To Err is Human; to Debug, Divine

Roger Staum, SAS® Institute, New York, NY

ABSTRACT

This tutorial describes four types of programming errors and how you can recognize them. It also discusses four approaches to handling errors: deter, detect, deflect, and debug.

INTRODUCTION

In this paper, I focus on errors which you may encounter, and tools you can use, in

- version 8.2 of base SAS® software
- all operating environments supported by that software[1]
- both interactive and non-interactive execution modes.

Because it is a beginning tutorial, this paper does not address errors which can occur

- in the SAS macro language. The only use of macro language is as a diagnostic tool.
- in the process of integrating your program into a system. All examples are assumed to be stand-alone programs.

The primary data set used in the examples is SASUSER.CLASS, which contains data about 19 boys and girls. There are two character variables (Name and Sex) and three numeric variables (Age, Height, and Weight). The data set is sorted by Name within Sex.

In order to illustrate merging issues, a second data set, SASUSER.CLASSCOLORS, is used. This data set contains two character variables (Name and FavoriteColor), 9 observations, and is sorted by Name.

CLASS.DAT and CLASS2RECORDS.DAT are external files which can be read to create SASUSER.CLASS. CLASS.DAT has one record, and CLASS2RECORDS.DAT has two records, for every observation of SASUSER.CLASS.

Excerpts from programs, logs, and outputs appear throughout this paper. I have edited them in order to save space, and emphasized some lines in bold font.

WHAT IS AN ERROR?

People use the term "error" in different ways, often resulting in confusion and miscommunication. In order to avoid this, I will state my definition up front, and use it consistently:

> ***An error in a computer program refers to a condition leading to an unexpected and usually undesirable result.***

This is, admittedly, a very general definition. It implies that an error may occur without being signalled by an error message in the SAS log. In fact, information about an error may be contained in warnings or notes, or may not even be hinted at in the log.

TYPES OF ERRORS

People also categorize SAS programming errors in different ways. The distinctions among these categories can be difficult to understand, especially for a beginning programmer. Therefore the categories which I have chosen are based primarily on what you can observe in the SAS log.

Recognizing the type of error which has occurred in your program is the first step in the process of debugging.

SYNTAX/RESOURCE ERRORS

A syntax/resource error is always identified by an ERROR message in the log, and the step or global statement in which it occurs is not executed. This means that you are likely to see additional messages in the log, indicating that a DATA step did not read or write data, a PROC step did not produce a report, or a global statement did not do its work.

Syntax errors are caused by violating the rules of the SAS language. Such violations include misspelled keywords, omitted semicolons, mismatched quotes, invalid options, etc.

The following is an example of a syntax error in a PROC step. Notice how the SAS system® does not process any data or produce the desired report.

```
8    PROC PRINT DATA sasuser.class;
                     -------------
                     73
ERROR 73-322: Expecting an =.
9    RUN;
NOTE: The SAS System stopped processing this
step because of errors.
```

Resource errors occur when the SAS system is unable to find a resource, such as a data set or external file, which the program needs.

The following is an example of a resource error caused by requesting a non-existent data set. Notice once again that the SAS system does not process data or produce a report.

```
10   PROC PRINT DATA=sasuser.children;
ERROR: File SASUSER.CHILDREN.DATA does not
exist.
11   RUN;
NOTE: The SAS System stopped processing this
step because of errors.
```

Sometimes a syntax error is accompanied by misleading resource error messages. Notice how the missing semicolon in the following example makes the SAS system believe that the program is trying to read non-existent data sets.

```
1    DATA a;
2        SET sasuser.class
3        IF Age > 14;
                -
                22
                  --
                  200
ERROR: File WORK.IF.DATA does not exist.
ERROR: File WORK.AGE.DATA does not exist.
ERROR 22-322: Syntax error, expecting one of the
following: a name, a quoted string, (, ;, END,
KEY, KEYS, NOBS, OPEN, POINT, _DATA_, _LAST_,
_NULL_.
ERROR 200-322: The symbol is not recognized and
will be ignored.
4    RUN;
NOTE: The SAS System stopped processing this
step because of errors.
WARNING: The data set WORK.A may be incomplete.
When this step was stopped there were 0
observations and 5 variables.
```

In the case of a slight misspelling of a keyword, the SAS system may guess what keyword you intended, display a warning instead of an error message, and execute the step or global statement.

Here is an example of a misspelled keyword leading to a warning message and step execution:

```
17   DATA a;
18       ST sasuser.class;
         --
         1
WARNING 1-322: Assuming the symbol SET was
misspelled as ST.
19       IF Age > 14;
20   RUN;
```

```
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
NOTE: The data set WORK.A has 5 observations and
5 variables.
```

The SAS system may not always guess correctly what you intended. Even when it does, you should *always* correct the error. Your goal should be an error-free program, a clean log, and correct output.

RUN-TIME ERRORS

A run-time error is also flagged with an error message in the log, but the step partially executes.[2] A DATA step may read or write some of the data, or a PROC step may produce part of a report, and then stop executing.

A run-time error is usually caused by a serious contradiction within your program, or between your program and the data. The SAS system is unable to detect this problem until it starts executing the step, and cannot easily recover.

Here are two examples of run-time errors. The first is an attempt to do BY-group processing when the data set is not properly sorted. Notice that PROC PRINT reads the data until it finds an observation out of sort order. It also produces a partial report.

```
7     PROC PRINT DATA=sasuser.class;
8         BY Name;
9     RUN;
ERROR: Data set SASUSER.CLASS is not sorted in
ascending sequence. The current by-group has
Name = Tammy and the next by-group has Name =
Alfred.
NOTE: The SAS System stopped processing this
step because of errors.
NOTE: There were 10 observations read from the
data set SASUSER.CLASS.
```

The second run-time error occurs when an executing DATA step attempts to access a non-existent 16th array element.

```
15    DATA a;
16        SET sasuser.class;
17        ARRAY X {15};
18        X{Age} = 1;
19    RUN;
ERROR: Array subscript out of range at line 18
column 5.
name=Philip sex=M age=16 height=72 weight=150
X1=. X2=. X3=. X4=. X5=. X6=. X7=. X8=. X9=.
X10=.
X11=. X12=. X13=. X14=. X15=. _ERROR_=1 _N_=16
NOTE: The SAS System stopped processing this
step because of errors.
NOTE: There were 16 observations read from the
data set SASUSER.CLASS.
WARNING: The data set WORK.A may be incomplete.
When this step was stopped there were 15
observations and 20 variables.
```

DATA ERRORS

Data errors occur only in DATA steps. They are identified by notes, rather than error messages, which tell you that missing values have been generated because an expression could not be evaluated or an external data field could not be read. In addition, the log displays a snapshot of the Program Data Vector (PDV) at the time the error was encountered. This snapshot shows that the automatic variable _ERROR_ is set to 1.[3] If an external data file is being read, the log will also show a snapshot of the input buffer, with a column ruler.

Data errors result from contradictions between program specifications and the data being processed. In these cases, the SAS system recovers by generating missing values. Execution continues.

Data errors can be either sporadic or systematic. If the errors are caused by data entry errors, for example, there will probably be only a few such messages, and therefore a few missing values as a result. If the errors are caused by incorrect specifications or informats, however, virtually identical data error messages may occur for every record or observation processed.

You can control the maximum number of such messages printed by using the ERRORS= system option. The default value is 20. This option does *not* affect the number of missing values generated.

Here is an example of a data error resulting from an expression which could not be evaluated. Apparently, the programmer assumed that every student's weight (in pounds) would exceed his/her height (in inches), and that it would therefore be safe to take the log of the difference. This assumption was not true for one student.

```
32    DATA a;
33        SET sasuser.class;
34        X = LOG(Weight-Height);
35    RUN;
NOTE: Invalid argument to function LOG at line
34 column 9.
name=Sandy sex=F age=11 height=51.3 weight=50.5
X=. _ERROR_=1 _N_=7
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).
1 at 34:9
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
NOTE: The data set WORK.A has 19 observations
and 6 variables.
```

Data errors can also occur when reading external files, if the data fields contradict the input specifications. You might be attempting to read character data into a SAS numeric variable, or to read data with an inappropriate informat.

In the following example, the programmer was trying to create SASUSER.CLASS from an external file, and apparently believed that the field representing the student's sex was coded as a number, rather than a letter. The variable Sex in the new data set now contains missing values.

```
239   OPTIONS ERRORS=3;
240   DATA sasuser.class;
241       INFILE 'class.dat';
242       INPUT @ 1 Name $8.
243             @10 Sex 1.
244             @12 Age 2.
245             @15 Height 4.1
246             @20 Weight 5.1;
247   RUN;
NOTE: Invalid data for Sex in line 1 10-10.
RULE:     ----+----1----+----2----+----3
1         Alice    F 13 56.5  84.0 24
Name=Alice Sex=. Age=13 Height=56.5 Weight=84
_ERROR_=1 _N_=1
NOTE: Invalid data for Sex in line 2 10-10.
2         Becka    F 13 65.3  98.0 24
Name=Becka Sex=. Age=13 Height=65.3 Weight=98
_ERROR_=1 _N_=2
NOTE: Invalid data for Sex in line 3 10-10.
ERROR: Limit set by ERRORS= option reached.
Further errors of this type will not be printed.
3         Gail     F 14 64.3  90.0 24
Name=Gail Sex=. Age=14 Height=64.3 Weight=90
_ERROR_=1 _N_=3
NOTE:  The  data  set  SASUSER.CLASS  has  19
observations and 5 variables.
```

LOGIC ERRORS

Logic errors are not as easily identified as the other types of errors. There are no error or warning messages, or even distinctive notes, in the log. The SAS system does not perceive anything wrong or unusual in the program.

Sometimes log notes may indicate a relationship between the amount of input and output data which you may recognize as incorrect. A logic error is a probable explanation. In other cases, examining the output may be the only way to discover logic errors.

Logic errors are caused by a completely legal use of programming tools which, nonetheless, do not produce the intended output.

Here is an example of a logic error, in which the programmer wants to create a summary data set by Sex. The desired result should have two observations, but since the programmer did not use a conditional OUTPUT or subsetting IF statement, the summary data set has as many observations as the original detail data set.

```
26   DATA avhts (KEEP=Sex AverageHeight);
27       SET sasuser.class;
28       BY Sex;
29       IF FIRST.Sex THEN DO;
30           SumOfHeights = 0;
31           Number = 0;
32       END;
33       SumOfHeights + Height;
34       Number + 1;
35       IF LAST.Sex THEN
36           AverageHeight =
SumOfHeights/Number;
37   RUN;
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
NOTE: The data set WORK.AVHTS has 19
observations and 2 variables.
```

AN APPROACH TO PROGRAMMING: THE 4 D'S

If your attitude towards programming errors is, "I'll wait until they are obvious or someone points them out to me, and then I'll correct them", then you will spend a lot of time explaining your incorrect results, as well as maintaining and correcting your programs. A comprehensive approach to errors in computer programs should include more than just debugging.

I recommend a multi-layer approach to dealing with errors in your SAS programs. It can be summarized as the 4 Ds:

- **Deter** – avoid errors in the first place
- **Detect** – notice and categorize errors
- **Deflect** – protect your data and program logic
- **Debug** – remove the errors

DETER

"An ounce of prevention is worth a pound of cure" is an adage which certainly holds true in programming. It is far less time-consuming and frustrating to avoid errors than to identify and remove them.

PROGRAMMING STYLE

A readable, consistent programming style is your first line of defense against errors. White space and indentation support readability. Ending every step with an appropriate RUN; or QUIT; statement will not only cause messages to appear in the right place in your log, but also make it easier for you to submit individual steps interactively.

Aim for simple, elegant solutions to programming problems. Remember that there is usually more than one solution to a problem in the SAS system. If your code seems complex or clumsy, look for a better way.

A liberal sprinkling of comments within your code will force you to think about and articulate the issues, and thus help you avoid logical errors. Comments can also facilitate subsequent debugging and program maintenance.

DEVELOPMENT STYLE

If you try to write a long, complex program all at once, the debugging process will almost always be more difficult. Instead, develop incrementally - one SAS step at a time, for example, checking your results after each step.

Interactive execution mode goes hand-in-hand with incremental development. When you run interactively, you can submit code fragments and get faster turnaround. Once you have an error-free program, non-interactive execution will use fewer resources and may be more convenient.

DEFENSIVE PROGRAMMING

Just as defensive driving can help avoid traffic accidents, defensive programming can help prevent errors – especially logic errors.

SAS software is very forgiving. For instance, there are many built-in defaults, and automatic type conversion occurs in many situations. These features can save time and typing, but the results may not be what you wanted. When you code more explicitly, you are more likely to think through issues such as variable type and length, and get optimal results.

For example, when you omit the DATA= option on a PROC step, the PROC will read the most recently created data set. But if you later move code within the program, that PROC might read the wrong data set.

As another example, the SAS system does *not* warn you about truncation of character data. The only way that you can tell that this has happened is by looking carefully at your output. There are a number of situations where the SAS system defaults to 8 bytes for character strings, which may not be what you want. An explicit LENGTH statement allows you to assign the correct number of bytes to a character variable, avoiding both truncation and wasted space.

KNOW YOUR DATA

There is no substitute for knowing your data. The better you know your data, the less likely you are to encounter run-time, data, and logic errors.

There are many tools in SAS for familiarizing yourself with your data:

- The FSLIST procedure allows you to browse external files.[4]
- The CONTENTS, DATASETS, and CATALOG procedures allow you to gather metadata about SAS data libraries and their members.[5]
- The interactive windows of the SAS Explorer allow you to gather similar metadata
- The SQL procedure dictionary tables contain similar information, as well as global metadata such as title settings.
- The PRINT procedure and VIEWTABLE windows allow you to look at the data portion of SAS data sets.
- The FREQ and MEANS procedures permit you to get a quick picture of the distribution of variables.

USE THE ENHANCED EDITOR

The enhanced editor is available in Version 8 under Windows. The visual appearance of your code can give you an early warning about syntax errors before you submit the program. For example, quoted strings usually have a unique color and font combination. Omission of a quote is immediately obvious, because all subsequent code looks like one long quoted string.

MERGING

Merging is an inherently complex operation. Here are three issues to think about in order to avoid logic errors:

Non-matches

If the data sets do not contain the same values of the BY variables, then there will be some non-matches. Always be aware of whether non-matches exist in your data, and whether you want to output them. The IN= data set option tells you which data set(s) are contributing data to the PDV at each execution of the MERGE statement, and thus allow you to include or exclude non-matches.

In the following example, we need to merge SASUSER.CLASS with SASUSER.CLASSCOLORS, which contains the results of a class survey on students' favorite colors. Since only the girls responded to the survey, there are many non-matches. If our purpose is to display the survey results without embarrassing the boys, then the program should eliminate non-matches.

```
147  PROC SORT DATA=sasuser.class
148            OUT=class;
149      BY Name;
150  RUN;
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
NOTE: The data set WORK.CLASS has 19
observations and 5 variables.
NOTE: PROCEDURE SORT used:
      real time           0.02 seconds
      cpu time            0.02 seconds

152  DATA responders;
153      MERGE class
154            sasuser.classcolors (IN=cc);
155      BY Name;
156      IF cc = 1;
157  RUN;
NOTE: There were 19 observations read from the
data set WORK.CLASS.
NOTE: There were 9 observations read from the
data set SASUSER.CLASSCOLORS.
```

Multiple matches

If there are multiple observations in 2 or more datasets with the same values of the BY variables, then you have multiple matches. This situation is usually problematic, since there is no a priori "correct" way to combine the data.[6] The DATA step responds by writing the following note to the log:

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

If you consider multiple matches as an error in your program, the usual causes are:

- a variable missing from the BY statement
- data sets which should not be merged in the first place.

Overwriting of commonly named variables

Usually the only variables in common when merging data sets are the BY variables. If there are other variables with the same names, the values in the first data set mentioned on the MERGE statement will be overwritten by those in the second data set.

By default, no special message appears in the log. If you set the system option MSGLEVEL= to I, then you will see a message such as the following:

```
INFO: The variable Y on data set WORK.A will be
overwritten by data set WORK.B.
```

In many cases the problem can be corrected by realizing that the additional variable should have been in the BY statement. In other cases - especially when the variables have a different underlying meaning - one of them should be renamed to avoid overwriting.

DETECT

Despite your best efforts, SAS errors happen. It is important to be aware that an error has occurred.

LOOK AT THE LOG

It cannot be over-emphasized: *always look at the log*. Beginning programmers tend to ignore the log if they get any output at all, or to stare only at the source code if they get no output or the output is clearly incorrect.

There are tools which make it easier to remember to look at the log. In interactive mode:

- Issue the commands AUTOPOP ON; WSAVE from the log window. The first command causes the log to pop up automatically if SAS software writes anything to it. The second command causes the software to store this behavior in your profile.
- Program your function keys to take you to the top of the log, and to search for common error keywords (such as INVALID).

In non-interactive mode:

- At a minimum, refer to the pages listed in the summary of error messages at the end of the log. Remember, however, that SAS error messages do not identify data and logic errors.
- You might also want to consider creating a post-processing routine which reads the log and produces a readable summary of syntax, run-time, and data errors.

ALL THE CODE

If you look at your program as echoed in the log, you may not see all of the code which actually executed. There are five system options which you should make sure are turned on:

- ECHOAUTO, which must be set at invocation or in your configuration file, ensures that any code in an autoexec file is echoed at the top of the log. Remember that an autoexec file may have been written by someone else and be executing without your knowledge. The default setting is NOECHOAUTO.
- SOURCE controls the echoing of your main program (primary source code). If all that appears in the log are messages, someone has probably turned this option off. This option may be (re)set at any point in your session. The default setting is SOURCE.
- SOURCE2 controls the echoing of secondary source code brought in by %INCLUDE statements. This option may be (re)set at any point in your session. The default setting is NOSOURCE2.
- MPRINT controls echoing of source code generated by macro execution. This option may be (re)set at any point in your session. The default value is NOMPRINT.
- SYMBOLGEN controls resolution of macro variables. This is especially useful in "open code" (when there is no macro program executing, and therefore MPRINT cannot be used). The default setting is NOSYMBOLGEN.

MAKE THE LOG MORE READABLE

Don't discourage yourself from looking at the log because it is cluttered and difficult to read:

- In interactive mode, clear the log periodically.
- In non-interactive mode, if your program has several sections, you may want to use the global statements SKIP <n>; or PAGE;.[7] The SKIP <n>; statement will cause n blank lines to appear in the log.[8] The PAGE; statement will cause the log to go to another page. Neither of these statements is echoed to the log.

LOOK AT THE OUTPUT

Do not restrict yourself to looking at the log. Remember that logic errors may be undetectable from the log. Look at the output of each step carefully, whether it is a report or a data file.

DEFLECT

Sometimes errors can harm your data or program logic before you even have a chance to notice them. In such cases, it is a good idea to anticipate the possibility of errors, and program around them.

DEFEND YOUR DATA

There are system options which, when turned on, will help you protect your data from the harmful effects of errors.

NOREPLACE prevents permanent data sets from being replaced. If your program is only supposed to be reading from permanent data sets, and the only data sets it should be replacing are temporary, then this might be a good option to set. It should be noted that this protection does *not* extend to techniques which modify the data in place. The default setting is REPLACE.

In the examples below, the NOREPLACE option keeps the SORT procedure from replacing SASUSER.CLASS, but does not block the SQL procedure from deleting observations (since this modifies the data set in place).

```
47   OPTIONS NOREPLACE;
48   PROC SORT DATA=sasuser.class;
49       BY Height;
50   RUN;
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
NOTE: The data set SASUSER.CLASS has 19
observations and 5 variables.
WARNING: Data set SASUSER.CLASS was not replaced
because of NOREPLACE option.

52   PROC SQL;
53   DELETE FROM sasuser.class
54   WHERE       Age > 14;
NOTE: 5 rows were deleted from SASUSER.CLASS.
```

RSASUSER puts the entire SASUSER library in read-only mode. It is valid only at session startup. The default setting is NORSASUSER.

DATASTMTCHK= allows you to prevent certain keywords from being used as one-level data set names. The possible settings are NONE, COREKEYWORDS (the default), and ALLKEYWORDS.

This option might seem unimportant to you until you see a particular type of error caused by omitting a semicolon. Notice how DATASTMTCHK=COREKEYWORD prevents the data set from being inadvertently destroyed.

```
6    OPTIONS DATASTMTCHK=NONE;
7    DATA a
8        SET sasuser.class;
9        IF age > 14;
10   RUN;
NOTE: Variable age is uninitialized.
NOTE: The data set WORK.A has 0 observations and
1 variables.
NOTE: The data set WORK.SET has 0 observations
and 1 variables.
NOTE: The data set SASUSER.CLASS has 0
observations and 1 variables.

17   OPTIONS DATASTMTCHK=COREKEYWORDS;
18   DATA a
19       SET sasuser.class;
         ---
         56
ERROR 56-185: SET is not allowed in the DATA
statement when option DATASTMTCHK=COREKEYWORDS.
Check for a missing semicolon in the DATA
statement, or use DATASTMTCHK=NONE.
20       IF age > 14;
21   RUN;
NOTE: The SAS System stopped processing this
step because of errors.
WARNING: The data set WORK.A may be incomplete.
When this step was stopped there were 0
observations and 1 variables.
WARNING: Data set WORK.A was not replaced
because this step was stopped.
WARNING: The data set WORK.SET may be
incomplete.  When this step was stopped there
were 0
observations and 1 variables.
WARNING: Data set WORK.SET was not replaced
because this step was stopped.
WARNING: The data set SASUSER.CLASS may be
incomplete.  When this step was stopped there
were 0 observations and 1 variables.
WARNING: Data set SASUSER.CLASS was not replaced
because this step was stopped.
```

MERGENOBY= allows you to specify what action SAS should take when it finds a MERGE statement with no accompanying BY statement.

A MERGE without a BY is a one-to-one merge. It is legal, but rarely useful. If you know that you never use this kind of merge, setting this option to ERROR could prevent incorrect creation or replacement of a data set. The possible settings are NOWARN (the default), WARN, and ERROR.

```
55   OPTIONS MERGENOBY=ERROR;
56   DATA a;
57       MERGE sasuser.class
58             sasuser.classcolors;
59   RUN;
ERROR: No BY statement was specified for a MERGE
statement.
NOTE: The SAS System stopped processing this
step because of errors.
WARNING: The data set WORK.A may be incomplete.
When this step was stopped there were 0
observations and 6 variables.
WARNING: Data set WORK.A was not replaced
because this step was stopped.
```

You may also want to consider password-protecting your data via one of three data set options, corresponding to three levels of protection:

- READ=, which prevents the data set from being read
- WRITE=, which prevents the data set from being modified in place
- ALTER=, which prevents the data set from being replaced or deleted

without the password.

In order to protect important data sets from being damaged by program errors, you should assign both WRITE and ALTER passwords.[9] Under UNIX and Windows, you can also apply security rules at the operating environment level. Regular backups are vital as a last line of defense.

The following example shows how to add an ALTER password to an existing data set, and what happens if you try to replace the data set with and without the password. Notice how the password is masked on the log.

```
31   PROC DATASETS LIBRARY=sasuser NOLIST;
32       MODIFY class (ALTER=XXXXXX);
33   QUIT;

36   DATA sasuser.class;
37       SET sasuser.class;
38       WHERE Age > 14;
39   RUN;
ERROR: Invalid or missing ALTER password on
member SASUSER.CLASS.DATA.
NOTE: The SAS System stopped processing this
step because of errors.

40   DATA sasuser.class (ALTER=XXXXXX);
41       SET sasuser.class;
42       WHERE Age > 14;
43   RUN;
NOTE: There were 5 observations read from the
data set SASUSER.CLASS.
NOTE: The data set SASUSER.CLASS has 5
observations and 5 variables.
```

One way you might choose to defend your data is to make sure that the missing values created by data errors are not added to important data sets. For example, we can modify an earlier example of a data error:

```
1    DATA cleandata problemdata;
```

```
2        SET sasuser.class;
3        X = LOG(Weight-Height);
4        IF (_ERROR_ = 0) THEN
5            OUTPUT cleandata;
6        ELSE
7            OUTPUT problemdata;
8    RUN;
NOTE: Invalid argument to function LOG at line 3
column 9.
name=Sandy sex=F age=11 height=51.3 weight=50.5
X=. _ERROR_=1 _N_=7
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).
1 at 3:9
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
NOTE: The data set WORK.CLEANDATA has 18
observations and 6 variables.
NOTE: The data set WORK.PROBLEMDATA has 1
observations and 6 variables.
```

There are also special statements and statement options which help your program recover from problems reading external files.

When you are reading multiple records in each iteration of the DATA step, and the correct number of records is not present, the LOSTCARD statement can help in resynchronization.

In the following example, the external file should have two records per student, with a unique identifier (Name) on each record. One student (Philip) has only one record. The programmer, having been burnt before, wrote defensive code. When the DATA step logic detects that Name1 (Philip) is not the same as Name2 (Robert), then the LOSTCARD statement executes. The SAS system recovers by assuming that the defective record is actually the first record for Robert.

```
43   DATA a (DROP=Name1 Name2);
44       INFILE 'class2records.dat';
45       INPUT #1 @1 Name1 $8.
46                @10 Sex $1.
47                @12 Age 2.
48             #2 @1 Name2 $8.
49               @10 Height 4.1
50               @15 Weight 5.1;
51       IF Name1 = Name2 THEN
52           Name = Name1;
53       ELSE
54           LOSTCARD;
55   RUN;
NOTE: Invalid data for Height in line 32 10-13.
NOTE: Invalid data for Weight in line 33 1-5.
NOTE: LOST CARD.
RULE:     ----+----1----+----2----+----3
32        Robert   M 12 13
33        Robert   64.8 128.0 19
Name1=Philip Sex=M Age=16 Name2=Robert Height=.
Weight=. Name=  _ERROR_=1 _N_=16
Name1=Philip Sex=M Age=16 Name2=Robert Height=.
Weight=. Name=  _ERROR_=1 _N_=16
NOTE: 37 records were read from the infile
'class2records.dat'.
NOTE: SAS went to a new line when INPUT
statement reached past the end of a line.
NOTE: The data set WORK.A has 18 observations
and 5 variables.
```

When using list-style input, you should always use the MISSOVER option on the INFILE statement in case there are missing fields on the record.

In the following example, the external file contains missing data (blanks) for Sharon's weight. An INFILE statement with default FLOWOVER behavior causes the INPUT statement to load another record into the input buffer to find a value for the numeric variable WEIGHT. Since the first field of each record contains character data for NAME, a data error results. The resulting data set has only 18 observations.

```
250  DATA sasuser.class;
251      INFILE 'class.dat';
252      INPUT Name $
253            Sex $
254            Age
255            Height
256            Weight;
257  RUN;
NOTE: Invalid data for Weight in line 9 1-5.
RULE:     ----+----1----+----2----+----3
9         Tammy    F 14 62.8 102.5 24
Name=Sharon Sex=F Age=15 Height=62.5 Weight=.
_ERROR_=1 _N_=8
NOTE: 19 records were read from the infile
'class.dat'.
NOTE: SAS went to a new line when INPUT
statement reached past the end of a line.
NOTE: The data set SASUSER.CLASS has 18
observations and 5 variables.
```

When you use the MISSOVER option, if SAS runs out of fields before the INPUT statement is satisfied, remaining variables are set to missing. SAS does not try to load a record prematurely. The correct number of observations is created.

```
275  DATA sasuser.class;
276      INFILE 'class.dat' MISSOVER;
277      INPUT Name $
278            Sex $
279            Age
280            Height
281            Weight;
282  RUN;
NOTE: 19 records were read from the infile
'class.dat'.
NOTE: The data set SASUSER.CLASS has 19
observations and 5 variables.
```

REROUTE THE PROGRAM

You may wish to change the execution path of your program if you detect errors. This applies primarily to non-interactive execution mode.

First, you need to find a programmatic way of detecting an error and be able to reroute the program accordingly. One way to do this is to write your program as a macro, check the macro return code after crucial steps, and branch based on conditional logic.

The global macro variable SYSERR is reset by the SAS system after DATA steps and many PROC steps. In the following program, the reports should be executed only if the the data set WORK.A is correctly created, and SYSERR receives a zero value. Since there is a missing semicolon in the DATA step, SYSERR will receive a non-zero value, the reporting procedures will be bypassed, and a special error page will be produced.

```
%MACRO TryIt;
DATA a;
       SET sasuser.class
       WHERE Age > 14;
RUN;

%IF &syserr = 0 %THEN %DO;
TITLE1 'Class Members Older Than 14';
PROC PRINT DATA=a;
RUN;
PROC TABULATE DATA=a;
       CLASS Sex;
       VAR Height;
       TABLE Sex, Height*(N MEAN);
RUN;
%END;

%ELSE %DO;
DATA _NULL_;
       FILE PRINT;
       PUT 'No output due to DATA step error';
```

```
RUN;
%END;
%MEND TryIt;

%TryIt
```

There are special global macro variables, called SYSLIBRC and SYSFILRC, which can tell you whether the most recent LIBNAME or FILENAME statement, respectively, worked correctly. As before, a zero value indicates success, and a non-zero value indicates failure.

In the following examples, if the file or directory does not exist, the program logic branches to write an error message to the log.

```
%MACRO TryIt;
FILENAME xyz 'Imaginary.dat';
%IF &sysfilrc = 0 %THEN %DO;
DATA sasuser.imaginary;
        INFILE xyz;
        INPUT @ 1 FirstName $10.
              @11 LastName $10.;
RUN;
%END;
%ELSE %DO;
%PUT Warning - External file error!;
%END;
%MEND TryIt;
%TryIt
```

```
%MACRO TryIt;
LIBNAME Mouse 'C:\Disney\Characters\Mickey';
%IF &syslibrc = 0 %THEN %DO;
PROC DATASETS LIBRARY=Mouse;
        CONTENTS DATA=_ALL_ NODS;
QUIT;
%END;
%ELSE %DO;
%put Fatal Error: Library unavailable or does
not exist.;
%END;
%MEND TryIt;
%TryIt
```

If you decide that you want to terminate the program whenever any error occurs, set the ERRORABEND system option. Data and logic errors are not considered official SAS errors, and so do not trigger termination.

This option can be (re)set at any time during the session. The default value is NOERRORABEND.

The SAS system reacts differently to errors when NOERRORABEND is in effect, depending on execution mode. In interactive mode, the session continues normally, since the programmer can intercede easily. In non-interactive mode, the SAS system displays the following message:

```
NOTE: SAS set option OBS=0 and will continue to
check statements.
This may cause NOTE: No observations in data
set.
```

Many programmers refer to this behavior as syntax-check mode.

In the following example, the session will terminate as soon as the DATA step error is detected, because ERRORABEND has been set.

```
OPTIONS ERRORABEND;

DATA a;
        SET sasuser.class
        IF Age > 14;
RUN;
/* More code */
```

In a DATA step, you can use the STOP or ABORT statements to terminate step execution in case of a data error or other undesirable condition. In addition to terminating execution of the DATA step, ABORT triggers a run-time error. ABORT also has an ABEND option which will terminate the entire program, and optionally send an error code to the operating environment.

```
1    DATA a;
2        SET sasuser.class;
3        IF Age > 14 THEN
4            STOP;
5    RUN;
NOTE: There were 6 observations read from the
data set SASUSER.CLASS.
NOTE: The data set WORK.A has 5 observations and
5 variables.
NOTE: DATA statement used:
      real time           0.06 seconds
      cpu time            0.02 seconds

7    DATA a;
8        SET sasuser.class;
9        IF Age > 14 THEN
10           ABORT;
11   RUN;
ERROR: Execution terminated by an ABORT
statement at line 10 column 9.
name=Mary sex=F age=15 height=66.5 weight=112
_ERROR_=1 _N_=6
NOTE: The SAS System stopped processing this
step because of errors.
NOTE: There were 6 observations read from the
data set SASUSER.CLASS.
WARNING: The data set WORK.A may be incomplete.
When this step was stopped there were 5
observations and 5 variables.
```

If you prefer that the program continue in syntax check mode when you encounter an error, even in interactive mode, you can adopt a programming standard in which you add macro variable references to RUN and PROC SQL statements. Then, at the point at which you detect an error, you can assign the value CANCEL to one macro variable and NOEXEC to the other. (The values are null otherwise.)

```
OPTIONS MPRINT;
%MACRO tryit;
%local  Cancel
        NoExec;

DATA a;
        SET sasuser.class
        WHERE Age > 13;
RUN;

%IF &syserr NE 0 %THEN %DO;
        %LET Cancel = CANCEL;
        %LET NoExec = NOEXEC;
%END;

PROC PRINT DATA=a;
RUN &Cancel;

PROC SQL &NoExec;
SELECT  Sex,
        AVG(Height) AS AverageHeight
FROM    sasuser.class
GROUP BY Sex;
QUIT;

%MEND tryit;
%tryit
```

Because of the missing semicolon in the DATA step, SYSERR is set to a non-zero value, the CANCEL and NOEXEC macro variables are set to values equal to their names, and the syntax of the PROCs is checked, but they do not execute. The end of the log looks like this:

```
MPRINT(TRYIT):   PROC PRINT DATA=a;
MPRINT(TRYIT):   RUN CANCEL;
NOTE: The procedure was not executed at the
user's request.

MPRINT(TRYIT):   PROC SQL NOEXEC;
MPRINT(TRYIT):   SELECT Sex, AVG(Height) AS
AverageHeight FROM sasuser.class GROUP BY Sex;
NOTE: Statement not executed due to NOEXEC
option.
MPRINT(TRYIT):   QUIT;
```

DEBUG

Once you detect an error, you need to understand why it happened and remove it. This is the process of debugging.

THE DATA STEP DEBUGGER

The DATA step debugger is an excellent tool for debugging DATA step logic errors in interactive mode. You invoke it by adding a DEBUG option to the DATA statement, e.g.:

```
        DATA a / DEBUG;
```

The debugger displays two special windows. The Debugger Source window shows the DATA step code, highlighting the statement which is about to execute. The Debugger Log window is used to enter commands and show you the results of those commands.

You control the debugger by issuing commands, which allow you to:

- step through execution one or several statements at a time
- move rapidly through execution until a predefined event occurs (e.g. a particular variable changes its value)
- evaluate expressions
- change the values of variables
- exit the debugger at any time.

If the logic error is data-dependent, you may want to apply tools such as a WHERE statement or the FIRSTOBS and OBS options to focus on those observations which are causing problems.

DATA STEP STATEMENTS

The PUT and LIST statements are useful for finding DATA step logic errors, especially in non-interactive mode. PUT writes values from the PDV or the input buffer. By default, PUT writes to the log, which is usually appropriate when your focus is on debugging. LIST writes values from the input buffer to the log.

Examining the contents of the PDV can often help you debug a DATA step. Since PUT statements can generate many lines, you may want to limit the number of PUT statements you use and the number of iterations in which you execute them. It is also a good idea to use named PUT syntax and text literals, so that it is easy to associate variables with values, and PUT statements with lines in the log.

In the following variation on a previous example (calculating average heights by Sex), the program correctly outputs once per group, but the value of AverageHeight is missing. Notes in the log also indicate that missing values are being generated.

```
361  DATA avhts (KEEP=Sex AverageHeight);
362      SET sasuser.class;
363      BY Sex;
364      IF FIRST.Sex THEN DO;
365          SumOfHeights = 0;
366          Number = 0;
367      END;
368      SumOfHeights = SumOfHeights + Height;
369      Number = Number + 1;
370      IF LAST.Sex;
371      AverageHeight = SumOfHeights / Number;
372  RUN;
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).
      17 at 368:33   17 at 369:21   2 at 371:34
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
NOTE: The data set WORK.AVHTS has 2 observations
and 2 variables.
```

By inserting two PUT statements and examining the log, the programmer discovers that SumOfHeights is not being incremented, but is being reset to missing value. Hopefully, the programmer then realizes that SumOfHeights needs to be retained.

```
DATA avhts (KEEP=Sex AverageHeight);
374      SET sasuser.class (OBS=3);
375      BY Sex;
376      IF FIRST.Sex THEN DO;
377          SumOfHeights = 0;
378          Number = 0;
379      END;
380      SumOfHeights = SumOfHeights + Height;
381      Number = Number + 1;
382      PUT Name= Height= SumOfHeights=;
383      IF LAST.Sex;
384      AverageHeight = SumOfHeights / Number;
385      PUT 'End of Group: ' AverageHeight= /;
386  RUN;

name=Alice height=56.5 SumOfHeights=56.5
name=Becka height=65.3 SumOfHeights=.
name=Gail height=64.3 SumOfHeights=.
End of Group: AverageHeight=.

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 380:33   2 at 381:21   1 at 384:34
NOTE: There were 3 observations read from the
data set SASUSER.CLASS.
NOTE: The data set WORK.AVHTS has 1 observations
and 2 variables.
```

The PUT statement can include special keywords, which can be useful for debugging:

- _ALL_ refers to the entire PDV (both user and automatic variables).
- _INFILE_ refers to the contents of the input buffer. PUT _INFILE_ can only be used after an INPUT statement.

The LIST statement also writes a record from the external file to the log, but it does not have to follow an INPUT statement, and also displays a column ruler.

GO FOR THE ROOT CAUSE

Errors rarely occur in isolation. They usually cause further errors downstream, as the assumptions of later statements and steps are violated. This means that your log may contain many warning and error messages, caused by only one mistake on your part.

Always start the debugging process with the first step in which an error message occurs. Look at all the messages for that step. Determine whether a syntax/resource error or a run-time error has occurred. It rarely makes sense to look at the messages in subsequent steps unless they recur after you correct the code in the first step and rerun the program.

Debugging is a skill which improves with experience. The more errors you make, the better you become at debugging.

SIMPLIFY

When a complex program has errors – especially logic errors - it may help to simplify the program in an attempt to isolate the problem.

Here are some guidelines:

- Focus on one step at a time.
- Carefully check the inputs and outputs of that step.
- “Comment out” statements which you think are unrelated to the problem. If the problem disappears, then add them back in one or two at a time.
- Focus on intermediate values, using tools such as the DATA step debugger or the PUT statement.
- Question your assumptions.

THE USUAL SUSPECTS

Whenever you see a syntax error message which makes no sense, a missing semicolon should be the first thing to consider. This type of error often confuses SAS software, since what you intended as two statements now look like one, statement keywords may look like options, etc. Examine the statement immediately preceding the one flagged with the error message.

One of the most annoying errors to correct is a mismatched quote. In non-interactive mode, the rest of the program will probably not run correctly, if at all. In interactive mode, some of the symptoms may vary:

- You may see a “step running” message on the title bar of the editor window.
- The log may display a warning such as:
  ```
  The TITLE statement is ambiguous
  due to invalid options or unquoted
  text.
  ```
- The log may display a warning such as:
  ```
  The quoted string currently being
  processed has become more than 262
  characters long.  You may have
  unbalanced quotation marks.
  ```

but the program will not execute and the usual log notes are absent.

The best solution for this error is to interrupt the currently executing step, correct the code, and resubmit the program. Methods for interrupting a step differ slightly across operating environments.[10]

SUSPICIOUS NOTES

There are certain notes which should always make you think twice, because they so often indicate an error. Data errors and reading past the end of a record were mentioned earlier. Here are some other notes which should sound alarms, and possible solutions:

Propagation of missing values occurs as a result of evaluating an expression, in which at least one operand has a missing value.

If your input data contain some missing values, then it may be both appropriate and expected for those values to be propagated through expression evaluation.

You may be able to avoid propagation of missing values by using functions instead of arithmetic operators. For example, the SUM function ignores missing values, whereas the + operator propagates them. In any case, it is your responsibility to determine whether propagation of missing values constitutes an error in your program.

An **uninitialized variable** note means that you have referred to a variable in a DATA step without providing a way for it to receive a value. The uninitialized variable is assigned a missing value.

In the following example, the programmer misspelled the variable Weight. Expression evaluation propagates the missing values.

```
1    DATA a;
2        SET sasuser.class;
3        WeightKg = Wait / 2.2;
4    RUN;

NOTE: Variable Wait is uninitialized.
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).
19 at 3:21
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
NOTE: The data set WORK.A has 19 observations
and 7 variables.
```

Data type conversion messages should make you ask yourself several questions. Did I intend SAS to do type conversion for me, or is one of my variables not the type that I expected? Are the default format and informats which SAS uses appropriate to the results that I want? Should I do the conversion myself with PUT or INPUT functions?

In the following example, the programmer is attempting to create a new variable called SexAge by concatenating Sex and Age. Since Age is a numeric variable, and the concatenation operator expects character operands, the SAS system does an implicit type conversion as it evaluates the expression. The format used in implicit numeric-to-character conversion is BEST12., which causes SexAge to be 13 bytes long with 10 embedded blanks in the values.

```
34   DATA a;
35       SET sasuser.class (OBS=3);
36       SexAge = Sex || Age;
37       PUT SexAge=;
38   RUN;
NOTE: Numeric values have been converted to
character values at the places given by:
      (Line):(Column).
      36:21
SexAge=F           13
SexAge=F           13
SexAge=F           14
```

Replace the default implicit conversion with an explicit conversion using the PUT function and the 2. format:

```
40   DATA a;
41       SET sasuser.class (OBS=3);
42       SexAge = Sex || PUT(Age,2.);
43       PUT SexAge=;
44   RUN;
SexAge=F13
SexAge=F13
SexAge=F14
```

A note saying that **at least one format was too small to be printed** means that the specified format is not wide enough for the data value. As a result, some parts of the displayed value may be modified: commas, dollar signs, places to the right of the decimal place, etc. If precision and appearance are important to you, you may consider this to be an error.

In the following example, we want to find out whether the entire class can safely fit on an elevator whose weight limit is 1200 pounds. The programmer ran a PROC PRINT step with a SUM statement for the WEIGHT variable. The format 4.1, although appropriate for individual weight values, is too small for the sum of all weights.

```
62   PROC PRINT DATA=sasuser.class NOOBS;
63       SUM Height;
64       FORMAT Height 4.1;
65   RUN;
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
NOTE: At least one W.D format was too small for
the number to be printed. The decimal may be
shifted by the "BEST" format.
```

```
Name        Sex     Age     Height     Weight

Alice        F       13      56.5        84.0
Becka        F       13      65.3        98.0
Gail         F       14      64.3        90.0
Karen        F       12      56.3        77.0
Kathy        F       12      59.8        84.5
Mary         F       15      66.5       112.0
Sandy        F       11      51.3        50.5
Sharon       F       15      62.5       112.5
Tammy        F       14      62.8       102.5
Alfred       M       14      69.0       112.5
Duke         M       14      63.5       102.5
Guido        M       15      67.0       133.0
James        M       12      57.3        83.0
Jeffrey      M       13      62.5        84.0
John         M       12      59.0        99.5
Philip       M       16      72.0       150.0
Robert       M       12      64.8       128.0
```

```
Thomas       M      11     57.5       85.0
William      M      15     66.5      112.0
                         ======
                          1184
```

By changing the format to 6.1, the programmer gets a more attractive and precise sum for the report.

```
67   PROC PRINT DATA=sasuser.class NOOBS;
68       SUM Height;
69       FORMAT Height 6.1;
70   RUN;
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
```

```
Name       Sex    Age    Height    Weight

Alice       F      13     56.5      84.0
Becka       F      13     65.3      98.0
Gail        F      14     64.3      90.0
Karen       F      12     56.3      77.0
Kathy       F      12     59.8      84.5
Mary        F      15     66.5     112.0
Sandy       F      11     51.3      50.5
Sharon      F      15     62.5     112.5
Tammy       F      14     62.8     102.5
Alfred      M      14     69.0     112.5
Duke        M      14     63.5     102.5
Guido       M      15     67.0     133.0
James       M      12     57.3      83.0
Jeffrey     M      13     62.5      84.0
John        M      12     59.0      99.5
Philip      M      16     72.0     150.0
Robert      M      12     64.8     128.0
Thomas      M      11     57.5      85.0
William     M      15     66.5     112.0
                         ======

                         1184.4
```

TRICKY ERROR MESSAGES

There are certain error messages which can be confusing to a beginning SAS programmer:

Symbol not recognized means that the SAS system does not understand a token[11]. This often means that the token is out of place, which in turn may mean a missing semicolon.

Illegal array reference means that you have tried to reference the array as a whole, rather than an individual element.

In the following example, the programmer wants to replace zero values in the numeric variables with missing values, but is referencing the array name inside the loop, rather than an array element.

```
101  DATA a (DROP=I);
102      SET sasuser.class;
103      ARRAY V {3} Age Height Weight;
104      DO I = 1 TO 3;
105          IF V = 0 THEN
ERROR: Illegal reference to the array V.
106              V = .;
ERROR: Illegal reference to the array V.
107      END;
108  RUN;
NOTE: The SAS System stopped processing this
step because of errors.
WARNING: The data set WORK.A may be incomplete.
When this step was stopped there were 0
observations and 5 variables.
WARNING: Data set WORK.A was not replaced
because this step was stopped.
```

The corrected program looks like this:

```
109  DATA a (DROP=I);
110      SET sasuser.class;
111      ARRAY V {3} Age Height Weight;
112      DO I = 1 TO 3;
113          IF V{I} = 0 THEN
114              V{I} = .;
115      END;
116  RUN;
NOTE: There were 19 observations read from the
data set SASUSER.CLASS.
NOTE: The data set WORK.A has 19 observations
and 5 variables.
```

RESOURCES

You are not entirely on your own. Other debugging resources available to you include:

- Hardcopy documentation, SAS OnlineDoc Documentation®, and SAS System Help
- SAS Institute Technical Support
- Colleagues
- Books By Users SAS Institute's Author Service®
- The SAS-L email list
- User group proceedings
- The SAS Institute website

DEBUGGING IS NOT TESTING

Remember that debugging is not a synonym for testing. Even though you may have debugged a program, it should be rigorously tested by other people if it will be run frequently, is mission-critical, or is part of a system.

CONCLUSION

Errors are inevitable in any programming project. The key to coping with them is to recognize the different types of errors, and to deter, detect, deflect, and debug.

REFERENCES

Burlew, M.M., Debugging SAS Programs: A Handbook of Tools and Techniques, 2001.

Cody, R., Cody's Data Cleaning Techniques Using SAS Software, 1999.

Delwiche and Slaughter: The Little SAS Book, 1995, Chapter 8: "Debugging Your SAS Programs".

Step-by-Step Programming with Base SAS Software, 2001, Chapter 24: "Diagnosing and Avoiding Errors".

SAS Language Reference: Dictionary, Version 8, 1999, Appendix 3: "DATA Step Debugger".

CONTACT INFORMATION

Contact the author at:

Roger Staum
SAS Institute
787 Seventh Ave., 47th Floor
New York, NY 10019
Email: Roger.Staum@sas.com

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

Other brands and product names are registered trademarks or trademarks of their respective companies.

[1] The program, log, and output excerpts which appear in this paper were run under Windows 2000. The principles which they illustrate, however, should apply under other variants of Windows, as well as UNIX and OS/390.

[2] A run-time error cannot occur in a global statement.

[3] This value is reset to 0 at the top of the next iteration.

[4] Of course, you can also use a system editor (such as Notepad under Windows) for this purpose.

[5] There is a system option called DETAILS which causes the CONTENTS and DATASETS procedures to provide additional information about library contents. (The default value is NODETAILS.)

[6] The DATA step and the SQL procedure generate different results when faced with this situation. The DATA step does a sequential match; PROC SQL produces a Cartesian product.

[7] These statements do work interactively, but they are more useful in non-interactive mode.

[8] SKIP; and SKIP 1; are equivalent.

[9] It is important not to forget these passwords if you ever need to modify or delete the data sets.

[10] See the SAS Companion manual for your operating environment.

[11] A token is the smallest group of characters which have meaning to the SAS system. The SAS language consists of name, numeric constant, character constant, and special character tokens.