

The SAS Debugging Primer

*Frank DiIorio
CodeCrafters, Inc.
Philadelphia, PA*

Background

- ✿ Bugs are a part of programming life
- ✿ Everyone creates them
- ✿ “If you don’t have bugs, you’re not trying hard enough!”
- ✿ They can always be corrected
 - but it requires an open mind
 - and a knowledge of available tools
- ✿ Remember ... “failure is information”



Organization

- ✿ Two sections
 - Approaching problem-solving
 - The debugging toolset
- ✿ Emphasis on Section 1



What *Is* Debugging, Anyway?

- ✿ It *is* “The process of identifying the root cause of an error and correcting it.”
- ✿ It *is not* testing. Testing:
 - is more systematic
 - requires a different mindset than that of the programmer
 - is often performed by non-programmers



Part 1: Behavior

Effective programming and debugging begins with the understanding that it's part instinct, part coding.

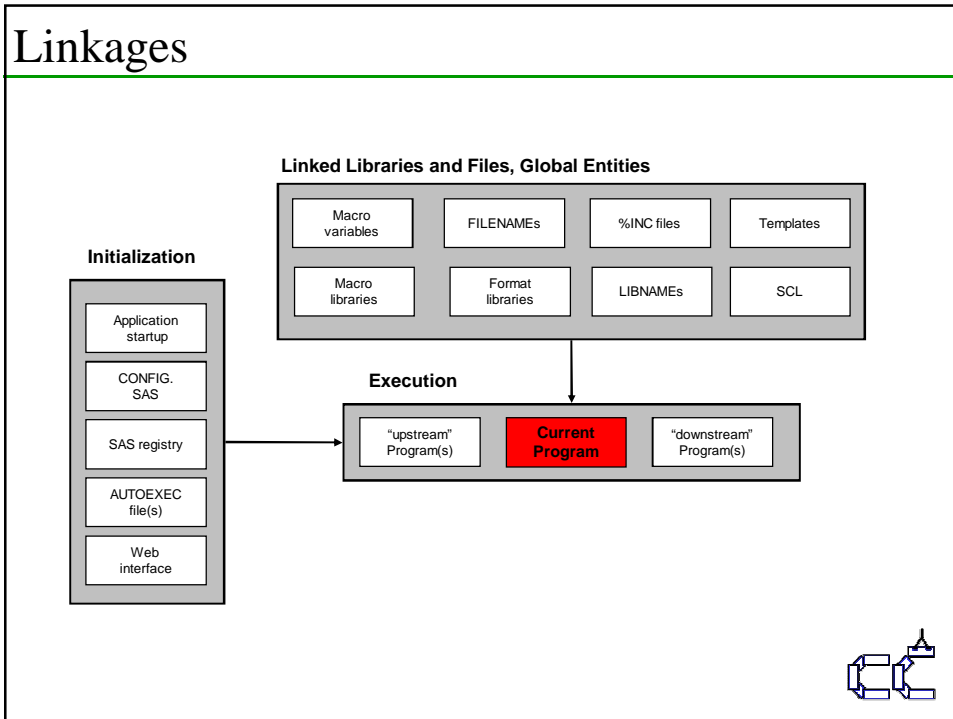
In Part 1, we discuss the background, behavioral aspects of problem solving.



Identify Possible Error Sources

- ✿ Syntax
- ✿ Logic (unit-level)
- ✿ Logic (system-level)
- ✿ Data
- ✿ Entity *example*
- ✿ Execution environment
- ✿ SAS, OS software





Understand Your Programming Behavior

- ☘ "Is the problem familiar?"
- ☘ "Have I seen it before, maybe in some other form"
- ☘ "How do I usually approach this sort of problem?"
 - "... how did I fix it when it broke?"
- ☘ Have the capacity for *objective* self-examination



Relax Your Assumptions about the Source of the Error

- ✚ Your experience is fine, but ...
 - It is limited to being only *your* experience
- ✚ Don't be a slave to routine, automatic ideas about why the code is problematic
- ✚ Open yourself to what are, for you, non-traditional approaches (e.g., SQL v. DATA step)
 - You'll learn more, and the problem may be fixed more readily
 - More on this later ...



Describe the Problem Thoroughly

- ✚ Don't fix "it" until you have a clear idea of what "it" is.
- ✚ Ideally, the problem should be
 - Describable
 - Replicable
- ✚ Begin to identify circumstances and behavior surrounding the bug
- ✚ If you can't stabilize and define the bug, you won't be able to fix the bug. Look for covariates.



Develop/Fix Incrementally

- ✿ Write initial code (or bug fix), then test it
- ✿ **Don't** write and test several items at once
- ✿ If there are “n” locations of debug code and the program is fixed, it may not be apparent where the fix actually occurred.
 - **Don't confound the debug (and learning) process**



Hope That It's a Simple Fix

- ✿ It could be simple
 - “**fat fingers**” **transposing letters**
 - **Missing semicolon**
 - **Etc etc**
- ✿ One error can spawn many downstream errors
- ✿ Debug from the top of the program
- ✿ Debug the *code*, not the *comments*!



Consider Alternative Approaches

- ✿ Sometimes it really isn't simple – bugs can reveal flaws not only in syntax, but (even worse) in logic and design
- ✿ SAS usually allows a multiplicity of approaches, even for simple tasks
- ✿ Consider the merits of redesign
 - **of the program**
 - **of the data**



Abandon Assumptions

- ✿ Never say “that can't happen” when you can clearly see that it is, indeed, happening
- ✿ Assume that *any* statement, program step can malfunction
- ✿ Be literal – “play computer” and read the program exactly as SAS would. This requires distancing yourself from the code.



Suspect Everything

- ✿ Maybe it isn't just your code that's at fault in this particular situation
- ✿ What other changes were made recently? Does the program have a history of being troublesome?
 - Become familiar with the revision (and testing!) history of the program.
- ✿ Look at related programs as well.



Consider This Guy as Well ...

From Wikipedia

- ✿ A **leprechaun** is a type of fairy in Irish folklore, usually taking the form of an old man, clad in a red or green coat, who enjoys partaking in mischief.



Learn and Re-learn

- ✿ Your skills should grow in tandem with the SAS System
- ✿ Most of what you know about a feature is acquired when you first learned it.
 - Updates and enhancements tend to get lost unless you consciously seek them out
- ✿ New features invariably make initial coding (and, later, debugging) easier



Take a Break

- ✿ Walk away from the problem if you can
- ✿ Get a fresh pair of eyes to examine the problem
- ✿ Spend some debugging time away from the keyboard



Ask “Did I Fix the Right Thing?”

- ✿ Did you fix the bug or simply swat one of its manifestations?
- ✿ A hard-coded workaround guarantees the same “fix” will be applied later on to other bugs.
- ✿ Attack all sources – data, program, design, user behavior, *et al.*



Reflect on the Solution

- ✿ What tools were most effective?
- ✿ Was it a logic error? Syntax?
- ✿ If logic or design issues arose, document them, along with tracking numbers, other memory joggers



It Might Be SAS's Fault

- ✿ It can overreact
- ✿ It can get flaky
- ✿ It might have a bug
- ✿ It *usually* tells you what's wrong
- ✿ It does not identify logic errors



Part 2: Tools

A good attitude and self-awareness about how you approach debugging is fine and well, but what about the tools that make it happen?

In Part 2, we discuss SAS tools for debugging.



Use Options to Control Debugging Output

- ✿ Remember when you can invoke the option
- ✿ System options, grouped by:
 - Echo source code in Log
 - Messages in Log
 - Invalid format, variable, dataset references
 - Reaction to errors



Use Options to Control Debugging Output (cont.)

- ✿ Macro-specific, grouped by:
 - Autocall usage (mautolocdisplay, sasautos, mautosource, mrecall)
 - Execution tracing (mlogic, mlogicnest)
 - Error handling (merror, serror)
 - Display of generated code (mprintnest, mprint, symbolgen)



Don't Ignore the Log!

Among the ways it helps you are:

- ✚ Echoes your code
- ✚ Notes data type conversions
- ✚ Identifies creation of missing values
- ✚ Identifies uninitialized variables
- ✚ Describes input and output data sets
- ✚ Contains user-written diagnostics
- ✚ Contains macro diagnostics



Debug with Macros, Macro Variables

- ✚ Various ways to display values of macro variables
 - `%put _global_;` `%put _local_;`
 - Display SASHELP.VMACRO, DICTIONARY.MACROS
- ✚ Macro variables can regulate the amount of debugging output
- ✚ Also use macros ...



Debug with Macros, Macro Variables (CONT.)

✿ Macros can be used to conditionally execute debugging code

- Same principle as macro vars, previous slide ...
- But they can more easily block out entire sections of code
- See next slide for an example



Controlling Debugging Output (1:3)

```

%macro test(parm1=,parm2=,dbg=yes);
  %let dbg = %upcase(&dbg.);
  %if &dbg. = YES %then %do;
    %let star = ; %let cancel = ;
  %end;
  %else %do;
    %let star = *; %let cancel = cancel;
  %end;

  data part1;
    ... DATA step statements ...
    &star. put _n_ = ptid=;
    ... more DATA step statements ...
    &star. put @10 "Count becomes " nFail=;

  proc print data=part1;
  run &cancel.;

  %if &dbg. = YES %then %do;
    ... diagnostic DATA steps, PRINTs, FREQs, etc. ...
  %end;
%mend;

```



Controlling Debugging Output (2:3)

What SAS sees when `DBG=yes`

```

data part1;
  ... DATA step statements ...
  put _n_ = ptid=;
  ... more DATA step statements ...
  put @10 "Count becomes " nFail=;

proc print data=part1;
run;

... diagnostic DATA steps, PRINTS, FREQS, etc. ...
    
```

Callouts:

- &star resolves to null. The PUT statement is executed
- &star resolves to null. The PUT statement is executed
- &cancel resolves to null. The PROC is executed
- &dbg resolves to YES. The diagnostic DATA steps, etc. are executed



Controlling Debugging Output (3:3)

What SAS sees when `DBG^=yes`

```

data part1;
  ... DATA step statements ...
  * put _n_ = ptid=;
  ... more DATA step statements ...
  * put @10 "Count becomes " nFail=;

proc print data=part1;
run cancel;
    
```

Callouts:

- &star resolves to *. The PUT statement is not executed
- &star resolves to *. The PUT statement is not executed
- &cancel resolves to cancel. The PROC is not executed
- &dbg resolves to other than YES. The diagnostic DATA steps, etc. are not executed



DATA Step Debugging Tools

The DATA step is the most problematic part of SAS coding. Lots of tools to help repair code, however:

- ✚ PUT statements (make them meaningful and descriptive)
- ✚ Automatic variables (`_ALL_`, `_INFILE_`)
- ✚ LIST statement
- ✚ Format modifiers → ? and ??
- ✚ IN data set option
- ✚ \$HEX. and other formats



Use “New” Stuff ...

- ✚ Enhanced editor
- ✚ DATA step debugger
- ✚ Dictionary Tables and Views



Use PROCs to Debug DATA Steps

- ✿ Don't limit DATA step debugging strictly to DATA step tools. Also use ...
- ✿ CHART, FREQ
- ✿ MEANS
- ✿ PRINT, REPORT
- ✿ CONTENTS, DATASETS



Comment Your Code

- ✿ While you're developing
 - ✿ While you're debugging
 - ✿ While you're enhancing
 - ✿ ... pretty much *any* time!
- ✿ Don't believe that "any code that was hard to write should be hard to understand"!



Use “Non-Program” Resources

- ✿ Gurus / 2nd set of eyes
- ✿ Manuals, Proceedings
- ✿ The Web (SAS Inst., SAS-L, others)
- ✿ SAS Sample Library; SAS-supplied AUTOCALL macros
- ✿ Tech support: SAS Institute, in-house



Thanks for Coming

Your comments are welcome and valued.
Contact: Frank@CodeCraftersInc.com

Visit www.CodeCraftersInc.com

