Paper CT14

Big SAS® Code Navigation & Management

Fred Ross, Quanticate International Ltd, Wilmslow, United Kingdom
Jessica Whittaker-Dixon, Quanticate International Ltd, Wilmslow, United Kingdom

ABSTRACT

Substantial quantities of code can be difficult to navigate, debug and manage if poorly planned and laid out. Therefore, when starting a program anew or taking over previously established code it is imperative that extra effort is made to make the code straightforward and easy to understand for not only yourself but anyone who may inherit the code from you in the future. This paper will explore techniques and best practices to achieve this in your SAS® programs, whether starting with existing code or from scratch. Focus will be on larger SAS programs and how they can be accessible and reusable to developers of any level, including and going beyond the traditional standard good programming practices to delve into more advanced techniques and ideas for good program management. Techniques include using SAS functions and procedures to provide summaries and critical information for navigation and debugging of code.

INTRODUCTION

At some stage in a programmer's career they will have to write or take over a program that is "big", i.e. having many lines of code. Greater care is required when approaching programs of bigger size and this paper will act as a guide and will explore ideas, techniques and best practices for navigating and managing large SAS programs, looking at code structure, comments, sections and naming conventions.

Lastly, a macro creating prefixes for datasets is presented as a proof of concept of topics covered in this paper acting as an example of dataset ordering and naming conventions. This paper is relevant to anyone attempting to develop a large SAS program from scratch or taking over an existing large SAS code. The ideas explored in this paper should be useful for any SAS working environment.

GOOD PROGRAMMING PRACTICES

Whilst every programmer has their own preferred style of programming, what must be at the forefront of every programmer's style is Good Programming Practice (GPP). Naturally we all follow our own internal style, we follow Standard Operating Procedures (SOPs) and also code things in a certain way or a way that we think is proper or correct. This paper proposes that a user takes some time to define their own style within the framework of compulsory/mandatory GPP, consistent at least for each individual program. It is important not to make any massive sweeping changes on how we code within a program and maybe even across a project.

Compulsory GPPs will include those from programmer's company, the client and any other mandatory SOPs that must be followed. These will form the base layer of a programming style that can be built off. Beyond this base there are many other sources from which a programmer can draw ideas and inspiration from whilst still retaining or conforming to GPP.

A good example of third-party sources is PHUSE's very own Good Programming Practice Guidance Document [5].

Other third-party sources include programming papers such as this one. Though other papers may cover some other topic the recommended practice within the paper can be considered GPP and influence how a programmer develops their programming style.

Base Step: Mandatory Practices from SOPs
Third Party Step: Ideas and concepts raised by other sources. Including GPP in SOPs that are optional.
Personal Style Step: Looking at the preceding steps and deciding on one's own final style whilst conforming to Good Programming Practice that the programmer wishes to follow tying ideas with the others already explored.

In continually developing and utilising Good Programming Practice big code projects can be approached more confidently and with a clear goal of what needs to be achieved from a technical quality standard perspective.

PLANNING & SECTIONING

Sectioning work is a common approach to any complex problem, so it is no surprise that it plays such a key role in handling large code. Programmers naturally make their problems into smaller, more manageable modules, to then tackle individually. As SAS is a procedural language, this modular approach is a natural part of programming, however modularisation [1] is a large topic in the wider world of computer science.

SECTION TYPES

The first stage of planning large code should be to consider the types of sectioning that will be applied to the code by creating an outline of the pseudocode. Below are four of the major section types; procedural, variable, observation and input. A well laid out large code is likely to combine all four in some way, however certain datasets may contain one type of section more than the others. It will vary by dataset, and it is the responsibility of the author to find the right sections for their programming style and output dataset.

- **Procedural Sectioning**

Procedural sectioning breaks code down by process, and often by applied SAS procedure. For example, a section dedicated to SAS PROC SORT statements, or to applying formatting. Other examples include transposing, summarising, baseline calculations, merging or macros.

- **Variable sectioning**

In many data models there are naturally grouped variables, such as timing variables or flags. These are often calculated together and will often make sense to be located together within the code. For example, SDTM IG 3.2 groups variables into Identifier, Topic, Timing, Qualifier and Rule [2]. Alternatively, code may be sectioned by variable category such as subject-level, character or numeric.

- **Observation sectioning**

In other cases, it makes sense to split code based on a vertical division of the intended dataset, rather than horizontal. A common example of this would be laboratory parameters, where it would be natural to create a section for different tests, or a questionnaire dataset. This could also appear on a more general scale, with a section for derived observations or for specific epochs. These sections are most likely to be appropriate for a repeat calling macro.

- **Input sectioning**

Depending on the structure of the input and output data, these sections may overlap with variable or observation sectioning. Here, the programmer has multiple sections in which all derivations can be performed from the datasets being read in; for example, a demography section wherein the demography dataset is read in, and all the required processing is completed for these variables.

As mentioned above, the ideal code will mix together all these styles of sectioning. A common approach is to begin and end with procedural sections in the form of dedicated sorting and merging. A vital signs dataset is likely to use a large number of observational sections, whilst a subject-level dataset will be more suited to variable sectioning. Please note that while preparing a section plan is important, it will always be subject to change. Study-needs can evolve, and new data can pose new challenges. Be aware that structuring of the code also needs to be adaptable.

NAMING CONVENTIONS

A good naming convention agreed upon early on can make a big difference in late stage debugging and updates. Below are six key rules for naming sections:

- **Unique dataset names**

Having a unique dataset name ensures datasets are never overwritten. This is also preferable for debugging as the programmer only needs to run the entire code once and then look in the work library without any risk that a dataset is overwritten by a different data step or procedure. Overwriting datasets is generally bad practice.

- **Suffixes and prefixes**

Using suffixes and prefixes for datasets and variables produced in certain sections, allows for easy tracking of workflows and sorting of the work library. Do note however that this rules out using numeric prefixes, as datasets cannot start with a numeric character in SAS.

- **Be succinct but descriptive**

Succinct sections and proper naming conventions for datasets makes it easier to manage the code as it gets bigger. Comparing and matching different names is easier to do when the name is shorter as many library

viewers may only show the first 8 characters by default. An example of a good length and dataset name might be A_BBB_X, where A is a section name and BBB is additional information.

- **Order datasets chronologically**

In the above rule an _X is placed at the end of the dataset name. This is to keep the datasets sorted in the work library and is a very widely used technique. However, note that the ordering of sections should be done with caution, if a new section must be added later in production, it could result in having to rename many datasets.

- **Stay relevant**

At first a simple 'A' 'B' and 'C' sectioning might seem ideal: it's short, alphabetical and easy to remember. However, updates and debugging will be made difficult. Whilst sections are fresh in the minds of the production programmers in the first week of production, after a few rounds of quality checking some of the section names may start to seem meaningless.

- **Well documented**

Much like every aspect of code development, a choice of naming convention should be well documented in the comments in case the code needs to be updated, adapted for new purposes or rerun by another programmer. It will also make code review easier at later stage.

Later in this paper there are some potential solutions for automating naming conventions using macros that will help in keeping section and dataset labelling consistent throughout big code.

SECTION & PROGRAM HEADERS

A section can begin with a comment, a macro call or even a bookmark. The section header is an excellent opportunity to document sectioning and naming decisions, and good documentation is at the cornerstone of good programming practice. There are many designs for section headers and dividers, formed from various comment formats. For large code, more information may be stored in a section divider than just a section header name, such as a description of the code within the section, the input and output datasets or any parameters to be used and updated. It can also be used to define section specific macros to help improve consistency throughout the section. Below are a few suggestions on ways that one could format section headers.

```
/*=====================*/
/* Main section        */
/*=====================*/
/*description:         */
/*                     */
/*=====================*/
```

```
*-------------------;
*Sub-section        ;
*-------------------;
*output datasets:   ;
*                   ;
*                   ;
*-------------------;
```

```
/*=====================*/
/* Main section        */
/*=====================*/
%let section_name = LAB;
%let input_ds = LAB1;
/*=====================*/
```

```
/**Main Section Macro Call**/
%macro main_section(in=
                    ,out=

,prefix=);
%*in = first used dataset;
%*out = output dataset for merging;
%*prefix = prefix applied to      all datasets;
%mend;
```

```
/*==MAIN SECTION==*/
%let prefix = LAB;

*===sub-section===*;
%let prefix = LAB1;
%let subprefix = &prefix.1;
%let subprefix = &prefix.2;

*===sub-section===*;
%let prefix = LAB2;
```

```
/*==================*/
/*===MAIN SECTION===*/
/*==================*/
%let section_name=LAB;
%section_starting_macro;
```

Figure 1: Suggestion for format of section headers

It should be noted that these examples use "forward slash comments", i.e. /*…*/ for main section headers, and for "regular comments", i.e. * …*; for sub-sections. This way, the programmer can easily comment out large chunks of code within sections and comment out entire sections if needed. Alternatively, to inactivate a section a large uncalled macro can be created around the section. Also, notice that the macro example uses macro comments (i.e. %*…;), which prevents the comment being written to the log.

Not only are section headers important but also the overall program header. The best programs will start with a program header containing all relevant information. Important things to include are information such as which external datasets are being read in, any outputs the program is creating, concise purpose of the program, among other things. Special consideration for big SAS code would be to include a small summary of how the program functions. A list of sections and purposes can be very helpful for future navigation. Should a programmer inherit or come back to a program after an extended period, a quick glance at the header should tell them exactly where to look in the program in order to adapt, debug or continue creating the program.

OTHER CONSIDERATIONS

Different environments for SAS will have different functionalities designed to help organise the code. This paper will not go through all features available, as the list is extensive and will be constantly evolving as new versions of SAS are released. However, do consider investigating the features of your SAS environment that can benefit your code layout. For example, in SAS 9.4 for Windows, by default, the combination of shift and F2 keys creates a bookmark. As many bookmarks as needed can be set, and pressing F2 alone will jump through all bookmarks, allowing for scrolling between sections easily.

ORDERING THE CODE

Ordering large code properly will not only improve the readability, but may also have a positive effect on the efficiency of the program. Much like working with big datasets, big code may result in a large amount of read and write processing that will slow down the run time, and in turn the programmer's own productivity. Sorting and merging are the biggest culprits of this, and whilst they are a cornerstone of most SAS programs, they can be minimised by ordering and sectioning.

SORTING

Starting big code with a procedural section focused on sorting is a very popular approach, known as data pre-processing. From a planning and sectioning perspective, it establishes naming conventions and indexes for the incoming raw data. A common mistake is not sorting uniquely at this stage, which leads to further sorting being required at a later stage. When starting with a sort procedure section, think whether further sorts will be needed later in the code, and if so, reconsider the initial sort to combat this.

```
proc sort data = rawdata.labs out = work.alb (where = (paramcd="ALB"))
                                    work.gluc (where = (paramcd="GLUC"))
                                    work.sodium (where = (paramcd="SODIUM"))
                                    nodupkey dupout = dups_labs;
    by usubjid visit param;
run;
```

In the above example of a proc sort, it should be noted that we have a single input dataset, but 3 output datasets, one for each parameter. Due to advance sorting, we know that we will need all three separated for the upcoming processing. By using the proc sort statement in this way, the program will only read in and sort `rawdata.labs` once, rather than three times. This is a common technique when handling big data, but it also works well for big code as the run time is improved, especially over multiple sorts of different raw dataset. In addition, there is a NODUPKEY and DUPOUT. Removing duplicates should be avoided, but by applying the following code we verify whether the duplicates dataset is completely empty:

```
proc sql noprint;
    select * from dups_labs;
    %if &sqlobs = 0 %then %put USER: rawdata.labs contains duplicates;
    drop table dups_labs;
quit;
```

Or alternatively:

```
data _null_;
    set dups_labs;
    put 'USER: rawdata.labs contains duplicates';
    stop;
run;
```

This makes it easier to spot issues in the code early on, as well as keeping the work library clear of unnecessary information. Many different suffixes can be used here, including but not limited to; USER, ISSUE, NOTE, WARNING and ERROR. However, the PHUSE GPP guidelines [5] suggest avoiding the WARNING and ERROR

options, as they may cause confusion with SAS generated warning and errors. Additionally, a programmer may wish to add the date and initials to a log printed comment, if this is consistent with GPP's. As always, check what the relevant programming SOPs state within your organisation before finalising a method.

As a final consideration, if SQL is being used regularly or the programmer would like to implement more advanced techniques for handling bulk datasets, the SQL primary key system could be used. By defining or amending a table in a PROC SQL statement, it can state which variables should act as the unique sort order. This information is stored in the SQL dictionaries, and can be referenced by PROC SQL in the future. This has powerful applications for bulk sorting, merging and accessing [3][4].

BRANCHING & MERGING

An issue that programmers may come across when updating their code, is finding out that after updating one dataset name, unexpectedly later merges cease to work. As a code begins to branch out from the original trunk of the program, it may become harder to keep track of the various uses of every dataset. But how can this be planned for and fixed? What can be done to stop this issue in its tracks, or to fix the issue in hindsight?

The first suggestion is perhaps a less sophisticated method, but also one of the easiest to implement. Take a pen and paper and begin to draw a simple flowchart or spider diagram, following the main chains of datasets and branching off as per the logic of the code. As SAS is procedural, the only deviation from a usual flowchart will be adding a macro, which can be included as a separate flowchart. Diagram 1 shows example illustration of this technique. Different layers will form naturally in the flow chart, which should line up with the sections of the code. Note that not every data step is listed, just enough for the flowchart to connect.

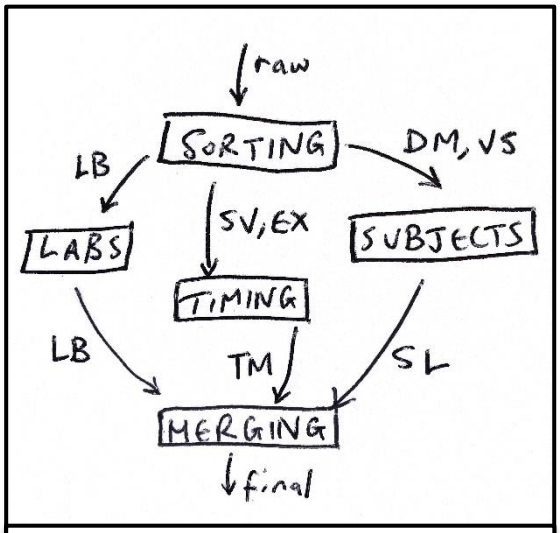

Diagram 1

Alternatively, the log can be utilised to get a better understanding of the code. Using simple statements such as:

```
data _null_;
    put "N""OTE AP: Section XXX has started";
    put "N""OTE AP: Input datasets = ";
    put "N""OTE AP: Output datasets = ";
run;
```

Or:

```
%put %str(N)OTE AP: Derivation for variables XXX, YYY, ZZZ complete;
%put %str(N)OTE AP: Merged variable AAA for calculating BBB;
```

This code will output information to the log, with initials printed on the line. Then either manually or using a utility SAS program, a programmer can read the log with only the lines which contain their initials. Consider printing the section headers discussed earlier in this paper, with as much or as little information as is required for the complexity of the code.

The final suggestion is to produce one system of overarching macros which can track a dataset and the section names throughout the code. The section below discusses how this would work and shows a prototype of what such a macro could look like, along with the advantages and disadvantages this would offer.

PREFIX & WILDCARD MACROS

The below macros are a proof of concept for ordering and numbering of dataset names. This macro may only be suitable in some cases and is not necessarily meant to be used in every program but is a further development of ideas already raised in this paper.

The idea of this macro is to generate a prefix, stored in the macro variable "&prefix", for each dataset created in a program. When the work library is opened it will mean every dataset is stored in order of creation. This will allow ease of debugging and can allow programmers to create a narrative for their work library alone.

A condition for the following to work, each dataset name after the prefix macro variable within the dataset should be entirely unique. Every time a new dataset is created %prefix should be called. In addition, when a dataset needs to be read, then the second macro %prefix_wildcard should be called once. This macro searches the work library for the unique dataset name and stores the actual dataset name including prefix in a macro variable matching the unique dataset name.

Should the previous guidance be followed as specified in this paper regarding dataset naming, by being descriptive in the purpose and function of a dataset, then through the work library it will be clear what is going on without looking at the code. This turns a work library of a large SAS program that may have many datasets, from chaos to a refined ordered map of a program.

```
*============================================================================;
* Macro Define Section: Section in which both the %prefix and %prefix_wildcard    ;
* macros are defined.                                                              ;
*============================================================================;

* The macro prefix creates a prefix for datasets created this will follow the form
section_sequence-number eg the first prefix in section A will be
A_001_unique_dataset_name;
%macro prefix;
%* Define global macros seqnumber prefix and section to create a unique prefix for
a new dataset;
  %global seqnumber prefix section;
%* If seqnumber macro variable already exists add 1 to it else create the macro
variable as equal to 1;
  %if %symexist(seqnumber)=1 %then %do;
      %let seqnumber=%sysfunc(putn(%eval(&seqnumber+1),z3.));
  %end;
  %else %do;
      %let seqnumber=1;
  %end;
%* Create prefix macro variable;
  %let prefix=&section._&seqnumber._;
%mend prefix;

* The macro prefix_wildcard allows to essentially put a wildcard before a dataset
name. This way every dataset should have a unique name then it will create a macro
variable from the unique_dataset_name that has the prefix worked out;
%macro prefix_wildcard(unique_dataset_name=);
  %global &unique_dataset_name;
%* Proc Sql to find the prefix of the unique_dataset_name in the sashelp.vmember
dataset;
  proc sql noprint;
      select memname
      into :&unique_dataset_name
      from sashelp.vmember
      where libname = "WORK" and memname like upcase('%'||"&unique_dataset_name")
      ;
  quit;
%mend prefix_wildcard;
```

```
*============================================================================;
* Section A: Section to show example of using %prefix and %prefix_wildcard macros.;
%let section = A;
*============================================================================;
%prefix;
data &prefix.sdtm_input_1;
    usubjid=1;
run;
%prefix_wildcard(unique_dataset_name = sdtm_input_1);

%prefix;
data &prefix.first_data_step;
    set &sdtm_input_1;
run;
%prefix_wildcard(unique_dataset_name = first_data_step);

%prefix;
proc sort data=&first_data_step out=&prefix.first_data_step_sort;
    by usubjid;
run;
```

REFERENCE CODE

When taking over and having to adapt a large SAS code it can be a daunting task. There could be any number or even all the following problems, ranging from the code being poorly commented, no comments at all, datasets being overwritten any number of times, unclear structure and poor indentation.

It is therefore important to do a quick read of the code to determine which course of action to take. A useful feature of SAS Enterprise Guide (EG) is the indentation tool, this can instantly make the code slightly more readable making skim reading a bit easier. If the working environment is not EG then code can be copied into EG then copied back into the working environment that is being used after indentation is applied.

Following a skim read, i.e. a quick read skipping some detail, of the code the decision to format and improve the existing code or start from scratch should be made.

STARTING FROM SCRATCH METHOD

Should the code be in a poor state the most efficient use of a programmer's time may be to setup a new program with its own structure and write the code from scratch utilising the original code in parts in the new program.

FORMAT AND IMPROVE METHOD

If it is chosen to format and improve the code, then essentially apply techniques and ideas raised in this paper. The process below gives an idea of what to follow, apart from the first step which should always be done first, the other steps can be done in the order that is found personally most appealing.

Check for dataset overwrites, if they occur remove the overwriting. The first step of this could be just to add suffixes such as _1 and _2 to duplicated dataset names.
If indentation hasn't already been applied, while skim reading the code, it can be applied here.
Separate the code out into sections and order data steps and procedures logically.
Add comments, not only are these important for the finished code but they are helpful to personally keep track.
Implement naming conventions of datasets.

CONCLUSION

Managing big SAS programs can be a daunting task should a programmer approach such programs unprepared or approach in such a way they would any other program. However, if they follow the advice and tips presented in this paper any programmer can develop their own programming style alongside Good Programming Practice. A programmer will then be prepared to not only preemptively avoid common issues encountered with large volumes of code but also rectify these if they are present in already existing code.

REFERENCES

[1] The Advantages of Modularization in Programming, G S Jackson
https://www.techwalla.com/articles/the-advantages-of-modularization-in-programming

[2] SDTM Model Concepts and Terms, Judy Li
http://pharma-sas.com/sdtm-model-concepts-and-terms/

[3] Create Index Guide, SAS Support
https://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/viewer.htm#a002473673.htm

[4] Assessing SQL Dictionaries, SAS Support
https://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/viewer.htm#a002473711.htm

[5] PHUSE Good Programming Practice Guidance
https://www.phusewiki.org/wiki/index.php?title=Good_Programming_Practice_Guidance

ACKNOWLEDGMENTS

Thanks to Will Greenway for his great input on the initial paper idea, Paula Finch and Piotr Karasiewicz for their review of the paper. Thanks to Daryna Khololovych & Rosie Fulton for all their help and support.

CONTACT INFORMATION

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

Author Name: Fred Ross
Company: Quanticate International Ltd.
Email: fred.ross@quanticate.com

Author Name: Jessica Whittaker-Dixon
Company: Quanticate International Ltd.
Email: Jessica.Whittaker-Dixon@quanticate.com