PhUSE 2016

Paper CC06

How Macro Design and Program Structure Impacts GPP (Good Programming Practice) in TLF Coding

Galyna Repetatska, Chiltern, Kyiv, Ukraine

ABSTRACT

Good program structure and macro design are critical to have streamlined statistical analysis and reporting. To develop efficient, easy-to-update and data-sensitive tools, programming conventions for open-code are not enough; one should think about program design. Combining simple methods and building them together like gears in a machine gives an effect that outreaches simply using individual components. This paper illustrates how GPP (Good Programming Practice) implementation in daily practices improves data traceability and makes programming a creative and exciting process. Samples of SAS macro and pseudo-code will be substantiated with GPP conventions and recommendations.

INTRODUCTION

The first priority in clinical trials is to follow a high standard in data collection and processing. Exploratory analysis needs more flexibility and adaptability to change. An attainable goal for data analysis is to optimize programming efforts and achieve a good result from the first draft, prior to validation. Simple rules make the program flexible to data changes and reflect any issues that are point of interest. This helps us as programmers to handle the most complex and routine work quite easily, create our own masterpieces and, finally, pay more attention to the data.

SAS® is used to provide code samples; GPP will serve as a binder to evaluate the effect of one approach or another.

OPEN CODE CAN BE UNIVERSAL

When we start programs from scratch, open code is a starting point to determine program structure and further generalization. Repeated lists of variables and other identifiers can be generalized in macro variables, programming patterns to be arranged in macro, and some macro variables to become parameters.

The macro tools is a universal way to optimize programming, and sometimes we are tempted use it for relatively simple steps and then repeat macro calls for each usable instance, such as decimal aligning or another type of formatting, calculating results separately for each table cell, category, column or row. This forces programmer to be very attentive to the data and track changes with each rerun and update macro calls accordingly. But as a matter of fact, the LOG file increases nearly at macro length, and it becomes difficult to find significant issues, even with the program full of user warnings and notes. Furthermore, when it comes to rearranging code to repeat for a sequence of outputs, we must deal with iterations or imbedded macros of two or more levels, whilst their own local variables and datasets that may overlap and become hard to debug. Thus, even for experienced programmers, attention concentrated on programming updates rather than analysis and following standards.

In fact, to make a more universal programs and macros for TLF analysis, we need to minimize the use of local macros, but still maintain high level of generalization. SAS provides the necessary tools for generalization of calculation and analysis, while programmers' tasks usually consist of a few common steps:

A. Subset and prepare data for analysis;
B. Perform calculations with standard procedures and get result; brief check for duplicates and other issues;
C. Format result to output cells and arrange to table structure;
D. Create and save TLF outputs.

An attainable goal is to minimize the number of repeated actions and programming efforts referring to data structure rather than repeated sub-macro sequences. Number of commands in the code can be compared not with a number of cells, rows, or columns, but rather with the number of statements in specification or layout necessary to describe.

Suppose layout contains:

Na = number of different types of analysis (numeric, categorical, different kinds of statistics);
Nv = number of variables involved in analysis (AVAL/BASE/CHG; ADSL variables for demographics);
Ng = number of treatment groups or other subject classification;
Np = number of parameters; Nt = number of protocol-defined time points.

There are several ways to arrange processing in a program, with different number of operations required:

- Calculate each cell individually, let it be in open code, macro sequence or loop. Largest number of operations, many intermediate datasets and efforts to combine them:
  $N_{operations}$ ~ Na * Nv * Ng * Np * Nt;
- Thanks to BDS structure with parameter and visit variables, SAS can do it in one loop with BY statement:
  $N_{operations}$ ~ Na * Nv * Ng, or $N_{operations}$ ~ Na * Nv.
  However, pooled subject groups and overall categories are often programmed in separate steps.
- Let us think not in terms of parameters, but types of analysis such as percent, univariates, or some kind of statistic. We can use OUTPUT and other tools to pre-define required categories and their specialties. Number of operations can be minimized and moreover, dependency from other N-s in code can be reduced to the reasonable minimum:
  $N_{operations}$ ~ Na, *or to be more specific,* $N_{operations}$ ~ Na + Nv+ Ng + Np + Nt;

Tools and factors helpful to minimize programming efforts are:

1) BY statement which allows to *repeat analysis by categories*, is settled by the list of variables.
2) SDTM structure for Interventions and ADAM BDS structure perfectly match the use of BY statement;
3) We can reinforce BY with the creation of new categories with OUTPUT statement;
4) Organize code according to GPP to do one operation in one place;
5) Use macro variables/parameters for variables, *lists of variables* and formats. SAS has quite flexible syntax which enables us to combine variable lists and select desired place for statements.

Steps suggested below are just simple elements that can be transformed and replaced. This is the result of some optimization which showed its sustainability in wide sequence of varying or complicated tasks. Although code is not minimal, it is well-protected from common programmer misprints and analysis errors, and draft results are close to final.

SDTM/ADAM DATASETS CONVENTIONS VS. TLF CONVENTIONS

- Define new variables with the ATTRIB statement in order to ensure that the variable properties such as length, format, and label are correct instead of allowing them to be implicitly determined by the circumstances in which they are initialized in the code [1, GPP recommendation]

Guideline 1:

- Improve traceability not just through the code and log, but through intermediate datasets and variables.
- Do not modify incoming variables for output, even in WORK datasets, keep original variables for reference.
- Where necessary, create new variables with independent names for analysis and output.
- Do not overload open code with eventual variables, keep them in macro statement to ensure consistency.
- On the contrary, constructions not allowed in open code, should not be misused in TLF macro. It is preferred to use %IF…%THEN only to provide different options, or %DO loop to repeat outputs we can't do otherwise.

Not Recommended:

```
Data adsl1;
  set adsl;
  output;
  TRT01AN=0;
  TRT01A = "Total";
  output;
Run;
```

+ Treatment variable explicitly shown;
- Modification to other variable not flexible: many changes through code;
- Dataset still named with ADSL* prefix whereas it is not subject-level yet;
- Assigned value makes ADSL.TRT01A(N) variable out of controlled terminology.

Recommended:

```
Data subjtrt;
  length TRTN 8 TRT $40;
  set adsl;
  flag=1; dummy;
  trtn = trt01an;
  trt = trt01a;
  output;
  if not missing(trtn) then do;
    trtn = 0;
    trt = "Total";
    call missing(trt01an, trt01a);
    output;
  end;
Run;
%let bytrt= trtn trt;*trt01an trt01a;
```

+ New TLF-conventional variable created for layout;
+ TRT01A(N) can be changed to another variable in one place;

EXAMPLE OF TLF-CONVENTIONAL VARIABLES

There are obvious benefits of creating new variables and records for report needs, or use macro variables for similar but varying patterns of data. Let's say, TLF-conventional variables are variables which do not overlap with standard terminology, but follow some logic and conventions of output.

1. Subject-level variables:

TRT(N) – numeric and character variable for treatment group. Defines order of treatment groups in output.
Macro variable: &BYTRT
GRP(N) – numeric and character variable for other subject classification and analysis by subgroups.
Macro variable: &BYGRP

Note: If groups are displayed as columns along with spanning treatments, makes sense to combine them into a new variable for convenience into TRTGRP(N). Simple mapping to TRT(N) or GRP(N) makes code more simple, but not replaceable.
Example: `%let bytrt = trtn trt; %let bygrp = BMIGR1N BMIGR1;`

2. Data-level variables:

CAT1(N), CAT2(N),.. – categories for analysis, usually grouping/indenting rows in report, which usually come without numbers, such as "Gender", "BMI (kg/m^2)", AVISIT(N), PARAM(N), PARCAT.
AEBODSYS/AEDECOD can be treated as data-driven categories, since one record per category is selected for analysis.
Macro variable: &BYCAT. Prior to calculation, ensure the uniqueness of record within &BYCAT USUBJID.
When BDS variables used as categories, there can be different sets of categories depending on their role in output: &BYPARM =VSCAT VSTESTCD VSTEST, &BYTPT = EPOCH VISITNUM VISIT.

COL1(N), COL2(N) – leading text columns with ordering used to display statistics for output.
Example: n, Mean (SD), Median, Min-Max for univariates; analysis values for categorical analysis, adjusted for output (formatted values of RACE, ETHNIC, BMIGRP(N), AVALCAT1(N), CRITxx, text explaining AE subset, etc.)
Macro variable: &BYVAL for analysis variable used as column.

Not Recommended:

```
ANRIND = "Overall";
```

- Do not redefine existing data variables – create TLF-conventional for analysis/output;

```
AEBODSYS= propcase(AEBODSYS,".");
AEDECOD = "  " || strip(aedecod);
```

- value changed within data variable and can't be used for subset or other statement anymore; - updated value can be trimmed.

Recommended:

```
col1  = "Overall";
```

* define TLF-conventional vars before (or after) analysis; original values can be kept in macro list;

```
cat1 = propcase(AEBODSYS,".");
cat2 = "  " || strip(AEDECOD);
%let bycat = AEBODSYS cat1 AEDECOD cat2;
```

+ original variables still remain and can be used as reference to original data.

AVALUE(N) – Numeric or character variable for analysis.
If statistical analysis requires another categorization, or some transformation (e.g. log), then PVALUE(N), LOGVALUE etc.

3. Variables to control alignment and decimal places for obtained results:

NDEC [=0,1,2,3…] – number of decimals for output of MIN, MAX of numeric variable to determine rounding for Mean, SD, Median, Q1-Q3. This can be either assigned programmatically for each instance, via format, or even derived automatically based on data AVALC appearance.
Note: This is beneficial to have additional variables rather than refer to eventual instances: if we are going to output all descriptive statistics for ADLB with Proc Means, it makes more sense to create an independent variable, which can be easily moved from one program to another.
&DEC0 - &DECn – global variables to maintain consistency in decimal point aligning. This works well without strip() function and space imputation.

```
%let dec0=3.; %let dec1=5.1; %let dec2=6.2; %let dec3=7.3; %let dec4=&dec3;
```

Note: W.D format outputs the result to right-aligned variable with length W filling leading (W-D-1) places with blanks.

&NDEC – macro parameter with the same role as NDEC. Conventional formats for Mean, SD can be derived as local variables based on &NDEC, &DEC0-&DECn with SYMGET function.

Use decimal formats in macro instead w.d:

```
length col1n 8 col1 $200
       rez $20;
col1n = 1; col1  = "n";
rez = put(n,&dec0.);
output;
col1n = 2; col1 = "Mean (SD)";
rez = put(Mean,&dec1.);
if ^missing(SD) then
rez=trim(rez)|| " (" ||
strip(put(stddev,&dec2.)||")";
output;
```

Utilize local dataset to create and track macro variables

```
%global decv decm decs;
Data _localvars_;
  DecV= symget("dec"||put(&ndec.,1.));
  DecM= symget("dec"||put(&ndec.+1,1.));
  DecS= symget("dec"||put(&ndec.+2,1.));
  _byvar_frq= tranwrd(compbl("&byvar"),' ','*');
  array lvars _ALL_;
  do over lvars;
      call symputx(vname(lvars),lvars);
  end;
run;
```

DecV	DecM	DecS	_byvar_frq
5.1	6.2	7.3	avalcat1n*avalcat1

HOW TO MAKE PROGRAM FLEXIBLE AND TRACEABLE WITH MACRO VARIABLES

- Structure your program to read in all external data at the top, do the processing then produce any outputs or permanent analysis datasets. [1, GPP requirement]
  - Group similar statements together. [1, GPP recommendation]
  - Perform only one task per module or macro. [1, GPP recommendation]
  - Use logical groupings to separate code into blocks. [1, GPP recommendation]

Guideline 2:

- Use BY statement to *minimize* the number of temporary datasets, and keep in one dataset for arrange/review:
- Try to keep *maximum information* from original categorical variables needed for the processing, understanding, ordering and distinction between categories. This will help to understand and track availability and changes within each given parameter and original variables.
- *Optimal solution*: use macro names instead original variables and lists of variables. Since different lists of variables are used at the subsequent stages of program, it makes sense to divide them by roles in and concatenate lists for different instances.

ONE-PROC CALCULATION WITH 'BY' AND 'OUTPUT' FOR THE CASE OF ADVERSE EVENTS BY SEVERITY

In this section sample steps will be demonstrated on how to make procedures output once for each type of analysis and avoid repeated code between categories.
Typically we have datasets of 1) subject-level and 2) analysis, and 4 steps A through D will be presented below for both datasets. Shown below is how to do each step without repeats.

A1: Subject-level categorization and ordering

VIEWTABLE: Work.Subjtrt

	TRTN	TRT	Unique Subject Identifier	Planned Treatment for Period 01	Planned Treatment for Period 01 (N)	FLAG
1	2	Treatment 2	S001	Treatment 2	2	1
2	3	Total	S001			1
3	1	Treatment 1	S002	Treatment 1	1	1
4	3	Total	S002			1

A2: Data-level categorization and ordering

For BDS datasets, simple defining of &BYCAT list would be enough. But for a typical AE table, each event representative must be analyzed at 3 levels of categorization, and needs *post-processing* to select one record per subject three times.
Now, with OUTPUT statement *(key action is to make original values missing for pooled categories),* AEs can be analyzed in one loop, with results prepared for analysis in one dataset.

- LVL (level of categorization) will be supplementary for further data-driven sorting based on frequency; in fact, *LVL=nmiss(*AEBODSYS,AEDECOD*)*. But we can operate with LVL independently from incoming variables.
- CAT1 can be created after processing, but earlier initialization is right in place to have a non-missing record for LVL=0, and TLF-conventional variable for output.

```
Data aecat;
length lvl 8 cat1 $200;
label cat1="SOC|   Preferred Term";
  set adae;
  lvl=2;
  cat1="  "||strip(aedecod);
  output;
  call missing(aedecod);
  lvl=1;
  cat1=propcase(aebodsys,'.');
  output;
  lvl=0;
  cat1 = "Subjects with at least|
one <TEAE>";
  call missing(aedecod, aebodsys);
  output;
run;
%let bycat=aebodsys aedecod lvl
                              cat1;
```

Note: Due to the optional list of variables **&BYCAT**, program becomes multi-level; the same logic applicable for 2-level AEs of Special interest, 4-level Concomitant Medication, and so on, plus original values will be kept for reference.

VIEWTABLE: Work.Aecat

	LVL	CAT1	Unique Subject Identifier	Dictionary-Derived Term	Body System or Organ Class	Treatment Emergent Analysis Flag	Analysis Severity/Intensity	Analysis Severity/Inte (N)
142	2	Thermal burn	S049	Thermal burn	INJURY, POISONING AND PROCEDURAL COMPLICATIONS		Moderate	2
143	1	Injury, poisoning and procedural complications	S049		INJURY, POISONING AND PROCEDURAL COMPLICATIONS		Moderate	2
144	0	Subjects with at least one TEAE	S049				Moderate	2

B1: Make subject subset, derive total numbers and default headers for subject groups.

```
%let rnum = 01; * identifier of output and supporting index in WORK tables;
%let insubj = subjtrt; * parameter for incoming subject-level data;
%let selsubj = %str(where saffl='Y'); * parameter for subset;

Proc Sort data=&insubj out=subj&rnum;
  &selsubj;
  by &bytrt usubjid;
run;

Proc Means data=subj&rnum;
  by &bytrt;
  var flag;
  output out=totals&rnum n=Nsub;
run;
```

VIEWTABLE: Work.Totals01

	TRTN	TRT	NSUB	TRTLABEL
1	1	Treatment 1	261	Treatment 1|N=261
2	2	Treatment 2	253	Treatment 2|N=253
3	3	Total	514	Total|N=514

C1: Create column labels for treatment groups and/or global variables for output;

```
Data totals&rnum(drop=_TYPE_ _FREQ_);
  set totals&rnum end=last;
  call symputx("trt" || strip(put(trtn,best.)), put(Nsub,best.));
  trtlabel = catx("|", trt, compress("N=" || put(Nsub,best.) ) );
  call symputx("lbl" || strip(put(trtn,best.)), trtlabel);
  if last then call symputx("ntrt",_N_); *number of subject groups;
run;
```

Dataset SUBJxx contains subjects with necessary **subset** and categorization. All group counts are kept in TOTALSxx dataset and used as denominators; the dataset will contain variables for further column labels/headers, and so on. Default &LBL1-&LBL3 will be used as macro variables for spanning, they are defined explicitly in TRTLABEL

B2: Make *data* subset (B2.1), calculate result with standard procedures (B2.2).

This step is to be repeated just once for each type of analysis. One additional point is to create mock and generate dummy rows for zeros. Some tips for automatic layout will be provided at the end of the paper.

Let's generate a table of TEAEs by Severity, arranged by spanning treatment columns (Mild/Moderate/Severe).

```
%let indata= aecat; * name of incoming dataset;
%let seldata= %str(where trtemfl='Y'); *subset for incoming data;
%let byvar= asevn asev; * analysis value (with ordering);
```

B2.1. In addition, *after subset* we need to select one record per subject within categories defined in &BYCAT.

```
Proc Sort data=&indata out=indata&rnum;
  by &bycat;
  &seldata;
run;
```

To not repeat subject-level categorization for AEs, merge many-to-many SUBJ* with DATA* categorization:

```
Proc Sql noprint;
  create table data&rnum as select * from subj&rnum s, indata&rnum d
  where s._usubjid = d.usubjid;
quit;
```

Select AE with maximum severity:

```
Data datasubj&rnum;
  set data&rnum;
  by &bytrt &bycat usubjid &byvar;
  if last.usubjid;
run;
```

B2.2. ***Calculation of counts in one procedure –*** to be repeated just for another kind of analysis: for AE counts with `data=indata&rnum` statement, or with `BY &bytrt &bycat` to have total counts within category for ordering.

```
Proc Freq data=datasubj&rnum noprint;
  by &bytrt &bycat &byvar;
  tables flag / out=count_subj&rnum(drop=percent);
Run;
```

C2.1: ***Format character cells for output***

- Variable NSUB from TOTALS** will serve as the denominator, no global variables or array needed. This makes code applicable for an arbitrary number of treatment groups without additional/optional programming.
- It has proved efficient to arrange results into table cells right after calculation (optionally merge with template layout), and transpose prepared text cells rather than their numeric compounds.

```
%let dec0 = 3.;
Data res_all&rnum;
  merge count_subj&rnum totals&rnum;
  by &bytrt;
  length rez $20 column $10 collbl $200;
  percent = 100*count/Nsub;
  length _perc $8;
  if percent > 0 then _perc = cats( "(", put(round(percent,1E-10),5.1),"%)" );
  rez = put(count,&dec0.) || "|" || right(_perc);
  *~Create columns to transpose~*;
  column= 10*trtn + asevn;
  collbl = ASEV;
Run;
```

C2.2. ***Transpose results to pre-defined rows/columns and final corrections.***

In general, TRTN and TRTLABEL from TOTALS&rnum are used to form treatment columns.
But for this special case new variables **COLUMN** and **COLLBL** are created. To arrange Severity/Treatment in columns, no need to transpose or merge three times. Just *combine* the variables (Treatment group and Severity) to get *desired* variable name with ID statement. IDLABEL will map Treatment, Severity or their combination to the variable label.

```
Proc Transpose data=res_all&rnum
out=result&rnum prefix=trt;
  by &bycat &byvar;
  var rez;
  id trtn;
  idlabel trtlabel;
Run;
```

```
Proc Transpose data=res_all&rnum
out=result&rnum prefix=trt;
  by &bycat;
  var rez;
  id column;
  idlabel collbl;
run;
```

```
Data result&rnum;
  set result&rnum;
  length cat1n 8; * Assign order by frequency here*;
  array trtcols(*) trt: ; * fill empty cells *;
  do over trtcols;
    if missing(trtcols) then trtcols = put(0,&dec0.);
  end;
Run;
```

D. Reporting. This part is data-specific, sensitive to `&bytrt/&byvar/&bycat` variables, number of treatment columns, category levels, data-specific pagination, etc. However, all additional variables included in `&bycat` (e.g. AECAT, APERIOD) can be included in the BY and WHERE statements, or #BYVAL( ) for titles and footnotes.

```
Proc Report data=result&rnum missing spacing=1 split='|' headline headskip ls=133;
  column aebodsys cat1n cat1 ("&lbl1."trt11-trt13) ("&lbl2." trt21-trt23) ("&lbl3."
trt31-trt33);
   define aebodsys / group width=10 noprint;
   define cat1n / order order=internal noprint;
   define cat1 / order flow width=24 spacing=0;
   define trt: / display flow width = 8;
   break after aebodsys/skip;
Run;
```

DIFFERENT TYPES OF ANALYSIS FOR DEMOGRAPHICS AND BASELINE CHARACTERISTICS

This kind of table reflects a sequence of categories based on different variables. Let's prepare datasets for each kind of analysis: Proc Freq for qualitative and Proc Means for quantitative analysis. To check the difference between treatment groups with Fisher p-value for 2x2 tables, PCAT will be populated.

```
Data data_qual;
  length group $40 cat1n 8 cat1 $200
         col1n 8 col1 $200 pcat $200;
  set subjtrt;
  group = "QUAL";

  cat1n=1;  cat1="Gender";
  col1n=sexn; col1=put(sex,$genderf.);
  pcat = sex;
  output;
  cat1n=3; cat1=vlabel(race);
  col1n=racen;
  col1=tranwrd(propcase(race),' Or ',
                              ' or ');
  pcat= ifc(race='WHITE',race,'NON-WHITE','');
  output;
Run;
```

```
Data data_quan;
  length group $40 cat1n 8
         cat1 $200 avalue ndec 8;
  set adsl;
  group = "QUAN";

  cat1n=2;
  cat1="Age";
  avalue = age;
  ndec = 0;
  output;
  cat1n=4;
  cat1="Duration at Study
(weeks)";
  avalue = DURSTUDY;
  ndec = 1;
  output;
Run;
```

Note: COL1N and COL1 will be assigned as "n", "Mean (SD)" etc. after Proc Means call

Output categories created to get all results in one step:

group	cat1n	cat1	col1n	col1	pcat	USUBJID
QUAL	1	Gender	1	Male	M	DEM005
QUAL	3	Race	1	White	WHITE	DEM005
QUAL	1	Gender	2	Female	F	DEM006
QUAL	3	Race	2	Black or African...	NON-WHITE	DEM006

```
Proc Freq data=data_qual;
  by trtn trt cat1n cat1;
  tables col1n*col1/out=freqs;
run;
```

```
Proc Means data=data_quan;
  by trtn trt cat1n cat1 ndec;
  var avalue;
  output out=means;
run;
```

group	cat1n	cat1	avalue	ndec	USUBJID
QUAN	2	Age	58	0	DEM005
QUAN	4	Duration at Study (weeks)	69.7	1	DEM005
QUAN	2	Age	55	0	DEM006
QUAN	4	Duration at Study (weeks)	18.2	1	DEM006

USEFUL TRICKS OF PROC SQL TO GENERALIZE TYPICAL STUDY-SPECIFIC PROGRAMMING

Lists of parameters, as well as data-driven formats can be created as macro variable and printed with SQL:

```
Proc Sql;
select distinct cats(avisitn,"='",avisit,"'") as _visfmt into:_visfmt
       separated by ' ' from data&rnum;
select distinct strip(paramcd) as _parmlst into:_parmlst
       separated by ' ' from data&rnum;
quit;

Proc Format;
  value avisfmt &_visfmt;
run;
```

Having *variable lists as BY parameters*, let's generate data-driven mock to fill empty cells.
It works fine if necessary &BYVAL values appear *at least once* in dataset.

```
%let byparm=PARAMCD PARAM; %let byvis= AVISITN AVISIT; %let byval=AVALC;

Proc Sort data= data&oid nodupkey out=byparm&oid(keep=&byparm); by &byparm; run;
Proc Sort data= data&oid nodupkey out=byvis&oid(keep=&byvis);   by &byvis;  run;
Proc Sort data= data&oid nodupkey out=byval&oid(keep=&byval);   by &byval;  run;

Proc Sql ;
  create table shell&oid as select * from byparm&oid, byvis&oid, byval&oid;
quit;
```

FROM OPEN CODE TO MACRO DESIGN

Science is what we understand well enough to explain to a computer.
Art is everything else we do (Donald Knuth)

Macro variables and TLF conventions suggested above are the background for powerful macros, whether designed for study-specific needs or arranged for certain type of analysis. Let us go back to the basic steps of programming as it is implemented to TLF outputs and see some reasoning for how to combine them into macros better.

A: Subset and prepare data for analysis. We can use different types of categorization inside one program.

- In exploratory analysis, variables are not always available in an ADAM dataset and need to be derived initially in TLF code. This should definitely be done outside of TLF macro.
- **Prepare data *structure and variables***. Sometimes we have different types and levels of categorization. Several kinds of categorization of ADSL and BDS-based datasets can be generated and used as input data. Macro is typical but not best place for it, since this way is task-specific. Moreover, we can create a few input datasets in one program and pull them into macro as parameters. That is, by means of TLF-conventional variables we can produce different AE categorization and treatment group within one program.
- **Subset of subjects and observations**. This is good starting point for the macro. Parameters &insubj and &indata with subsets &selsubj and &seldata will be used to transform categorized data.

B (perform calculation) and C (format result) can be divided, but often go together, since analysis in one procedure does not need much macro programming. Makes sense to create a sub-macro for each type of analysis.

D: Output result. This is more specific to the data and variables. But moreover, pulling output in one macro does not allow us to run program for reviewing WORK results and keep output files unchanged. Also output paths and options not always replaceable from study to study. It had proven its efficiency to split result and output into separate macros:

```
%result(01, <list of parameters>);
%reportA(01);
```

```
%result(02, <list of parameters>);
%reportB(02);
```

CONCLUSION

Overall, this process is useful to align programming both with GPP and ADAM/CDISC standards, and for thinking forward about results, be it output, macro or program. Practically, the first point is to have reasoning what and why to do (or not) things. Combination of known methods, adding one step outside of given task can have effect of consolidating hundreds line of code, like SAS usually does.

REFERENCES

1. http://www.phusewiki.org/wiki/index.php?title=Good_Programming_Practice

CONTACT INFORMATION

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

Galyna Repetatska
Chiltern
51B Bohdana Khmelnytskogo str.
Kyiv / 01030, Ukraine
Mobile: +380 96 311 3445
Email: Galyna.Repetatska@Chiltern.com