler to document, and less susceptible to typos. And you can change the style of the XML output as simply as changing the value of the XMLTYPE= option.

We mentioned the DOW-loop (also known as the Whitlock do-loop) in passing, a couple sections ago. Let's look at it now, as one more Right Tool to have in your toolbox. Suppose someone comes to you with a large database of university teachers. There are some problems with the data, and the one you are asked to help on involves the DEGREE_FROM variable. It may legitimately be blank. But once it is filled in, it should remain constant. So in this example subset of the data, the first line should not be changed, but the last line needs to have DEGREE_FROM filled in from the previous record.

ID	Term	Rank	Degree_From	VarA	VarB	VarC
12345	2001	Assist	.	abc	def	ghi
12345	2002	Assist	NCSU	abc	def	Zhi
12345	2003	Assist	NCSU	abc	def	ghi
12345	2004	Assoc	NCSU	abc	def	ghi
45678	2001	Assist	Chicago	Xbc	def	ghi
45678	2002	Assist	Chicago	abc	def	ghi
45678	2003	Assist	Chicago	abc	def	ghi
45678	2004	Assoc	.	abc	def	ghi

This could be done with LAG() and checking FIRST.xx and LASTY.xx variables. It could be tackled with RETAIN, with similar checking. But the DOW-loop provides a natural way of handling the problem. Consider this code, which SAS-L Hall of Famer Howard Schreier provided for a poster on SAS-L :

```

data filled (drop=prevDF);
  do until (last.id);
    set YourData;
    by id;
    if missing(degree_from) then degree_from = prevDF;
    output;
    prevDF = degree_from;
  end;
run;

```

The DO UNTIL loop is our DOW-loop. It explicitly reads the data set YourData record by record, until the UNTIL condition is met. The usual re-setting of variables in the PDV (Program Data Vector) only occurs when the code makes an implicit loop through the DATA step. Here, that happens when LAST.ID is 1 and the program drops out of the UNTIL condition to loop back to the top of the DATA step. This means that we can let the code do the initialization and clean-up for us. So here, PREVDF is automatically cleared every time we start a new ID group.

This could be done with other SAS tools, but the DOW-loop gains in utility as we increase the complexity of our code. If we go back to the DATA step on page 3, we can see that using the DOW-loops makes our code a lot more manageable. The DOW-loop is a way of permitting code before and after blocks of data, without cluttering up our DATA step. The more you learn about it, the more things you will find to do with it.

FEWER STEPS GET YOU FARTHER

SAS is a 4GL that is focused on procedure steps and DATA steps. You can generally count on it taking a lot longer to run a big data set through 12 steps than 3 steps. Many times, we can reduce the number of steps used, and as a result, reduce CPU requirements and time needed for the program to run. We may also be reducing programmer effort and maintenance issues. And yet we regularly see people who code like this:

```

data two; set one; keep id--var73;

data three; set two; rename id=NewID;

data four; set three; if NewID=4325;

proc print data=four; var NewID cost ickfactor var73; title 'Ickiness factors for ID 4325';

```

This kind of programming is just adding to code maintenance when the data sets are small. (I won't complain here about the lack of RUN statements, or the combining of multiple statements on each line. But those are problems too.) Still, as the data sets get larger, there is considerable extra time required to run through the entire data set three times, and through the subset of the data set where ID=4325 a fourth time as well. There is also the problem of extra storage space required for multiple copies of the original data set. As data sets grow, this becomes more and more of a problem. Surely this could be done using data set options instead:

```

proc print data=one (
  keep=id cost ickfactor var73
  rename=(id=NewID)
  where=(NewID=4325) );
var NewID cost ickfactor var73;
title 'Ickiness factors for ID 4325';
run;

```

If it takes a microsecond to read the data set, this does not save us much time. If it takes an hour to read the data set, then we have a very different situation! Once again, it helps if we know our problem.

A subtle point here is the order of the data set options. How do we know whether the WHERE clause needs to use ID or NEWID? There is a useful mnemonic we can use. The data set options DROP, KEEP, RENAME, and WHERE are performed in an order that happens to be alphabetical. That makes it easy to remember.

Okay, that was a really obvious application of the idea that fewer steps get you farther. Surely we have some better examples out there. Here's a common type of problem:

"I have to read in an entire directory of identically-structured text files, all with .txt extensions. How do I do this?"

Now our first thought may be that a macro would be vastly preferable to a huge number of hardcoded data steps. That's certainly true. But a macro would end up generating as many DATA steps as there are files. Is there a simpler approach that gets us farther in a single data step? Of course.

```
data AllTxtFiles;
  infile 'C:\YourDirectoryPath\*.txt' <any other needed options> ;
  input <your list of variables here> ;
run;
```

Now what could be easier than that? And how much easier is this to maintain, or to adapt, than our earlier ideas?

But we do not always want all the files that look like *.txt . What do we do if we only want a specific set of, say, three of these files to be read in? Well, we still do not need to write that macro. Instead, we just use the FILEVAR= option of the INFILE statement. Note that we use INPUT to read in the name of the file, which we pass to INFILE. Then we use a DOW-loop to read in the entire contents of the file before we move to the top of the DATA step and repeat the process.

```
data SomeTxtFiles;
  length fileloc myinfile $ 100;
  input fileloc $ ;
  infile dummy filevar=fileloc filename=myinfile end=done;

  do until(done);
    input <your list of variables here>;
    output;
  end;

  put 'Finished reading ' myinfile=;
  datalines;
C:\YourDirectoryPath\File007.txt
C:\YourDirectoryPath\File222.txt
C:\YourDirectoryPath\File247.txt
;
run;
```

Another typical example of "Fewer Steps Get You Farther" will appear below, in the section "Too Much of a Good Thing Is Bad", when we see how to replace a large looping construct in a macro with by-variable processing which requires no looping at all. We'll go from an unwieldy looping macro which requires 3001 DATA and proc steps, down to a 3-step program.. which runs a lot faster.

STAY TALL AND THIN

All too often, we end up struggling with arrays or macros or other approaches to deal with the fact that the data set we get is poorly designed. SAS is designed to work with normalized data: data which are arranged into record groups for efficient processing. Normalized data also means that data is stored in only one place, instead of being copied multiple times in multiple rows. And it means that important data is held in variables in our data sets, instead of stuck in the variable *names*, where it is hard to use.

In SAS programming, we want the data sets to be narrow and long, instead of wide with lots of information strung out along each record so that it is really hard to work with. We also want meaningful variable names which are not storing important data as part of the names. That just makes data manipulation that much harder. Here's an example of this:

"We just heard yesterday about this Y2K thing. We have a data set with IDs and three years of monthly costs named VAR1 to VAR36. We have our missing values for these variables stored as 999 or 9999. We want to change all the 999 values to a missing value .A and all the 9999 values to a missing value .X. Do we need a macro or an array?"

The better answer is not to store the data like this at all! If we leave the data like this, we will forever be struggling to find the right month in our processes. We will have to process data across these rows by hand, instead of down columns as SAS procedures are designed to do. We will need more chunks of complex code every time we need to make any change or manipulation of the data.

Instead, let's just use PROC TRANSPOSE to normalize the data, and then we can solve the original problem with a very simple DATA step.

```
proc transpose data=OurWideData out=OurThinData(rename=(col1=MonthlyCost));
  var var: ;
  by ID;
run;

/* at this point, our data have variables ID, MonthlyCost, and _NAME_ , with the
MONTH information still buried in the variable _NAME_ . Let's first pull out the
MONTH numbers 1 to 36 from _NAME_ , and then fix the missing values. */

data OurFixedData(drop=_NAME_);
  month = input( substr(_NAME_,4) , best8.); /* pick the MONTH off the var name */
  if MonthlyCost = 999 then MonthlyCost = .A;
  if MonthlyCost = 9999 then MonthlyCost = .X;
run;
```

Or let's suppose we have a data set OUR.ICKYFILE with a StoreID variable, and for each store we have variables Cost2003Q1, Cost2003Q2, . . . , Cost2006Q4 . What if we need to increase all second-quarter costs, starting with the second quarter of 2005, by 14% ? Now we have a real headache, because the year and quarter data are trapped in the variable names, instead of where data ought to be.

And this problem is endemic with this kind of data structure. Any time we need a particular quarter's data, we can't just use a WHERE clause and subset the file. We would have to hardcode the variable name, or else we have to cram everything into macros that take the variable names as parameters. But we wouldn't need these macros if the data were designed properly.

Any time we need to analyze the data, we will need it in a tall-and-thin structure anyway: SAS procedures are designed to work with data in this kind of structure. If you think that your data need to be in a short-and-wide structure for a SAS procedure, then you are probably doing the wrong thing. If you are keeping your data in a short-and-wide structure for reporting purposes, then you need to learn about PROC REPORT and PROC TABULATE. ("Use the Right Tool", please.)

By making the data tall and thin (instead of short and wide, as it is now), and moving the data out of the variable names, we make our future processing much easier. Which, in turn, makes our lives easier. Don't forget that one kind of 'efficiency' is 'programmer efficiency'. Spending one hour getting your database into a functional structure is much nicer than spending an entire week every month writing and debugging macro code to handle the next sets of changes and output requests because the database is such a nightmare to work on.

It is easy enough to normalize OUR.ICKYFILE using PROC TRANSPOSE, and then to pull the year and quarter out of the _NAME_ variable:

```
proc transpose data=our.ickyfile out=trans1(rename=(col1=cost));
  var cost: ;
  by storeid;
run;

data newfile;
  set trans1;
  year = input( substr(_name_, 5,4), 4. );
```

```

qtr = input( substr(_name_,10,1), 1. );
run;

```

But it is almost always easier to maintain date information in SAS date variables, instead of strings or (in this case) separate year and quarter values. So here is an alternative version of the above DATA step:

```

data newfile;
  set trans1;
  date = input( substr(_name_,5,6), yyq7.);
run;

```

At this point, the year and quarter information are available from the DATE variable with SAS functions. And we can format the results to make the display of the data look any way that we require. So now we're ready to tackle the original problem. But at this point, the problem is simple. We have the date information in the DATE variable and the cost information in a COST variable. All we have to do is use SAS date functions to pull the year and quarter information out of the date, and use that to choose which costs to modify. Our complete code becomes:

```

data fixedfile;
  set newfile;
  if year(date) >= 2005 & qtr(date) = 2 then cost = cost * 1.14;
run;

```

Isn't that a lot simpler than fighting with the variable names every time? The moral of the story is simple. Never store data in variable names, and keep your data normalized as long as it is in SAS.

TOO MUCH OF A GOOD THING IS BAD

Lots of times, the approach to a SAS problem will be a section of SAS macro code. This can be a good thing. The SAS macro language gives us ways of automating complex tasks. It gives us ways of making our process flow conditional on our data or our situation. It gives us ways of passing data across SAS steps. It gives us ways of dynamically creating code based on user requirements. It gives us ways of creating code which is more general, and more extensible. These things are all good.

Unfortunately, the most abused feature of SAS is also the SAS macro language. Too many people use macro code to re-invent the wheel when SAS already has tools designed to perform the tasks faster and simpler. (These people need to "Use the Right Tool".) Too many people write macros to generate unnecessary repetitive code which can take much longer to run than tools that are already a part of SAS. (These people should remember that "Fewer Steps get you Farther".) And too many people use macro code, but do not use it at the right level of their process. (These people need to be reminded that they should "Know Your Problem".)

A good example of this problem is a tool like bootstrapping. Bootstrapping is a statistical tool in which we re-sample the sample data in such a way that we can estimate the mean and variance of estimators that are too complex for us to do the computations analytically, or which have deviations from the underlying assumptions on which those analytical formulas depend. Even if you are not a statistician, you run the risk of having a statistician ask you to implement his or her code, in order to get statistical estimates. There is a %jackboot macro distributed by SAS, but few people know about it. The typical hand-coded bootstrapping macro looks something like this:

```

%macro boot( input=YourData, reps=1000 );
  %do i = 1 %to &REPS;
    data repdat;
      do i=1 to nob;
        rec = ceil(nob * ranuni(0));
        set &INPUT nob=nob point=rec;
        output;
      end;
    run;

    proc mixed data=repdat;
      /* proc mixed code here */
    run;

    proc append base=repall data= OutFromProcMixed; run;
  %end;

  proc univariate data=repall;
    var YourParm;
    output out=final pctlpts=2.5, 97.5 pctlpre=ci;
  run;
%mend;

```

But 1000 replicates here, which is not uncommon in bootstrapping, will require 3001 separate steps: 1000 DATA steps, 1000 calls to PROC MIXED, 1000 calls to PROC APPEND, and the PROC UNIVARIATE at the end of the process. At a minimum, you need to run this in batch mode instead of from the Display Manager, because you will have to worry about this flooding your log and output windows, locking up the system with pop-up messages until you clear the windows, over and over again. Even if you run this in batch mode, you will have a mammoth log to sift through for problems. And the process may take hours, or even days, to complete.

Or you could build a tall-and-thin data set of all the replicates, and run the whole thing through PROC MIXED using by-processing. (Remember to "Stay Tall and Thin".) And you could simplify the data generation by using the SAS/STAT procedure PROC SURVEYSELECT instead. (Remember to "Use the Right Tool" too.)

```

sasfile YourData load;
proc surveyselect data=YourData out=outboot
  seed=30459584
  method=urs samprate=1 outhits
  rep=1000;
run;
sasfile YourData close;

ods listing close;
proc mixed data=outboot;
  by Replicate;
  /* same proc mixed code here, and this time OUTALL has all by-levels */
run;
ods listing;

proc univariate data=outall;
  var YourParm;
  output out=final pctlpts=2.5, 97.5 pctlpre=ci;
run;

```

PROC SURVEYSELECT is designed to build common sampling designs, and this takes advantage of the fact that a bootstrap sample is the same as K independent replicates of a 100% URS (Unrestricted Random Sample, or Simple Random Sample With Replacement). So the one procedure builds a single data set OUTBOOT as 1000 replicated bootstrap samples, with a variable REPLICATE as the by-variable you can use to work with the samples individually.

The SASFILE statement is another tool you should learn about. SAS tries to buffer your data automatically. Often, the internal buffering is good enough for our purposes. But SASFILE will specifically load your data set into RAM for faster access – as long as your data will fit in RAM.

The MIXED procedure here, with its BY-statement, will run faster than 1000 separate calls to PROC MIXED. In some circumstances, it will run several orders of magnitude faster. The ODS LISTING [CLOSE] statements simply turn off the output window until the procedure is complete, and then restore it. This is not essential, but the relevant data are all output to the OUTALL data set, so the extra output is not needed. The above code is simpler to debug and maintain than the macro. More importantly, it will run faster.. sometimes orders of magnitude faster, depending on the problem and the data set size and the procedures used.

Let's consider another example in this category. Someone comes to us and tells us:

We need to take our huge database TRAFE.PORK and change every variable name from xxx to P_xxx in order to meet the new corporate naming standards.

Whether this is a bad idea from a database management standpoint is outside our control. We still have to do it. And there are several ways we could do it. We could get the list of variable names and write a macro to run a sequence of RENAME statements. But a lot of the time, when the process is data-driven and there already is a data set with the information in it, there are better options. CALL EXECUTE is one such option, but we won't discuss it in this paper. Another choice is using PROC SQL with the INTO: clause to build macro variables to do all the work for us.

SAS now provides dictionary tables that are meta-data, holding all the information that we want to know about libnames, data sets, variables, indices, macro variables, etc. The DICTIONARY.COLUMNS file is accessible from PROC SQL , and provides details on each of the columns (variables) in all the data sets in all the libnames. (A view of this is available through the DATA step, if you're interested in experimenting with this.)

```
Proc sql noprint;
  select Name || ' = P_' || Name into : VARS separated by ' '
  from dictionary.columns
  where Libname='TRAFE' and Memname='PORK';
quit;

Proc datasets lib=trafe;
  modify pork;
  rename &VARS;
run;
quit;
```

The PROC SQL procedure reads each of the selected records from DICTIONARY.COLUMNS , but this has been restricted to only the variables in TRAFE.PORK by the WHERE clause. So this builds a rename piece for each variable, and puts all these pieces into the macro variable VARS, carefully separated by a space each time. Then the macro variable is ready to be used in a RENAME statement. This is simpler to use and easier to maintain than a large macro which does a similar renaming process.

There are some additional points that this example brings up.

- [1] The SAS dictionary tables like DICTIONARY.COLUMNS may be much slower than using PROC CONTENTS, if some of our libnames point at external data sources, or if the WHERE clause cannot get optimized due to the way it is written.
- [2] A single macro variable may be inadequate if the list of variables is long enough to overflow the 65,534-character length that can be used for a macro variable. PROC SQL can also load the information into multiple macro variables, if that is needed.
- [3] PROC DATASETS can be significantly faster than a data step if the data set is large enough, so that reading in the data takes a long time.
- [4] The choices on these points depend on what we know about our problem.

SKIP THE EXPENSIVE STUFF

Some of the most common things we do in SAS are sorts and merges. Unfortunately, sorts and merges are the most expensive, time-consuming, disk-space-eating tasks we could choose. (Massive repeated looping constructs are one of the other major time-eaters, but we already talked about them. Some statistical and mathematical algorithms are major CPU consumers too, but dealing with them is outside the scope of this paper.) So we need to learn when to sort or merge, and when we can avoid doing so altogether.

Unless your data sets are relatively small, don't sort unless you have to. SAS requires roughly three to four times the data set size in free disk space in order to perform that sort, so the larger your data become the longer it takes to sort the data, and the more problematic a sort becomes.

But people routinely sort their data in order to join it with other data, or to help them select subsets of records, or to use by-processing in later procedures, or any of a dozen other reasons. For many of these tasks, we don't really have to sort the data.

If we are sorting the data in order to join it with other data sets, then we need to think about alternative ways of merging/joining our data. A data step merge or a PROC SQL join can require a lot of time, and a huge amount of disk space. So, as we discussed under "Know Your Problem", we need to look at our process and study our problem before we choose which techniques we will use for joining our data sets. Tools like user-defined formats are a straightforward way of making a single satellite variable available. This has the added advantage that you do not have to store the new satellite data in your already-large data set, thus saving on disk space.

But formats tend to get slower as the cardinality of the format (the number of unique values) increases. This is why DATA step hashes have such a huge value with large data sets. The speed of the look-up is essentially independent of the size of the hash, so the time to look up a matching value is roughly the same whether you have a look-up table of 100 records, or a look-up table of 1,000,000 records.

Even a hash cannot beat a DATA step array for look-up speed. If your keys are integers within a reasonable range – a range small enough that you can build an array in memory – then you can use an ordinary DATA step array for key-indexing.

Let's go back to our hashing example. If we have a key variable that is numeric, say zipcode, and two satellite variables, say latitude and longitude for the geographic center of each zipcode, then we can build an array that provides all the data when we make a pass through our main data set. In this example, we will use a two-dimensional array so we can store all our satellite variables by the same array index. We will also use two DOW-loops, one to load the array, and one to read from the array as we pass through our main file. Note that we are not assuming the variables have the same names between data sets either:

```

data merge_by_keyindex;
  array ll{00001:99999, 1:2} _temporary_ ;

  /* load the array with the satellite info, using a Whitlock do-loop */
do until(eof1);
  set satelliteinfo end=eof1;
  ll{zip,1} = lat;
  ll{zip,2} = long;
end;

  /* use the array to load data into the main file, using a Whitlock do-loop */
do until(eof2);
  set big_main_file end=eof2;
  ziplat = ll{zipcode,1};
  ziplate = ll{zipcode,2};
  output;
end;

run;

```

If our data structures are sufficiently complex, we may even want to think about using user-defined formats to link in some satellite data, while we use hashes or key-indexing to link in larger data tables. SAS gives us a great deal of flexibility in the DATA step and within PROC SQL, so we can choose the methods that help us the most.

If we only need to pull out a subset of the data, then we might prefer to use an index instead. This is a common database management technique, and if you work with Oracle or DB2 or MS SQL Server databases, you may find that these databases are indexed instead of sorted. Indexing works by building a separate file which is sorted in the requested order, with pointers leading back to the relevant records in the original data set. It is simple to build one or more indices in PROC DATASETS. And it takes a lot less time to build an index than it does to sort a data set. But it takes a lot longer to read an entire indexed data set than it does to read an entire sorted data set. The indexing means that your computer can no longer read sequentially from your data set when it pulls out records in your indexed order – your data set is still in its prior order.

A general rule of thumb is that if you are pulling out less than 10% or 15% of your data set for your process, then indexing is going to be faster. The way that you pull out your data will matter too! You should use WHERE clauses instead of IF statements. The WHERE clause is designed to use your index, as long as your WHERE clause is using your index variables and not, say, complex functions of the index variables, or some index variables and some un-indexed variables.

If we are sorting in order to use by-processing in later steps, we should look to see if sorting is really necessary. A number of SAS procedures also accept a CLASS statement, which works like the BY statement but does not need to receive the 'class' variables in sorted order. Instead, it reserves memory and accumulates all the needed information for the entire data set before computing resultants for display.

Or perhaps the data are not sorted, but are grouped by the key variables. Then the NOTSORTED option on the BY statement may be all that you need, instead of having to sort the grouped data. Let's consider our 'missing values in database of university teachers' example that we used to show the features of the DOW-loop. If the data are properly grouped by ID and TERM, but the groups of ID-TERM blocks are not sorted, then we still do not need to sort. Here's our DOW-loop solution, modified for this grouped-but-not-sorted data set:

```

data filled (drop=prevdf);
  do until (last.id);
    set yourdata;
    by id NOTSORTED;
    if missing(degree_from) then degree_from = prevdf;
    output;
    prevdf = degree_from;
  end;
run;

```

Notice that we only made one change: we added the NOTSORTED option into the BY statement. The rest of the process runs unchanged, because the DO UNTIL loops only depend on the data being sorted within a block, not that the IDs are sorted.

If we really do have to sort our data, then we need to think about the type of sorting we need to do. PROC SORT has an option TAGSORT, which is an option to consider when the standard sort requires more disk space than we have to spare. (Remember, the usual sort requires about 3.5 times the size of the data set in free disk space.) The tagsort usually takes longer to run, but it requires less hard drive space. This is because it actually sorts only the sort-keys of the data set, along with a pointer back to the original record that goes with said sort-keys. Then it makes a second pass through the data set, using these pointers to pick out the records in order to get all the remaining variables.

SHARPEN THE SAW

"Sharpen the Saw" means the same thing that it did in the book "The 7 Habits of Highly Effective People". If you want the saw to be useful over time, you have to keep it sharp. For SAS programmers, that means that we need to keep learning about SAS, and we need to keep trying out these new techniques.

The highest-profile source is the SAS Global Forum (SGF), which used to be called SUGI. Not only is there a wealth of knowledge available at each SGF, but the papers from each conference become available at the SAS website. The conference papers are available all the way back to SUGI 22 (1997) at <http://support.sas.com/events/sasglobalforum/previous/online.html>.

In addition to the big SGF conference, there are also regional user groups which have annual conferences. There is not a complete source for papers from all of these conferences, but <http://www.lexjansen.com/> has SUGI papers, NESUG (Northeast SAS Users Group) papers, PharmaSUG papers, and phUSE papers. For more access to papers from your regional conference, try attending one and picking up the proceedings CD. Also, many cities and sub-regions have their own local user groups which have regular meetings with useful presentations.

One of the most useful and little-known resources is SAS-L, the international mailing list for SAS users. This mailing list is connected via gateway to the UseNet group comp.lang.soft-sys.sas, which means that it is also available under Google Groups. The archives of this list are extensive, and cover everything ever posted in SAS-L. That means that the archives cover everything from statistical analyses with SAS to details of formatting reports to communicating with Microsoft Office products. It is a very wide-ranging group, with experts around the world contributing their knowledge on all manner of SAS-related problems. The searchable archives can be found at <http://www.listserv.uga.edu/archives/sas-l.html>.

SAS has its own forums now. These are not a mailing list or UseNet group, nor do they try to replace them. These forums are small discussion areas on specific topics of interest to the SAS Institute people who set up and maintain the forums. As of January 2007, there are only seven forums, covering: ODS and SAS/Base reporting procedures; integration with Microsoft® Office; SAS Enterprise Guide; SAS stored processes; SAS and clinical trials; SAS in health-care fields; and operations research with SAS. The URL for the "User Forums" page is <http://support.sas.com/forums/index.jspa>.

A better-known resource is the SAS Publishing group, which puts out scores of books on SAS programming and SAS applications. In addition to books put out by SAS Institute personnel, there are many helpful Books By Users on a wide variety of SAS-related topics.

The SAS OnlineDoc® is an invaluable resource, with thousands of pages of documentation and example code. It should be available with every installation of SAS, and it is available for free at sas.com as well. But there are other

places where SAS help shows up unexpectedly. Some ODS tagsets have a help option which gives you documentation on the options for that tagset!

So now you can use these resources to learn more about the topics we have discussed in this paper: user-defined formats; hashing and key-indexing; PROC PLAN; the XML destination and tagsets; wildcards in the infile statement and the FILEVAR= option; PROC SURVEYSELECT; the Whitlock do-loop; indexing and WHERE clauses; the SASFILE statement; the NOTSORTED option; and all the other parts of this paper that had you stopping and going "Huh?!?"

CONCLUSION

These 'habits' look easy, but they require effort on your part. You have to make an effort to learn new things. Then you have to learn how to apply them, and you have to figure out where they work, and where other tools work better. You have to think about your data and your process and your project goals, and apply your SAS skills to those problems. You need to think about your data structures and whether they help you or not. You need to look for bottlenecks in your processes, and examine ways to re-engineer them or avoid them completely. You need to look for appropriate uses of the SAS macro language, and avoid the abuses.

But you can do all of this if you give it a try. Everyone who uses SAS can learn to use it effectively.

REFERENCES

SAS OnlineDoc® 9.1.3, Copyright © 2002-2005, SAS Institute Inc., Cary, NC, USA; All rights reserved. Produced in the United States of America.

Carpenter, Arthur L. "Table Lookups: From IF-THEN to Key-Indexing". SAS Institute Inc., Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference. Cary, NC: SAS Institute Inc., 2001. <http://www2.sas.com/proceedings/sugi26/p158-26.pdf>

Cassell, David L. "A Sort of a Mess – Sorting Large Datasets on Multiple Keys". SAS Institute Inc., Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference. Cary, NC: SAS Institute Inc., 2001. <http://www2.sas.com/proceedings/sugi26/p121-26.pdf>

Dorfman, Paul M. "Table Lookup by Direct Addressing: Key-Indexing – Bit-Mapping – Hashing". SAS Institute Inc., Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference. Cary, NC: SAS Institute Inc., 2001. <http://www2.sas.com/proceedings/sugi26/p008-26.pdf>

Dorfman, Paul M. "DATA Step Hash Objects as Programming Tools". SAS Institute Inc., Proceedings of the Thirty-First Annual SAS Users Group International Conference. Cary, NC: SAS Institute Inc., 2006. <http://www2.sas.com/proceedings/sugi26/p236-31.pdf>

Karp, Andrew H. "My Friend the SAS® Format". SAS Institute Inc., Proceedings of the Thirtieth Annual SAS Users Group International Conference. Cary, NC: SAS Institute Inc., 2005. <http://www2.sas.com/proceedings/sugi26/p253-30.pdf>

Karp, Andrew H. and David Shamlin. "Indexing and Compressing SAS® Data Sets: How, Why, and Why Not". SAS Institute Inc., Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference. Cary, NC: SAS Institute Inc., 2003. <http://www2.sas.com/proceedings/sugi26/p003-28.pdf>

Morgan, Derek. "Using the FILEVAR= Option for Input and Output". SAS Institute Inc., Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference. Cary, NC: SAS Institute Inc., 2002. <http://www2.sas.com/proceedings/sugi27/p082-27.pdf>

ACKNOWLEDGMENTS

The author would like to thank the readers of SAS-L (the international SAS mailing list) and comp.lang.soft-sys.sas (the UseNet group covering SAS topics) for putting up with assorted kvetching on this subject for several years now.

This paper has no connection whatsoever with the book "The 7 Habits of Highly Effective People", the book's author Stephen R. Covey, or the book's publishers the Franklin Covey Co.

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.
Other brand and product names are trademarks of their respective companies.

CONTACT INFORMATION

The author welcomes questions and comments. He can be reached at his private consulting company, Design Pathways:

David L. Cassell
Design Pathways
3115 NW Norwood Pl.
Corvallis, OR 97330
DavidLCassell@msn.com
541-754-1304