

Debugging Made Easy

Andrew Ratcliffe, Ratcliffe Technical Services Limited

If we cannot eliminate human error we are unlikely to be able to eliminate the creation of bugs in computer programs. However, the inevitability of bugs in all but the smallest of programs should not deter us from attempting to minimise the number of bugs; and those bugs that we do produce should be easy to investigate and resolve.

This paper focuses on making the process of bug investigation easier. It lists and discusses coding techniques, tools, and debugging techniques that the author has found to be successful. The paper focuses on Base SAS coding, but includes occasional mention of SAS/AF issues.

Introduction

All of my programs have bugs in them. And even if my programs appear bug free, you can be pretty sure that other aspects of the system on which they run have got bugs in them, i.e. SAS software and the operating system. So there's a fair chance that even the simplest of my programs will not run as designed on certain days of certain years under certain circumstances. It's a fact I have to live with!

Having come to terms with this fact, I have accrued and developed a number of tools and techniques for getting to the root cause of problems (potential bugs) as/when they occur. These tools and techniques are for use in a combination of situations and at a mixture of occasions in the software development life cycle. Most, however, are for use during the programming phase. If you accept the inevitability of bugs then you can see the benefit of making extra effort in the programming phase to make the bugs easier to investigate and resolve when they occur.

Expect The Unexpected

In my childhood days I was a member of the Boy Scouts. Their motto is "Be

prepared" and I've kept that motto with me all these years. I try to write my programs to expect the unexpected and to handle that situation to a reasonable degree.

Select instead of if

To help me with trapping unexpected values, I generally use *select* statements instead of *if* statements when I am testing for specific values. For instance, if I have a variable that contains gender as 'M' or 'F', I might be tempted to code:

```
if gender eq 'M' then ...do male things...  
else ...do female things...
```

But a safer option is to use the *select* statement:

```
select (gender);  
  when ('M') ...do male things...  
  when ('F') ...do female things...  
end
```

The advantage of using the *select* statement is two-fold. Firstly, you are making the valid values of the gender variable very clear; secondly, the program will bomb if gender contains an invalid value. Thus, you get the earliest warning that something is wrong. If I had used the *if* statement, my program would have continued with an incorrect belief that we were dealing with a female observation.

I use the *select* statement's *otherwise* clause to trap unexpected values, but that presupposes that you can do something in the event that an unexpected value arises.

If it didn't work, do something

There's nothing worse than a program that continues after a problem has occurred. The program will inevitably destroy evidence about the original cause of the problem. It will do this by over-writing variables and data sets, etc. If you want to get to the root of a problem, you need as much good evidence as possible.

After doing something, check it worked. You can use low-level techniques such as the *select* statement I discussed above, and you can use the &syserr macro variable to check the success of a PROC or DATA step.

If it didn't work, use ABORT or STOP or ENDSAS, etc. to indicate failure. You can use ERRORABEND in a batch job.

The topic is discussed in great detail in "Taking Control and Keeping It" by Justina M Flavin et al, Proceedings of SUGI 26. See www2.sas.com/proceedings/sugi26/p074-26.pdf.

In the event of an error, you will want to prevent subsequent steps from being executed. A simple means of doing this is to use the *cancel* option on your *run* statements. *run cancel* tells SAS to just perform a syntax check on the step without executing it. I code *run &cancel* for each of my steps; I initialise &cancel to blank at the beginning of my program; I set &cancel to 'cancel' when I detect a problem and don't want subsequent steps to run. Here's an example.

```
178 options mprint;
179 %macro herbert;
```

```
180 %let cancel = ;
181
182 proc summary data=sashelp.class;
183   class sex;
184   var height wait;
185   output out=summ sum=;
186 run &cancel;
187
188 %if &syserr ne 0 %then
189   %let cancel = cancel;
190
191 data result;
192   set summ;
193   if height gt weight then put 'h>w';
194   else put 'w>h';
195 run &cancel;
196
197 %if &cancel ne %then
198   %put Program did not complete
successfully;
199 %mend herbert;
200
201 %herbert;
MPRINT(HERBERT):  proc summary
data=sashelp.class;
MPRINT(HERBERT):  class sex;
MPRINT(HERBERT):  var height wait;
ERROR: Variable WAIT not found.
MPRINT(HERBERT):  output out=summ sum=;
MPRINT(HERBERT):  run ;
```

NOTE: The SAS System stopped processing this step because of errors.

WARNING: The data set WORK.SUMM may be incomplete. When this step was stopped there were 0 observations and 0 variables.

WARNING: Data set WORK.SUMM was not replaced because this step was stopped.

NOTE: PROCEDURE SUMMARY used:
real time 0.02 seconds
cpu time 0.02 seconds

```
MPRINT(HERBERT):  data result;
MPRINT(HERBERT):  set summ;
MPRINT(HERBERT):  if height gt weight then
put 'h>w';
MPRINT(HERBERT):  else put 'w>h';
MPRINT(HERBERT):  run cancel;
```

NOTE: Data step not executed at user's request.

NOTE: DATA statement used:
real time 0.01 seconds
cpu time 0.01 seconds

Program did not complete successfully

Notice how the data step did not run because of the *run cancel*. The execution of the %if means that the code must be part of a macro, which may not suit all purposes. The same technique can be used if you are calling base SAS code from SAS/AF SCL code.

Conditional test code

I include a lot of conditional test code in my work. For example, I include *put* statements that are only executed if the &debug macro variable has been set to a non-zero value beforehand. This is very useful for re-running code that you know is failing but for which you don't have enough information to figure-out the cause of the problem. I discussed this technique in greater detail in my SEUGI '99 paper entitled "Proactive Debugging". You can get a copy from www.ratcliffe.co.uk/res_papers.htm.

Make It Understandable

It may not be you

It may not be you that has to debug this program. So show some thought for your poor co-worker. Make the purpose of your program as clear as possible. Even if it is you that has to debug it later, you'll be grateful for your earlier consideration.

Neatness is a simple means of raising the level of clarity of your work. Don't underestimate the value of neatness in making your code more understandable for you and others.

No macho coding

We've all seen it! Functions embedded within functions embedded within functions, with buckets of parentheses thrown in for good measure; usage of esoteric bits of SAS functionality that were dropped from the documentation years ago but are still in the product; clever use of advanced mathematical functions to save using a larger number of basic mathematical functions.

Don't do it! Nobody will thank you when they have to maintain your program.

Write for clarity; don't try to impress. If you feel compelled to write complex code in order to achieve significantly better performance or some other goal, make sure you document your reasoning and the logic that you have applied. Use comments copiously.

Coding standards

Adopt coding standards. It is the simplest route to ensuring consistency of code and ease of maintenance for programmers. Gary E. Schlegelmilch's SUGI26 paper entitled "Structuring Base SAS for Easy Maintenance" is a good source of reference. You can get a copy at www2.sas.com/proceedings/sugi26/p220-26.pdf.

Make it explicit

I find it a tremendous help when I look at somebody else's program and I can easily trace back the source of some data. You can help to make data traceable in many ways. I'll discuss just a few here.

Scope your macro variables with %local, and by using methods in SCL. The "scope" of a variable is the range of sections within the program in which the variable is valid. In the following example, the macro variable named jaguar is globally scoped, i.e. it is accessible from any part of the SAS environment:

```
%macro rover;  
  %put #2- &jaguar;  
  %let jaguar = top;  
  %put #3- &jaguar;  
%mend rover;
```

```
%let jaguar=1;  
%put #1- &jaguar;
```

```
%rover;
```

```
%put #4- &jaguar;
```

So when we run the macro, jaguar's values are as follows:

```
#1- 1  
#2- 1  
#3- top  
#4- top
```

But if we tell SAS that jaguar is locally-scoped within the rover macro, the results are different:

```
%macro rover;  
  %local jaguar;  
  %put #2- &jaguar;  
  %let jaguar = top;  
  %put #3- &jaguar;  
%mend rover;
```

```
#1- 1  
#2-  
#3- top  
#4- 1
```

The results differ because we have two macro variables named jaguar in the second example. Within the rover macro we have a second variable who exists only for the life of the macro.

By restricting the life of information in this way, it becomes easier to trace the origin of information. Clearly it's not good practice to have variables with the same name in both the global and local macro environment, but what the examples demonstrate is that if we have a problem with jaguar inside the rover macro, the %local statement tells us that the invalid value for jaguar can only have been set within the macro. So we don't have to look too far for the cause of the problem.

In class-based SCL code, *do/end* blocks have their own scope too.

Another means of making it clear where information has come from is to make it clear what information is being passed between sections of code. In base, I never use `_LAST_` (the last created data set), I always code the data set name.

I avoid using global macro variables and the local/global SCL list.

Parameter lists (for macros and SCL methods) should be treated as your friends. Parameter lists form excellent documentation of what is being passed into and out of a sub-routine, i.e. a macro or SCL method.

Make It Easy On Yourself

Preserve evidence

If you have to investigate a problem, you will want to follow the chain of events backwards to find the original cause. You won't be able to do this if some of the information in that chain is missing.

I never overwrite intermediate (temporary) data sets. Those data sets may not be important in terms of your final results, but they are important if you are debugging. If you call your intermediate data sets something like "temp" then it's preferable to create temp1 from your input data set, and create temp2 from temp1. If you create temp and then over-write it then you have lost the "paper trail".

This approach can result in a large number of TEMPnn data sets, so you should delete these data sets when they are no longer required, e.g. delete temp1 after temp2 has been created from it, otherwise your work library may run out of space. But if you delete these data sets you've lost your paper trail! So, I make the deletion conditional upon a macro variable that indicates whether I am in debug/development mode or in production mode.

Good syntax

Make your syntax good: remove warnings such as type-mismatches. Consider removing messages like uninitialised variables. Compare the following two DATA steps.

```
1  data _null_;
2      numeggs = 12;
3      msg = 'Old Macdonald has '
4          !! compress(numeggs)
5          !! ' eggs' ;
6      put msg= never=;
7  run;
```

NOTE: Numeric values have been converted to character values at the places given by:
(Line):(Column).

4:21

NOTE: Variable never is uninitialized.
msg=Old Macdonald has 12 eggs never=.

```
8
9  data _null_;
10     numeggs = 12;
11     msg = 'Old Macdonald has '
12         !! compress(putn(numeggs,'BEST.'))
13         !! ' eggs' ;
14     never = .;
15     put msg= never=;
16 run;
```

msg=Old Macdonald has 12 eggs never=.

The second DATA step shows how to remove the type mis-match and the uninitialised variable message.

The V8 SAS/AF SCL compiler has compile-time binding, thereby offering more validation at compile time

Write Things Down

A specification

Write it down in words before you code it. If you doubt yourself later (or lose focus) you'll be able to judge if you're doing what you were intended to do.

The simplest of specifications is better than none at all. You may be grateful for it if you loose track of things whilst coding. And if anybody has to maintain your code at a subsequent time, they will certainly thank you for the documentation.

Compile time

If I am distributing compiled code, I have found it useful for some documentation on compile dates/times so that I can cross-reference the distributed code that is causing trouble with the version of the source that I hold. I have a small macro that does this for me in my SAS/AF SCL code. The macro is described in my aforementioned "Pro-Active Debugging" paper.

The macro could be adapted to be used in stored macros and stored data steps in addition to its existing use with SCL.

Build a Good Foundation

Clean the environment

At the start of my programs I try to clean the environment and to validate it. This helps give me a predictable environment in which to execute.

To clean the environment I begin by clearing-out the work library. You can do this with PROC DATASETS:

```
proc datasets lib=work kill;
run;
```

In addition, I often delete any permanent data sets and/or files that I expect my program to create. However, it is not always appropriate to do this. You have to consider whether you would prefer to retain those old data sets and files in the event of the program crashing.

You can validate the environment in a number of ways. Depending upon the type of application that you're producing, you might want to make sure you're running on a platform that your application supports; make sure that the necessary SAS modules are licensed; make sure you have enough

spare disk space for the work library and/or permanent libraries.

You can check to see what platform you're running on by querying the &sysscp or &syscpl macro variables. Use PROC SETINIT to see what modules are licensed.

On Windows, you can find the amount of free disk space using the getDiskFreeSpaceA DLL routine as follows:

```
libname samples 'C:\Program Files\SAS
Institute\SAS\V8\core\sample\';

filename sasctbl catalog
'samples.pcsamp.sasctbl.source';

data;
  success = modulen('*e'
                    , 'GetDiskFreeSpaceA'
                    , 'C:\'
                    , numSectorsPerCluster
                    , numBytesPerSector
                    , numFreeClusters
                    , numTotalClusters );
  freeSpace = numFreeClusters
              * numSectorsPerCluster
              * numBytesPerSector ;
  put success= freeSpace= commal6.;
run;
```

Incremental is best

I much prefer to develop incrementally. It allows me to discover problems one at a time. I can resolve problems as I go. I know that other developers prefer the big-bang approach where they do a great tranche of coding and then try to run and try to debug it. But to my mind, having more than one bug to resolve at the same time serves to complicate the process.

Old is best

Re-use code. Leverage your investment in earlier debugging. If I were looking for an example and I were to take this to an extreme, I would say use PROC REPORT instead of DATA _NULL_.

Sources of Information

The log is your best friend

If a problem occurs, the log could be your best source of information. In fact it may be your only source of information!

In a batch situation, make sure you are retaining the log file. And try to avoid using one log file that is over-written each time the program is run – if the program is re-run after a problem has occurred you may lose valuable information about why the problem occurred on the first occasion.

The log will be a crucial source of information in your debugging session. Write information to the log yourself, don't just rely on SAS. Paul D Sherman's SUGI26 paper "Intelligent SAS Log Manager" has a lot of good information on this topic. See www2.sas.com/proceedings/sugi26/p108-26.pdf.

Bug Mining

So maybe you've written a program, and maybe you've followed some of my recommendations. And now you find you need to investigate a bug. Frank Dilorio's SUGI26 paper entitled "The SAS Debugging Primer" is a very good source of information and guidance. You can find it at www2.sas.com/proceedings/sugi26/p054-26.pdf.

Start at the very beginning

Start at the **top** of the log and look for the **first** warning or error. Find the root cause of your problem. After one problem has occurred, subsequent sections of code are likely to fail. But don't focus on those subsequent sections, start back at the beginning and focus on the first warning/error.

Fix the first problem and you'll probably find that your subsequent problems disappear.

One step at a time

Don't try to rush what you're doing. Take your time and think carefully. Don't jump to conclusions. Try to remove one possible cause of the problem at a time. Keep removing possible causes until the code works. At that point you know that the thing you just removed is your culprit.

When you begin to drill-down into the problem, if you find it difficult to pin-down the problem, take into consideration that it might not be the fault of the SAS program. The fault may lie with any of the other components of your program, e.g. SAS itself or the operating system.

Data step debugger

Be on intimate terms with the DATA step debugger! To find out more, refer to "How to Use the Data Step Debugger", a SUGI25 paper by S. David Riba. See www2.sas.com/proceedings/sugi25/25/btu/25p052.pdf

Code generated by macros

If you have a problem with a macro then things can get extra tricky because the macro debugging tools are so limited. I find it often helps to remove the macro code from the "equation" by copying the generated Base SAS code from the log to the editor (having set *options mprint* before running the code).

However, if you have a lot of code being generated it can be tiresome to remove all those copied line numbers and other paraphernalia from the log. In this case you can use *options mfile* (*options reservedb1* in V6). Used in

conjunction with the *mprint* option and an *mprint* filename statement, *mfile* gives you a "clean" copy of the code generated by your macro. See below:

```
options mfile mprint;

filename mprint
'c:\temp\mprint.txt';

%macro jimbo;
  data;
    length
    %do i=1 %to 10;
      fred&i
    %end;
  8;
  run;
%mend jimbo;

%jimbo;
```

In the log we see:

```
MPRINT(JIMBO):  data;
NOTE: The macro generated output from
MPRINT will also be written to external
file c:\temp\mprint.txt while OPTIONS
      MPRINT and MFILE are set.
MPRINT(JIMBO):  length fred1 fred2 fred3
fred4 fred5 fred6 fred7 fred8 fred9 fred10
8;
MPRINT(JIMBO):  run;
```

And in the external file we see:

```
data;
length fred1 fred2 fred3 fred4 fred5 fred6
fred7 fred8 fred9 fred10 8;
run;
```

Information sources

MSGLEVEL

Using *options msglevel=i* will offer you additional information such as when indexes are being used.

```
95  options msglevel=i;
96
97  data billy (index=(demo));
98    do demo=1 to 1e4;
99      output;
100    end;
101  run;
```

```
NOTE: The data set WORK.BILLY has 10000
observations and 1 variables.
NOTE: Simple index demo has been defined.
NOTE: DATA statement used:
      real time          0.13 seconds
      cpu time           0.10 seconds
```

```
103 data subset;  
104     set billy;  
105     where demo lt 20;  
INFO: Index demo selected for WHERE clause  
optimization.  
106 run;
```

NOTE: There were 19 observations read from
the data set WORK.BILLY.

WHERE demo<20;

NOTE: The data set WORK.SUBSET has 19
observations and 1 variables.

In the example shown above, the msglevel=I option has caused two extra informational messages to be written to the log. The first tells us that the simple index was created; the second tells us that the index was used.

SASTRACE

The SASTRACE option displays detailed information about the commands that SAS/ACCESS sends to your DBMS. Use it in conjunction with the SASTRACELOC option.

DEBUGLEVEL

The “internal” SAS start-up option – debuglevel is something that offers plenty. I’ve not had time to make use of it yet! It has five potential values: testing, normal, debug, fulldebug, and demo. Invoking SAS with debuglevel=fulldebug will result in informational debugging dialog boxes.

More interesting tips like this can be found in Phil Mason’s SUGI26 paper entitled “SAS Tips I Learnt While At Oxford”. See www2.sas.com/proceedings/sugi26/p020-26.pdf.

Conclusion

Bugs are an inevitability of contemporary programming. In this paper I have tried to demonstrate some “defensive” programming techniques that I use in order to make the debugging process as painless as possible.

I encourage you to experiment with these techniques and to develop your own. In my experience it is an investment worth making.

Biography

Andrew Ratcliffe is a freelance SAS software consultant with over 15 years experience of SAS software. He specialises in object-oriented application development. Through his company (Ratcliffe Technical Services Limited) he is able to offer services including analysis and design, consultancy, and mentoring. Andrew is editor of the NOTE : free e-newsletter.

Email: andrew@ratcliffe.co.uk

Web: www.ratcliffe.co.uk

Newsletter subscription:
www.ratcliffe.co.uk/note_colon