Leave Your Bad Code Behind: 50 Ways to Make Your SAS® Code Execute More Efficiently.

William E Benjamin Jr, Owl Computer Consultancy, LLC

Abstract:

This laundry list of tips, gathered from over 25 years of SAS programming experience, will show 50 ways to help SAS programmers make their code run faster and more efficiently. General topics will include doing more than one thing in each DATA step, combining steps to make simple tasks take less code, using macro variables to simplify maintenance, using built in features rather than writing your own code, ways to save disk space, using sorts for more than just sorting data, and ways to make the program code just read better (code that is easier to read is easier to maintain). The list is broken into categories to allow readers to find something they can use faster. This list can work as a primer for making code changes that will improve the systems and processes to which it is applied.

Keywords:

Base SAS, Code tips, Macro variables, procedures, conditional code

In over 25 years of SAS programming this author has encountered code of all shapes and sizes, both good and bad. All of the code tips listed below have been used by the author over the years to help speed code, or make it more readable. The tips are grouped into nine different groups to help locate a tips based upon what it is or does to help the programmer. For this paper some of the tips have been tested to see which runs faster. The last tip at the end of the list is a macro that segments code and allows testing of code units while building programs. The macro sets up segments of code that can be executed by setting switches to control what segments of code execute. This works well with Top-Down development of a program by not re-running all of the code while testing a small segment at the bottom. Of course this can also be done by highlighting code and clicking on the “running-man”; but if you forget to highlight the code to test and click run, then it all runs.

The tests were done by executing the code 11 times (6 times the first way and 5 times the second way). The first execution is discarded because it is often higher than all of the other tests. A possible explanation for that is the time required to set up for the first execution of the submitted code. The test file was created by using the SASHELP.SHOES data file and enlarging the file. The records were read and a random number was generated using the SAS RANUNI function (x=ranuni(01261950)) and 150,000 records were generated for each of the 395 records in the SASHELP.SHOES data file. (395 * 150000 = 59,250,000 records)

The first test file was then sorted on the random number stored in the variable “x”. When code below is tested against a number of records, it means the 5 test runs executed 1/5 of the compares for each test run of the code. (5 * 10,000,000 = 50,000,000).

A second test file was created by building a file with 200,000,000 records that contain a random variable named x, and nine other variables Q1 to Q9 that are set to either ‘Y’ or ‘N’ based upon the value of x. if x > .1 then Q1 is set to ‘Y’, if x > .2 then Q2 is set to ‘Y’, etc.

NOTE – The code segments below are for illustrative purposes only and may contain “Editorial Comments” like . . . MORE SAS CODE . . . and are not intended to execute without syntax errors. The code segments are intended to guide programmers in building executable code that meets the needs of the reader. Additionally, differences in character fonts may present some characters that do not translate well when a “Cut-n-Paste” method is used to copy code to the user’s SAS session.

A) Processing more than one file in each DATA step

The following table shows ways that two or more data steps can be combined into one data step. While generally the same amount of output occurs for each process, the input steps are reduced to one pass over the input data file. Reading something once will always be faster than reading it more than once. The time it takes to read a file from a disk or a CD/DVD is orders of magnitude slower than the CPU can operate, so adding more instructions between reading records will not generally slow a computer.

#	This works:	This is more efficient:

1	**This does one thing at a time:**
	`Data temp_file_1;`
	`  Set Perm.input_file;`
	`Data temp_file_2;`
	`  Set temp_file_1;`
	`  Rename var_1 = var_a;`
	`Run;`
		This combines the two data steps and reads the data once, not twice.
		`Data temp_file_2;`
		`  Set Perm.input_file`
		`  (Rename var_1=var_a);`
		`Run;`
		(var_a is available in the data step)

2	**This does one thing at a time:**
	`Data temp_file_1;`
	`  Set Perm.input_file;`
	`Data temp_file_2;`
	`  Set temp_file_1;`
	`  Rename var_1 = var_a;`
	`Run;`
		This combines the two data steps and reads the data once, not twice.
		`Data temp_file_2(Rename var_1=var_a);`
		`  Set Perm.input_file;`
		`Run;`
		`(var_1 is available in the data step)`

3	**Instead of reading and writing the data twice to make the same output dataset:**
	`Data temp_file_1;`
	`  Set Perm.input_file;`
	`Data temp_file_2;`
	`  Set Perm.input_file;`
	`Run;`
		Read the data once and write it twice:
		`Data temp_file_1`
		`  temp_file_2;`
		`  Set Perm.input_file;`
		`Run;`

4	**Instead of reading and writing the data twice to make a different output dataset based upon different variables:**
	`Data temp_file_1;`
	`  Set Perm.input_file;`
	`  If a then output;`
	`Data temp_file_2;`
	`  Set Perm.input_file;`
	`  If b then output;`
	`Run;`
		Read the data once and write it twice into two different output datasets. This code uses independent tests, and output files are independent of each other;
		`Data temp_file_1`
		`  temp_file_2;`
		`Set Perm.input_file;`
		`If a then output temp_file_1;`
		`If b then output temp_file_2;`
		`Run;`

5	**Two text files can be read one at a time and merged. With an extra pass over the data required to merge the files.**
	`filename one_file 'G:\SESUG_2010\file1';`
	`filename two_file 'G:\SESUG_2010\file2';`
	`data read_one;`
	`  infile one_file linesize=160 truncover;`
	`  input  @01 text  $char80.`
	`  @81 more_data $char80. ;`
	`data read_two;`
	`  infile two_file linesize=160 truncover;`
	`  input  @01 text  $char80.`
	`  @81 more_data $char80. ;`
	`run;`
	`data merged;`
	`  set read_one read_two;`
	`run;`
		Or two text files can be read at the same time and no merge step is needed.
		`filename two_file ('G:\SESUG_2010\file1'`
		`  'G:\SESUG_2010\file2');`
		`data read_two;`
		`  infile two_file linesize=160 truncover;`
		`  input @01 text  $char80.`
		`  @81 more_data $char80.`
		`  ;`
		`run;`
		Two input files with the same format can be linked together in the filename statement and read as one file.

6	**Read and merge two text files, one at a time.**
	
	filename one_file 'G:\SESUG_2010\file1';
	filename two_file 'G:\SESUG_2010\file2';
	
	`data read_one;`
	`  infile one_file linesize=160 truncover;`
	`  input  @01 text  $char80.`
	`  @81 more_data $char80. ;`
	`data read_two;`
	`  infile two_file linesize=160 truncover;`
	`  input  @01 text  $char80.`
	`  @81 more_data $char80. ;`
	`run;`
	
	`data merged;`
	`  set read_one;  output;`
	`  set read_two;  output;`
	`run;`
	
	An extra pass over the data is required to merge the files. The data files are interleaved in this output file.
		Or two text files can be read at the same time and no merge step is needed.
		
		filename one_file 'G:\SESUG_2010\file1';
		filename two_file 'G:\SESUG_2010\file2';
		
		`data read_two;`
		`  infile one_file linesize=160 truncover;`
		`  input @01 text  $char80.`
		`  @81 more_data $char80.;`
		`  Output;`
		`  infile two_file linesize=160 truncover;`
		`  input @01 text  $char80.`
		`  @81 more_data $char80.;`
		`  Output;`
		`run;`
		
		Two input files with the different formats can be read by defining two filename statements and reading both in the same data setup. This code interleaves the records in the output file.

7	**Read and merge two text files, one at a time.**
	
	filename one_file 'G:\SESUG_2010\file1';
	filename two_file 'G:\SESUG_2010\file2';
	
	`data read_one;`
	`  infile one_file linesize=160 truncover;`
	`  input  @01 text  $char80.`
	`  @81 more_data $char80. ;`
	`data read_two;`
	`  infile two_file linesize=160 truncover;`
	`  input  @01 text  $char80.`
	`  @81 more_data $char80. ;`
	`run;`
	
	`data merged;`
	`  set read_one read_two;`
	`run;`
	
	An extra pass over the data is required to merge the files. The datasets are not interleaved in the output file in this step.
		Two different files can be read in one data step, this example uses a DO UNITL clause and only executes one pass of the data step code.
		
		`data read_two;`
		`counter + 1;`
		`infile one_file linesize=160 truncover end=eof1;`
		`  do until(eof1);`
		`  input @01 text  $char80.`
		`  @81 more_data $char40.`
		`  ;`
		`  output;`
		`  end;`
		`infile two_file linesize=160 truncover end=eof2;`
		`  do until(eof2);`
		`  input @01 text  $char80.`
		`  @81 diff_data $char80.`
		`  ;`
		`  output;`
		`  end;`
		`Put counter;`
		`Run;`
		
		At the end of the data step counter will equal 1; The datasets are not interleaved in the output file in this step.

8	**A simple way to subset records In a file is to create a new file to use later. This code will do that task.**
	
	`Data Subset_file;`
	`  Set whole_file;`
	`  If a = 'Y' then output;`
	`Run;`
	`  ... use this file in a later step  ...`
	
	In this code all of the values, of all of the variables, of all of the records are loaded into the program data vedtor; and only the records that pass the test (if a = 'Y') are written out to the output file.
		Both of these code segments are more efficient because the condition is tested before the Program Data Vector is filled. That means that the test occurs before data is moved into the variables. So only records that pass the test are processed by the data step.
		
		`Data Subset_file;`
		`  Set whole_file;`
		`  Where a = 'Y';`
		`  ... more SAS code ...`
		`- -  OR -  -`
		`Data Subset_file;`
		`  Set whole_file (where=(a = 'Y'));`
		`  ... more SAS code ...`
		`Run;`

9	**Instead of reading and writing the data twice to make a different output dataset based upon the same variable:**
	
	`Data temp_file_1;`
	`  Set Perm.input_file;`
	`  If a eq 'OK' then output;`
	`Run;`
	
	`Data temp_file_2;`
	`  Set Perm.input_file;`
	`  If a ne 'OK' then output;`
	`Run;`
		Read the data once and write it twice to two different output datasets:
		
		`Data temp_file_1`
		`  temp_file_2;`
		
		`Set Perm.input_file;`
		`If a = 'OK' then output temp_file_1;`
		`  else output temp_file_2;`
		`Run;`

10	**Build three data sets, each with different records and variables from the same file.**
	
	`Data temp_file_1;`
	`  Set Perm.input_file;`
	`  Keep var1 var2 var3;`
	`  If a eq 'OK' then output;`
	`Data temp_file_2;`
	`  Set Perm.input_file;`
	`  Keep var4 var5 var6;`
	`  If b eq 'OK' then output;`
	`Data temp_file_3;`
	`  Set Perm.input_file;`
	`  Keep var7 var8 var9;`
	`  If c eq 'OK' then output;`
	`Run;`
		Build three data sets, each with different records and variables from the same file. But, read the input file only once.
		
		`Data`
		`temp_file_1 (keep=var1 var2 var3)`
		`temp_file_2 (keep=var4 var5 var6)`
		`temp_file_3 (keep=var7 var8 var9);`
		
		`Set Perm.input_file;`
		`If a eq 'OK' then output temp_file_1;`
		`If b eq 'OK' then output temp_file_2;`
		`If c eq 'OK' then output temp_file_3;`
		
		`Run;`

11	**If your site has the SAS/ACCESS Interface to PC File Formats licensed then the following code will allow you to output a SAS file and an Excel file. Note this process is done in two data steps.**
	
	`libname aa 'c:\SESUG_dir';`
	`libname bb 'c:\SESUG_dir\Excel_file.xls';`
	
	`data aa.shoes;  * SAS output file;`
	`  set SASHELP.SHOES; * Excel output file;`
	`run;`
	
	`data bb.'My_sheet'n;`
	`  set SASHELP.SHOES;`
	`run;`
		If your site has the SAS/ACCESS Interface to PC File Formats licensed then the following code will allow you to output a SAS file and an Excel file. Note this process is done in one data step, and does the same thing as the step on the left.
		
		`libname aa 'c:\SESUG_dir';`
		`libname bb 'c:\SESUG_dir\Excel_file.xls';`
		
		`data aa.shoes  * SAS output file;`
		`  bb.'My_sheet'n; * Excel output file;`
		`  set SASHELP.SHOES;`
		`run;`
		
		The input file is read only once.

12	**Most programmers code individual steps for the SAS PROC FREQ and create code that looks something like this:**
	
	`proc freq data = sashelp.shoes;`
	`table region  / list out=region_freq1;`
	`run;`
	
	`proc freq data = sashelp.shoes;`
	`table region*product/ list out=region_freq2;`
	`run;`
	
	`proc freq data = sashelp.shoes;`
	`table region*stores / list out=region_freq3;`
	`run;`
		But, when the input dataset is the same for all of the tables that are created the code can be reduced to the following. This also reduces processing time because only one pass over the input dataset is required.
		
		`proc freq data = sashelp.shoes;`
		
		`table region  / list out=region_freq1;`
		`table region*product/ list out=region_freq2;`
		`table region*stores / list out=region_freq3;`
		`run;`
		
		Both examples in this section also output a SAS file that has the results of the freq that can be used by later steps.

13	**Many programmers code individual steps to subset multiple data files and then merge the files with code that looks something like this, flagging which file the record came from. This requires two passes over each data file.**
	
	data file_1;
	set sample_data1 (where=(var1='Y'));
	file_numb = 1;
	data file_2;
	set sample_data2 (where=(var2='Y'));
	file_numb = 2;
	data file_3;
	set sample_data3 (where=(var3='Y'));
	file_numb = 3;
	
	data file_all;
	set file_1
	file_2
	file_3;
	run;
		The same result can be achieved by doing the following, with only one pass over the data.
		
		data file_all;
		set sample_data1 (where=(var1='Y') in=a)
		sample_data2 (where=(var2='Y') in=b)
		sample_data3 (where=(var3='Y') in=c);
		
		select;
		when (a) then file_numb = 1;
		when (b) then file_numb = 2;
		when (c) then file_numb = 3;
		otherwise;
		end;
		
		run;

Table 1. In the table above the code on the left does more work than code on the right.

B) Combining steps to make simple tasks take less code

Tips in this area are shown one at a time. They can of course be combined together in a data step. The object of showing the code this way is to enable testing of each instruction. The test file described above has over 59 million records. The general test procedure this author used was to execute a predetermined number of tests. Usually five sets of 10 million records were read from the test file, and the code executed. Since the object was to test the code and not the I/O speed, only the input record was read and the test preformed. No output records were written, and the data step timings from the five runs were averaged together and compared for both sets of code.

#	**This works:**	**This is uses less code:**
14	**Reading data from a text file into an array can be done in several ways. One of the simple ways is to just read the variables like this:** Filename text 'D:\SESUG_2010\Text_file'; Data file_with_an_array; Array myvars{3} $ 25 var_1-var_3; Infile text linesize=100 truncover; Input @01 key $char25. @26 var_1 $char25. @51 var_2 $char25. @76 var_3 $char25.; Run;	**A second method is to read the same array using built in functions of the input statement as in the following code.** Filename text 'D:\SESUG_2010\Text_file'; Data file_with_an_array; Array myvars{3} $ 25 var_1-var_3; Infile text linesize=100 truncover; Input @01 key $char25. @26 (var_1-var_3) ($char25.); Run;

15	**At times the input data structures are even more complex that single arrays, They can be arrays of groups of variables. While these are easy to define in some of the older computer languages like COBOL, the structures are much harder to define in SAS. This set of variables is two arrays with 3 variables each.**
	
	`Filename text 'D:\SESUG_2010\Text_file';`
	
	`Data temp;`
	`Array myvars1{3} $ 10 vara_1 vara_2 vara_3;`
	`Array myvars2{3} $ 15 varb_1 varb_2 vara_3;`
	
	`Infile text linesize=85 truncover;`
	
	`Input @01 key  $char10.`
	`  @11 vara_1 $char10.`
	`  @21 varb_1 $char15.`
	
	`  @36 vara_2 $char10.`
	`  @46 varb_2 $char15.`
	
	`  @61 vara_3 $char10.`
	`  @71 varb_3 $char15.;`
	
	`Run;`
		A simple way to read these structures into an array is by using built in functions of the SAS input statement as follows.
		
		`Filename text 'D:\SESUG_2010\Text_file.txt';`
		
		`Data temp;`
		`Array myvars1{3} $ 10 vara_1 vara_2 vara_3;`
		`Array myvars2{3} $ 15 varb_1 varb_2 varb_3;`
		
		`Infile text linesize=85 truncover;`
		
		`Input @01 key  $char10.`
		`  @11 (vara_1-vara_3) ($char10. +15)`
		`  @21 (varb_1-varb_3) ($char15. +10);`
		
		`Run;`
		
		Note that this code reads the same set of two arrays with 3 variables each as the code on the left. What if there were 10, 100, or even 1000 array elements, The code on the left becomes annoying to write and update, but the code above can be updated to 1000 array elements like this and still only use two lines of code:
		
		`input @01 key  $char10.`
		`  @11 (vara_1-vara_1000) ($char10. +15)`
		`  @21 (varb_1-varb_1000) ($char15. +10);`

16	**Computers are good at doing the same thing over and over again. But, programmers do not like to code the same thing over and over again. Take this code for an example.**
	
	Data test;
	
	Set sample;
	
	If vara > 1 then do;
	  A = vara * amount * commission;
	  B = A / 2;
	  C = A + B;
	End;
	If varb > 1 then do;
	  A = varb * amount * commission;
	  B = A / 2;
	  C = A + B;
	End;
	If varc > 1 then do;
	  A = varc * amount * commission;
	  B = A / 2;
	  C = A + B;
	End;
	
	Run;
		One way to rewrite the code on the left is to use subroutines. The following is one example.
		
		Data test;
		
		Set sample;
		
		Temp =vara;
		If vara >1 then link test;
		
		Temp =varb;
		If varb >1 then link test;
		
		Temp =varc;
		If varc >1 then link test;
		
		Return; * end of the main datastep;
		
		* this subroutine code is executed three times;
		
		Test:
		  A = Temp * amount * commission;
		  B = A / 2;
		  C = A + B;
		Return; * end of the subroutine;
		Run;

17	**This is the same code as the previous example, but a second solution is presented that makes the code work with fewer variables. (i.e. Temp is not required)** Data test; Set sample; If vara > 1 then do; A = vara * amount * commission; B = A / 2; C = A + B; End; If varb > 1 then do; A = varb * amount * commission; B = A / 2; C = A + B; End; If varc > 1 then do; A = varc * amount * commission; B = A / 2; C = A + B; End; Run;	**The use of a macro to do the same task over and over again works well too.** %macro test(test_var); A = &test_var * amount * commission; B = A / 2; C = A + B; %mend test; Data test; Set sample; If vara >1 then do; %test(vara); end; If varb >1 then do; %test(varb); end; If varc >1 then do; %test(varc); end; Run;
18	**Inserting information into a SAS character variable is sometimes coded in ways that are harder to work with than they need to be, for instance replacing the 5th character in a character variable with "Y".** My_variable = 'PASSNFAIL'; My_variable = substr(My_variable,1,4) || 'Y' || Substr(My_variable,6,4);	**The SAS SUBSTR function can be used on either side of the equal sign. The example on the left shows it on the right side of the equal sign, here it is on the left side.** My_variable = 'PASSNFAIL'; substr(My_variable,5,1) = 'Y';

19	**Many programmers use the following code to test a variable and set an indicator to either 0 or 1:**
	
	Data _null_;
	Set Perm.input_file;
	If region = 'Africa'
	then flag_1 = 1;
	else flag_1 = 0;
	Run;
	user_cpu time used = 10.67 sec for 50 Million tests.
		This code does the same thing:
		
		Data _null_;
		Set Perm.input_file;
		flag_1 = region = 'Africa';
		Run;
		
		user_cpu time used = 10.47 sec for 50 Million tests.

20	**Many programmers use counters in their code like this.** Data step_one; Retain counter 0; ... more SAS code ... Counter = counter + 1; ... more SAS code ... Run; **Two instructions are used here.**	**Consider this code instead to use a counter in code like this.** Data step_one; ... more SAS code ... Counter + 1; ... more SAS code ... Run; **One instruction is used here, and the variable is automatically retained.**

21 **In this test case a closer examination on the SASHELP.SHOES dataset is needed. The file has 395 records and the test file that was created has 150,000 records for each of those 395 records. The file was sorted based upon a random number that was added to the file, but the same relative number of records exist in each file. The code presented below does not take any action when a condition is met, it is testing the number of compares. You would not usually code a SAS SELECT statement as it is coded below. The test was designed to measure the amount of time required for the testing. Since the input records and conditions were exactly the same any decrease in the execution time is as a result of the number of compares executed.**

Region (Alphabetical order)	count	percent	Cumm count	Cumm percent
Africa	56	14.18	56	14.18
Asia	14	3.54	70	17.72
Canada	37	9.37	107	27.09
Central America/Caribbean	32	8.10	139	35.19
Eastern Europe	31	7.85	170	43.04
Middle East	24	6.08	194	49.11
Pacific	45	11.39	239	60.51
South America	54	13.67	293	74.18
United States	40	10.13	333	84.30
Western Europe	62	15.70	395	100.00

Region reordered by the relative percent of records in the region. Note that no one group contains a disproportionate amount of the data records.

Western Europe	62	15.70
Africa	56	14.18
South America	54	13.67
Pacific	45	11.39
United States	40	10.13
Canada	37	9.37
Central America/Caribbean	32	8.10
Eastern Europe	31	7.85
Middle East	24	6.08
Asia	14	3.54

Test of region values in alphabetical order

```
Select (region);
   When('Africa');
   When('Asia');
   When('Canada');
   When('Central America/Caribbean');
   When('Eastern Europe');
   When('Middle East');
   When('Pacific');
   When('South America');
   When('United States');
   When('Western Europe');
  Otherwise;
End;
```

The average user_cpu time used = 15.172 sec to process the compares for 59,250,000 records. Total time to read the records and process the file was 5 minutes 39.744 seconds. The only work done by the data step was to read the records and test the REGION variable values.

Test of region values in relative percentage order

```
Select (region);
   When('Western Europe');
   When('Africa');
   When('South America');
   When('Pacific');
   When('United States');
   When('Canada');
   When('Central America/Caribbean');
   When('Eastern Europe');
   When('Middle East');
   When('Asia');
  Otherwise;
End;
```

The average user_cpu time used = 14.038 sec to process the compares for 59,250,000 records. This is a 7.47% increase in speed. Total time to read the records and process the file was 5 minutes 22.526 seconds.

*** NOTE * times are an average of 5 tests.**

22	**One of the tasks that a programmer is given is that of merging a very very large file with a small file, This author has processed files in excess of 40 GB that needed to have a file containing a few thousand bytes added to the file. (like a set of flags or constants for a limited number of records within the larger file. Invariably the large file is not in the order needed to match the records from the small file. If you are short on disk space then sorting can be out of the question (it takes free space equal to 2 ½ times the size of the input file to sort a file) A Data Base Administrator (DBA) will squeeze any byte out of a main database file if the same data can be stored in a small file and applied to the big file later. (SQL does this well, but these tips are not for SQL) PROC FORMAT can be used to transfer data from one file to another, as long as it can be squeezed into the "LABEL" of the format.**

```
Proc Format;
Value $flags

'123456' = 'Y5 9 3  wed 02152009'
'234567' = 'Y612 5  tue 03122008'
'345678' = 'N9 3 9  mon 06012007'
'Other'  = 'X';
Run;

Data new_master;

  Set my_big_file;
      If (put(cust,$flags.) in: ('Y','N')) then do;
          Flag_text = (put(cust,$flags.));
          Store = substr(flag_text,2,1);
          Bin   = substr(flag_text,3,2);
          Shelf = substr(flag_text,6,1);
          WDay  = substr(flag_text,9,3);
          month = substr(flag_text,13,2);
          day   = substr(flag_text,15,2);
          year  = substr(flag_text,17,4);
          Date  = mdy(month,day,year);
      end;


Run;
```

Table 2. In the table above the code shows alternative methods of processing.

C) Using Macro variables to simplify maintenance

This set of tips show simple ways to use macro variables to reduce the number of changes that have to be applied manually when code needs to be updated. (NOTE – special characters shown here may not translate properly if the code is cut and pasted into SAS. Some characters may need to be retyped after being pasted into the SAS editor. Specifically the quotation marks.) Think of macro variables as strings that can be placed anywhere they are needed. Macro variables are valid almost any place in SAS code. While it was not shown as one of the tips, macro variables can even be used to "Build" variable names. Macro variables are resolved before the SAS DATA step executes. The number of ampersands (&) that proceed a macro variable control how the value of the macro variable (or group of macro variables) is treated when the data step executes.

#	This works:	This requires less maintenance:

23	**Note each reference needs to be changed if the directory ever changes.**
	`Libname file_01 ‘c:\SESUG_2010\project_dir\input_data’;`
	`Libname file_02 ‘c:\SESUG_2010\project_dir\output_data’;`
	`Filename in_file1 ‘c:\SESUG_2010\project_dir\my_text.txt’;`
		Note the Double Quote will cause the macro variable to be resolved, and one change is applied to all three commands.
		`%let My_Dir = c:\SESUG_2010\project_dir;`
		`Libname file_01 “&My_dir.\input_data“;`
		`Libname file_02 “&My_dir.\output_data“;`
		`Filename in_file1 “&My_dir.\my_test.txt“;`

24	**Here three changes are needed if the array size changes.**
	`Data array_test;`
	`Array counters {15} var_01-var_15;`
	`Do I = 1 to 15;`
	`  Counters(i) = 0;`
	`End;`
		Note here the macro variable was used as part of a variable name and as a numeric constant, only one change is required to change the array size and usage.
		`%let max_size = 15;`
		`Data array_test;`
		`Array counters {&max_size} var_01-var_&max_size;`
		`Do I = 1 to &max_size;`
		`  Counters(i) = 0;`
		`End;`

25	**If your programming environment includes more than one computer platform and operating system (OS) the file definition code (LIBNAME statements) can be coded into the program and comments (/* … */ or *…;) placed around the code needed for the other OS.**
	/* comment out the statement not being used */
	`*LIBNAME data1 ‘/unix/directory/name’;`
	`LIBNAME data1 ‘d:\windows\directory\name’;`
		The SAS Automatic Macro variable SYSSCPL can be used to determine the operating system in use. (with SAS 9.1.2 or later). The code used picked at runtime by SAS, you never change it again.
		%let My_os = &SYSSCPL;
		%assign_libs;
		`  %if &My_os = SunOS %then %do;`
		`  LIBNAME data1 ‘/unix/directory/name’;`
		`  %end;`
		`  %if &My_os = XP_PRO %then %do;`
		`  LIBNAME data1 ‘d:\windows\directory\name’;`
		`  %end;`
		`%mend assign_libs;`
		`%assign_libs;`

26	**Most people code the libname and filename statements like this:**
	`Libname file_01 ‘c:\SESUG2010\proj_dir\in’;`
	`Libname file_02 ‘c:\SESUG2010\proj_dir\out’;`
	`Filename text ‘c:\SESUG2010\proj_dir\my.txt’;`
		Note the Double Quote will cause the macro variable to be resolved, and the period is required to avoid a space. The %let value does not need quotes, it is inserted into a quoted string.
		`%let My_Dir = c:\SESUG2010\proj_dir;`
		`Libname file_01 “&My_dir.\in“;`
		`Libname file_02 “&My_dir.\out“;`
		`Filename text “&My_dir.\my.txt“;`

27	**Most people code the dates into file names like this:**
	
	`Data myfile.Monthly_Data_for_2009_feb;`
	`  Set myfile.Monthly_Data_for_2009_Jan;`
	`Run;`
		But using macro variables allows for one change at the top of the program. (no period at the end is required, spaces do not matter here)
		`%let old_month = 2009_Jan;`
		`%let new_month = 2009_Feb;`
		
		`Data myfile.Monthly_Data_for_&new_month;`
		`  Set myfile.Monthly_Data_for_&old_month;`
		`Run;`

28	**Variable names can be built, coded, and changed every time the program data changes, for every time the variable is used.**
	
	`Data test;`
	`…`
	`Wigits_Jan_2009 = var1 + var2;`
	`…`
	`Wigits_Jan_2009 = Wigits_Jan_2009  + var2;`
	`…`
	`Run;`
		Variable names can be built using macro variables:
		
		`%let month = Jan;  %let year = 2009;`
		
		Data test;
		…
		Wigits_&month._&year = var1 + var2;
		…
		Wigits_&month._&year = Wigits_&month._&year + var2;
		…
		Run;
		
		***** Note *** The period is required after &month because variable names are not allowed to have spaces, and eliminating the period would insert a space. No period is needed after &year because a space is allowed here.**

29	**A list of variables can be pulled from several datasets at the same time in the following manor:**
	
	`Data Selected;`
	`Set`
	`  file1(keep=key var1 var2 var3)`
	`  file2(keep=key var4 var5)`
	`  file3(Keep=key var1 var2 var3 var4 var5); run;`
		Another method of listing variables is to use a macro variable to identify the variables needed.
		
		%let list1 = var1 var2 var3;
		%let list2 = var4 var5;
		
		`Data Selected;`
		`  Set file1 (keep=key &list1)`
		`  file2 (keep=key &list2)`
		`  file3 (Keep=key &list1 &list2);`
		` run;`

Table 3. In the table above the code shows alternative methods of processing using macro variables.

D) Using built in features rather than writing your own code

This next table shows how using built in features or functions can save on coding by taking advantage of routines that are part of the SAS system. When these features are used the extra code is (1) not written, and (2) not executed. It is a good bet that the built-in features run faster than the extra code required to do the same thing.

#	This works:	This is uses SAS Built-in functions:

30	**Arrays for temporary variables can be programmed more than one way. If the value in the array needs to survive more than one iteration of the SAS DATA step (i.e. not set to missing at the end of the DATA step) then the variables need to be retained.**
	`Data test;`
	`Array counters {3} var1-var3;`
	`Retain var1 var2 var3 0;`
	`Set my_data;`
	`By key1;`
	`Counter(1) = Counter(1) + 1;`
	`Counter(2) = sum(var4,var5);`
	`Counter(3) = var6 * 3;`
	`If last.key1 then do I = 1 to 3;`
	`Counter(i) = 0;`
	`End;`
	`Drop i var1 var2 var3;`
		The SAS ARRAY _TEMPORARY_ option has the ability to reduce the code needed to do the same tasks, and the variables are automatically retained and not output to the final dataset.
		`Data test;`
		`Array counters {3} _temporary_;`
		`Set my_data;`
		`By key1;`
		`Counter(1) = Counter(1) + 1;`
		`Counter(2) = sum(var4,var5);`
		`Counter(3) = var6 * 3;`
		`If last.key1 then do I = 1 to 3;`
		`Counter(i) = 0;`
		`End;`
		`Drop i;`

31	**Instead of coding for all possible options of test cases in character strings like this:**
	`if (a = 'YES' or`
	`a = 'YEs' or`
	`a = 'YeS' or`
	`a = 'yES' or`
	`a = 'yeS' or`
	`a = 'yEs' or`
	`a = 'Yes' or`
	`a = 'yes' ) then x = 1;`
		Use the SAS functions to compensate for minor variations in the variable formatting. The upcase function does not change the value of the variable in the following code, but it tests for all of the combinations of 'YES' as in the column on the left.
		`if (upcase(a) = 'YES') then x = 1;`

32	**Arrays of variables are sometimes used to calculate values across several records or for the whole dataset, like the following code:**
	`DATA only_test_var_totals (keep=(t_1-_3));`
	`Array test_vars {3} t_1-t_3;`
	`Retain t_1-t_3;`
	`If _n_ = 1 then do; *clear t_1-t_3;`
	`Do x =1 to 3;`
	`Test_vars{1} = 0;`
	`End;`
	`End;`
	`SET my_data end=eof;`
	`* sum t_1-t_3 over whole dataset;`
	`Test_vars{1} = Test_vars{1}+var_1/var_2;`
	`Test_vars{2} = Test_vars{2}+(var_3/var_2)*var_4;`
	`Test_vars{3} = Test_vars{3}+var_1/(var_2*var_5);`
	`If (eof=1) then output;`
	`Run;`
	This data step writes one record with totals.
		This code could be re-written as:
		`DATA only_test_var_totals (keep=(t_1-t_3));`
		`Array test_vars {3} t_1-t_3 0 0 0;`
		`Retain t_1-t_3;`
		`SET my_data end=eof;`
		`Test_vars{1} = Test_vars{1}+var_1/var_2;`
		`Test_vars{2} = Test_vars{2}+(var_3/var_2)*var_4;`
		`Test_vars{3} = Test_vars{3}+var_1/(var_2*var_5);`
		`If (eof) then output;`
		`Run;`
		This data step writes one record with totals, and the default value of the code "if (EOF)" is one (1), when the end of the SAS file has been reached and there for does not need to be coded as "EOF=1".

33	**The SAS SUBSTR function can be used to restrict the number of characters that are included in a compare. As in this example that compares 3, 4, and 5 characters of a string to a constant.**
	
	`data _null_;`
	`array test {3} $ 9 test1-test3;`
	`test1 = 'abcdef';`
	`test2 = 'abcdefghi';`
	`test3 = 'abcdkfirf';`
	
	`flag1 = 'n'; flag2 = 'n'; flag3 = 'n';`
	`if 'abc' = substr(left(test(1)),1,3)`
	`then flag1='y';`
	`if 'abcd' = substr(left(test(2)),1,4)`
	`then flag2='y';`
	`if 'abcde' = substr(left(test(3)),1,5)`
	`then flag3='y';`
	`run;`
		The same result can be achieved by changing the operator from an equal sign (=) to an equal sign followed by a colon (=:). This automatically restricts the compare to the size of the shortest string, and returns a compare condition if they match.
		
		`data _null_;`
		`array test {3} $ 9 test1-test3;`
		`test1 = 'abcdef';`
		`test2 = 'abcdefghi';`
		`test3 = 'abcdkfirf';`
		
		`flag1 = 'n'; flag2 = 'n'; flag3 = 'n';`
		`if 'abc' =: left(test(1)) then flag1='y';`
		`if 'abcd' =: left(test(2)) then flag2='y';`
		`if 'abcde' =: left(test(3)) then flag3='y';`
		`run;`

34	**When processing for specific values or lists of flags the code can end up with a lot of tests and counts. For instance if more than 50% of the flags must be set to true for a record to be valid then the following code will process the record.**
	
	`Counter = 0;`
	`If (Q1 = 'Y') then counter +1;`
	`If (Q2 = 'Y') then counter +1;`
	`If (Q3 = 'Y') then counter +1;`
	`If (Q4 = 'Y') then counter +1;`
	`If (Q5 = 'Y') then counter +1;`
	`If (Q6 = 'Y') then counter +1;`
	`If (Q7 = 'Y') then counter +1;`
	`If (Q8 = 'Y') then counter +1;`
	`If (Q9 = 'Y') then counter +1;`
	`If ((counter/ 9) > .5) then Passed=1; * passed;`
	`Else Passed=0; * failed;`
		The same result can be achieved without the use of a temporary counter variable as in the following code.
		
		`Passed = sum((Q1 = 'Y'),`
		`(Q2 = 'Y'),`
		`(Q3 = 'Y'),`
		`(Q4 = 'Y'),`
		`(Q5 = 'Y'),`
		`(Q6 = 'Y'),`
		`(Q7 = 'Y'),`
		`(Q8 = 'Y'),`
		`(Q9 = 'Y'))/9 > .5;`
		
		The result is a binary value (0 or 1) that is derived by adding the binary results of nine tests (Q1 = 'Y' …) and dividing the result by 9. Then comparing that result to > .5 as in the code on the left with 0 = False, 1 = True.

35	**Some systems still use text files to transport data from one system to another. Programmers are sometimes called upon to read in one text file and make minor changes (like changing a field) and write out another text file. Here is one method.**
	
	`Filename old 'Q:\SESUG_2010\Old_text.txt';`
	`Filename New 'Q:\SESUG_2010\New_text.txt';`
	
	`Data _null_;`
	`Infile old linesize=40;`
	`File new linesize=40 noprint notitles;`
	
	`Input @01 text $char20.`
	`@21 test $char1.`
	`@22 Var1 $char4.`
	`@26 fill $char15.;`
	
	`If test = 'Y' then var1 = 'FINE';`
	
	`put @01 text $char20.`
	`@21 test $char1.`
	`@22 Var1 $char4.`
	`@26 fill $char15.;`
	
	Run;
		An alternative method is to use the system option SHAREBUFFER and the automatic variable _INFILE_ to do the same work. Since the input and output buffers are the same when this option is in effect the code can be reduced somewhat;
		
		`Filename old 'Q:\SESUG_2010\Old_text.txt';`
		`Filename New 'Q:\SESUG_2010\New_text.txt';`
		
		`Data _null_;`
		`Infile old linesize=40 SHAREBUFFER;`
		`File new linesize=40 noprint notitles;`
		
		`Input ;`
		
		`If substr(_infile_,21,1) = 'Y' then`
		`substr(_infile_,22,4) = 'FINE';`
		
		`put _infile_;`
		
		Run;

E) Ways to save disk space

Here are some tips on how to save disk space when using SAS files, Some of these tips are for a Windows or UNIX based system.

#	The standard installation:	User Options available :
36	**The standard installation may not activate compression options for disk usage. As a result creating a SAS Dataset does not usually save disk space.**	**The SAS Option COMPRESS= has two options that produce output files that are smaller than the default option of COMPRESS=NO.** COMPRESS=YES - Produces a smaller SAS Dataset COMPRESS=BINARY – Produces the smallest files. Some files can be reduced in size by 80% or more. **Both types of compressed files can be read without an "uncompress" step.**
37	**Use Operating system commands to compress files.**	**On Windows the Program Winzip can be used to compress files. But the files need to be "uncompressed" for SAS to be able to read them.** **On UNIX the gzip command can be used to compress files. But the files need to be "uncompressed" for SAS to be able to read them.**
38	**Computer disk options can be used to compress files.**	**The Windows NTFS formatted disk drives can have compression turned on for mapped drives. Mapped drives can point to a space as small as one directory.** No "uncompress" step is required to access SAS files on compressed drives, but a slight slowing of the access may occur. This author has never found that to be enough to out weigh the space savings. ***** NOTE *** Always use caution when compressing a NTFS disk drive Your System Administrator may not want individual users to apply these changes to the computer disk drives.**
39	**Use the SAS ATTRIBUTE command to change the size of numeric variables, but remember that the maximum size of the number is smaller if the size of the variable is smaller.**	**Command Syntax =** `ATTRIB var1 LENGTH = n;` **Where n is an integer in the range of 3 to 8.**
40	**Use the SAS LENGTH command to change the size of numeric variables, but remember that the maximum size of the number is smaller if the size of the variable is smaller.**	**Command Syntax =** `LENGTH var2 = n;` **Where n is an integer in the range of 3 to 8.**

F) Using sorts for more than just sorting data

After many years of working with large datasets (the ones that take 2 or more hours to sort on a Windows based computer) this author spent many hours looking for ways to "BEEF-UP" the output of a sort step. Any pass over the data that can be eliminated means the job runs faster.

#	This works:	This process has fewer steps:

41	**Many programmers write the following code which requires the data to be processed twice:**
	
	`Data Temp_file;`
	`  Set old_file (keep=(key_1 var_1 var_2`
	`  var_3 var_4 var_5));`
	`Run;`
	
	`PROC SORT data=Temp_file out=new_file;`
	`By key_1;`
	`Run;`
		The following only requires one pass of the file and keeps only the requested variables:
		
		`PROC SORT Data = Temp_file(keep=(key_1 var_1`
		`  var_2 var_3 var_4 var_5)) out=new_file;`
		`By key_1;`
		`RUN;`
		
		This method processes the file directly and does not create a temporary subset file of the original input file. Only a Sort step reads the file.

42	**Many programmers write the following code which requires the data to be processed twice:**
	
	`Data Temp_file;`
	`  Set old_file (rename=(var_1=var_a`
	`  var_2=var_b));`
	`Run;`
	
	`PROC SORT data=Temp_file out=new_file;`
	`By key_1;`
	`Run;`
		The following only requires one pass of the file and renames the requested variables:
		
		`PROC SORT Data = Temp_file`
		`  (rename=(var_1=var_a var_2=var_b))`
		`out=new_file;`
		`By key_1;`
		`RUN;`
		
		This method processes the file directly and does not create a temporary subset file from the original input file.

43	**Many programmers write the following code which requires the data to be processed twice:**
	
	`Data Temp_file;`
	`  Set old_file (where=(var_1=var_a`
	`  var_2=var_b));`
	`Run;`
	
	`PROC SORT data=Temp_file out=new_file;`
	`By key_1;`
	`Run;`
		The following only requires one pass of the file and keeps only the records that qualify:
		
		`PROC SORT Data = Temp_file`
		`  (where=(var_1=var_a var_2=var_b))`
		`  out=new_file;`
		`By key_1;`
		`RUN;`

44	**Use the sort routine to combine any or all of the last three tips to subset your records at the same time as it is sorted and remove duplicates at the same time.**
	
	`PROC SORT data=Temp_file(keep =(key_1 var_1 var_2 var_3 var_4 var_5)`
	`  Rename =(var_1=var_a var_2=var_b)`
	`  Where =(var_1='A' and var_2='B'))`
	`  out=new_file nodupkey;`
	`By key_1;`
	`RUN;`
	
	***** Note *** The output file will have 6 variables (key_1 var_a var_b var_3 var_4 var_5) where var_a='A' and var_b='B'. Variables var_1 and var_2 are available in the where clause but var_a and var_b are output to new_file.**	

G) Ways to make the program code just read better

When defaults do not match the data (a 200 character variable to store “ABC”) take control and define what you need exactly. Add labels if you want too.

#	This works:	This defines variables or clarifies code:

45	**SAS as a programming language allows old and new programmers to get away without having to worry too much about the data definitions. The SAS files have numeric and character variables. The first time a variable is used the context of the usage defines the character type.**
	`Data My_file;`
	`A = ‘Y’;`
	`B = 3;`
	`C = mdy(‘01’,’26’,’1950’);`
	`Output;`
	`Run;`
	A simple file with one record, two numeric variables (B and C), and one character variable (A) is created.
		An experienced programmer can read the code on the left and quickly determine the default options used to create the variables. But SAS also allows the programmer to define the variable characteristics before the variables are used. Like some other languages.
		`Data My_file;`
		`Attrib A LENGTH= $ 1 label = ‘Flag’`
		`informat= $char1. format = $char1;`
		`Attrib B LENGTH= 8 label = ‘Counter’`
		`informat= 5. format = 5.;`
		`Attrib A LENGTH= 8 label = ‘Birthday’`
		`informat=mmddyy10. format=mmddyy10.;`
		`A = ‘Y’;`
		`B = 3;`
		`C = mdy(‘01’,’26’,’1950’);`
		`Output;`
		`Run;`

46	**At times a cascading “IF…THEN …ELSE” series of clauses can become hard to read, when indented over and over again, the SELECT statement structure operates in the same way as the “IF…THEN …ELSE”, but is easier to read. Note the following:**
	`IF var_a = ‘A’`
	`then var_1 = 1;`
	`else if var_a = ‘B’`
	`then var_1 = 2;`
	`else if var_a = ‘C’`
	`then var_1 = 3;`
	`else var_1 = 4;`
		Consider this instead:
		`Select(var_a);`
		`When(‘A’) var_1 = 1;`
		`When(‘B’) var_1 = 2;`
		`When(‘C’) var_1 = 3;`
		`Otherwise var_1 = 4;`
		`End;`
		Both these code segments run in about the same time, the SAS interpreter generates nearly the same code for both.

47	**Given data where the values have coded meanings, like the following for the variable footwear:**
	1 = a shoe
	2 = a boot
	3 = a slipper
	instead of coding the numbers which will have to be changed every where they are used in the following code:
	`if footwear=1 then sum(shoe_sales,amount);`
	`if footwear=2 then sum(boot_sales,amount);`
	`if footwear=3 then sum(slipr_sales, amount);`
		Then code the following change it so that only one change is needed to update every occurrence and usage of the values.
		`%let shoe = 1;`
		`%let boot = 2;`
		`%let slipper = 3;`
		`*footwear can only have one value at a time;`
		`select(footwear);`
		`when(&shoe) sum(shoe_sales , amount);`
		`when(&boot) sum(boot_sales , amount);`
		`when(&slipper) sum(slipper_sales, amount);`
		`otherwise ;`
		`end;`

H) Extras – Advanced coding techniques that are hard to find

These last few items are not for the weak of heart; this paper was designed to have some examples that will challenge your ability to use them.

#	This works – Try it sometime: Bit testing – combine multiple tests
48	**One feature that is not used too often is the ability to use the WHERE= clause on a SAS data step or procedure to subset a file based upon multiple conditions. The next trick will show how to test for up to 31 conditions with one WHERE= clause using a technique called bit masking. SAS code allows binary constants to be created. The math behind this tip is binary math in the form of numeric constants. For a computer with a "32" bit operating system the range of the constants is 0 to 2**32-1 [4,294,967,295 or in the common language of today 4GB-1]. Negative numbers are not included because the left most bit is a sign bit and is set for all negative numbers. With computers today using "Two's Complement" math, Zero always means no bits set. (some computers that used "one's Complement" math could have negative and positive zero conditions – All ones or all zeroes) But, enough of the math. The object of this tip is to enable the selection of more than one condition without making a WHERE= clause stretch on forever. Let us look at the following code:**

```
Data Master_file;


    flag = '00000000000000000000000000000000';   * a character string of 31 zeros;

    Set my_sas_file;

    If (cond_1 = 1) then substr(flag,31,1) = '1'; * set any condition to a zero or a one;
    If (cond_2 = 2) then substr(flag,30,1) = '1';
    If (cond_3 = 3) then substr(flag,29,1) = '1';

    *. . . more conditions . . . ;

    If (cond_29 = 4) then substr(flag,3,1) = '1';
    If (cond_30 = 5) then substr(flag,2,1) = '1';
    If (cond_31 = 6) then substr(flag,1,1) = '1';


    Conditions_flag = input(flag,ib4.) ; * convert the flags to a real numeric variable;
                                         * IB4. informat is Integer Binary for 4 bytes.;
                                         * It knows there were 31 binary digits not 32;

 run;


  proc sort data = Master_file
                 (where=(Conditions_flag=input('10100000000000000000000000000010',ib4.)))
             Out = New_subset_file;
  by key;
  run;
```

If the INPUT function on the WHERE= clause is not used (just the bit constant) then the Log will tell you that Bit constants are not compatible with the WHERE= clause. But the INPUT function converts the constant to a number before the WHERE= clause sees the constant. This way one master file can be used and subsets of data can be extracted for any proc that supports the SAS Dataset option WHERE=, like PROC FREQ, PROC PRINT, PROC SORT, and of course within the WHERE clause in a SAS DATA step. A creative programmer would use macro variables to hide some of those zeros.

#	**This works – Try it sometime: Send me an e-mail – I will answer it.**

49	**One of the cool built in features of SAS is the ability of the software to send an email. With cleaver usage of macro variables and the email system the status of the program can be transmitted to your email address as the jobs finish. This head-up can sometimes mean the difference between having a good day with your jobs done and coming in to work on Monday to a disappointing day.**
	
	This code skeleton outlines the general way the output filename (wbenjam - an email address) is defined and gives the programmer a chance to send messages about the status of the executing program.
	
	`filename wbenjam email "william@owlcomputerconsultancy.com"; * output email address;`
	
	`data one_step;`
	
	`set my_file end=eof; * set-up processing at the end of the input file;`
	`. . . your SAS code . . .`
	
	`If eof then do;`
	`call symput('LasT_Good_Step','one_step_was_ok'); * conditionally set the macro value;`
	`end;`
	
	`run;`
	
	`data Two_step;`
	
	`. . . your SAS code . . .`
	
	`call symput('LasT_Good_Step','two_step_was_ok'); * conditionally set the macro value;`
	
	`run;`
	
	`data _null_;`
	`file wbenjam subject='Job Completion Status'; * open the email address for output;`
	`put 'Job update';`
	`put "Last good step was &LasT_Good_Step"; * macro variable set to last good step;`
	`put ' bye ';`
	`run;`

I) A Macro routine that will conditionally execute SAS Code

For everyone that wanted to make something run “part of the time” – here it is – The conditional execution functions (%if %then %else). It only works in a macro, but for code that either takes a long time to run or a long time to test, here is an answer. Define the LIBNAME “USER” and all “WORK” level files are sent to “USER” these files are on permanent disk space and retained between sessions. Set up the macros as below, and only **`Conditional_Step_D`** will execute.

#	**This works – Try it sometime:**
50	**This group of five macros will set up conditional processing of code steps based upon the %let defined flags placed at the beginning of the program. By setting the flags to anything except YES the steps are conditionally executed. Normal macro variable scoping rules apply, so and macro variables defined within Conditional_Step_x (A, B, C ,or D) are not available out side of that macro, unless the macro variable is declared Global (i.e. %GLOBAL var1 var2;)**

```
*********************************************************************;
** code Execution flags – YES will execute – .YES will not           **;
*********************************************************************;
%let extract_data_files = .YES; * Flag to run the Extract data macro  **; *not run;
%let process_data_step1 = .YES; * Flag to run the Process step 1      **; *not run;
%let process_data_step2 = .YES; * Flag to run the Process step 2      **; *not run;
%let process_data_step3 =  YES; * Flag to run the Process step 3      **; * runs;
*********************************************************************;
* The USER libname is a reserved libname that sends all work directory files to the defined space;
Libname USER ‘c:\SESUG_2010\Work_area_that_is retained’;

%macro Conditional_Step_A;
 ... SAS code to extract the data ...
%mend Conditional_Step_A;

%macro Conditional_Step_B;
 ... SAS code to process the data ...
%mend Conditional_Step_B;

%macro Conditional_Step_C;
 ... SAS code to process the data ...
%mend Conditional_Step_C;

%macro Conditional_Step_D;
 ... SAS code to process the data ...
%mend Conditional_Step_D;

%macro execute;
   %if &extract_data_files  = YES %then %do;
         %Conditional_Step_A;
   %end;

   %if &process_data_step1 = YES %then %do;
         %Conditional_Step_B;
   %end;

   %if &process_data_step2 = YES %then %do;
         %Conditional_Step_C;
   %end;

   %if &process_data_step3 = YES %then %do;
         %Conditional_Step_D;
   %end;

 %mend execute;
 %execute;
run;
```

CONCLUSION

This paper has been fun to write, the tips span the simple one liner, to a complex set of macros to develop and test code. I wish I could say that all of this code can be “Cut-and-Pasted” directly into your programs, but the editor and character differences between SAS and Microsoft Word/2007 do not always translate correctly when the data is copied using that method. I tried to use tips available in BASE SAS only. My e-mail address is below if you need help.

CONTACT INFORMATION

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

Name	William E Benjamin Jr
Enterprise	Owl Computer Consultancy, LLC
Work Phone:	602-942-0370
Fax:	602-942-3204
E-mail:	William@OwlComputerConsultancy.com