Paper TT04

Make 'em %LOCAL: Avoiding Macro Variable Collisions

Arthur L. Carpenter
California Occidental Consultants, Oceanside, California

ABSTRACT

Have you ever run a macro, perhaps someone else's, and found that it inadvertently changed the value of a totally unrelated macro variable in some other seemingly unrelated macro symbol table? You may have experienced a macro variable collision

It is not unusual to use the Global macro symbol table to make macro variable values available throughout an application. Doing so, however increases the risk that a macro variable created in a macro within the application will, by mistake, overwrite that value. The risk even extends beyond the global symbol table whenever you have macros that call macros.

This paper discusses the concept of the macro variable collision, what causes it, and how you can protect yourself and others from its potentially crippling effects.

KEYWORDS

Macro variables, macro symbol tables, %LOCAL, macro variable collisions

INTRODUCTION

Macro variable collisions can occur because a macro variable name does not have to be unique within a SAS session or within an application. When macro variables with the same name are written to different symbol tables there is no problem. The collision itself is usually caused by two different macros that both address a macro variable with the same name AND both inadvertently write to the same symbol table.

Very often the author of a macro assumes that a given macro variable will be written to that macro's local symbol table when in fact it may, under certain circumstances, be written to an entirely different symbol table.

Consider the following portion of a program which has been working without an error. Recently the call to the macro %DATSERNUM (which has also been working without error) has been added. Now the SET statement is resulting in a syntax error.

```
%* Define the data set of interest;
%let dsn = clinics;

%* Determine the Data Serial Number;
%DatSerNum(adjust=5)
* Create the new data;
data new;
set &dsn;
 . . .  code not shown . . .
run;
```

Although the programmer has yet to discover the real problem, she has just experienced a macro variable collision.

THE PROBLEM IS....

When a macro creates a macro variable it is assigned to a symbol table. There is one symbol table for each macro, as well as, a global symbol table. When macros calls are nested, so are their respective symbol tables. This allows any macro to recognize any of the macro variables that reside in the current (local) or higher symbol table, including the global table, which is always the highest.

There are a series of fairly arcane rules surrounding the decision as to which symbol table is to receive a macro variable when it is first defined (Carpenter, 2004, Section 13.6). Generally speaking, when a macro variable is to be defined and it does not already exist on the current local table, the macro facility checks to see if it is already defined in a higher table. If it already exists in a higher table, that is where the value is written. When this is not what the programmer desires, a macro variable collision very well may occur.

The process becomes increasingly complex as macro calls (and their associated symbol tables) become more and more deeply nested. When a macro variable is defined inside of a macro, it is *usually* assigned to the local symbol table; this is what the programmer comes to expect. Here the '*usually*' is what is getting us in trouble. And the exceptions are what cause the collisions.

If we look more closely at the problem above we can see that the macro variable &DSN will be written to the global symbol table (assuming that the snippet of code is indeed outside of a macro definition *i.e.* it exists in open code). If any of the called macros, such as %DATSERNUM in this example, also create or modify a macro variable &DSN a collision can result.

For the example above it turns out that the problem really resides in the macro %DATSERNUM, a portion of which is shown here:

```
%macro datsernum(adjust=0);
   %if &adjust= %then %let adjust=0;
   %* Define the Data Set Number;
   %let dsn = %eval(5 + &adjust);
   %if &dsn> 5 %then %SerNumRpt(&dsn);
%mend datsernum;


%* Define the data set of interest;
%let dsn = clinics;

%* Determine the Data Serial Number;
%DatSerNum(adjust=5)
* Create the new data;
data new;
set &dsn;
 . . .  code not shown . . .
run;
```

When the variable &DSN is defined in the macro %DATSERNUM its value would normally be placed in the local symbol table for %DATSERNUM, however since &DSN already exists in a higher table, the value of &DSN in the higher table will be replaced (and &DSN will not be written to the local table for %DATSERNUM at all!). In this case the name of the data set (CLINICS) is replaced by a number. It is this number that causes the syntax error when it is used as a data set name in the DATA step that creates NEW.

A more subtle example of a macro variable collision, and one that can cause horrid errors, while leaving the programmer blissfully unawares, is contained in the program fragments below. In this case a secondary macro is called from within a %DO loop.

```
%macro primary;
   .... code not shown....
   %do i = 1 %to &dsncnt;
      %chksurvey(&&dsn&i)
   %end;
   .... code not shown....
%mend primary;
```

The %DO seems to work without error. But a closer inspection of the inner macro %CHKSURVEY reveals a hidden problem.

```
%macro chksurvey(dset);
   %do i = 1 %to 5;
      .... code not shown....
%mend chksurvey;
```

The %DO loop in %CHKSURVEY also uses &I as the index variable. Again this variable would normally be local to %CHKSURVEY, however since &I already exists in the higher table of the calling macro %CHKSURVEY will modify the value of &I in the higher table of %PRIMARY. In the case shown above, &I will have a value of 6 after %CHKSURVEY has been called. When &DSNCNT is less than 7, the %DO loop in %PRIMARY will terminate after having only executed once! If &DSNCNT is greater than 6 an infinite loop will have been established, since the &I in %PRIMARY will be reset to 6 over and over again. At least the programmer is likely to discover the latter problem.

ELIMINATING COLLISIONS

The above macro variable collisions can be completely eliminated simply by using the %LOCAL statement to identify all macro variables that you intend to be assigned to the local table of the inner macro. %CHKSURVEY becomes:

```
%macro chksurvey(dset);
   %local i;
   %do i = 1 %to 5;
      .... code not shown....
%mend chksurvey;
```

The %LOCAL statement establishes the macro variable on the local symbol table. When a value is assigned to the macro variable, in this case by the %DO statement, the value is forced to be written to the local table, regardless of whether or not it also exists in a higher table. This eliminates all possibility of a macro variable collision with a higher table.

A good rule of thumb is to **ALWAYS** use the %LOCAL statement in **ALL** of your macros. Since we do not always know how our macros will eventually be used, we need to protect any program or macro that might call our macro.

SUMMARY

If the macros that you call do not use the %LOCAL statement, you cannot protect yourself from macro variable collisions . However, if you religiously use the %LOCAL statement in all of your macros, you can protect the macros that call your macros.

REMEMBER: If you intend that a macro variable be written to the local table, force it there through the use of the %LOCAL statement.

When you control to which symbol table a given macro variable is to be written, you can be assured that your macro will not cause a macro variable collision.

ABOUT THE AUTHOR

Art Carpenter's publications list includes three books, and numerous papers and posters presented at SUGI and other user group conferences. Art has been using SAS® since 1976 and has served in various leadership positions in local, regional, national, and international user groups. He is a SAS Certified Professional™ and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

AUTHOR CONTACT

Arthur L. Carpenter
California Occidental Consultants
P.O. Box 430
Oceanside, CA 92085-0430

(760) 945-0613
art@caloxy.com
www.caloxy.com

REFERENCES

Carpenter, Art, 2004, *Carpenter's Complete Guide to the SAS® Macro Language 2nd Edition,* Cary, NC: SAS Institute Inc.,2004. Section 13.3.5 discusses macro variable collisions.

TRADEMARK INFORMATION