Paper 165-2008

The Art of Debugging

Ian Whitlock, Kennett Square, PA

Abstract

Program bugs are a fact of life, yet I never met a programmer that did not believe his/her program was bug free. Bugs are almost always a surprise because the programmer thought one thing and the program another. You can even have a bug before the first line of code is written.

Do you plan for bugs? Can you spot the symptoms of a bug? Can you locate the bug? Do you recreate it? Do you know the difference between the symptom and the problem? Do you know that even programs that work and produce correct results can have bugs?

If you don't know the answer or answered no to any question, then you probably need a lesson in the gentle art of debugging.

This talk is aimed at the beginning level SAS® programmer, but it takes an expert to really foul up a program, so almost anyone interested in SAS may find something of interest. Although the examples come from using the SAS/BASE software under Windows, the ideas are general enough to be of interest for any platform and any SAS software product.

Introduction

The term "bug" comes from the early days of vacuum tube computers, when a moth gumming up an electrical connection was found to be the cause that stopped a computer from operating. The term was soon used for programs that didn't work and has been with us ever since. I suspect that one of the chief reasons is that it implies an external agent, thus allowing the programmer to not feel responsible. I also think it makes the debugging process more difficult for the same reason - a denial of responsibility. A bug is a programmer's error.

The paper is presented from the point of view of bug classification. Bugs have been classified into:

1. Syntactic mistakes
2. Other compile time errors
3. Execution time errors
4. Logical mistakes
5. Data problems

Although bugs can occur in any step, the discussion here is only about the DATA step since I feel that the general knowledge is easily transferred to any procedure and that a discussion of bugs special to a particular procedure does not belong in a general discussion. The DATA step debugger is not discussed since I do not find it helpful in my work.[1]

Errors can be made on many different levels ranging from a simple syntactic mistake to designing the wrong program.[2] Even a program that is completely correct according to a good specification can be buggy when run on buggy data.

The basic steps to debugging are:

1. Become aware that there is an error.
2. Locate the error.

[1] For an enthusiastic and complete discussion of this tool see Riba, "*How to Use the DATA Step Debugger*". Some of the other SAS material listed in the reference section also discuss this topic in the more general setting of debugging.
[2] From the cost point of view the most important mistakes are those made in the analysis phase of development and before. For a general programming discussion of this topic see Hunt and Thomas, "*The Pragmatic Programmer*".

3. Understand the error by isolation and recreation.
4. Fix the error in the program.
5. Fix why you made the error or limit the consequences of making it again.

Depending somewhat on the classification of the bug, these steps are fairly obvious for the bugs discussed in this paper. You should consider them as practice for the more difficult problems you will face. In particular, for step 5 it is left to you to develop a style that will minimize and catch the kind of mistakes that you favor. When I am asked to debug someone else's code, I often find that I could not have made that kind of mistake because my style would have prevented it. On the other hand a missing semi-colon is a favorite of mine because I also program in line oriented languages that do not tend to use the semi-colon.

Syntactic Bugs

Mistakes in syntax are trivial in the sense that the compiler tells you about the mistake. The compiler underlines the point at which it cannot understand the code and issues some kind of message about it. Probably the most common message is error 180.[3]

```
           data _null_ ;
              mistake
              _______
              180
ERROR 180-322: Statement is not valid or it is used out of proper order.

              put _all_ ;
           run ;

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

The error message cannot say much because the compiler had no idea what to expect, and it had no meaning for what it got, but it does indicate the point at which the mistake occurred, so it is pretty clear what is wrong, and it is left to you to know how to fix it.

Now consider a mistake where SAS has more information.

```
           data _null_ ;
              mistake = 1 200 ;
                          ___
                          22
ERROR 22-322: Syntax error, expecting one of the following: !, !!, &, *,
              **, +, -, /, <, <=, <>, =, >, ><, >=, AND, EQ, GE, GT, LE,
              LT, MAX, MIN, NE, NG, NL, OR, ^=, |, ||, ~=.

              put _all_ ;
           run ;

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

Again we see underlining at the point where the compiler failed to understand the code. This time it saw "mistake = " as the beginning of a SAS statement. Hence it knows there is some kind of assignment taking place. It is expecting some kind of operator to connect the 1 and 200. Note that the compiler's assumption is that everything is correct up to this point. It is entirely possible that we wanted

```
mistake = cats(1, 200) ;
```

[3] Notes about execution time messages have been removed from all logs in this paper. Errors and warnings have been made bold to emphasize the message.

or something else, but there is no way it makes sense to suggest this. What about the suggestion that we might want to use concatenation? Well concatenation is legal, although it would cause a conversion message when the concatenation is attempted. It should be clear that in fact all of the above operators are in fact possible.

Not all syntax mistakes are mistakes. Consider:

```
1          data _null_ ;
2             length mistake $ 4 ;
3             mistake = "aaaa ;
4             x = 3 ;
5             y = 4 ;
6             z = 5" ;
7             put (_all_)(=/) ;
8          run ;


mistake=aaaa
```

Here we do not find any error messages because from the point of view of the compiler there are none. We assigned a long character string to MISTAKE which was truncated by the LENGTH statement. The only indication that there is something wrong is that the other variables are not mentioned in the log report.

However, from the point of view of the programmer there are two syntactical mistakes. On line 3, the constant literal string was started with a double quote[4], but not ended. It is not a true syntax error because line 6 provides the closing double quote. The situation here is fairly obvious because we said it was a mistake and the program is tiny. Do you have any confidence that you would notice the mistake in the following log?

```
12         data fixed ;
13            set problem ;
14            if id = "1234" then
15            do ;
16               zip = "12345 ;
17               age = 36 ;
18               status = married" ;
19            end ;
20         run ;

NOTE: There were 795 observations read from the data set WORK.PROBLEM.
NOTE: The data set WORK.FIXED has 795 observations and 4 variables.
```

You could do a before and after print of records that were fixed and then you would see the problem. Or you might get lucky and not make a second mistake leading to syntactically correct code. Then the log might be confusing.

```
22         data fixed ;
23            set problem ;
24            if id = "1234" then
25            do ;
26               zip =
26       ! "
           _
           49
26       ! 12345 ;
```

[4] Although SAS accepts single or double quotes, you should standardize on double quotes. Most of the time it doesn't matter, but when you start programming with macro, double quotes will be correct more often when it does matter. Moreover, you become sensitized to seeing single quotes as unusual. Hence many bugs are eliminated, others become more visible, and as a reader you are alerted when single quotes are required.

```
NOTE 49-169: The meaning of an identifier after a quoted string may
             change in a future SAS release.  Inserting white space
             between a quoted string and the succeeding identifier is
             recommended.

27                age = 36
28                status = "married" ;
                                       _______________________
                                       388    200
                                       76
ERROR 388-185: Expecting an arithmetic operator.

ERROR 200-322: The symbol is not recognized and will be ignored.

ERROR 76-322: Syntax error, statement will be ignored.

29             end ;
30          run ;

NOTE: The SAS System stopped processing this step because of errors.
NOTE: SAS set option OBS=0 and will continue to check statements.
      This may cause NOTE: No observations in data set.
WARNING: The data set WORK.FIXED may be incomplete.  When this step was stopped
         there were 0 observations and 4 variables.
WARNING: Data set WORK.FIXED was not replaced because this step was stopped.
```

Here the first note looks pretty mysterious and irrelevant. It doesn't even claim there is an error. But it should draw your attention to the fact that there is no closing quote after the 5. The real error messages begin where there is nothing wrong, so they are completely misleading. Their import is to say, the compiler went off the track here, there must be a mistake earlier. To be fair, the compiler thinks you are assigning ZIP and have just finished a quoted expression. It could not handle the bare word "married" in this context and hence complained.[5] There doesn't have to be any indication of where the real problem is, but in this case, it was very fortunate that it brought the eye to the problem for erroneous reasons. Every message from the compiler is a hint about how it sees your code. Hence it can be revealing even when it is not relevant to your error.

Any time something slightly unexplainable happens, you should spend time trying to understand the situation because this might be the only hint that a mistake has happened. You should view compiler messages as a kindness, not a right, and certainly not a right to a clear way to correct the code.

Most syntax mistakes are easy because they stop the program and therefore are very apparent. Often the indication of the first error is very near the point where the mistake is made, and a lot of the time the error reported is accurate and leads you to a solution when you understand the message. It is well worth spending time trying to understand how the error message might be relevant. On the other hand, the thing you do wrong need not be the point at which the compiler issues a message because your error may mean something. The compiler can only report at the point it no longer understands the code. The reason it no longer understands may be quite irrelevant if your mistake misled the compiler to this point.

Compensating mistakes can be difficult because the compiler has no way to know where the closing symbol belongs. Single or double quotes can be particularly difficult because the quote hides semicolons. Hence the problem can appear statements or even steps before any message. Using an editor that color codes quoted material is the best solution here. Leaving out the END statement of a long DO-block can be a problem. This is one reason why good programmers religiously insist on a sound system of indentation. Fortunately, such blocks cannot cross the step boundary. Hence in SAS, it takes a really complex big DATA step to be much of a problem.

[5] The first NOTE appears before the problem to indicate the beginning of the quote that ends with an "m" immediately following the closing quote mark. The remaining ERROR messages are caused by the unknown word, married.

If multiple errors are reported you should concentrate on the first one. After understanding the first, it is worth looking at the later ones since you may be able to find and fix some simple errors, but it is not worth working hard on them since they may automatically go away or become clearer when the first mistake is fixed. Conversely one error can hide another that only becomes apparent when the first is fixed.[6] Since making one mistake is a good indication that you might have made another, it is worth looking at the code near your first mistake for a hidden one.

I have used syntax mistakes to highlight two key points about debugging:

1. Awareness of an error.
2. Location of the error.

Syntax problems are usually helpful with both these problems. By definition a syntax mistake has a location, and the compiler makes you aware of the bug by some sort of message. However, the compiler can only see your code, not your intent. In general you as the programmer have to learn how to code so that you can make yourself aware that a problem exists, and how to code so that you can locate the error. Even with syntax mistakes, the compiler can help only some of the time.

Other Compile Time Errors

Syntax mistakes, if caught, are always caught at compile time. Sometimes a mistake can be caught at compile time, but there is nothing wrong with the syntax of the code. It is a mistake because something is missing. Consider the following problem very close to a syntax mistake, but not a syntax mistake from the point of view of the compiler.

```
           data out
              input x dollar4. y  ;
           cards ;

ERROR: Libname DOLLAR4 is not assigned.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: SAS set option OBS=0 and will continue to check statements.
      This may cause NOTE: No observations in data set.
WARNING: The data set WORK.OUT may be incomplete.  When this step was
         stopped there were 0 observations and 0 variables.
WARNING: The data set WORK.INPUT may be incomplete.  When this step was
         stopped there were 0 observations and 0 variables.
WARNING: The data set WORK.X may be incomplete.  When this step was
         stopped there were 0 observations and 0 variables.
```

In this case there is no underlining and no syntax error indicated. In fact, there is no syntax error. The message is clear, but it seems mysterious. Your first thought should be - why did it think DOLLAR4 was a LIBREF? If you have an answer then you know what is wrong. If not then it is wise to look at the WARNING's. Why does it say that the data set WORK.INPUT and WORK.X may be incomplete? It seems as if the INPUT statement has been misunderstood as a list of data sets. That should suggest that the mistake occurred earlier. Look at the DATA statement. It is missing the semi-colon. Now we can see that INPUT, X and DOLLAR4.Y[7] are part of the DATA statement.

A missing semi-colon can be notorious for misleading error messages. The compiler depends on a sequence of key words to identify the type of each statement. If you leave out a semi-colon then you hide the key word of the next statement. The compiler is likely to find something wrong, but it is usually not the real the mistake - the missing semicolon. Hence the errors and warnings are just hints about what the compiler is seeing instead of the underlying problem.

[6] Did you catch the missing-semicolon on line 27 in the last log in the previous section? The problem was hidden by the quoting problem.

[7] You probably are used to seeing no spaces around the period in a data set reference. However the period is really a separator between the LIBREF and the member reference. Hence, spaces are allowed. You should see errors as an opportunity to learn new things about SAS.

Just as the format caused an execution time error, a true LIBREF can cause a problem when there is no supporting LIBNAME statement.

Consider the difference between the log

```
1          data q.w ;
2             x = 1 ;
3          run ;

ERROR: Libname Q is not assigned.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: SAS set option OBS=0 and will continue to check statements.
      This may cause NOTE: No observations in data set.
```

and the log

```
5          data w ;
6             set q.w ;
ERROR: Libname Q is not assigned.
7          run ;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.W may be incomplete.  When this step was
         stopped there were 0 observations and 0 variables.
```

In the first case, there is no error caught before the RUN statement because the message occurs after the RUN statement.[8] In the second case, it is caught during compile time at the point of reference.[9] In the second case the compiler must look at the header dictionary of the data set to determine any new variables that must enter the PROGRAM DATA VECTOR (PDV) so the compiler learns of the missing LIBREF.

Another common problem is the "uninitialized" NOTE that usually indicates an error.[10]

```
1          data _null_ ;
2             x = y ;
3             put x=  ;
4          run ;

NOTE: Variable y is uninitialized.
x=.
```

Here Y was never assigned a value and the compiler issues the NOTE to warn the programmer. Why did the compiler have to wait until the step boundary to issue the message? A RETAIN statement could appear below the first occurrence of Y that assigned a value at compile time. Another possibility is that Y is the name of a system variable that will be supplied at compile time. For example,

```
           set w nobs = y ;
```

[8] The LIBNAME can occur before the DATA step or anywhere in the DATA step. On the other hand, it is better style to place the LIBNAME statement before the step. In general, global statements (TITLE, FILENAME, OPTIONS, etc.) can occur anywhere in a step, but it is better style to place them before the step because they are not really part of the step. Often they belong at the top or near the top of the program because they announce conditions for the program.
[9] Roger Staum (see references) classifies this kind of error as a resource error.
[10] This might be considered a DATA error in the sense that Y has an unexpected value, missing. However, the cause here is an outright coding error as opposed to having a value that was not planned for. The classification is not always clear due to overlapping problems and the fact that different programmers can see the errors in different ways.

Consequently the compiler cannot know until the step boundary that some future line will not provide the initialization required. Sometimes the uninitialized message is wrong because the initialization is carried out by some means the compiler cannot be aware of. If you determine this to be the case then the use of CALL MISSING routine or the RETAIN statement can be used to avoid the message. You should always be able to identify where the initialization takes place when the message is issued, and it is a good idea to eliminate the message for the sake of a clean log, even if it requires unnecessary code to do so.

Similarly, SAS converts from character to numeric or vice versa whenever needed, but usually issues a message. Consider the following log.

```
1          data _null_ ;
2             length x y z $ 4   ;
3             n = 763 ;
4             x = n ;
5             y = substr(n, 1, 2) ;
6             intended_y = substr(put(n, best12. -l), 1, 2)   ;

7             put x= y= z= ;
8          run ;

NOTE: Numeric values have been converted to character values at the places
given by:
      (Line):(Column).
      4:8    5:15
x=763 y=  intended_y=76
```

Why is Y blank? The SUBSTR function requires the first argument to be a character expression. Hence SAS automatically converts to a character string. This is done using the BEST12. format. Hence a temporary string 12 bytes long is created and the digits 763 are placed in the last 3 bytes because of right justification. The only indication that line 5 does not work as expected is the NOTE about automatic conversion, since SAS does not issue messages about truncation.[11] However, in some cases SAS is more careful about the conversion.[12] Note that line 4 was also automatically converted and squeezed into 4 bytes, but it worked! However the leading character in X is a blank and likely to cause trouble in a more serious program. Consequently, the NOTE might indicate a problem and it might not. The only safe practice is either to never allow the message in the program log, or to categorize all places where the note may be safely ignored. Most good programmers take the former approach, and I do not know of any place where one can find the latter approach taken seriously. Note that line 6 shows a reasonable way to make the conversion of N explicit, thus it produces the intended value of Y.[13]

From the programmer's viewpoint the only real difference between the mistakes in this section and those in the section on syntax mistakes is that the messages occur at the bottom of the step rather that where the mistake was made and that the mistake is often labeled as a NOTE or WARNING rather than an ERROR. Often the mistakes in this section are either classified as syntax mistakes or execution time errors. I prefer to call them compile time mistakes to point out who is reporting the error. It is the compiler, not the SAS supervisor component that executes the step. Often you can glean information about what went wrong by knowing what part of the system is reporting the error.

Some programming languages always require explicit conversion between character and numeric. It is worth asking yourself why they would do this. For some it is the desire to eliminate possible confusion or mistakes, and for some it is necessary because "+" is used to indicate both addition and concatenation.

---

[11] This is probably a good decision because many SAS programmers take advantage of this when only the initial part of a string is wanted.
[12] In version 9+ some functions, e.g. CALL SYMPUTX and the CAT's family of functions, automatically convert numeric values to character without message and without doing it incorrectly.
[13] See the documentation on the PUT and INPUT functions for learning how to correctly convert numeric to character and character to numeric respectively. The "-l" requests left justification for the digits of N.

Execution Time Errors

An execution time error is one where the compiler can construct a program, but then something goes wrong during the execution of the program. Three of the most common reasons that do not stop execution are illustrated in the log below. They are division by 0, arithmetic expressions involving missing values, and functions asked to evaluate outside their domain.

```
1          data _null_ ;
2             x = 4 ;
3             y = 5 - x ;
4             do i = 1 to 2 ;
5               z = 7 / (y - 1) ;     /* division by 0    */
6               ms = z + y ;          /* missing value    */
7               lg = log ( x - 8 ) ; /* negative number */
8               put z= lg= ;
9             end ;
10         run ;


NOTE: Division by zero detected at line 5 column 12.
NOTE: Invalid argument to function LOG at line 7 column 11.
z=. lg=.
NOTE: Division by zero detected at line 5 column 12.
NOTE: Invalid argument to function LOG at line 7 column 11.
z=. lg=.
x=4 y=1 i=3 z=. ms=. lg=. _ERROR_=1 _N_=1
NOTE: Missing values were generated as a result of performing an
      operation on missing values.
      Each place is given by: (Number of times) at (Line):(Column).
      2 at 6:13
NOTE: Mathematical operations could not be performed at the following
      places. The results of the operations have been set to missing
      values.
      Each place is given by: (Number of times) at (Line):(Column).
      2 at 5:12   2 at 7:11
```

First of all note that there are no messages labeled "ERROR:" and next that these problems did not stop execution of the program. In fact the complete step was executed as shown by the results of the two PUT statements. Next observe that the notes about division and invalid argument occur when (i.e. each time) the problem is encountered during execution.[14] These notes cannot occur after the problem line because the complete log of the step code was written before execution began. Fortunately, these notes contain a reference to the problem line so they are still easy to locate and usually to fix.[15] Finally observe that the last two notes occur once at the end of the step as a summary of the two types of problems shown.

The system dump

```
x=4 y=1 i=3 z=. ms=. lg=. _ERROR_=1 _N_=1
```

is produced at the bottom of the DATA step iteration because the system set the automatic variable _ERROR_ to 1. You can do the same by explicitly setting _ERROR_ to 1 for your own conditions, or using the ERROR statement to set the value to 1. The advantage of the ERROR statement is that it looks less mysterious and you can include an explanatory message about why you want the dump. To prevent the dump you can set _ERROR_ to 0.

[14] The system option ERRORS= controls how many times these errors will be reported. By default it is set to 20.
[15] Unfortunately, when the code is generated by a macro, the line number refers to the line where the macro is called. Hence these simple problems can become more difficult to track down when using macros. Often the simplest approach is to save the generated code (see system option MFILE) and then execute it to get explicit line numbers.

Unfortunately you usually want to know the values of variables at the time of the error rather than at the bottom of the DATA step iteration. In this case, you can create your own dump with

```
put _all_ ;
```

at the appropriate place in the code. In such situations, you can run the code once to find out where you want the PUT statement and then again after inserting the PUT statements.

The previous example dealt with some missing value problems arising purely out of code. Consequently it is placed in this section on execution time errors rather than under data errors. The next example is more questionable since it deals more with the code-data interaction and might be more appropriate to the DATA section. I chose to place it here because there are SAS messages to indicate the problems. The example uses are three lines of data.

```
???  1
2
3 4
```

Here is the log.

```
1          data w ;
2             input x y ;
3          cards ;

NOTE: Invalid data for x in line 4 1-3.
RULE:      ----+----1----+----2----+----3----+----4----+----5----+-- ...
4          ???  1
x=. y=1 _ERROR_=1 _N_=1
NOTE: SAS went to a new line when INPUT statement reached past the end of
      a line.
NOTE: The data set WORK.W has 2 observations and 2 variables.
```

There are two problems here. The first, indicated by the first NOTE, is relatively simple. SAS could not read "???" as a number, so it did the best it could and made the value of X missing. It is really up to you whether it is important to fix the data or treat "???" as a signal for a properly missing value. In the latter case it would probably be better to read X with an INFORMAT that explicitly allowed "???" to mean missing value in order to avoid mention of what is otherwise an error.

The second problem is the more serious because there is no indication of where things went wrong, but at least there is a message to indicate the possibility of an error. We expected three lines of data in W, but there are only two. What happened? The second line is short, i.e. missing the value of Y, so SAS picked up a value for Y on the next line. The message is only a NOTE because SAS recognizes that you may want this activity. However, in this case it was a real mistake, we lost a pair of values and placed an incorrect value of Y in the second observation. Either the line should include a dot to indicate that Y is missing, or the desired value for Y.

The second error could have been avoided with an INFILE option.

```
infile cards truncover ;
```

Now SAS would assign Y the value missing in the second observation and not try to find the value on the next line, but then there would be nothing in the log to alert you to what happened. I tend to prefer this solution because it avoids the more serious problem of definitely incorrect data, and because I tend to pay careful attention to what is giving fields the missing value so the error will be caught in the checking process. Note that in either case the step and the program are not stopped. If you prefer that activity then use the INFILE option STOPOVER.

To see an execution time error serious enough to stop processing, consider the following log.

```
7          data _null_ ;
8             set w ;
9             by group ;
10         run ;

ERROR: BY variables are not properly sorted on data set WORK.W.
group=2 FIRST.group=1 LAST.group=1 _ERROR_=1 _N_=1
```

Here W has not been sorted properly by the variable GROUP. The use of the BY statement asks SAS to check that in fact the data are in the order specified. Suppose you want to do group processing where each group of observations occurs together, but the groups are not in order. In this case, add NOTSORTED to the list of BY variables. NOTSORTED tells SAS to use BY processing but stop checking. Sometimes it is handy. The most common case is when processing states that are in state abbreviation order then they will be grouped by state name but not necessarily in state name order. Of course, when used, the programmer must take responsibility for knowing that all of the observations in a group are together.

In the early days of SAS "NOTSORTED" could not be the name of a variable. Recent versions allow 32 characters for variable names. Hence "NOTSORTED" is a legal name, but it is an accident/error waiting to happen when put in a BY statement.

Another example occurs when an array index is out of range.

```
1          data _null_ ;
2             array a (3) ( 1 2 3 ) ;
3             a[4] = 99 ;
4          run ;

ERROR: Array subscript out of range at line 3 column 4.
a1=1 a2=2 a3=3 _ERROR_=1 _N_=1
NOTE: The SAS System stopped processing this step because of errors.
```

In this case the index value 4 is clearly out of range. Had an index variable, say X, been used with the value 4, then you could see from the dump what the bad value was. However, remember that the dump is made at the end of the iteration of the DATA step and therefore may no longer indicate the incorrect value.

Error control

We saw a number of examples where SAS provided error messages to help you understand problems with the code. There are several system options to help you control how serious the error is and whether you want to see a message.[16]

```
1          proc options group = errorhandling ;
2          run ;

    SAS (r) Proprietary Software Release 9.1  TS1M2

 BYERR              Set the error flag if a null data set is input to the
                    SORT procedure
 CLEANUP            Attempt recovery from out-of-resources condition
 NODMSSYNCHK        Do not enable syntax check, in windowing mode, for a
                    submitted statement block
```

[16] I have emphasized the SAS Institute's grouping of options and how to limit the options to a group. DiIorio, "*The SAS(r) Debugging Primer*", has a very nice section using his own classifications which are more oriented toward debugging.

```
DSNFERR             Generate error when SAS data set not found condition
                    occurs
NOERRORABEND        Do not abend on error conditions
NOERRORBYABEND      Do not abend on By-group error condition
ERRORCHECK=NORMAL   Level of special error processing to be performed
ERRORS=20           Maximum number of observations for which complete
                    error messages are printed
FMTERR              Treat missing format or informat as an error
QUOTELENMAX         Enable warning for quoted string length max
VNFERR              Treat variable not found on _NULL_ SAS data set as an
                    error
```

The group, LOGCONTROL, also produces some system options that help tailor the log to your requirements. From the debugging point of view the most important is MSGLEVEL=I. When set, it provides more information about possible errors that were not considered as errors in the early development of SAS.

A missing semi-colon on the DATA statement can be very serious because it can wipe out a permanent data set. For example:

```
data w
   set project.main ;
...
```

In this case, SET and PROJECT.MAIN are treated as output data sets. Since the code may easily pass the compile phase, PROJECT.MAIN can be corrupted. Two options are offered in the SASFILES group to handle this problem. The more general and traditional is to use the option NOREPLACE. In this case every output file with a LIBREF that is not WORK must be new. You can be more specific with the LIBNAME statement option ACCESS=READONLY. In this case, only that particular library using that LIBREF is protected. If a second LIBREF is used to refer to the same directory then the data set can still be replaced using the alternate LIBREF, so one has to be careful how this technique is used.

The newer option DATASTMTCHK in the SASFILES group treats the problem more directly. You can set it to abort the step whenever a SAS key word (i.e., the first word starting SAS statements) appears in the DATA statement, or when just the most common problem key words (MERGE, RETAIN, SET, UPDATE) appear in the DATA statement.

Another important example from the option group SASFILES is MERGENOBY. Traditionally, if you use a MERGE statement without a BY statement, it is not an error since SAS will do a one-to-one merge in that case. On the other hand, it is such a common error that many SAS programmers want to know about it, so the option can be set to WARN, ERROR, or the default NOWARN.

As you can see from the above examples, system options to help control errors and make you more aware of problems in the code can be found in many different option groups.

Logical Errors

Logical errors occur when you misunderstand something or fail to take account of some condition. A classic example is the "one off" mistake. Suppose you have to process information for 38 data sets. You are ready to begin processing data set number 23. How many data sets do you have yet to process? If your answer is 38 - 23 then you have fallen into the "one off" problem. The easiest way to see this is look at the 24th data set. Now 24 - 23 is 1, but two sets are to be processed the 23rd and 24th, so you need to add one to the difference. However, you cannot always add one. Suppose the problem had been, you just finished the 23rd data set. Now the difference, 15, would be correct.

Most stair case accidents occur on the first or last step. Why? It is because these steps are special - one has no preceding step, and the other has no next step. In programming mistakes are often made on the first or last of something for the same reason. Let's start with some play data.

```
ID  VALUE
1       1
2       1
2       2
2       3
```

The variable ID represents the group identifier. We want to read this data set and output one observation for each group. The plan is to read the values into an array V and output whenever the value of ID changes. We use the RETAIN statement to save the ID and values. (Assume we have reason to believe there will be no more than 5 values for any group.) First we read the identifier as TEST and then compare it with the previous value of ID. When it is new we know that a new group has started, so we output the data saved from the old group, reset ID, and start a count at 1. Then we read the current value and increment the counter.

Here is the code.

```
options nocenter ;
filename mydata "H:\Documents\_SAS\SASTalk\SGF08\FF\Testing\Log1.dat" ;
data w (keep = id v1-v5) ;
   retain id v1 - v5 ;
   array v (*) v1 - v5 ;
   infile mydata ;
   input test @ ;
   if id ^= test then
   do ;
      /* output last record */
      output ;
      id = test ;
      cnt = 1 ;
   end ;
   input v[cnt] ;
   cnt + 1 ;
run ;
```

Do you see any mistakes? None are reported in the log and we get two observations as expected. But a print then reveals the problem.

```
Obs    id    v1    v2    v3    v4    v5

 1      .     .     .     .     .     .
 2      1     1     .     .     .     .
```

Oh, the first step! On the first observation ID is missing so TEST and ID are not equal, but there is no previous group because we are on the first step. That is what makes the first step special - nothing came before. We want to output every time except the first one. So the line for output becomes

```
if _n_ > 1 then output ;
```

Now there is a hint of something wrong in the log. It is the observation count for the output. There is only one output observation, but we expected two. Oh, the last step! Our idea was to output each time a new group is encountered, but the last group has no next group. That is what makes the last step special. We need a special output at the end of file. So we change the INFILE statement to include END=EOF, and add the following line to the bottom of the step.

```
if eof then output ;
```

Finally the log is clean and the listing shows:

```
Obs    id    v1    v2    v3    v4    v5

 1      1     1     .     .     .     .
 2      2     1     2     3     .     .
```

Are we done? No, we have taken care of the first and last step problems but there is still a logical problem. To see it, add a third group with only one record, say with VALUE equal to 0. Now the print shows:

```
Obs    id    v1    v2    v3    v4    v5

 1      1     1     .     .     .     .
 2      2     1     2     3     .     .
 3      3     0     2     3     .     .
```

Why are there three values instead of one? We needed to retain the values in the array V because we used the standard processing of one record in per iteration of the DATA step, but by doing so we lost the standard SAS processing of reinitializing the values of V. Hence we need code to clean up after OUTPUT in the block where we detect each new group. There is also the question of what happens if the data set happens to have more than 5 values in some group. Fortunately the system will detect such an error, but you ought to feel that it is really the programmer's responsibility to find such problems rather than a system responsibility.

I chose the code to illustrate thinking about logical bugs. However, the number of problems encountered suggests that maybe there is a simpler approach. What if we allow two DATA steps? This might lead to a problem with longer processing, if the amount of data is very big, because the data is read twice - once as external data and once as SAS data. A data VIEW provides a solution. The first step just creates a view, i.e. no data are actually read. The code is saved as a special engine for producing the data when required. The second step does the reading, but allows us to treat the view PREP as a SAS data set.

```
options nocenter ;
filename mydata "H:\Documents\_SAS\SASTalk\SGF08\FF\Testing\Log1c.dat" ;
data prep (keep = id value) / view = prep ;
   infile mydata ;
   input id value ;
run ;

data w ( keep = id v1-v5 ) ;
   array v (*) v1 - v5 ;
   do i = 1 by 1 until (last.id) ;
      set prep ;
      by id ;
      if i > 5 then abort ;
      v[i] = value ;
   end ;
run ;

proc print data = w ;
run ;
```

The second step is unusual in that 1) it places the SET statement in an explicit loop, 2) it uses a test for LAST.ID defined by the BY statement in the loop, and 3) it does not have a TO value ending the loop.[17] Note that the first two

[17] This style of loop has been promoted by Paul Dorfman because it solves the incrementing problem, saves a test of the index, and relieves the programmer from naming an upper bound.

facts allow SAS to clean up the values of the array V in the normal fashion and to handle the incrementing of the loop index automatically.

What if we wanted to use a WHILE loop instead of UNTIL? SAS initializes LAST.ID to 1 at the beginning of execution because there is a BY statement defining the LAST dot variable.[18] Consequently there is another trap waiting here for you if want to change the type of loop. While I strongly prefer the WHILE loop, there are times when it is much better to use an UNTIL loop.

Another lesson to learn from debugging is that sometimes many bugs can indicate that there may be a better way to handle the problem. You may have to make the choice between CPU efficiency with tricky code or coding simplicity to avoid bugs. In most cases it wise to choose simplicity over efficiency and/or shorter code.

Another problem open to logical difficulties is the problem known as the "find the best" of something. Suppose we have STATE, STORE_ID, SALES_AMOUNT. The problem is to find the three stores in each state with the highest sales. Here is the code.

```
proc sort data = lib.sales out = sales ;
   by state descending sales_amount ;
run ;

data w ;
   set sales ;
   by state descending sales_amount ;
   if first.state then cnt = 0 ;
   cnt + 1 ;
   if cnt <= 3 then output ;
run ;
```

When the code is run and the output investigated everything looks good - there are three records for each state with the sales listed in descending order. So what's wrong?

What about ties? If there are five stores in Texas all with the same top sales amount why should two of them go unlisted? One possible fix is to restrict when the count is incremented.

```
   if first.state then cnt = 1 ;
   else
   if sales_amount < lag(sales_amount) then cnt + 1 ;
```

This seems to work even when we test where there are more than three winners in a state. In fact the code might work, but the code still has a big classic bug. The LAG function is executed conditionally, i.e. it is not executed for the first observation in each state. Consequently, the sales amount for the second observation is compared with the last observation of the previous state.[19]

In both of the above examples, there are real logical problems with the code. While it is true that some data may be processed correctly, not all reasonable data can be. The mistake is in the code, not the data.

Sometimes the logical mistake is made in the analysis stage of development rather than in the code. You can argue that the second example is the fault of the programmer rather than the analyst because the analyst did not make clear what "best three" should mean. On the other hand, I would expect a good programmer to raise the issue.

Deeper mistakes in the analysis of a problem can be much more costly than the ones we have illustrated here, but we leave those to the more sophisticated analysts.

[18] If there is a FIRST dot or LAST dot variable without a BY statement, then the compiler initializes the variable to zero. Otherwise it initializes the value to one.
[19] Note that even this statement contains a bug. If the previous state contains only one store, then the second observation is compared with the last observation of the previous state with more than one store.

Data Problems

So the code works on your data. Is it bug free? Not necessarily! Most programs are an interaction between code and data.[20] Perhaps your code is just waiting for the data to show it. One common example comes from a chain of IF - ELSE IF conditions.

```
data w ;
   set q ;
   if x <= 10 then x_edit = "OK" ;
   else
   if x <= 20 then x_edit = "PROB" ;
run ;
```

With one glaring exception[21], the code looks all right and can work, but what happens when a bad negative value shows up in X? It falls in the OK category. Then suppose that a new value of 21 is added. In that case X_EDIT is blank. If there is a bad value in the variable Y, you can argue that it is not the responsibility of this step. But it definitely is the responsibility of this step to know about X and to protect the integrity of the code. In some sense this chain of IF - ELSE IF's is another illustration of the "first step/last step" problem. You could also argue that it belongs in the logical class of mistakes, but it is the data that make that logic wrong.

Here is another type of example where the code is all right (in it is fact quite normal and traditional) for merging some data sets, but it is wrong for the data it runs on. It is less clear than in the previous case, whether it is the responsibility of the code to catch this problem or that of the programmer. Ultimately it is the programmer's responsibility to know that the merge is correct. This knowledge can come from an external investigation or internally from code in the program.[22] In any case it something that should be checked for, and is a big problem when first encountered. Consider the following log in which two data sets are sorted and then merged by the variable X.

```
1          libname dat "H:\Documents\_SAS\SASTalk\SGF08\FF\Testing" ;
NOTE: Libref DAT was successfully assigned as follows:
      Engine:        V9
      Physical Name: H:\Documents\_SAS\SASTalk\SGF08\FF\Testing
2          options nocenter ;


3          proc sort data = dat.w1 out = w1 ;
4             by x state ;
5          run ;

NOTE: There were 3 observations read from the data set DAT.W1.
NOTE: The data set WORK.W1 has 3 observations and 2 variables.

6
7          proc sort data = dat.w2 out = w2 nodupkey ;
8             by x lkup ;
9          run ;

NOTE: There were 4 observations read from the data set DAT.W2.
NOTE: 0 observations with duplicate key values were deleted.
NOTE: The data set WORK.W2 has 4 observations and 2 variables.

10
```

[20] All programs when you consider literal values in the code as data.
[21] The length of X_EDIT is determined by the first occurrence. Hence it has length 2 and there is no room for "PROB". If this did not jump out at you, you need to develop your code reading intuition.
[22] If the merge appears in a macro that will be run with many different programs creating the data, then the check should probably be in the code. In terms of simple stable SAS programs the check would almost always be external, but you must still be ready to handle the situation when it arises.

```
11          data q ;
12             merge w1 ( in = w1 ) w2 ( in = w2 ) ;
13             by x ;
14             file = w1 + 2*w2 ;
15          run ;

WARNING: Multiple lengths were specified for the BY variable x by input
         data sets. This may cause unexpected results.
NOTE: There were 3 observations read from the data set WORK.W1.
NOTE: There were 4 observations read from the data set WORK.W2.
NOTE: The data set WORK.Q has 4 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.02 seconds

16
17          proc print data = q ;
18          run ;

NOTE: There were 4 observations read from the data set WORK.Q.
NOTE: The PROCEDURE PRINT printed page 1.
```

Note that the second sort indicates that there were no duplicate values of X, but the MERGE step has the warning that both files had duplicate values of X. How is that possible? Here is the listing from data set Q.

```
Obs    x     state    lkup    file

 1     aa     AK        1       3
 2     ab     AL        2       3
 3     ab     AL        3       3
 4     bb     AR        4       3
```

You can review this code for hours without finding a mistake if you ignore the relatively new **WARNING** after line 15. In fact there is nothing wrong with the code! As a second step we add a print to see those observation in W1 that have X = "ab".

```
19
20          proc print data = w2 ;
21             where x = "ab" ;
22          run ;

NOTE: There were 1 observations read from the data set WORK.W2.
      WHERE x='ab';
NOTE: The PROCEDURE PRINT printed page 2.
```

Impossible, there is only one observation in W1 with X = "ab". Again, there is nothing wrong with the code! Programs are an interaction of code and data to produce output. If the result is wrong and there is nothing wrong with the code, then there must be something wrong with the data. Here is the complete data for W1.

```
Obs     x     lkup

 1     aa       1
 2     ab       2
 3     aba      3
 4     bb       4
```

What happened? Clearly all the values of X are distinct, as reported in the first sort. What about the third observation? It is not in the report. Why not? In the report the value of LKUP is 3 for the second value, "ab", of X. This indicates that the third observation went into the merge, but the value of X was "ab". Why?

A PROC CONTENTS of the output data set Q shows that the length of X is 2, The length of X in W2 is clearly longer. Now go back to the MERGE statement.

```
12              merge w1 ( in = w1 ) w2 ( in = w2 ) ;
```

The length of X is chosen by the length in W1, the first data set listed in the MERGE statement. Apparently the length of X in W1 is 2 and it is longer in W2.

In this case it was fairly apparent that something was wrong because of the WARNING and it was fairly easy to track down because of the small size of the data sets. Had this problem observation been one out of thousands and had the problem record had a distinct value not in the other file then the bug could easily go unnoticed for a long time if you do not carefully inspect the log and explain every message from the compiler. The only other hint of a problem might be that the number of output records was one too large.

The previous example used a problem with merging by a character variable. Here is another example using numeric variables that show the common numeric variable X has the same length. The two data sets have just one record and X is shown as 0.3 in both. Since we now know that length can be a problem the length of X was captured using the VLENGTH function when the data sets were made and those lengths printed in the log.

```
1           libname lib "H:\Documents\_SAS\SASTalk\SGF08\FF\Testing" ;
NOTE: Libref LIB was successfully assigned as follows:
      Engine:        V9
      Physical Name: H:\Documents\_SAS\SASTalk\SGF08\FF\Testing
2
3           data w ;
4              merge lib.f1 lib.f2 ;
5              by x ;
6              put _n_= x= lenx_f1= lenx_f2= ;
7           run ;

_N_=1 x=0.3 lenx_f1=8 lenx_f2=.
_N_=2 x=0.3 lenx_f1=. lenx_f2=8
NOTE: There were 1 observations read from the data set LIB.F1.
NOTE: There were 1 observations read from the data set LIB.F2.
NOTE: The data set WORK.W has 2 observations and 3 variables.
```

Note that LENX_F1 is 8 on the first observation since it comes from data set LIB.F1 and is missing on the second since it comes from LIB.F2. So why don't the values of X match?

A more careful listing of the values in the merge reveals the answer.

```
8
9           data _null_ ;
10             merge lib.f1 lib.f2 ;
11             by x ;
12             put _n_= x= best17. x= hex16. ;
13          run ;

_N_=1 x=0.29999999999999 x=3FD3333333333300
_N_=2 x=0.3 x=3FD3333333333333
NOTE: There were 1 observations read from the data set LIB.F1.
NOTE: There were 1 observations read from the data set LIB.F2.
```

Here, even the width 16 is not enough for the BEST format to show the difference in decimal between the two values of X. Now the width 17 is required. The format HEX16. is very special and important for any sound understanding of real floating point numbers in SAS. This format shows each byte as it is stored in 8 bytes. In this case we see that the two numbers differ in the last byte - 00 versus 33.

If we drop the idea that we are debugging for a minute, we can make the point stronger with the following log.

```
data _null_  ;
   x = 0.1 + 0.1 + 0.1 ;
   y = 0.3 ;
   put x= best32. y= best32. x= hex16. y= hex16. ;
   different = not ( x = y ) ;
   put different= ;
run ;

x=0.3 y=0.3 x=3FD3333333333334 y=3FD3333333333333
different=1
```

Here there is no merge, just one DATA step and not question of length. Still there is a difference in the way the computer sees 0.3 and 0.1 + 0.1 + 0.1. This SAS program was run under Window XP on an Intel processor, but the problem is really inherent in the way computers work. It is not a defect of SAS, the operating system, or the CPU.[23]

Both of the above examples have concentrated on merge problems. But in each case the cause of the mistake need not appear in a merge problem. In the case of character variables, if you assign a short character variable with the value of a long one, truncation will occur and there will be no warning message. Yet the mistake can be fully as serious as the merge example presented. In the case of the numeric comparison of variables, the last DATA step shows that even an increment of the last bit by 1 can change two values from testing equal to being different. It is surprising how well SAS does with 8-byte floating point numbers, but that means the computer problems with handling real numbers are all the more difficult for their unexpected nature. You simply cannot trust exact numeric comparisons without detailed knowledge of what is being compared.

Protection and Prevention

Most of this paper has concentrated on classifying and looking at various types of mistakes. This is important for the beginning programmer because he must get past this stage in order to learn to program. However, it is only the beginning. As you learn to write more sophisticated programs, you become more aware that some of your mistakes (bugs) are inherent in the process of programming. Consequently you must learn how to avoid them and failing that, become aware of them, locate them, recreate them, and fix them.

Modular development is probably the single most important tool for preventing mistakes, catching them, and locating them. SAS as a language is very helpful in that it is step oriented. Each step has explicit inputs and should produce specific outputs.   By concentrating on the step you already have an approximate location to look for the error. Moreover, there is less to go wrong in any one step than in the whole program. Consequently one can add code to show that the input and output requirements were actually met. Sometimes this means pre or post processing steps such as PROC FREQ and PROC PRINT or more complex statistical analysis. Sometimes it means extra code within a step to alert that certain requirements are not met. Consequently a good program should be able to inspect and prove itself.[24]

Modular development often leads to reusable units of code. Once you begin to think this way, you can afford more time for checking because the code will be used in many places. Hence the cost is spread over more projects. On the other hand, the cost of an unrecognized error goes up, again because the code is used in multiple places.

[23] See, TS-DOC: TS-230 - Dealing with Numeric Representation Error in SAS listed in the reference section for more information on handling real numbers in SAS.
[24] Prove in the sense of test. Rarely can one really have a formal proof of program correctness in anything but the simplest cases. However, one can gain confidence with the elimination of potential problems.

A detailed knowledge of the data, how SAS works, and the project problem are the next best defense against bugs. As one modest programmer put it, "I am so good because I have made every mistake and studied it carefully". The lesson is half right, a great deal can be learned from your mistakes, but the lesson that this program is correct, is wrong. Never forget to study the log carefully.

It is important to recognize that it is your mistake and that you are responsible. I find that I spend hours trying to prove that it is the system that is wrong, or that the code was misused. After these possibilities have been eliminated I can usually quickly zero in on the problem with a confidence that it is my code that is wrong.

It is important to be able to simplify. Often a simple DATA step with CARDS data can skip over lots of irrelevant code in preparing for the crucial step. Then eliminate features one by one in the problem step that you think are not relevant until you come down to the kernel of the problem. If you still do not understand and know how to fix the problem, then let it sit a while doing something else. If you have narrowed the problem down, and let it sit, but still cannot recognize what is wrong then show it to a friendly colleague. Often the problem is that your mind automatically changes what is there to what should be there so you do not see. Each time you have conquered a mistake, learn why you made it and consider how to avoid it. Does your style need some change that would help you prevent repeating the problem? Is it in the nature of the data? Perhaps a restructuring of the data would avoid such errors. Perhaps the names of the variables are meaningless or maliciously alike. Is it something about SAS that you did not understand? Ask your self, why not, and what are the related things that you should learn about SAS.

Conclusion

Although very important, in preparing this paper I explicitly ruled out any mention of SAS macro programming as inappropriate to what I wanted to cover. Some of the papers listed in the reference section mention macro problems, and one deals exclusively with this point of view.

A traditional classification of bugs into syntactic mistakes, execution time errors, logical mistakes, and data problems has been presented with simple self contained examples that hint at the danger presented to programs. However, you must read with much imagination and apply that imagination to your programming in order to write programs with minimal errors. The problems you will face usually occur in a much bigger and more complex context than those shown here. You must learn to simplify and extract the kernel of the bug. These examples really show you the end step in the process and present a number of topics which typically cause problems.

Each bug class is important for different reasons. Syntactic and execution time mistakes are important for learning the importance of the SAS log and how to read it. Logical mistakes form the deepest of bug problems, but perhaps the hardest to discuss. Data mistakes are important because one often spends hours trying to debug a correct program only to find that the problem lies with unexpected data values. Never forget that data is half of the system that you probably didn't look at.

One of my more valuable lessons came in my second course in programming. We were assigned a problem each month with a description of what good data looked like. Then a week before the code was due, we were given the data. Four out of five records were there only to test whether we had accounted for things our program couldn't handle.

Two of the books listed in the references are about general programming rather than SAS. Remember that bugs are a general programming problem and that there is much to learn about SAS from authors who may never have used SAS. These authors know about programming languages and bugs.

Contact information

Ian Whitlock

29 Lonsdale Lane
Kennett Square, PA 19348
Ian.Whitlock@comcast.net

References

Burlew, M., *Debugging SAS Programs: A Handbook of Tools and Techniques*, SAS Publishing.

DiIorio, Frank., *The SAS(r) Debugging Primer*, http://www2.sas.com/proceedings/sugi26/p054-26.pdf.

Hunt, A. and Thomas, D., *The Pragmatic Programmer*, Addison-Wesley 2006.

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

Riba, D., *How to Use the DATA Step Debugger*, http://www2.sas.com/proceedings/sugi25/25/btu/25p052.pdf.

Slaughter, S. and Delwiche, L., *Errors, Warnings, and Notes (Oh My): A Practical Guide to ...*, http://www2.sas.com/proceedings/sugi22/BEGTUTOR/PAPER68.PDF.

Staum, R., *To Err is Human; to Debug, Devine*, http://www2.sas.com/proceedings/sugi27/p064-27.pdf.

TS-DOC: TS-230 - Dealing with Numeric Representation Error in SAS, Applications, http://support.sas.com/techsup/technote/ts230.html.

TS-DOC: TS-654 - Numeric Precision 101, http://support.sas.com/techsup/technote/ts654.pdf.

Whitlock, I., Macro Bugs - How to Create, Avoid and Destroy Them, http://www2.sas.com/proceedings/sugi30/252-30.pdf.