

Defensive programming techniques

John Woods, ICON Clinical Research, Dublin, Ireland
Jennie McGuirk, ICON Clinical Research, Dublin, Ireland

ABSTRACT

Often when running programs in SAS®, we come across unexpected values, or get unexpected results from a proc/data step. Code should be written to handle all case scenarios, not just for the scenarios of the expected data. Implementing defensive programming techniques as standard practise when coding in SAS will minimise the risk for errors when processing real data to obtain statistical output for Tables, Figures and Listings.

The aim of this poster is to describe some useful programming techniques to minimise these risks, such as implementing error handling code around proc/data steps using if/case statements, and writing warning messages to the log file for the programmer's attention. Problems will be identified from some of the more significant messages in SAS log files, and solutions given on how these can be overcome. The poster will describe how to implement these practices in every day programming.

INTRODUCTION

Errors and warnings in the log file immediately alert the programmer to a problem. However, some SAS statements which also need attention may go unnoticed. The option '*nonotes*' should never be used, as some NOTE statements can identify problems in the code. This poster will identify NOTE messages and other significant SAS statements in log files which need attention, and will offer suggestions on how to overcome any potential problems arising from these statements, as well as offering helpful hints on guarding against other common pitfalls when coding.

The poster's audience is the SAS programmer and those new to SAS programming, as the aim is to instill good programming techniques.

POTENTIALLY PROBLEMATIC LOG STATEMENTS (WITH SOLUTIONS)

Specific SAS statements from the log file will be identified as potentially problematic. These statements should either not appear in the log file, or have an explanatory note attached to show that the programmer has considered the potential risks and appropriate action has been taken. The problems will be described, and solutions provided.

The following are typical examples:

[1] Catering for unexpected data

```
'NOTE: Missing values were generated as a result of performing an operation on  
missing values'
```

This statement is put to the log when the program tries to perform calculations with variables which have not been assigned a value. The easiest way to overcome missing/invalid data is to check the data before processing it and assign defaults; if statements and case statements are most efficient to catch any unexpected values.

```
data newsals;  
  merge salaries payrise;  
  by id;  
  
  newsal = 0; * set default value;  
  if sal eq . then  
    put "Warning: Sal is null for employee:" id;  
  else if payrise eq . then  
    put "Warning: Payrise is null for employee:" id;  
  else newsal = sal + payrise;  
run;
```

or

```
proc sql;
  create table newsals as

  select s.id,
  case when (s.sal ne . and p.payrise ne .)
    then s.sal + p.payrise
    else 0 end as newsal,

  case when (s.sal eq .)
    then 'Warning: Sal is null for employee:' || s.id
  when (p.payrise eq .)
    then 'Warning: Payrise is null for employee:' || s.id
  else '' end as errflag

  from salaries as s
  left join payrise as p
  on s.id=p.id;
quit;
```

The 'missing values' note is also a common note to find in the log file when variables are uninitialized. If the note 'NOTE: Variable x is uninitialized' accompanies the 'missing values' note, then this is a serious issue where the variable is not recognized, and the program should be checked (i.e. for variable spellings, missing set/merge statements, missing retain statements, etc.).

```
'NOTE: Variable x is uninitialized.'
'NOTE: Division by zero detected at line x column y'
```

Both of these can be checked with an 'if' or 'case' statements. If they are allowed, then variable x will be set to missing and the result of any further calculations using mathematical operators involving x will also be set to missing. If there is no need to check for missing values, then there are functions available in SAS to ignore missing values. For example, the sum() function can be used to sum up all non missing values. A note would also be sent to the log for any uninitialized variables if present.

```
'NOTE: Numeric values have been converted to character'
'NOTE: Character values have been converted to numeric'
```

If a number is recorded as a character variable, SAS will do the conversion and add the conversion of character to numeric note to the log. However, the SAS functions input(x, format) and put (x, format) should be used for character to numeric and numeric to character conversion respectively. For example;

```
data a;

  a = 2;
  b = '4';
  c = a + input(b, best.);

run;
```

```
'NOTE: Invalid numeric data'
```

The invalid numeric data message results from SAS not being able to perform a mathematical operation because it cannot determine the numeric value of the character variable, as above. A new function, notdigit(), in SAS 9 provides an easy way to check for numeric data in character fields. It checks the parameter for the position of the first non numeric character and returns a value of 0 if not found. For earlier versions, the following example would suppress the log message by using format modifiers '??', allowing the programmer to handle the check.

```

data test;

    x = '32M';
    y = 0;

    if input(x,?? best.) eq . then put "ERROR: ....";

    /* for SAS 9 use: if notdigit(x) ne 0 then put "ERROR: ..."*/

    else y=input(x,5.)+1;

run;

```

[2] Highlighting potential issues

```

'NOTE: Input data set is empty.'
'NOTE: The data set ... has 0 observations and ...'

```

This can be overcome by first checking the number of observations in the dataset using a macro variable, and if 0, writing this to the log so that it can be flagged as checked.

```

%macro Checkobs(data=);
    %global obs;
    proc sql noprint;
        select count(*) into :obs from &data;
    quit;

    %if &obs eq 0 %then
        %put WARNING: ** No obs in &data.;
    %else %put &data has %trim(%left(&obs)) observation(s). ;

%mend;
%checkobs(data=...);

```

Or, if the number of observations is 0, then the program can be prevented from processing data.

```

%macro process();

    %if &obs ne 0 % then %do;

        proc print data = ....;
        run;

    %end;
%mend process;

%process;

```

```

'NOTE: At least one W.D format ...'

```

This note arises from trying to fit values which are too large into specified formats. This can be checked by using a put statement to print the maximum values (in a macro variable) to the log. By using **proc sort** to sort by the descending highest x values (for example) first, and then select the first observation to isolate the highest value. Then we can reference the log before creating the variable in a data step and assigning the correct format.

```

proc sort data=all_vals out = max_val;
    by descending x;
run;

data _null_;

    set max_val(obs=1);
    if x ne . then call symput('xmax', trim(left(put(x, best.))));
    else put "WARNING: x is null";

run;

```

```

%put WARNING: max x is: &xmax;

data test;

    format x 4.1; * formatting x to preferred length based on value of
                  xmax;
    ...
run;

```

Similarly, the problem causing the SAS note containing

```
'...outside the axis range ...'
```

can be highlighted by creating macro variables to find the most extreme points on an axis, and printing these to the log or using them in the axis statement.

```

%let min_x=0;           *create default value;
%let max_x=100;         *create default value;

proc sql;

    select min(x) into :min_x
    from graphdata;
    select max(x) into :max_x
    from graphdata;
quit;

%put Min X is: &min_x;
%put Max X is: &max_x;

```

Or

```

axis2 label = (font = ...)
              order=(&min_x to &max_x )
              ...

```

These are not preventative solutions but they highlight the problems immediately, so that under the pressure of producing final outputs, the programmer doesn't need to spend valuable time debugging these issues.

GUARDING AGAINST COMMON PITFALLS

There are some common pitfalls which can be overcome by adapting a 'presume the worst' approach to the data which the program is being written for. Below are some useful tips which the programmer should be aware of during programming.

```
'NOTE: MERGE statement has more than one data set with repeats of BY values.'
```

If there are more than one set of unique key variables in more than one dataset in a merge statement, SAS will output the above note to the log file. However, since this can result in corrupted data, it is important to check the sorted datasets before the merge and highlight the discrepancy with an error message. A variation on the **%checkobs()** macro above yields something like the following, where x is the sorted dataset before the merge has occurred.

```

%macro key_vars(data=, vars=);

    proc sql noprint;
        select count(*) as keys into :keys from &data
        group by &vars * key variables;
        having calculated keys > 1;
    quit;

    %if &keys > 1 %then %put
        "ERROR: Repeats of key variables &vars in &data";

%mend;

```

```
%key_vars(data=x, %str(age,id));
```

A method of eliminating array memory errors by assigning array lengths dynamically using DIM() or LBOUND() and HBOUND() could be used to specify lower and higher bounds in the do loop:

```
array t{*} pt1-pt20;  
k=dim(t);           * length of the array t;  
  
do i=1 to k;  
    ... (some code) ...  
end;
```

or

```
array t{*} pt1-pt20;  
low=lbound(t);      * lower bound of the array t;  
high=hbound(t);     * upper bound of the array t;  
  
do i=low to high;  
    ... (some code) ...;  
end;
```

Using these array functions means that during the running of the code, the lengths are assigned dynamically, so as to minimize the risk of memory errors, and only using what's needed.

CONCLUSION

The poster specifically points out potential hazards lurking within SAS programs and offers some defense techniques to guard against corruption. The aim, however, is to encourage the programmer to consider all possibilities when working with data, and that their programs are written so that any unforeseen eventualities are caught. Checking the log file is paramount to identifying problems in the code, and these techniques, if not preventative, will highlight any potential issues clearly. I have written 'Warning' notes to the log in most cases. A more stringent approach would be to write 'Error' notes to the log file, and force the SAS program to fail so that the programmer must check these messages for data or programming issues.

CONTACT INFORMATION

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

Author Name	John Woods
Company	ICON Clinical Research
Address	South County Business Park, Leopardstown
City / Postcode	Dublin
Work Phone:	+353 (0) 1 291 2418
Fax:	+353 12912200
Email:	woods@iconirl.com
Web:	www.iconclinical.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.