

Good Programming Practice In Macro Development

Table of Contents

Introduction	2
Importance of Good Programming Practices for Macros.....	2
Gathering User Requirements	3
Architecture/Design of a Macro.....	3

Introduction

We focus on recommended Good Programming Practices (GPP) that should be followed when developing SAS Macros and are meant to compliment the PHUSE [Guidance on Good Programming Practice](#) from Good Programming Practice in Health and Life Sciences and summarised in the [PHUSE GPP Summary](#).

The *macro* capability is offered in many computer programming languages. A SAS macro implements *code generation*, based on dynamic, data-driven, run-time conditions. The macro language variables, functions, and statements are interpreted by the macro processor during compilation. The text that is generated (resolved) by the macro language can then be interpreted as SAS code and run by the SAS compiler. A SAS macro is used where the underlying program is well understood, and certain parts need to be modified based on changes in run-time conditions.

Importance of Good Programming Practices for Macros

Following Good Programming Practices help to enhance the quality of a macro. An effective macro is one that is simple, readable and adaptable. If the macro is poorly designed, does not have a logical flow, is not adaptable, or not readable, it defeats the very purpose of being a macro.

Because a macro is composed of SAS code and generates resolved SAS code based on the macro parameter settings, Good Programming Practice (GPP) for coding a SAS program applies equally to coding the SAS code within a macro and the SAS code that the macro generates, as discussed in [PHUSE GPP](#) and summarised in [PHUSE GPP Summary](#). GPP for macro coding extends these concepts to the SAS macro language and the unique capabilities it brings to the SAS programming environment.

Macros generally have a long use life and are often updated by different programmers who adopt them in their work. The scope of a macro may initially be small and grow over time as multiple programmers make updates to the macro. If GPP are not followed the programs often become overly complex, hard to maintain and hard to use. A few overarching guiding principles follow.

- **Standardisation**
 - Maintaining a similar look and feel across macros for a protocol, indication or organisation enhances the usability of the macro as the users know what to expect. Standardisation across macros may include a consistent program layout, uniform parameter names and consistent error checks and displays. Standardisation will assist in building consistent code and help to standardise the development process.
- **Keep it simple and maintainable**
 - Programs that have a clear focus and smaller scope are easier to maintain and to use. Trying to do too much in one program usually requires many parameters making the macro overly complicated and difficult to use. Include requirements that will be needed for 80% to 90% of the studies.

- Macro updates happen over time and will often be done by many different people. Follow the general PHUSE GPP guidance to make the programs readable and include thorough commenting. Refrain from using obscure or undocumented coding elements and aim for clear and concise program code.
- **Efficient**
 - Some macros that read through large datasets or process many domains for a protocol can be time consuming to run. It's important to implement efficient coding strategies to optimise runtime and system resources.

Gathering User Requirements

Like any other program, a macro is built based on the requirements that triggered its need. Understanding the requirement clearly is the key to building a strong base for a macro. As a programmer, when you start to write a macro, you will straight away head for the %macro statement. This is not necessarily the best approach. The starting point is summarising the requirements.

The scope and requirements of a macro must be thoroughly documented in order to facilitate easy execution of the macro, ensure the correct output is generated and assist the developers in understanding the reasoning and methodology so they can accurately develop or modify the program. It is extremely important to gather the user requirements accurately and completely, and document them clearly and concisely. Documenting clear requirements will reduce the chance of ambiguity and misunderstanding in the development stage of the program and will reduce rework. It's important to identify and involve the correct users in the requirement phase, especially when much functionality is needed. The developer must collaborate closely with the users and communicate possible coding or technical challenges. User requirements are often collected in an indexed manner as described in Appendix I. It is helpful, in consolidating the requirements input, to store the requirements documentation in a shared space that all involved users and developers have write access to. Some of the user requirements may be copied to the program specification document and can also be used when drafting user-guides.

Architecture/Design of a Macro

Sharing a standard macro template that models a consistent layout and includes standard programming code for routine functions will help maintain consistency in design of all the macros in an organisation or department. See a sample in Appendix II.

- **Program Header**
 - As PHUSE GPP states every program should start with a header. A macro header should contain all the GPP required/recommended contents as well as the following additions.
 - Assumptions and Constraints – Describe the assumptions or pre-requisites of the macro. State the limitations/constraints of the macro.

- Dependencies – State if the macro is a successor or predecessor, if applicable or if it calls other macros.
- Parameter Description – Describe each of the parameters to be passed in the macro, their use, description, example values, whether the parameter is optional or mandatory/required, and indicate any assigned default values.
- It's recommended to put the start of the macro out to the SAS log along with the parameter values passed to the macro.

```
%put -----;

%put --- Start of %upcase (&sysmacroname) macro          ;

%put ---                               ;

%put --- Macro parameter values                        ;

%put ---  input_dataset  = &input_dataset                ;

%put ---  output_dataset = &output_dataset              ;

%put -----;
```

- **Defining Macros and Parameters**

- The following GPP should be followed when defining the macro and the macro parameters:
- Name of the macro should be meaningful and should be relative to the purpose of the program.
- File name of the macro should match the macro definition name to work correctly with the SAS autocall facility.
- Macro parameter names should be meaningful and should not have generic names.
- Keyword parameters should be used instead of positional parameters.
- Order of the parameters should be consistent with the program logic.
- All parameters should be clearly described in the header.
- Optional parameters with defined default values should be used and the use of required parameters should be limited.

- **Modular Structure**

- While writing a macro, it's important to maintain a logical flow, moving from the one logical step to another, like a story. This can be achieved by breaking your macro down into logical modules (this is especially important for large multi-functional macros). Modularly structured macros are easier to read, debug, upgrade and maintain. In a modular structure each module or section has a specific purpose such as reading input data, transforming data, loading data, presenting data etc., and as with any program the modules or sections are connected/linked to each other in a logical structure.

- **Error Checks**

- A bullet-proof error section should be included in the top of the macro that outputs clear errors to the SAS log and exits the program when invalid parameter values are entered. Additional error checks should be made when unexpected scenarios are encountered in the macro, such as when insufficient data exists. The macro generated error messages are especially important in situations where the error messages generated by SAS are unclear.
- **Minimal Impact on SAS Session**
 - Executing the macro should have minimal impact on the SAS session. All macro variables defined within a macro should be declared as local variables with the %local statement unless needed outside of the macro. Having single or double underscores in the beginning of each local macro variable and temporary dataset assures these elements will not overwrite others defined in the SAS session or calling program.
 - If SAS options need to be changed in a macro, it's recommended to keep the user defined settings/default values in macro variables and reset to the original settings in the bottom of the macro.

```
%let old_mprint      =%sysfunc(getoption(mprint));      à Beginning of the
macro

options &old_mprint ;                                à End of the macro
```

- **Additional Design Best Practices**
 - Macros should be data driven. Hard-coded information should be avoided when it can be extracted from the data. Typical examples are determining the number of treatments in a study, dynamically defining array sizes and determining the type of a variable.
 - When evaluating equality/inequality conditions for macro variables, use either double quotes to surround the parameters and comparator values, or SAS quoting functions to mask the effect of special characters, such as the dash ('-') which would be interpreted as a minus sign in the condition.
- **Example**
 - Instead of: %if &display_pvalue eq N and &DocLibr = %then

Use: %if "&display_pvalue" eq "N" and "&DocLibr" = "" %then

or %if %nrquote(&tablspprs) eq "N" and %nrquote(&doclibr) = ""
%then

- Statisticians often want to review the exact statistical code that is executed when a macro is run. Setting the SAS option MPRINT on right before executing statistical procedures and turning it off right after the procedure will send the resolved statistical procedure statements to the log without cluttering up the log with unnecessary MPRINT output.

- One should avoid including nested macros within iterative loops as it makes it difficult to read and maintain. It is important to keep the macro simple and avoid unnecessary complicated processing steps.
- Before ending your macro program, always clean up the unnecessary temporary datasets. Include the name of the macro in the %mend statement. This improves readability and is particularly useful when there are iterative loops and several lines of code.
- **Executable Code**
 - Some regulatory agencies are requesting program code to assist them with understanding the analysis logic used to generate the numbers in the tables, listings and figures (TLFs). Programs containing heavy macro code are hard for the reviewers to understand and can slow down the review process. Macro code is often required in the error checking section and in the code to format the TLFs, and the agencies are not interested in these sections of the program. A clean, focused program containing resolved macro code can easily be created by changing the mprint and mfile options. The code below backs up the system options to be changed and assigns a program location to the mprint filename. Turning the mfile and mprint options on sends the resolved code to the output_filename.sas program file.

```
%let omautolocdisplay=%sysfunc(getoption(mautolocdisplay));
```

```
%let omprint=%sysfunc(getoption(mprint));
```

```
filename mprint "directory_path/output_filename.sas" lrecl=2048;
```

```
options nomautolocdisplay mfile mprint;
```

- It's optimal to include these statements right before reading the input data. The following code clears the mprint filename, turns off generation of the resolved code and sets the mautolocdisplay and mprint options back to what they were before changed.

```
filename mprint clear;
```

```
options nomfile &omautolocdisplay &omprint ;
```

- Closing the resolved code file after creating the dataset used to generate the table focuses the output program to only the section important to the reviewer.
- Care should be taken to assure there are no exit paths in the program between the statements that start the resolved code and the statements that close the file or the file will remain open and may cause issues the next time the macro is executed.
- **How to Structure Logic**
 - Unlike traditional SAS, the SAS macro language is not straight forward to troubleshoot. It's helpful to include a DEBUG parameter, that enables one to

switch the debugging options on and off. Strategically placed put statements should be added to write macro variable values and comments on the logical execution of the macro out to the SAS log when the debug option is on. Users don't like to see a lot of unnecessary output in the SAS log and the debug output can be suppressed when debug is turned off. When the debug option is on the temporary datasets created by the macro should be retained so they can be used to debug.

When reading user input (parameters or pre-existing macro variables) using a SAS macro, one needs to keep the concept of GIGO (Garbage in, Garbage out) in mind. Everyone makes mistakes and every good piece of code should at least make provision for some amount of input error. Input should be validated in the form of unit tests and regression testing (backward compatibility). Code in general, especially macros, needs to be robust. Robustness may be facilitated by using %goto statements and strategic %if conditions where applicable.

It is important to keep in mind that macro variables can be overwritten and may not exist in some scenarios (local vs global macro variables). As per GPP, it is always advantageous to remove extra macro variables which will not be utilised at a later stage (%symdel) and to make sure that a macro variable exists (%symexist) and has the expected value when coding