



Techniques for Writing Robust SAS Macros

Martin Gregory

PhUSE Annual Conference, 19-21 Oct 2009, Basel

Overview



- Why robustness?
- Summary of techniques
- Types of macros considered
- An example from real life: finding duplicates
- Applying the techniques
- Summary

Why robustness?



- a robust program should
 1. be well documented
 2. be well tested
 3. validate inputs
 4. handle failures in a controlled way
 5. notify caller or calling program of outcome
 6. clean up after itself
- we consider points 1 and 2 to be self-evident truths and concentrate, in this paper, on points 3-6

Why robustness?



- Robustness involves some extra effort, so when and why strive for it?
 - macros which will be used by many other programmers
 - macros which are part of a system of macros
- Benefits are:
 - minimizes support effort for the macro
 - avoids unnecessary execution of code
 - provides an interface for calling programs to react appropriately

Summary of techniques



- parameter design
- parameter validation
- checking and reacting to outcomes
- returning status to the caller
- restoring the environment
- testing
- documentation

- Definition of robustness
- Summary of techniques
- Types of macros considered
- An example from real life: finding duplicates
- Applying the techniques
- Conclusions

Types of macros considered



- **Full-step** macros
 - generate one or more complete steps or global statements
- **In-step** macros
 - generate code fragment which are part of a step
- **Pure** macros
 - use only macro statements and generate no code to be compiled by the SAS supervisor
- Techniques described apply primarily to full-step macros. Some apply to in-step and pure macros as well
- We consider only **full-step** macros

- Definition of robustness
- Summary of techniques
- Types of macros considered
- **An example from real life: finding duplicates**
- Applying the techniques
- Conclusions

Example: finding duplicates

- We wish to examine observations with duplicate ids in a dataset to determine what differences there are
- This is what PROC SORT in SAS 9 can do, i.e. one of each duplicate in in the OUT= dataset:

DATA=

Id value n

Aa 1.056 1
Ab 3.141 2
Ca 0.155 3
Ac 2.712 4
Ca 0.666 5
Aa 1.888 6
Bb 0.137 7
Ca 0.333 8

OUT=

Id value n

Aa 1.056 1
Ab 3.141 2
Ac 2.712 4
Bb 0.137 7
Ca 0.155 3

DUPOUT=

Id value n

Aa 1.888 6
Ca 0.666 5
Ca 0.333 8

Example: finding duplicates

- This is what the macro %finddups will do, i.e. produce a dataset with **all** observations having duplicate ids:

DATA=

Id value n

Aa 1.056 1
Ab 3.141 2
Ca 0.155 3
Ac 2.712 4
Ca 0.666 5
Aa 1.888 6
Bb 0.137 7
Ca 0.333 8

OUT=

Id value n

Aa 1.056 1
Aa 1.888 6
Ca 0.155 3
Ca 0.666 5
Ca 0.333 8

%finddups: initial version

```
%macro finddups(data=_last_, out=_dups, by=, where=) ;  
    %local i lastby ;  
  
    proc sort data=&data out=&out ;  
        by &by ;  
        %if (%quote(&where) ne ) %then %do ;  
            where &where ;  
        %end ;  
    run ;  
  
    %let i=1;  
    %do %while(%scan(&by,&i,%str( )) ne );  
        %let lastby=%scan(&by,&i,%str( ));  
        %let i=%eval(&i+1);  
    %end;  
  
    data &out ;  
        set &out end=__end;  
        by &by ;  
        if first.&lastby + last.&lastby < 2 then output &out ;  
  
    run ;  
%mend ;
```

Applying the techniques



- parameter design
- parameter validation
- checking and reacting to outcomes
- returning status to the caller
- restoring the environment
- testing
- documentation

- named parameters generally preferred:
 - reduces risk of assigning values to wrong parameters
 - sensible defaults may be specified
 - order is irrelevant
 - helps to maintain backward compatibility
- naming convention:
 - follow SAS names where possible
 - be consistent
- parameters with fixed or categorical values:
 - be consistent
 - be case insensitive

Parameter design



```
%macro finddups(data=_last_, out=_dups, by=, where=) ;
```

- uses SAS names
- standard SAS default for data= parameter
- default for out= parameter

Applying the techniques



- parameter design
- parameter validation
- checking and reacting to outcomes
- returning status to the caller
- restoring the environment
- testing
- documentation

- motivation:
 - avoid running code with incorrect or incomplete input
- strategies:
 1. no validation: may be valid for certain circumstances
 2. formal, i.e. required parameters are supplied, have correct type and have formally correct values
 3. content, i.e. supplied parameters have valid values in the context, e.g.
 - specified datasets exist
 - specified variables exist on the corresponding datasets
 - supplied parameters have valid values in range

Parameter validation: %finddups – first version



- Add before the proc sort step the code:

```
%if %quote(&data) eq %str() %then %do ;  
    %put ERROR: &sysmacroname: Parameter DATA is required.;  
    %return ;  
%end;  
%if %quote(&by) eq %str() %then %do ;  
    %put ERROR: &sysmacroname: Parameter BY is required.;  
    %return ;  
%end;
```

- Formal validation only
- proc sort step will determine the existence or otherwise of the input data set and by variables

Parameter validation: %finddups – generic version



```
%local i _params _param nmiss ;
%let _params=.data.out.by.;
%let i=1;
%do %while(%scan(&_params,&i,.)^=%str()) ;
    %let _param=%scan(&_params,&i,.);
    %if %quote(&&&_param) eq %str() %then %do ;
        %put ERROR: &sysmacroname: Parameter %upcase(&_param) is required.;
        %let nmiss=%eval(&nmiss+2**(&i-1)) ;
    %end;
    %let i=%eval(&i+1) ;
%end;
%if &nmiss>0 %then %return;
```

Parameter validation: %finddups – generic version



```
%local i _params _param nmiss ;
%let _params=.data.out.by.;
%let i=1;
%do %while(%scan(&_params,&i,.)^=%str()) ;
    %let _param=%scan(&_params,&i,.);
    %if %quote(&&&_param) eq %str() %then %do ;
        %put ERROR: &sysmacroname: Parameter %upcase(&_param) is required.;
        %let nmiss=%eval(&nmiss+2**(&i-1)) ;
    %end;
    %let i=%eval(&i+1) ;
%end;
%if &nmiss>0 %then %return;
```

Parameter validation: %finddups – generic version



```
%local i _params _param nmiss ;
%let _params=.data.out.by.;
%let i=1;
%do %while(%scan(&_params,&i,.)^=%str()) ;
    %let _param=%scan(&_params,&i,.);
    %if %quote(&&&_param) eq %str() %then %do ;
        %put ERROR: &sysmacroname: Parameter %upcase(&_param) is required.;
        %let nmiss=%eval(&nmiss+2**(&i-1)) ;
    %end;
    %let i=%eval(&i+1) ;
%end;
%if &nmiss>0 %then %return ;
```

Parameter validation: %finddups – generic version



```
%local i _params _param nmiss ;
%let _params=.data.out.by.;
%let i=1;
%do %while(%scan(&_params,&i,.)^=%str()) ;
    %let _param=%scan(&_params,&i,.);
    %if %quote(&&&_param) eq %str() %then %do ;
        %put ERROR: &sysmacroname: Parameter %upcase(&_param) is required.;
        %let nmiss=%eval(&nmiss+1) ;
    %end;
    %if &i=some_value %then %do ;
        specific checking for param &i ;
    %end ;
    %let i=%eval(&i+1) ;
%end;
%if &nmiss>0 %then %return ;
```

Applying the techniques



- parameter design
- parameter validation
- checking and reacting to outcomes
- returning status to the caller
- restoring the environment
- testing
- documentation

Checking and reacting to outcomes



- motivation: after a step or global statement, does it make sense to continue execution?
- strategies:
 - check &SYSERR after step execution: >4 means error
 - check &SYSLIBRC or &SYSFILRC after LIBNAME or FILENAME statement: 0 means success
 - check actual outcome of step: e.g. if the sort step in %finddups returns zero observations, the data step is unnecessary

Checking and reacting to outcomes



```
proc sort data=&data out=&out ;  
  by &by ;  
  %if (%quote(&where) ne ) %then %do ;  
    where &where ;  
  %end ;  
run ;  
%if &syserr gt 4 %then %do ;  
  %put ERROR: &sysmacroname: error sorting the dataset. ;  
  %return ;  
%end;
```

Applying the techniques



- parameter design
- parameter validation
- checking and reacting to outcomes
- **returning status to the caller**
- restoring the environment
- testing
- documentation

Returning status to the caller



- Use global macro variable
- Allow caller to name this macro variable through a parameter to the macro
- Define convention for return codes, e.g.:
 - 0 is success
 - n is a specific status, where n is documented
 - 9999 is returned when a situation other than success and not anticipated by the macro occurs

Returning status to the caller: adding the *RC* parameter



```
%macro finddups(data=_last_, out=_dups, by=, where=, rc=rc_dups);  
  %if %quote(&rc) eq %str() %then %do ;  
    %put NOTE: &sysmacroname: Using RC_DUPS for the RC parameter ;  
    %let rc=rc_dups;  
  %end ;  
  %global &rc ;  
  %let &rc=9999 ;  
  
  /* macro body here */  
  
  %let &rc=0 ;  
%DONE:  
%mend ;
```

Returning status to the caller: setting the *RC* parameter



```
%macro finddups(data=_last_, out=_dups, by=, where=, rc=rc_dups);  
  %if %quote(&rc) eq %str() %then %do ;  
    %put NOTE: &sysmacroname: Using RC_DUPS for the RC parameter ;  
    %let rc=rc_dups;  
  %end ;  
  %global &rc ;  
  %let &rc=9999 ;  
  
  /* sort step here */  
  %if &syserr gt 4 %then %do ;  
    %put ERROR: &sysmacroname: error sorting the dataset. ;  
    %let &rc=101 ;  
    %goto DONE;  
  %end;  
  
  /* data step here */  
  
  %let &rc=0 ;  
%DONE:  
%mend ;
```

Applying the techniques



- parameter design
- parameter validation
- checking and reacting to outcomes
- returning status to the caller
- restoring the environment
- testing
- documentation

Restoring the environment



- Restoring the environment helps avoid interfering with subsequent calls to the macro or to other macros
- Temporary datasets should be deleted except for debugging purposes
- Temporary file or library references used only within the macro should be cleared
- If the macro sets system options, it should restore the original values
- Restoring the environment may be required after abnormal termination: if not use %return, otherwise use %goto as in the following example

Restoring the environment



```
/* body of macro */

%let &rc=0 ;
%DONE:
  proc datasets lib=work nolist ;
    %if %sysfunc(exist(_tmp)) %then %str(delete _tmp);
    ...
  quit;

  %if %sysfunc(fileref(_tmpfref)) %then filename _tmpfref clear ;

%mend ;
```

Applying the techniques in practice

- Define templates in editor/development tool
 - minimal impact on development time
- Add checking code as each step is added to the macro
 - using template has minimal impact on effort
- Include tests for the techniques, in particular for parameter validation and checking outcomes
 - additional effort
- Include information on the status codes in the documentation
 - additional effort
- Support of the macro
 - reduced effort

Summary



- The techniques described are simple and easy to code
- For re-usable macros, the effort of attaining robustness is repaid in reduced support
- Using an appropriate development tool (emacs, eclipse, multiedit), the effort can be further reduced
- Simplifies building and testing of macro systems

Questions

