Paper 54-26

The SAS® Debugging Primer

Frank C. DiIorio
Advanced Integrated Manufacturing Solutions, Co.
Durham NC

All reasonably complex programs are born flawed. A hallmark of an experienced programmer is a talent for effectively identifying and formulating solutions to programming problems. This paper discusses several aspects of debugging in general, and SAS program-fixing in particular. We will see that with the right attitude and the right tools, debugging can be turned into a relatively painless process.

While reading this paper, remember several things. First, *all* programmers have syntactical and logical "lapses" during the course of program development. Second, while debugging a program, you can learn a good deal about how the language works in addition to fixing a problem. It's worthwhile to remember Henry Kaiser's observation: *problems are opportunities in work clothes.*

The paper is divided into two sections. The first discusses the nature of the bug beast, then identifies some of the "off program," psychological aspects to the debugging process. The second section provides a quick review of the debugging tools available in SAS. Throughout, the paper implicitly emphasizes the importance of good coding style. Indeed, if the code is written well there will be much less opportunity for using debugging techniques. Write clearly, avoid tricks, consider all possible logic paths, and you will see most of your programming time being dedicated to development and not to debugging.

Part I: It's About Art and Craft

Knowing how to debug a program is more than simply knowing how to use debugging statements. Effective problem solving begins with understanding that debugging is part instinct and technique, part coding. This section describes the some of art, instinct, and behavior that drive proper use of debugging tools.

Know What Debugging Is and Is Not

The study of bug entomology is knowing what debugging is and knowing what it is not.

- **Know What Debugging *Is*.** A good definition of debugging is taken from Steve McConnell's *Code Complete* (p. 623):

  > [Debugging is] the process of identifying the root cause of an error and correcting it.

  The description is powerful: it points out the importance of identifying the underlying ("root") cause of a program error. It describes debugging as an ongoing set of steps (a "process"), which ultimately leads to a tested solution. The definition stresses the need for a thoughtful, methodical approach to the description, identification, and correct elimination of the programming error. As we will see later in this paper, the process is art and logic in varying proportions.

- **Know What Debugging Is *Not*.** Debugging is not testing. Debugging deals with specific programming issues. Testing is part of program validation. Debugging asks: "How can we fix this problem?" Testing asks: "Is the program performing as expected? It is doing the correct thing, and doing it correctly?"

  The distinction is more than academic – the activities take place in different phases of the program life cycle, they often involve different human resources with different skill sets, and they require different tools for effective and efficient performance.

Success Hinges on Behavior and Attitude

Knowing the debugging tools and how to use them is all fine and well. Remember, however, that behavior and attitude are *at least* as important as tools when fixing a program. Let's assume that you regard the problem-solving process as an opportunity to correct not only the problem, but also to come away from the experience a better programmer. Here are some procedural and behavioral items to consider (some are SAS-related, others are not):

- **Become Familiar with the Range of Error Sources.** Where are bugs born? Anywhere in the program development life cycle, and in any entity used in the system. This needn't be depressing. Part of the psychology of debugging is to relax your assumptions about the source of the error. A thorough investigation of the error's origin may ultimately assign the root cause to one or more of the following sources. Some are discussed in greater detail later in the paper.
  - **Syntax.** The problem may be a simple, misspelled keyword, an omitted semi-colon, or anything else that leaves SAS unable to understand your intent.
  - **Logic.** Correct syntax can perform a task incorrectly. The gap between the actual and intended functionality of the program may be due to misspecification of requirements, incorrect interpretation of requirements by the programmer, or a simple mistake in typing. SAS may or may not highlight the inaccuracy, since the failure of specification may or may not result in program failure. This uncertainty emphasizes the importance of careful examination of the Log and other program outputs.

- **Data.** A happily executing program on day "x" will fail miserably on day "x+1" if its input data sources change. A source can disappear, data types can change, or the specificity of the rows may be altered. Alternately, the data may be correctly updated, but the new values may create conditions that the program was not designed to handle.
- **System Architecture.** Even simple end-user requirements result in the division of the task into logically separated modules. Traditional design metrics require that a program/module has specific input and output values. The programs may be coded and tested individually, but may fail when integrated into the system. Failure may be ascribed to the program itself, the integration process, or even the system specification documents.
- **Entity Linkage.** Most programs require links to external entities such as data files, data sources, format and macro libraries, etc. A program will fail if the linkage is not made correctly. For example, a LIBNAME may point to a test location rather than production, an AUTOCALL path may identify the macro library search path incorrectly, or format libraries may be concatenated in the wrong order.
- **Execution Environment.** Even with tested, time-honored code, subtle or catastrophic errors can occur if anything in the system environment changes. The program that was written for SAS Version 6.12, running in Windows 95, accessing SQL Server Version 7 tables *may* fail if changes are introduced anywhere in the system configuration.

With any of the above, always assume the error is, to some degree, your fault. Failing that, consider the error sources of last resort:

- **SAS Software.** A well-conceived program will come to naught if the tool itself is flawed. Errors do sometimes occur in SAS software. Fortunately, they are rare, and work-arounds and/or program patches are almost always available.
- **System Hardware and Software.** Just as SAS can have an occasional flaw, so can the operating system, its associated utilities, and its hardware. A network connection, for example, may fail for reasons beyond the control of the program.

- **Understand Your Programming Behavior.** The breadth and depth of your experience is, ultimately, your strongest ally during debugging. Your capacity for self-examination is equally important. Look at the problem at hand and ask yourself: "Is this an error I've seen in the past? If so, how did I fix it?" For example, "I know I've seen subscript errors in the past. I remember accidentally resetting the DO loop index, and the index was also used as the subscript. Maybe that's what's going on here."

  This train of thought can be applied throughout development – each of us has idiosyncratic coding habits, and we can continually draw on our experience with these techniques.
- **Relax Your Assumptions.** The flip side of relying on your experience is that it is, indeed, limited to being only *your* experience. It's important to accept the idea that the bug may have arisen from factors beyond those in your personal history. Draw from your experience, but don't be a slave to routine. Otherwise, the range of solution options is artificially constrained.
- **Describe the Problem Thoroughly.** Before attempting to solve the problem, be sure the program is ready for debugging. This requires a clear description of the errant program behavior and being able to replicate the error. This can be problematic in GUI (SAS/AF and Web-based environments) since many screen objects can interact with each other. Detailed, exact descriptions of the program's status before the failure and end-user actions causing the problem will make the fix far easier.

  If the problem can, in fact, be replicated, see if the errant behavior follows a pattern. If parameters from a Web page are submitted for background processing, for example, and the program fails, ask how this submission differs from the others. You may learn that the user selected two objects from a pick list rather than the usual single selection. Does a subsequent single-selection program run successfully? If so, the problem may lie somewhere in the program's handling of variable lists generated by the Web interface.

  The more stable and well-described the problem, the quicker it will be resolved. It's likely that if you can't stabilize and describe the problem, you are overlooking the factor common to all failures.
- **Develop and Fix Programs Incrementally.** Even simple programs are best developed in small steps. This becomes even more important when you are writing logically complex or long DATA steps or when you are using new PROCs. The reason is simple – if you take a functioning program, make a single change, and see the program fail, the failure can be attributed to the change just made. Make five changes, however, and the identification of the errant code can become problematic. Note that "single change" does not necessarily mean a single statement or parameter. Rather, it is a small, logically related group of statements or options.

  By way of example, assume you need to modify a regression model. The change requires a new analysis variable. After making the required changes and rerunning the program, the regression output makes no sense – the number of observations used in the analysis has dropped drastically and several statistics could not be computed.

Is the problem in the DATA step or in the specification of the regression model? You probably can't tell. The better approach is to attack the problem in pieces. First, modify the DATA step and convince yourself that the new variable has the values you expected. Successfully creating the variable means the regression procedure will have solid, proven data as input. Next, run the regression model with the new variable. If it fails, you now know that it was a model specification issue. Making data and analytic changes at once confounds interpretation of results.

- **Don't Panic – It May Be Simple.** Few SAS programming experiences are more dispiriting than seeing dozens of error messages and warnings in the Log. Bear in mind that many errors can be propagated by a single, simple mistake early in the program. A badly coded macro can spew hundreds of lines of hard-to-read error messages. A missing semi-colon or quotation mark can create similarly distressing results.

  Always debug from the top of the program – correcting the first error often fixes dozens of downstream problems. Consider the following program:

```
/* Written by Joe Novice, April 13, 2001
   Summarize current month's values and
   append to year-to-date
%let curr=jun;

data this_mth;
/* moved to c: in jan '98 */
infile "c:\fin\curr\&curr..dat";
retain month "&curr.";
input acct 1-6 suffix 7-9 $3. tranno 5.
      trantype $2. amt 8.2;
run;

proc print data=this_mth(obs=100);
title "first 100 obs. from &curr.";
id branch;
run;
```

  There is a single, simple mistake here: the failure to close the opening comment (a missing */ in the third line). The header comment is not terminated until the comment after the DATA statement. This means the macro variable CURR will not be defined and the DATA statement will not be executed, since both are contained within the unexpectedly long comment. SAS rightfully gets hysterical, as seen in the Log:

```
1    /* Written by Joe Novice, June 3 1998
2       Summarize current month's values and
3       append to year-to-date
WARNING: Apparent symbolic reference CURR not
         resolved.
4    %let curr=jun;
5
6    data this_mth;
7    /* moved to c: in jan '98 */
8    infile "c:\fin\curr\&curr..dat";
     ------
     180

ERROR 180-322: Statement is not valid or it
      is used out of proper order.

WARNING: Apparent symbolic reference CURR not
         resolved.
9    retain month "&curr.";
     ------
     180

ERROR 180-322: Statement is not valid or it
      is used out of proper order.

10   input acct 1-6 suffix 7-9 $3. tranno 5.
     -----
     180
11         trantype $2. amt 8.2;

ERROR 180-322: Statement is not valid or it
      is used out of proper order.

12   run;
13
14   proc print data=this_mth(obs=100);
ERROR: File WORK.THIS_MTH.DATA does not
       exist.
15   title "first 100 obs. from &curr.";
WARNING: Apparent symbolic reference CURR not
         resolved.
16   id branch;
17   run;

NOTE: The SAS System stopped processing this
      step because of errors.
NOTE: The PROCEDURE PRINT used 0.11 seconds.
```

  All the errors can be quickly resolved by starting at the top of the program, thinking about why a perfectly good INFILE statement would be rejected, and eventually inserting the prodigal "*/". Sometimes things aren't as bad as they appear (especially when a single program change spawns a torrent of error messages).

- **Be Adaptable – Consider Alternatives.** Other times, things really *are* as bad as they seem. If the problem is intractable or can only be solved with unreasonably complex coding, chances are that there is another, better solution. One of SAS's greatest strengths is that it offers you a robust programming environment. If you program yourself into a corner using familiar tools, chances are that you can save the day by learning new tools and techniques. The set-oriented IML or SQL procedures, for example, can be useful when traditional DATA step coding becomes too ponderous. Likewise, data-driven formats can duplicate some of the functionality of resource-hungry DATA steps.

  Alternatives aren't confined to the software. If the coding for a task becomes convoluted and error-prone, a possible solution may be a redesign of the data. Transposition, normalization, consolidation into a single table, and other approaches may greatly simplify the program and, in turn, reduce error flow to a trickle.

- **Be Literal – Abandon Assumptions.** A guaranteed way to increase debugging effort is to make assumptions about the program's behavior. Even though a parameter passed to the program could *never* be null, it's likely that at some point the program creating and passing the parameter will, indeed, cough up a null value. Be open to all possibilities, and never catch yourself looking at a problem and saying "that can't

happen." Not being open-minded confines the problem-solving effort to known territory, even though the solution may be found elsewhere.

One way to defeat behavior based on assumptions is to be literal about the program's function, to "play computer." Examine the program as if you were the SAS compiler – read the statements in the order in which they were executed and describe the function of each statement *as stated in the statement's syntax*. This requires you to make the sometimes difficult distinction between your knowledge of the program's context and your knowledge of SAS syntax. The payoff comes when being literal about a statement like

```
input range_low 6. range_high 6
      spec_low 5. spec_high 5.;
```

An assumption-*bound*, non-literal reading of this statement is "read two variables six bytes long, then two more that are five bytes long." An careful, assumption-*free*, literal reading begins with "read RANGE_LOW in the first six columns, then read RANGE_HIGH in column six." This reveals the likely source of the problem – without a trailing '.' SAS read a single digit, from column six, rather than a value six-columns wide. Remember – SAS is *always* literal, and cannot make context-sensitive judgments (and remind yourself that this is a good thing).

- **Be Suspicious – Accuse Everyone.** Part of freeing yourself from assumptions is recognizing that *any* part of a program or system of programs can be the source of the error. Ask questions such as: Were other parts of the program recently changed? If so, how thoroughly were the changes tested? How does the new code differ from the old? Would this have any impact on the current problem? Is the program a "problem child," with a history of bugs? If so, it may be worthwhile to redesign and rewrite. Such lines of questioning can lead toward seemingly unrelated parts of the program.
- **Learn and Re-learn.** Although not strictly a debugging practice, this point warrants inclusion, especially if you are looking for alternate solutions. The SAS System is constantly evolving. It acquires features, many requested by users, which invariably simplify the life of the programmer. Never assume you know everything about a PROC or DATA step statement. A review of the documentation may reveal an important new feature. It may also uncover an "old" feature that you overlooked when you were first learning the language. For example, if you initially used PROC REPORT for its multi-column capabilities, you may never discover its ability to summarize within groups, create temporary variables, and create datasets. You won't learn more about any features unless you seek them out – the "Resources" section, below, identifies numerous on-line, print, and human resources.
- **Take a Break If Possible.** If you are stuck, and you have the luxury of time, use it to your advantage. Leave the problem program and work on something else, even for a short while. It's remarkable how often even experienced programmers get stuck in one path of attack on the problem. Coming back to the program with a fresh attitude clears the debris from the old path and often identifies alternate solutions. Failing that, get a colleague's fresh pair of eyes to examine the problem – it's striking how many times someone will immediately locate the missing semicolon that eluded you for hours.

  Don't rush the process. Believe, as Gerald Weinberg says, that you should "never debug standing up." If you don't have the luxury of a break, at least proceed a bit more deliberately. The write-run-review cycle in most program development environments encourages rapid resubmission of programs. Very often, this simply means that a panicky programmer at loose ends tries one rapid-fire solution after another without really examining the underlying problem. Slow down, spend some time away from the keyboard, and *carefully* examine the Log and any other outputs.
- **Reflect on the Solution.** Once debugging is complete, reflect on what the correctly coded program looks like. Also consider how you got there. How was the problem identified (syntax or logic)? How did the problem-solving process proceed? Did you rely on experience, intuition, documentation, or other resources? Which were most effective? What types of false leads did you encounter? Would different logic, design, or program presentation make similar problems less frequent in the future?

  Make notes about the process by using comments in the program or keeping a written log containing both the problem description and its solution. Even if you think you'll never forget the road to the solution, you probably will. Documentation will save you a good deal of head scratching later on.
- **Ask "Did I Fix the Right Thing?"** Successful debugging means addressing the bug's source, not just its current manifestation. A data source, for example, may have unexpected values. The solution may be to insert an IF statement whose condition sets the bad values to missing. This is a stopgap measure at best. It does not address the reason why the values occurred and whether they are legitimate. It fairly guarantees that another bug will be detected later, identifying yet more unexpected values. Take the time to understand and deal with the reason the data were unanticipated, and not simply code around the problem.

Consider That It Might be SAS's Fault

SAS has some characteristics that directly affect both your reaction to a failed program and how you fix the program. Consider the following:

- **It can overreact.** As we saw in the "Don't Panic" section, above, errors can cascade, one error spawning many. Work from the top down. Correct the first er-

ror and resubmit the program. That is often all that's needed.

- **It can get flaky.** Sometimes, especially during a long-running session, the system can begin to behave erratically. Performance will degrade, widgets will become invisible, and other, transient problems will develop even though you are making small and legitimate changes to your program. In these situations it is usually best to save your work and exit. Restarting SAS or the computer often works wonders. In most cases, the only strategy you have to work with is restarting.
- **It might have a bug.** SAS's quality assurance procedures are among the most thorough in the industry, and it's a testament to their effectiveness that a system with millions of lines of code is so bug-free. However, bugs do occur. Some of these are known and documented. Other, undocumented bugs are also uncovered on an ongoing basis. What this means to the programmer is that there will be times that syntactically correct code will simply fail – the session can lock up, results are counterintuitive, etc. It's true that "a poor craftsman blames his tools," but keep in mind that if you have exhausted all reasonable solutions, the fault may, in fact, lie with the tool.
- **It *usually* tells you what's wrong.** Sometimes it will even fix the problem for you. But don't rely on its corrections – if you entered NOCOLUMN in a PROC FREQ TABLES statement, SAS will say it assumed you meant NOCOL and continue executing. Fix the spelling! Warnings and notes about shared assumptions are distracting, and you can't be sure that the correction will be made in future releases of the SAS System.

  In some cases when SAS cannot make assumptions about your intent, it will display a list of possibilities in the Log. Coding

  ```
  proc freq data=prod.demog nopr;
  ```

  creates an error. SAS sees NOPR as an invalid option, rather than an ambiguous reference to the NOPRINT option. It lists all keywords valid in the PROC FREQ statements, then attempts to execute the next program step.
- **It does not identify logic errors.** If you tell SAS to do the wrong thing in a syntactically valid way, it is up to you to detect the error and make the changes. For example, if you read a variable with a $5. format instead of a $6. SAS will *not* bring the inadvertent truncation to your attention. Likewise, specifying

  ```
  avg = mean(of v1-v5);
  ```

  and

  ```
  avg = mean(v1-v5);
  ```

  are syntactically correct, but produce very different results. The first correctly averages variables V1 through V5, while the second takes the average of the difference of variables V1 and V5. It's your responsibility to track such errors down.

Part II: It's Also About Tools

There's no shortage of programming statements and options to aid the debugging process. In this next section, we'll review system options, the SAS Log, macro variables, macros, DATA step statements, the DATA step debugger, procedure-related options, and dictionary tables and views. Given the breadth of the coverage, it is necessarily cursory, written only to give a feel for the breadth of error handling and debugging activities within your control.

Use Options to Define the Debugging Environment

System options can create a diagnostic paper trail, control macro output, restrict the size of the data sets being used in the program, and prevent their inadvertent replacement. Here are some points to consider when using system options in a debugging context:

- Remember when you can invoke the option. Some options can be set in a configuration file, during startup, in an autoexec file, or any time during the SAS session. Among these are DATE, CENTER, and OBS. Others, such as MACRO and ECHOAUTO, can only be specified early in the program execution process – in the configuration file or during SAS invocation.
- System-level options. These options control the amount of diagnostic output you receive, and the source statements that are displayed in the Log. They also tell the SAS Supervisor how to react to abnormal conditions – that is, whether a particular event (such as an unknown format) should be considered an error.

Category	Options
print source in Log	ECHOAUTO, SOURCE, SOURCE2
messages in Log	MSGLEVEL, NOTES, DETAILS, PRINTMSGLIST
invalid references	BYERR, DATASTMTCHK, DKRICOND, DKROCOND, DSNFERR, ERRORABEND, ERRORCHECK, FMTERR, VNFERR
reaction to errors	CLEANUP, ERRORABEND, ERRORS, INVALIDDATA
data set creation	MERGENOBY, REPEMPTY, REPLACE, WORKTERM
using observations	FIRSTOBS, OBS, WHERE
option settings	OPLIST

- Macro-specific options. These options allow extended searches for macro references and control the amount

of diagnostic information related to macro variables and macro references.

Category	Options
macro invocation	CMDMAC, IMPLMAC, MACRO, MRECALL, MSTORED
autocall facility	MAUTOSOURCE
execution tracing	MFILE (RESERVEDB1 in Version 6), MLOGIC, MPRINT, SYMBOLGEN
error handling	MERROR, SERROR

Ignore the SAS Log at Your Peril!

The SAS Log performs many useful activities. Don't assume that your program ran correctly if it produced "reasonable" looking output. Examine the Log for errors, warnings and notes. The Log has many useful features. It:

- **Echoes your code.** Be sure all the code you *think* should be executed actually *was* submitted for execution.
- **Notes character-numeric conversions.** If you must do this, at least be sure it's being done in the places you anticipated.
- **Identifies creation of missing values.** The Log contains the location (line, column number) of each calculation that resulted in a missing value. Some of these may be acceptable, but watch for an excessive number of locations or a location that generates as many missing values as there are observations in the dataset. The latter is a warning that there may be something systematically wrong with the calculation.
- **Identifies uninitialized variables.** This may be due to any number of problems: a misspelled variable name; incomplete removal of a group of statements (thus "orphaning" a variable reference); omission of a semicolon, thus identifying a SAS keyword as a variable name (see "SAS *cannot* identify logic errors," above).
- **Describes the size of output SAS datasets.** The datasets should meet your a *priori* expectations about the number of observations and number of variables.
- **Describes external data sources.** As with output SAS datasets, above, you should have some idea of how large the data are. The Log lists the number of records read from a data source, the line length, and other characteristics useful in debugging.
- **Contains text/diagnostics.** PUT and %PUT statements, by default, write to the Log.
- **Contains macro diagnostics.** Macro source, generated code, and output from options such as MPRINT and SYMBOLGEN are written to the Log.

Debug with Macros and Macro Variables

Code written with the macro language is often not pretty to look at and almost always difficult to debug. Issues that do not arise in the rest of the Base SAS language become salient – and problematic – in macros. Matters of variable scope, value quoting, and the distinction between when code is generated versus when it is executed make even seasoned programmers tear at their hair. Many of the stylistic traits associated with good Base SAS programming can also be applied to good advantage in macros – documentation, indentation, logical closure, and the like contribute to readability and, in turn, a lower error rate. Here are some good practices to employ during the almost inevitable macro debugging process.

- **Display Values via Automatic Macro Variables.** The %PUT statement supports an automatic variable, _ALL_. This dumps all current macro variable names and values to the Log. The scope of the displayed variables may be limited by coding _GLOBAL_, _LOCAL_, or _AUTOMATIC_. For example, %PUT _GLOBAL_; displays only globally available user-defined macro variables.

  Another way to display macro variables is to use the PRINT or REPORT procedures, or any other viewing tool, to display the view SASHELP.VMACRO. This view is automatically generated and maintained and contains the name and setting for all available macro variables.

- **Use Macro Variables to Control Debugging Output.** Macro variables can control the amount of diagnostic output. They can be used for substitution in RUN statements (the CANCEL option) and can toggle execution of PUT and other statements in DATA steps. This technique allows diagnostic code to remain in the production version of the program without having an appreciable impact on execution speed. It is, in effect, a way to automate turning debugging statements on and off. Some up-front coding time is required, but is well worth the effort. Refer to the following example.

```
%let rundiag = ; /* blank=run prints,
                    cancel=don't run prints */
%let runput = *; /* *=comment out PUT stmts,
                    blank=execute PUT stmts */

<<< lots of intervening code >>>

proc print data=dataset1;
title "Print of non-critical failures";
run &rundiag.;

data test;
set master.round3;
&runput. put "This is input value of var."
             " DIAG " diag;
calc = diag * (1 - ratio);
&runput. if calc = . then
            put "CALC was missing. "
                diag= +3 ratio=;
```

If RUNDIAG is blank, the RUN statement delimiting the end of the PRINT step resolves to RUN; and the procedure is executed. When RUNDIAG is CANCEL, the RUN statement expands to RUN CANCEL; printing is bypassed. Likewise, setting RUNPUT to its two values will turn diagnostic statements into comments (RUNPUT = *;) or let them execute (RUNPUT =;)

- **Use Macros for More Sophisticated Diagnostics.** As an extension to the macro variable example above, consider using macros when you need a greater amount of control over debugging output. You can keep debugging code in production programs without consuming too many additional resources or cluttering the visual look and feel of the program. Consider the following program:

```
data temp;
Set in.master;
<lots of intervening code>
run;

proc print data=temp(where=(id < 100));
title "ID's less than 100";
run;

proc freq data=temp;
tables race salgroup / missing;
title "From TEMP";
run;
```

If you want to turn the PRINT and FREQ procedures off when we are not debugging, you could make the change manually, or rewrite the program as follows:

```
%macro runit(debug=y);
%let debug = %upcase(&debug.);
Data temp;
Set in.master;
<lots of intervening code>
run;

%if &debug. = Y %then %do;
    proc print data=temp(where=(id < 100));
    title "ID's less than 100";
    run;

    proc freq data=temp;
    tables race salgroup / missing;
    title "From TEMP";
    run;
    %end;
%mend;

%runit(debug=y)
```

The debugging output is handled with a simple macro option, rather than manual changes. The macro can be further parameterized to control the ID values that PRINT displays, the tables that FREQ produces, etc.

Know the Range of DATA Step Debugging Tools

The DATA step is, arguably, where the most things can go wrong. Fortunately, you have a host of statements and features to help you:

- **PUT statements.** Use PUT statements to display problematic variables, signal that a portion of code is executing, and any other significant DATA step event. The display should be meaningful! When you are trying to fix a problem, especially under deadline pressure, few things are worse than seeing an unlabeled, enigmatic "20000" in the Log. Take some time to annotate the output, and use formats to make values readable. Rather than

```
put salary;
```

enter

```
put 'Salary has changed! New value is '
    salary comma8.2;
```

The debugging task often comes down to understanding the flow of execution through IF-THEN-ELSE and SELECT statements. Multiple PUT statements, sometimes displaying variable values, sometimes not, can shed light on the gap between the intended and the actual execution. This is shown below:

```
put 'SCRDATE, before LENGTH calc.' scrdate
     mmddyy10.;
if . < scrdate < '01jan2000'd then do;
   put 'In 1/1/2000 group';
```

A final note about PUT statements – too much of this type of output can be hard to use, and can make other diagnostic aspects of the SAS Log less effective. Chose diagnostic content carefully, and consider writing it to a separate file, apart from the Log.

- **_INFILE_ and _ALL_ automatic variables.** These display the last record accessed by an INPUT statement and all variables currently in the Program Data Vector (PDV), respectively. Note that in Version 8.0 and later, the _INFILE_ variable may be used in most contexts as you would any other data set variable:

```
if index(_infile_, '09'x) > 0 then do;
   put 'Obs. ' _n_ ' has tabs. Line is: ' /
       _infile_ $hex100.;
```

- **LIST statement.** This writes to the Log the last record accessed by an INPUT statement. It also displays a column ruler. It helps identify errant INPUT statement column specifications.
- **_ERROR_ automatic variable.** This variable toggles a dump of all variables on (1) or off (0). Assign it a value of 0 if you want to suppress the dump associated with such conditions as zero divides, invalid INPUT values, and the like.
- **IN dataset option.** Use it to identify which dataset(s) a particular observation came from during a MERGE, SET, MODIFY, or UPDATE. The value can be used to create a "pattern" variable that shows which data sets contributed to the current observation:

```
data curr_prev;
merge repos.fy2001(in=_2001)
      repos.fy2000(in=_2000);
by dept_id group_id;
```

```
length pattern $2.;
pattern = '..';
if _2000 then substr(pattern, 1, 1) = '0';
if _2001 then substr(pattern, 2, 1) = '1';
format pattern $char2.;
run;
```

It is important not to make the PATTERN creation with an IF-THEN-ELSE structure. Both statements must execute for every observation being processed. The result is that PATTERN has values of '01', ' 1', '0 '. It can be used in the DATA step, tabulated with the FREQ procedure, used as a filter in a VIEWTABLE window, and so on.

- **? and ?? format modifiers.** Use these to suppress the Log messages generated when SAS encounters invalid raw data. Coding input var 4. ?; will suppress the NOTE about invalid data and only print a dump of the offending line. Coding input var 4. ??; will suppress both the NOTE and the dump. If you know a particular variable is problematic and do not want the dumps and NOTEs cluttering the Log, use these options.
- **$HEX and other formats.** Sometimes the way data is displayed determines how effective the PUT statement will be. Years ago, the author needed to parse directory structures on a VAX/VMS system. On the screen, the directory name, file name, and other items appeared to be separated by blanks. The program contained the appropriate "&" and ":" format modifiers, but they appeared to have no effect. A PUT statement of the variable containing the _INFILE_ data produced no clues (this was Version 5.18; Version 8 is a bit more forthcoming). Finally, a $HEX format was used, and the output showed that the file was tab-delimited (hex 09's separated the fields). The tabs were recognized and used on the screen, but the program was not aware of their presence. Adding options to the INFILE statement fixed the problem.

Use the Enhanced Editor

One of the most significant improvements to SAS software in Version 8 has nothing directly to do with DATA step or procedure functionality. The Enhanced Editor provides a context-sensitive editing environment for SAS programs. Possibly its most significant feature is its color-coding of the different pieces of SAS programs. Keywords, comments, constants, formats, macro names, and many other elements of even the most basic programs are readily identified. Writing a program with unclosed comments, as shown earlier in this paper, becomes almost impossible. Other common syntactical miscues (invalid text, unresolved macro references, *et al.*) are also highlighted as you type.

Use the DATA Step Debugger

Versions 6.11 and higher of the SAS System support the DATA step debugger. This is an interactive tool used in the Display Manager Environment. It lets you step through a program line by line, examine or change variable values, and perform other activities that give you insight into not only your immediate debugging problem, but also into how SAS "thinks" while running the DATA step.

The debugger is a programming environment unto itself and is beyond the scope of this paper. Refer to SAS Institute's "SAS Software: Changes and Enhancements, Release 6.11" for a complete description of the debugger's syntax and usage.

Keep in mind that the debugger can also be used in batch mode. This somewhat counterintuitive activity can be accomplished by stacking debugging commands after the DATA step.

Use Procedures to Debug DATA Steps

DATA step debugging does not have to take place entirely within the confines of the DATA step. Several procedures can be used to help you get a sense of what's happening in the data. Here are a few:

- **FREQ** Use this to get n-way distributions of categorical variables or for FORMATted values of interval, ratio-scale variables. The distributions can verify that, say, the categories of a value calculated in a DATA step are, in fact, in the expected range or take on legitimate values.
- **CHART** This serves somewhat the same purpose as FREQ, but with a more visually oriented display.
- **MEANS** Just as FREQ is used with categorical data, MEANS is used to display information about continuous, interval- and ratio-scale data. Minimum and maximum values and other univariate statistics, possibly presented by groups, can identify patterns in the data.
- **PRINT** Use PRINT to display all or part (rows/columns) of the dataset. Use the WHERE statement or dataset option to control what is printed. You could, for example, use WHERE to display observations with a variable value outside a specific range.
- **REPORT** This procedure serves roughly the same purpose as FREQ, but is more tree-friendly, since it can easily "panel" across the page (the PANELS=n option in the PROC statement). Consider using it if you are printing hundreds of observations but only a few, narrowly formatted variables.
- **CONTENTS/DATASETS** These procedures help display variable names, order, data type and other characteristics of the data.

Use Dictionary Tables, Procedurally and Interactively

Dictionary tables and associated views are organized collections of meta data describing the current program. They

are automatically generated by SAS during system startup, and are maintained throughout the session by the SAS Supervisor. They contain much of the information found in the CONTENTS and CATALOG output datasets, as well as non-graphics option settings and macro variables. The tables can be accessed through the SQL procedure. The views can be accessed through any SAS procedure.

To get a feel for their contents, execute SAS interactively, enter DIR SASHELP on the Command line. Sort by "Type", then scroll to the bottom of the list. You should see about 16 views beginning with the letter "V" (VCOLUMN, etc.). Double click on a view's icon to open it for browsing.

As an example of the tables and views used in debugging, consider this scenario. You are merging several datasets, each of which contains a character variable DGRP. A print of the variable shows that it is truncated in some observations. Examine the variable length in the dictionary tables and views by doing one of the following:

```
proc sql;
select memname, name, length
from dictionary.columns
where name = 'DGRP';
quit;

from the command line:
viewtable sashelp.vcolumn
where name = 'DGRP'

proc print data=sashelp.vcolumn;
where name = 'DGRP';
run;
```

The tables will reveal which datasets have the variable in question. The dataset with the inadvertently short DGRP can be readily identified.

As Always, Use Comments!

While not a debugging tool *per se*, comments are tightly woven into the fabric of a good, maintainable program. They explain the program's purpose, describe the typical inputs and outputs, and identify portions of the program that may be unusual (e.g., non-standard methods of handling a calculation). In the context of fixing a program, they can note SAS help desk tracking numbers related to the problem, and describe alternative solutions that were tried and rejected.

One purpose served by comments merits some elaboration. As errors – or enhancements, for that matter – are made to a program, they should be identified by a change code. The code itself, date, author, and a brief description should be placed in a header comment, and a comment containing the code number should be at or near the statements affected by the change *in all programs affected by the change*. This is shown in a highly abbreviated format below:

```
<< header comment >>
/* R003  FCD 2001/01/24  Add file name var. */
data dir_scan;
<< intervening code >>
f_name = scan(line, -1, '\');  /* R003 */
<< more intervening code >>
keep status plant area /* R003 */ f_name;
run;
```

The small amount of time needed for the extra commenting pays off handsomely when you want to review how the change was made. Simply search for the change code (here, "R003") to identify the appropriate statements. The benefits of this technique are more emphatically realized when a change touched several programs. An extended search for the change code will identify *all* statements in *all* programs in the system related to the change. Contrast this to not commenting, or to using different codes in multiple programs.

Locate "Off Program" Resources

The debugging process does not have to take place within the confines of a SAS program. Many other resources are available to, if nothing else, lend moral support and show you that you are not the first person to experience this problem.

- **Gurus.** Unless you are working in a vacuum, it's almost inevitable that you will encounter someone who seems to know everything about SAS and can communicate this knowledge in a clear and non-patronizing manner. It is very likely that the way they achieved this status was by making, and recovering from, every imaginable error. They made beginner's mistakes, intermediate mistakes, and are very likely now at the level of incredibly esoteric mistakes. But the key point is that they learned from their mistakes and were able to plunge ahead into even greater levels of program complexity and scope. They are the rulers of Arcania and should be sought out for their wisdom, *but not as a first resort*.
- **Manuals.** This includes both the SAS Online Doc and hard-copy manuals and books. SAS's publications are the definitive reference for syntax. Some Institute publications and most of their Books By Users series approach programming from more real-world standpoints – light on syntax, perhaps, but rich on practical "how to" in a wide variety of disciplines.
- **Proceedings.** Numerous SAS user group conferences take place regularly at the international, regional, and local levels. Many produce a Proceedings, a collection of presentations at the conference. Some groups produce a CD-ROM in addition to the more traditional bound volumes. Conference Proceedings are a good blend of tutorial and "hands on" information. If you are looking for an alternate explanation of how to define a column percentage in PROC TABULATE or want to see if someone has already solved your immediate problem (possibly using a different approach), this is a good place to look.

  Proceedings are included with the cost of registration

or may later be ordered directly from SAS Institute or a regional users group. See the Institute's Web Site for ordering information.

- **The Web.** SAS Institute's World Wide Web site contains marketing information, office locations, training schedules, downloads for bug fixes and product upgrades, and much more. The Uniform Resource Locator (URL) is `www.sas.com`. Other, non-Institute sites are growing in popularity and user acceptance. Two such sites (as of Winter 2001) are SAS for the Masses (http://faith.hypno.net/sasmass/) and Qualex Consulting Services (http://www.qlx.com). New sites and services are regularly announced on the SAS-L list server.
- **SAS-L.** This is a world-wide list server with more than 2,000 subscribers. It is the definitive on-line resource, and it is free. It is a "peered" list, with administrative duties shared by multiple hosts. To subscribe, send an email to a host. One host to use for subscription is `listserv@listserv.uga.edu`. No subject is needed. The body of the message should be

  ```
  sub sas-l Your Name
  ```

  No quotes are needed around your name. Just be sure to separate the sub, sas-l, and name fields with at least one blank. You will receive a confirmation message containing instructions on using other helpful list server commands. SAS-L sends dozens of messages in a typical weekday. To receive all the messages in one long email, use the SET DIGEST command once you've subscribed. SET is fully described in the confirmation email mentioned earlier.

  To post a question, send email to

  ```
  sas-l@listserv.uga.edu
  ```

  You do not have to be subscribed to SAS-L to post a question, but you do have to be subscribed to receive responses. Make sure the subject field gives a clear idea of the nature of your problem. In the body of the message, state the problem as succinctly as possible, describe solutions you've already tried, and supply any appropriate Log or output.

  The SAS-L archives are available on the Web: www.listserv.uga.edu/archives. The site contains over a decade's worth of your colleagues' discussions, and is easily filtered by topic, sender, and other criteria.
- **The SAS Sample Library.** The SAS System comes with useful sample programs. The samples are for Base SAS procedures (including SQL) and for any other installed products, such as SAS/GRAPH, and SAS/FSP. To access the sample library in a Windows environment, start Display Manager and then click on Help, Sample Library. The process is essentially the same in other releases and platforms.
- **SAS Institute Technical Support.** SAS Institute's Technical Support staff may be reached at (919) 677-8008. Yes, it's a toll call, but the support is free. Be prepared to provide your site number to the support staff, and have on hand as many Logs, output listings, and notes as possible. If your problem cannot be handled by the person initially handling your call you will be issued a "tracking number" and the problem will be referred to senior staff. Most problems requiring tracking numbers are resolved within 24 hours.
- **In-House Technical Support.** Most companies and universities have Help Desks or similarly-named locations that can be called or visited for help. The advantage to using these rather than SAS Institute technical support is that they have site-specific knowledge about network addressing, job submission and retrieval, and other issues not readily apparent to the more product-oriented people at the Institute.
- **You!** In the long run, you are your best resource. Nothing compares to developing a body of knowledge built on application and industry-specific experience. As you build more complex programs, break them, and repair them, the speed and reliability of the repairs will increase. Most important, the *number* of errors will decrease as you write savvier, more bullet-proof code.

Comments? Questions?

Your input is always welcome. Contact the author:

102 Westbury Drive
Chapel Hill NC 27516-9154

fcd1@mindspring.com

This paper is part of a chapter in the author's new book, *The Elements of SAS Programming Style*, to be published by SAS in (hopefully!) 2001.

Further Reading

Delwiche, Laura and Susan Slaughter, *The Little SAS Book: A Primer, Second Edition*, Cary, NC: SAS Institute, Inc., 1998.

Gill, Paul, *The Next Step: Integrating the Software Life Cycle with SAS Programming*, Cary, NC: SAS Institute, Inc., 1997.

Hunt, Andrew and David Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Reading, MA: Addison Wesley Longman, Inc., 2000.

Kernighan, Brian and P. J. Plauger, *The Elements of Programming Style*, New York, NY: McGraw-Hill Book Company, 1974.

Ledgard, Henry F, *Programming Proverbs for FORTRAN Programmers*, Rochelle Park, NJ: Hayden Book Company, 1975.

McConnell, Steve, *Code Complete: A Practical Handbook of Software Construction*, Redmond, WA: Microsoft Press, 1993.