Macro Bugs - How to Control Them
Ian Whitlock, Kennett Square, PA

ABSTRACT
The process to control the production of macro bugs is far more significant in creating successful programs than the process of eliminating a bug once you have produced it. This paper will concentrate on the conditions that cause a bug to develop. However some time will be spent discussing the system tools most fruitful for locating bugs.

The reader may be relatively new to writing macros, but he/she should bring some SAS(r) and some macro experience to the lecture for it is hard to pick up all the background even when all terms are explained. This discussion of bugs should be relevant to anyone interested in writing code and the quality of that code.

The material for this talk has been developed on the PC under Windows using SAS 8.2 and 9.1.2. However, all operating systems and platforms that can execute SAS are relevant.

INTRODUCTION
Good macro development is hard work when compared to writing small and simple SAS programs. In general macros are used to generate SAS code. They are harder to test and debug because:

- Macros usually can generate different versions of SAS code each of which may have SAS bugs.
- Mistakes can be made in the generation of the code, or in the code generated.
- There is less help available from the SAS system because the macro facility has no knowledge of SAS; macro is a text manipulation language.
- You have to consider the SAS code for each program that you want to generate.

Consequently, the process of writing macros deserves much more planning and adherence to good technical practices than are required by short SAS programs. Before attempting macro it pays to understand SAS code well; otherwise macro programming degenerates into a guessing game on what program you want.

The general principles of debugging are:
- Understand the macro facility and how it works.
- Believe that it is your code that is making the mistake.

More specifically:
- Recognize who is reporting the mistake.
- Locate the mistake.
- Recreate the mistake to prove that you understand it
- Fix the mistake.

In large and complex macro programs a big problem is caused by the fact that an action in one part of the program can cause problems in quite a different part of the program. In a pure SAS program information is passed between steps via data sets; hence there is little interaction between the steps. When the steps are small there is less chance for bugs to develop. The macro language provides a tool, macros, for organizing a program into parts (macros), and it provides macro variables (and parameters) as a means of communication between macros and between SAS steps. Consequently, the dangers of interaction between, both physically distant places in the code, and execution-time distant parts of the program, are greatly increased with the misuse of the macro language. Hence, you must guard against the opportunities that macro presents for creating bugs that are difficult to fix.

Working with two languages, SAS and macro, having separate but intertwined compile and execute times requires a great deal of understanding. Again there is much opportunity to develop bugs that are difficult to understand.
We begin with problems that arise from misunderstanding the macro facility.

**TIMING**

Errors can occur at four different times in SAS job:

- Macro compile time when the code between the %MACRO and %MEND statements is read.
- Macro execution time when the SAS code is generated by the compiled macro instructions.
- SAS compile time\(^1\) when the generated SAS code is compiled.
- SAS execution time when the generated SAS code is executed.

The times are inherently intertwined. At macro execution time as the SAS code is being generated, it is also being read and compiled by the SAS compiler. This means that macro execution must be held in abeyance during the step execution. Hence control of step boundaries becomes critical in developing macros.

Unfortunately many examples of SAS code given by the SAS Institute and by SAS programmers ignore step boundaries. Consequently the beginning macro programmer often has to overcome bad habits that he has absorbed reading code that does not adhere to good technical practices.

**CLASSIC MISTAKES**

Many of the classic mistakes can be attributed to timing issues and misunderstandings about the executing macro environment. Hence it pays to understand and recognize these mistakes.

**Step Boundary Problem**

When macro instructions are placed within a step, they are executed as the step compiles. This means that any macro instructions before the step boundary will also execute. For example, the invocation of %NEXTSTEP in the macro PRINTPLUS below will generate some code before the print, generated by %PRINTPLUS, is finished.

```sas
%macro printplus ( data = ) ;
    title1 "My important print" ;
    proc print data = &data ;
        format _all_ ;
    %nextstep()
%nextstep(data = ) ;
%mend printplus ;

%macro nextstep ( data = ) ;
    title1 "Basic Analysis Variables" ;
    proc means data = &data ;
        run ;
%nextstep();
%mend nextstep ;

%printplus( data = w )
```

When the macro %PRINTPLUS starts to execute, it generates a title, procedure statement and a FORMAT statement. Since there is no step boundary to stop the compilation of PROC PRINT, the macro instruction invoking the macro %NEXTSTEP is executed during the compilation of the PROC PRINT, not after the procedure's execution.

---

\(^1\) Technically, procedure steps are not compiled, but rather parsed. However, because the English is simpler, I will continue to use the word "compile" for procedures on the grounds that parsing is a part or form of compiling.
First `%NEXTSTEP` generates a new TITLE statement wiping out the intended title generated by `%PRINTPLUS`. Then the MEANS procedure statement is generated. This causes the macro facility to suspend operation while the previous PROC PRINT executes.

It is important to understand which macro owns the mistake. The macro `%NEXTSTEP` did not make a mistake, and it would be wrong to place a RUN statement before the TITLE1 statement in this macro. Why? Because it destroys the integrity i.e. coherence of `%NEXTSTEP`, whose job is to run PROC MEANS. When the RUN statement is added to `%NEXTSTEP`, the job becomes – fix possible boundary problems and run PROC MEANS. To see what this loss in coherence means, suppose that a new request asks for the removal of the MEANS step. The invocation of `%NEXTSTEP` is removed from `%PRINTPLUS`, and suddenly the problem fixed months ago reappears because code following the invocation of `%PRINTPLUS` creates a similar situation. `%PRINTPLUS` generated the print step and is responsible for saying when it ends. Consequently, the RUN statement belongs in the macro `%PRINTPLUS`. Note that, in either placement of the RUN statement, the same SAS code is generated; hence we are not really looking at a SAS mistake. Instead the issue is a macro one issue of design concerned with avoiding bugs. Placing the RUN statement in `%NEXTSTEP` opens `%PRINTPLUS` to future bugs. A further indication that the RUN statement belongs in `%PRINTPLUS` comes from the fact that a reader of the single macro `%PRINTPLUS` cannot know when the PRINT step is to be finished.

Here, the boundary problem was clear because the code was made deliberately simple to illustrate the problem. However, it becomes harder to follow complex situations when you start fixing one macro's mistake in another. Suppose we add another macro `%OTHERSTEP` and replace the last line of `%PRINTPLUS` with:

```sas
%if %sysfunc(inputn(&systime,time.)) < %sysfunc(inputn(12:00,time.)) %then
   %nextstep();
%else
   %otherstep();
```

Now, if the RUN statement were placed in `%NEXTSTEP` to fix the problem, the system might work correctly in any job run before noon and give the wrong title for the same macro code in any job run after noon. The situation is still simple because titles are an inherently simple situation, but the example does indicate how easily bugs develop and how important it is to understand the real problem and fix it correctly.

I have presented the issue as a timing problem that usually comes from bad coding habits; however, it is also a design issue in the sense that good design can help to avoid bugs in the first place. It is also a debugging issue in the sense that it is easier to debug macros when each macro is readable and complete in itself. Moreover, in a realistic situation it can be much harder to debug, particularly when the problem applies to something more complex than a TITLE statement.

**Macro instruction timing in DATA steps**

When macro code appears inside a DATA step, there is another timing issue. Suppose you have written and executed the following code.

```sas
data w ;
```

2 On occasion, one may want a macro `%MAC` to generate the beginning of a step without finishing it because the managing macro `%DRIVER` is responsible for calling other macros that will add to the same step. However, that is not the case with `%PRINTPLUS` since `%PRINTPLUS` causes its own problem. One should also note that the design mentioned in this footnote causes the macros invoked by `%DRIVER` to have a restricted independent use because they really need the overall management services of `%DRIVER`. On very rare occasions a macro must be designed to allow the consumer to add code to the last generated step. In this case, the situation should be well documented, and a parameter provided to make the consumer aware that he is responsible for finishing the step begun by the macro.
do obs = 1 to 10 ;
    if I <= 5 then
    do ;
    %let x = 5 ;
    end ;
else
    do ;
    %let x = 0 ;
    end ;
y = &x ;
output ;
end ;
run ;

You might have expected to see that Y has the value 5 on the first five observations and 0 on the remainder. However, Y is always 0. What happened? It is important to remember that the %LET statement, as a macro instruction, goes to macro facility for execution during the compilation of the DATA step. Hence neither %LET statement is conditional. Both DO groups are empty, and the last assignment of X wins. The %LET statements were executed long before the step finished compiling. Thus X was set first to 5 and then to 0 (once!). Since Y is set to &X, Y always has the value 0.

Note that in the above form there is no error message; nothing is wrong. You just don't like what your code said must be done. This is a common beginner mistake based on the idea that an assignment to Y is needed that can change and that macro variables can provide that change. No, the flexibility is needed at SAS execution time and it cannot be implemented at SAS compile time. That is why it is an important timing mistake.

However, there would have been error messages had you written the code without DO groups.

data w ;
do obs = 1 to 10 ;
    if obs <= 5 then
    %let x = 5 ;
else
    %let x = 0 ;
y = &x ;
output ;
end ;
run ;

Here, the message is a mysterious, "No matching IF-THEN clause." Note that there is still no macro error, since the message came from compiling the generated SAS code. The macro facility issues no error message because macro instructions may be placed anywhere between two SAS tokens. This time you just didn't get SAS code that could compile. The message is correct, but very confusing. What happened? What ends the %LET statement? The semi-colon at the end. So what ended the SAS IF statement? It is not the same semi-colon because that went to the macro facility to end the %LET statement. That is the problem! The SAS IF statement was not completed; hence the SAS compiler issued the error message indicating that there is no IF-THEN to match the ELSE. Just as the macro facility cannot help with SAS errors; SAS cannot help with macro errors, and error messages can be misleading.

---

3 Macro instructions placed between two single quote marks are treated as text; hence they will not execute, but the code is not incorrect to the macro facility. In fact, it is an important tool for delaying the execution of macro instructions.
Note that in the first case the problem is one of timing because the %LET statements are executed before the step executes. In the second case it is still a timing issue because the semi-colon that you thought present for the IF statement was taken earlier by the macro facility for the %LET statement. Both are caused by the desire to have one assignment:

\[ y = \text{<value> ;} \]

where the correct value changes from iteration to iteration, but is not determined by a DATA step variable. Compiled code cannot act that way and adding macro cannot make it act that way. The ultimate mistake is one of a desire that cannot be filled in a compiled language such as SAS. To understand why, you must understand the timing issue. Something that will happen in the future cannot affect something that was done in the past. In other words anything done at the execution time of a step cannot affect how the code in that step is compiled. In SAS macro, the trick is that it can affect how the next step is compiled because each step is executed before the next step is read or even generated.

**CALL SYMPUT timing**

CALL SYMPUT is an important DATA step function for creating or assigning macro variables with values determined by data values. We continue the previous example illustrating the timing issues involved with CALL SYMPUT.

```sas
data w ;
  do obs = 1 to 10 ;
    if obs <= 5 then
      call symput ( "x" , "5" ) ;
    else
      call symput ( "x" , "0" ) ;
    y = &x ;
  output ;
  end ;
run ;
```

Here you do get a macro error message that the macro variable X does not exist, followed by the SAS message that it doesn't understand the line

\[ y = \&x ; \]

(because assignments cannot begin with the AND operator). To make matters more confusing suppose you add

```sas
%let x = Not initialized ;
```

before the DATA statement, and

```sas
%put x=>>>&x<<< ;
```

after the RUN statement. Now the step executes and you see from the PUT statement that X is 0. However, there is a mysterious message "NOTE: Variable initialized is uninitialized." Moreover, when you look at Y in the data set W, all values are 1. What happened? This time X has a value at SAS compile time, (Not initialized); so the DATA step executes. NOT INITIALIZED is assigned to Y. Note that this is equivalent to

\[ y = \text{not ;} \]

Now (.) is false; hence Y is true, and in SAS, that means Y is given the value 1. Now what about the %PUT statement? Well the DATA step executed, so X was reassigned by the CALL SYMPUT statements 5 times to 5, and 5 more to 0. Since the assignment to 0 came last, the %PUT statement shows this value. So why does Y still have
the wrong value when X changed values? The assignment of Y needs the value, &X, when it is compiled, but at this
time the step has not executed and &X is (Not initialized).

To make matters worse, had you developed the code interactively, and simply rerun the DATA step (without the initial
%LET statement), but commenting out the offending line, then it would also have executed! Why? Since you
removed the offending assignment statement, there is no error. Now a %PUT will show that X is 0. Now if you add
back the offending assignment of Y, there is no longer a problem with execution because the previous step executed
and left X with the value 0.

To correctly use CALL SYMPUT in the assignment of Y the DATA step function SYMGET is needed. SYMGET
returns in character form the value of the macro variable named in the argument. Hence the code should be:

```sas
data w ;
  do obs = 1 to 10 ;
    if obs <= 5 then
      call symput ( "x" , "5" ) ;
    else
      call symput ( "x" , "0" ) ;
    y = input ( symget ( "x" ), best12. ) ;
    output ;
  end ;
run ;
```

In terms of DATA step/macro interaction, the mistake throughout this section is the same - you cannot use at compile
time, a value that will be created during the execution time of that step. In each case it is trying to use the macro
variable X at the DATA step compile time when the desired value is not created until execution time.

The problem merges with step boundary problem in the following form:

```sas
Data _null_ ;
  call symput ( "CurrentTime", put(time(),time.)) ;
  title "Data set W printed at &CurrentTime" ;
  Proc print data = w ;
  run ;
```

Remember that CURRENTTIME will be created when the DATA step executes, but the title will compile during the
compilation of the DATA step because there is no step boundary and TITLE is a global SAS statement.

**Single quote problem**

Macro instructions are not seen inside a pair of single quotes. For example in

```sas
%let root = c:\project\data ;
filename in '&root\stuff.dat' ;
```

the FILENAME will not execute as intended because the macro reference &ROOT is not seen by the macro facility.
The simple cure is to use double quotes instead

```sas
filename in "&root\stuff.dat" ;
```

Usually the simple solution is best because the macro facility does see inside double quotes. However, if single
quotes must be used because they will be required by a system that will receive the value or because the
programmer doesn't see how to avoid them, then macro quoting is required to hide the single quote marks at macro
compile time and reveal them during macro execution time. For the above fileref the code is
At macro compile time the expression, `%str('`), hides the single quote mark; hence the reference can be seen at macro execution time. At execution time the material inside the %UNQUOTE will be executed first, so the reference is resolved. Then the function %UNQUOTE removes the hiding from the single quote marks so that at SAS compile and execution time the desired quoted expression is seen.4

CALL EXECUTE timing

CALL EXECUTE is another important DATA step function that interfaces with the macro facility. In this case, it sends a character string to the macro facility for macro execution; hence it is useful for calling a macro many times with parameter values that are determined by the data. Suppose we want a print of 5 observations from the first two data sets in SAS help. Here is a good solution using CALL EXECUTE and the SQL dictionary view for listing current SAS data sets. However, a subtle timing issue is present.

```sas
%macro look ( data = , obs=5);
  proc print data = &data ( obs = &obs );
  run ;
%mend look ;

Data _null_ ;
set sashelp.vtable ( where = ( libname = "SASHELP" ) ) ;
data = trim(libname) || "." || memname ;
macstring = '%look(data=' || data || ")" ;
call execute ( macstring ) ;
if _n_ >= 2 then stop ; /* long list of data sets */
run ;
```

Note the use of single quote marks in the CALL EXECUTE statement. Here, we have used the fact that the macro invocation will be hidden from the macro facility at SAS compile time. Remember that in SAS the quote marks indicate a character string they are not part of the string, so when CALL EXECUTE sends the string to the macro facility the invocation is not quoted. Had double quotes been used; the macro would have been invoked at compile time.5

%IF or IF?

The confusion between the macro statement %IF and the DATA step statement IF causes many bugs which should stand out as timing issues. However, it might be better, first, to simply understand what the two statements do. A %IF statement must be in a macro, and it makes a decision about what code to generate. An IF statement must be in a DATA step and it makes an execution time decision about what code to execute. Note that when the condition fails, in a %IF statement, code is not generated, and in an IF statement, the code is not executed, but it is present

---

4 Usually the macro facility automatically unquotes any macro quoting when the generated code is passed to the SAS compiler. However, this activity is suspended for quote marks; hence the %UNQUOTE is required. Without it, there will be a mysterious message that there is an error in the FILENAME statement, with no indication of what is wrong in a normal looking FILENAME statement in a macro when the option MPRINT is used.

5 In this example, double quotes work because of the simple nature of %LOOK. However nasty bugs can be created when macro instructions other than variable referencing appear in the macro invoked by CALL EXECUTE.

6 IF statements are also allowed in some procedures that essentially act as specialized programming languages, e.g. PROC IML and PROC REPORT.
and compiled in the step. You can usually decide which statement to use by knowing what you want to accomplish.
Consider:

```sas
%macro bug ( dummy = )
  data _null_
  x = 1;
  %if x = 1 %then
  %do
    put x= ;
  %end;
  run;
%mend bug;
%bug()
```

Here the %IF statement should stand out as wrong. We do not want to decide whether to generate the PUT statement; we want to decide whether to execute the PUT statement. Once again, the mistake is caused by thinking that macro can create flexible code by changing it at SAS execution time. It is also a timing issue when you look at the condition \(X = 1\). What is \(X\)? It is simply a letter to macro facility. Remember the DATA step has not started to execute, and the macro facility knows nothing about SAS. Of course the letter, \(X\), is not the character, 1, so the consequent, i.e. the PUT statement, is never generated.

Now consider:

```sas
%macro bug ( dummy = )
  data w;
  set something ( keep = x );
  if x = 1 then do;
    proc print data = w||put(x,1.) ;
    run;
  end;
  run;
%mend bug;
%bug()
```

Here we do want to generate the PROC PRINT, but the execution of that code cannot be done in the middle of a DATA step. The solution is to use CALL EXECUTE.

```sas
%macro makeprints ( dummy = )
  data w;
  set something ( keep = x );
  if x = 1 then call execute
    ( "proc print data = w" ||put(x,1.) " ; run ;" ) ;
  run;
%mend makeprints;
%makeprints()
```

7 The parameter DUMMY is not used. It is present because after version 5 and prior to version 9 a macro could not be defined and called correctly with an empty set of parameters. However, good programming practices should always provide for the later addition of parameters.
This works because the string passed to the macro facility is SAS code. What does the macro facility do with SAS code? It dumps the code into the input stack for execution when it is read. When will it be read by the SAS compiler? After the DATA step finishes executing.

A subtle bug arises when using CALL EXECUTE to invoke some macros, since a macro invoked via CALL EXECUTE cannot pause at step boundaries while SAS execution takes place. This means that any changes to the macro environment made by an executing step will not be made. Hence, when macros directly invoked via CALL EXECUTE contain code that uses CALL EXECUTE, CALL SYMPUT, or the SQL SELECT INTO statement, they probably will not execute correctly. A simple cure is to wrap the call in the %NRSTR quoting function. For example,

```sas
%mend makeprints ;
%makeprints()
```

As given, the code will not execute correctly because the TITLE statement will be generated before the SQL code executes and assigns a value to COUNT. However, replacing the CALL EXECUTE line with

```sas
call execute ('%nrstr(%titledprint( data=' || data || ', var=' || group || '))');
```

does work. The single quotes hide everything from the macro facility at SAS compile time. The %NRSTR hides the macro invocation from the macro facility when CALL EXECUTE sends it the string. Consequently the string is dumped in the input stack for later execution. As part of the automatic activity of the macro facility, the macro quoting is removed as the string passes to the input stack. Hence the invocation is a normal one, and the generated SQL code will execute before the TITLE statement is generated. This is not a step boundary problem, but a problem of how macros are executed when invoked by CALL EXECUTE.

**Executing macro environments**

When SAS starts, a macro environment called the global environment is created. Whenever a macro is invoked, it begins executing, i.e. generating SAS code, in a new local environment that is nested in the caller’s environment.
This local environment ends with the end of execution of the macro. A local environment is a property of a macro’s execution, not of the macro.

You can think of a SAS program using macros as an upside-down tree of environments (a program environment tree) with the global environment as root, each node as the invocation of a macro, and each branch as a sequence of nested macro invocations, where only one branch is active at a time. The complete tree represents a history of the executing environments created as the SAS program executes. In this picture, branches are not pruned as each node (i.e. macro) stops executing. The program environment tree is important in macro debugging because a macro mistake must be located at a node in this tree, i.e. a particular execution of a macro. Locating a macro bug can mean more than just locating the code because it may be a bug only in a special sequence of macro invocations.

Outer environments know nothing of variables stored in an inner environment, i.e. further from the root on the same branch or further down the branch. This feature is good in that it allows data encapsulation, but it can easily lead to novice bugs when one macro is called to export a variable to another. By default a macro variable assigned in an executing macro is local to the macro's environment, unless the variable already exists in an outer executing environment. This means that a macro, %B, cannot export a variable value to a macro, %A, without using global macro variables unless some explicit preparation has been made. In other words, %A must be closer to the root than %B on the same branch, and %A must declare the macro variable that %B exports. Using SQL to get a list of data set variable names provides a classic example. Consider:

```sas
%macro makelist ( lib = work, mem= ) ;
  proc sql ;
    select name into :list separated by " "
    from dictionary.columns
    where libname = "%upcase(&lib)"
    and memname = "%upcase(&mem)"
  ;
  quit ;
%mend makelist ;
```

The variable LIST is created; however, it is not available outside the macro unless it has been declared in a still executing environment when %MAKELIST is invoked. A quick solution to the problem is to add the line

```sas
%global list ;
```

at the top of the macro.

However, from the design point of view global variables have disadvantages:

- Programmers who easily use global macro variables fail to develop good macro design habits.
- Nasty bugs occur when the value of a global macro variable is accidentally changed in another macro without declaring the name of the variable local.
- Macros that require the use of certain global variable names are restricted in use to environments that know about these names.
- Macros that use global variables cannot be understood independent of a large block of outside code because one cannot know where the values are coming from.

Sometimes you must use global variables because the macro facility provides inadequate communication tools between disjoint executing macro environments. However it is worth limiting their use as much as possible.

There is still a possibility of accidental interaction when one macro invokes another because the execution of the inner macro can see and change all of the local variables of the outer environment. The cure is to always declare any macro variables created in the inner macro with a %LOCAL statement. This has three good consequences:
• It prevents the inner macro from changing the value of any outer variable using the same name.
• It announces to the reader any variables that will be used in the macro.
• It provides an opportunity to document each variable with a short comment.

```
%local
   i /* looping index */
   count /* return variable */
```

Note that the calling macro has no means of protecting its variables; it is the responsibility of the called macro to declare its variables local so that it will not inadvertently change values of variables known to the caller. In this sense macro does not support privacy, but it does allow the macro programmer to practice it.

ROLE OF GOOD MACRO DESIGN

A macro can be viewed as providing a service. In many cases the service is to generate some SAS code to achieve an objective. The parameters in the macro definition provide the stated part of the contract – if the following parameters are appropriately assigned then the macro will provide the service. There are three sources of hidden parts of the contract:

• The macro may expect that certain global statements have been executed, such as TITLE and LIBNAME statements.
• The macro may expect certain data sets to exist and have certain names for some of its variables.
• The macro may expect that certain macro variables have been created.

Usually the first two sources of expectations are so tied up with the service that they cause little difficulty in understanding how the macro works. In contrast, every macro variable used by the macro, but not created by it, causes difficulty in reading the macro and verifying that this part of the service has been provided because there is no hint of where such activity took place. Any of these expectations can cause the macro to have reduced reuse, but again it is the third source that causes the most difficulty.

First Principle of good design

The first principle of good design is that both the service and contract of the macro be clear. A high level description of the service should be capable of reduction to a sentence. This restriction usually means that the macro has good coherence, i.e. all of its code is aimed at one objective. A good contract means the macro is encapsulated; the only outside influence on the action of the macro should be via the visible contract and the only influence the macro should have over the rest of the program should be in the explicit service that it provides. For example, if a macro changes an option for its implementation of a service, then it should return the option to the original value because the change in value was not a part of the service.

Both ideas are important for locating bugs and preventing them. If the macros are well encapsulated by the contract and the services are well defined, then if a service fails, the problem either lies within the macro that provides the service or there is a defect in one of the providers to the contract. In other words, you do not have to look at the program as a whole to locate a bug, for each part may be investigated independent of the other parts in a good design. This goes a long way toward identifying the problem. In contrast, suppose the macro's service depends on global variables. Now one may have to search through all branches of the program's environment tree of macro calls to locate where a bad value was assigned. Suppose that a called macro does not declare a local variable local and it changes the value of a variable in macro closer to the root on the same branch. Here you may see the error's effect in a macro (node) and have to search all branches extending from the node. Once you have a sound understanding of how the macro facility generates SAS code, it is this problem of interaction between two parts of a program at a

8 Other services include generating a number or character string. A macro need not generate anything; instead it assigns macro variables or writes messages to the log.
distance that often provides the bugs that are hardest to track down, because the cause of the error is nowhere near
the point that the error appears and there is little hint of where to look for the cause.
A well encapsulated macro also means that it will not cause bugs in other parts of the program. Moreover, a macro
with a general service and a contract that is specified by its parameters will also be reusable in other programs;
hence once debugged, it contributes a good section of code in any program requiring the service.

For debugging, good encapsulation can mean that a macro can be tested independent of the rest of the program by
simply supplying the parameters from quick scaffolding and making simple stubs for any macros called. If a program
has not been designed as an assembly of pieces (macros with relatively little interaction) then there are no pieces
that can be independently verified. You must work with the whole program. That also makes debugging
considerably harder.

I like to think of a group of macros as a group of little managers. Each must have a specific task and be kept
unaware of all other managers' tasks. Whenever adding another feature to the program you must make sure it goes
to the right manager in order to prevent the managers from mixing into each other's business. The top boss should
not have to worry about details and the bottom level managers should not be trying to direct the flow of execution.

Sometimes a new task requires a new macro. For example, years ago I developed a system of macros with a
manager that, given a SAS data set and any SAS variable specification (e.g. X1-X5, A--B, C-\textsc{numeric}-D, E-
\textsc{character}-F or G:), would provide an expanded space-separated list to the caller. Recently it was suggested that
I add a parameter to allow separators other than a space so that the system could be used in an SQL SELECT
statement. The task sounded simple, but a quick look at the code showed that such a change would permeate
almost every macro in the system. The old manager provided a good well defined service. What I needed was a
new macro to provide a new service - change a space separated list into a list with another separator. The new
macro took one line using \%SYSFUNC and the TRANWRD function. I tested and found quoting problems with my
first attempt. Since the old code was rock solid in providing the space separated list the problems had to be in the
new macro. In fact I tested the new macro independent of the rest of the system. This is another indication of a good
design. With only one line to work with, it did not take long to fix. Then I wrote a new manager with the same
parameters as the old one, plus a parameter for specifying the separator. This manager called the old manager to
get the space separated list. Then it called the new macro to change the list separator, and finally returned the new
list. With just two calls to debugged macros the new service was provided. The result - I had a new generally useful
tool, and I avoided a great deal of work.

Second Principle of good design

The second principle of good design is that the macro must be locally readable. Coherence helps readability, but
does not guarantee it. One way to destroy readability is too much intertwining of macro code with SAS code.
(Another is the use of names that do not convey intent.) Ultimately it is the SAS code that is important; hence the
structure of the SAS code, i.e. the constant part of the macro that does not involve macro instructions, should be
easily seen. For example compare the following two blocks of code. Either could appear in a macro and accomplish
the same thing.

```sas
data
  %if &pl = 1 %then
  errs ( keep = id1 id2 a;
    %if &p = 1 and &p2 = 1 %then id3 ;
  )
  %if &p3 = 1 %then
  main ( keep = id1-id3 a b c ) ;
  %else
  sub ;;
```

---

12
Neither block of code is fun because 1) the requirements are messy, and 2) the names P1, P2, and P3 do not help to convey anything. However, the second block, although longer, is preferable because:

- The macro part of the code has been separated from the SAS part.
- The task has been broken up into 3 parts – assign ERRSPEC and DSSPEC, then make the DATA statement.
- Each of the parts can be verified, when needed, with a simple %PUT statement.
- If the requirements for ERRSPEC and DSSPEC get more complex, these tasks can easily be handed off to helping macros.

Notice how much easier it is in the second block to see that all this code is doing is generating one DATA statement.

The first principle of good design can be viewed as a global readability requirement. If the tasks are well defined with well defined contracts then one should be able to easily (relative to the difficulty of the problem) grasp a high level idea of how the program will accomplish its task. The second principle forces local readability, i.e. when reading any small group of statements one can see how this group contributes to the immediate objective.

Good design influences the debugging process in the following ways.

- Bugs due to bad interaction between macros are avoided.
- Bugs created by unreadable code are avoided.
- When the system is self checking bugs are often revealed.
- The remaining bugs can usually be quickly located and fixed.

**The role of multiple ampersands in good design**

At times it is convenient to construct variable names. The most common form is X&I where X is constant text and I is a macro variable. The value is then referenced via I as &X&I. The macro facility rules for evaluation are:

Form a new expression by:

1. Reduce two consecutive ampersands to one.
2. Evaluate any macro variable reference.
3. Copy constant text.
4. Repeat 1-3 (called rescanning) until the text contains no variable references.

In the following line "==>" represents rescanning and I has the value 2

```
&&x&i ==&gt; expression &x2 ==&gt; value of x2
```
The form of expression is important because it allows you to think of the expression as part of an array indexed by \( I \).

The form \&\&\&X is also important because it allows one variable to name another. Suppose \&X is VAR, then

\[
&&&x \Rightarrow \text{expression} \quad \&\var \Rightarrow \text{value of var}
\]

In particular it allows a parameter to specify or point to the name of a variable that is to be assigned a value; hence the calling macro can explicitly allow the called macro to change one of its variables. For example,

\[
\%\let \&pointer = \text{<value>} \; \%** to make assignment \; \\
\%put \&\&\&pointer \; \%** example reference of the value;
\]

In the following code, the macro %DEMO declares a local variable, MYLIST, and then invokes the macro %MAKELIST (fixed from the preceding environment section) to assign the appropriate value to MYLIST.

```sas
%Maketest (lib = work, mem=, returnlist = list ) ;
   proc sql ;
   select name into :&returnlist separated by " " /*revision*/
   from dictionary.columns
   where libname = "%upcase(&lib)"
   and memname = "%upcase(&mem)"
   ;
   quit ;
%Mend makelist ;

%macro demo ;
   %local mylist ;
   %makelist ( mem = w , returnlist = mylist ) ;
   %put mylist= &mylist ;
%Mend demo ;
```

Consequently:

- No appeal to a global macro variable is required for this type of communication.
- The assignment need not be based on an "understanding" of the contract.
- The reader of DEMO knows exactly who assigned MYLIST for the %PUT statement.
- %MAKELIST is more generally useful because the consuming macro decides the name.
- The code in %MAKELIST is clearer because the service provided is not based on an "understanding".

All of the consequences either help to prevent bugs or make them easier to find and fix.

**CONCLUSION**

The macro debugging process is largely one of knowing how to locate and understand macro and SAS bugs. Using principles of macro design helps enormously to avoid or locate bugs; hence any effort in this area pays well in debugging time for the time it costs in writing the code.

Your comments and questions are valued and encouraged. Contact the author (email preferred) at:

Ian Whitlock
29 Lonsdale Lane
Kennett Square, PA 19348