Paper CT06

SAS DATA Step Debugger in SAS® Enterprise Guide®

Andy Bayliss, SAS Institute, Marlow, UK

Joe Flynn, SAS Institute, Cary, USA

ABSTRACT

Have you ever run SAS® code with a DATA step and the results are not what you expected? Tracking down the problem can be a time-consuming task. To assist you in this common scenario, SAS® Enterprise Guide® 7.13 added a DATA step debugger tool. The simple and interactive DATA step debugger allows you to visually walk through the execution of your DATA step program. You can control the DATA step execution, view the variables, and set breakpoints to quickly identify data and logic errors. Come see the full capabilities of the new SAS Enterprise Guide DATA step debugger. You'll be squashing bugs in no time!

INTRODUCTION

The DATA step is a core feature of the SAS programming language. It is the primary method to create SAS data sets and process data. The DATA step can read input data, combine multiple data sources, loop to perform calculations, and use if-then-else logic for analysis. In many ways, it is a programming language unto itself. Similar to programming in any language, your DATA step will often not work as intended the first time you write it. SAS provides errors in the log, but many errors are more complex and require a debugger. Or, you might not even get an error. Your program could be syntactically correct, but not do what you expect. SAS Enterprise Guide added an interactive debugger to find and fix your DATA step problems. The DATA step debugger allows you to interactively explore your DATA step and quickly identify data and logic errors. It provides complete control of your DATA steps execution, making it easy to walk through your program line by line, or target a specific block of code. It even allows you to suspend execution of your program when a variable value changes. Variables are an important part of debugging, which is why the DATA step debugger provides a window to monitor all your variables and their values during program execution. When a variable changes, the debugger highlights this value to draw your attention to it. These are just a few of the many features provided by the DATA step debugger.

DEBUGGING BASICS

Before diving into the DATA step debugger a few basic debugging concepts need to be covered. When debugging of a DATA step begins, execution is suspended at the very first executable line of the DATA step. The program is now in a waiting state and will not proceed until it receives a command to do so. There are three ways to control the execution of your program: the **STEP**, **JUMP**, and **GO** commands.

The **STEP** command executes the highlighted source line in the debugger then suspends execution. This is great for walking through the execution of your program. Using this control mechanism is great for understanding the flow of your program. You can easily walk through nested logic and loops making even the most complicated program easier to understand.

The **JUMP** command moves program execution to a specific line. None of the intermediary statements are executed. This allows you to jump your programs execution into a loop or logic block where it would not normally execute. It can also be used to jump back to the top of your program so you can walk a specific portion again.

Finally, The **GO** command continues program execution until execution is suspended or the DATA step

completes. Suspending execution of a program is done by using the **BREAK** and **WATCH** commands.

The **BREAK** command can be used to set breakpoints throughout your program. Breakpoints allow a source line number to be specified where the execution of your DATA step should suspend. Execution is suspended when the source line is about to be executed. More advanced uses of the breakpoint command are covered later in the paper.

The **WATCH** command behaves similarly to the **BREAK** command. Instead of specifying a line of code, a variable is specified. The debugger watches that variable for changes to its value. When a change occurs, the program is suspended and the old and new values are printed to the debug console.

DISCOVERING THE DATA STEP DEBUGGER

ENABLING THE DATA STEP DEBUGGER

To use the DATA step debugger, you first have to enable it. Open a SAS program view. There is a button featuring a bug icon which toggles the enabled state of the DATA step debugger.

Run ▾ Stop | Selected Server: Local ▾ | Analyze Program ▾ | Export ▾ Send To ▾ Create ▾ | Changes Commit History

```
1 data results1;
2    set sashelp.class;
3    where sex='M';
4 run;
```

Figure 1. Enhanced Editor Options Dialog

Alternatively, the feature can be enabled from the ‘Enhanced Editor Options’ dialog by pressing Program -> Editor Options from the main Enterprise Guide menu. In this dialog check the ‘Enable DATA Step debugger’ option in the ‘General Options’ group box.

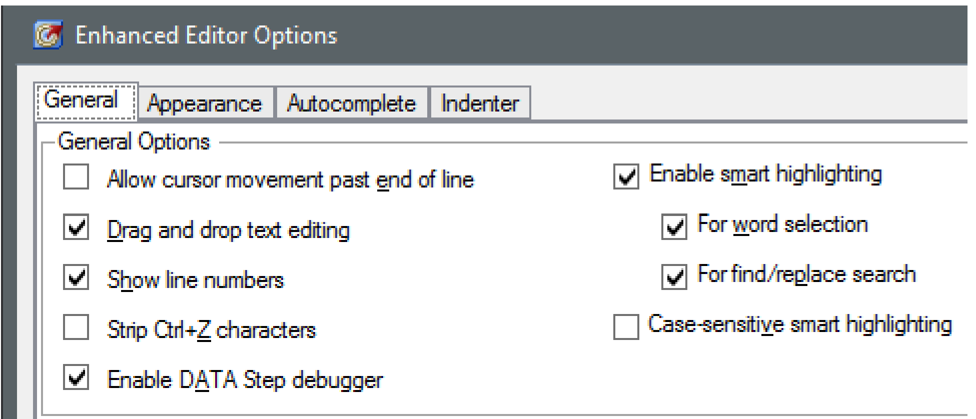


Figure 2. Enabling the DATA step debugger from the program toolbar

After enabling the feature, a green bug icon appears next to the beginning of each valid DATA step in the program. A green line also extends to the bottom of the DATA step to clearly mark the region available for debugging. At this point you notice only DATA steps are marked available for debugging. This tool cannot be used to debug SAS Procedures or Macro code.

Save ▾ Run ▾ Stop | Selected Server: Local ▾

```
1 data results1;
2    set sashelp.class;
3    where sex='M';
4 run;
```

Figure 3. DATA step debugger indicator

Launch the debugger by pressing the F5 key when the editor cursor is anywhere inside this DATA step region. Alternatively, you can click the green bug or any part of its ”bug trail” to launch the debugging window.

DEBUGGER WINDOW TOUR

The DATA Step debugger window can be broken down into four main sections: Toolbar, Debug Source Window, Watch window and Debug Console.

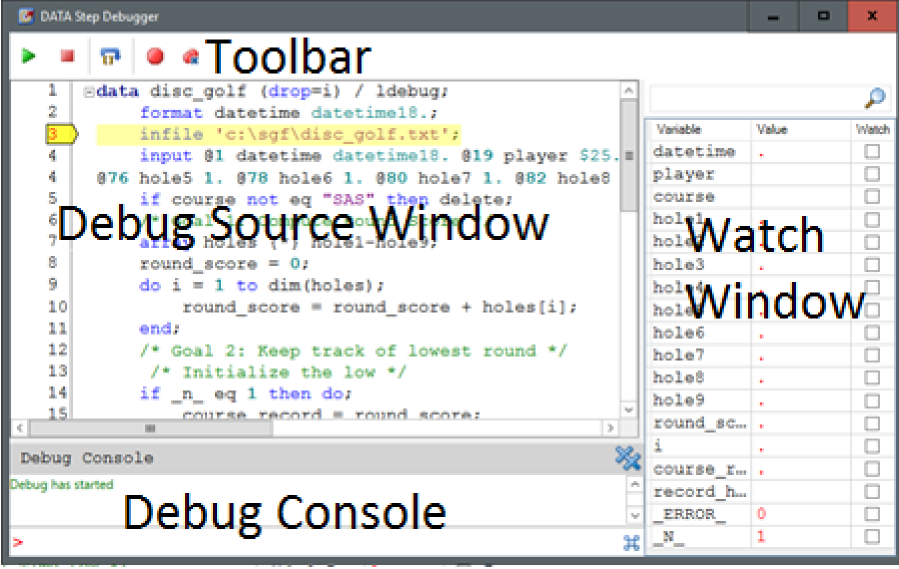


Figure 4. DATA step debugger window

Toolbar

The toolbar consists of a number of frequently used commands:

- The icon executes the GO command. If debugging is complete, this button restarts the debugger.
- The icon terminates your debugging session.
- The icon executes the STEP command.
- The icon sets a breakpoint on the selected line of the editor.
- The icon clears all current breakpoints set in the debug session.

Debugger Source Window

The debug source window provides a number of features to visually assist you when debugging. A yellow indicator is drawn on the current line number and the code is highlighted yellow to provide a quick and easy way to identify where the program execution has been suspended. One thing to note is that not all lines of your SAS program are executable. Therefore, your debugger may not suspend execution on every line of code. Format statements, informat statements, array definitions, and comments are some examples of non-executable source lines.

Any breakpoints that are set in the debugging session are displayed as red circles in the left margin. The code on a line where a breakpoint is set is highlighted with a red background.

```
3        infile 'c:\sgf\disc_golf.txt';
4        input @1 datetime datetime18. @19 player $25.
4    @76 hole5 1. @78 hole6 1. @80 hole7 1. @82 hole8
5        if course not eq "SAS" then delete;
6        /* Goal 1: Compute Round Score */
7        array holes {*} hole1-hole9;
8        round_score = 0;
9        do i = 1 to dim(holes);
10           round_score = round_score + holes[i];
11       end;
12       /* Goal 2: Keep track of lowest round */
13        /* Initialize the low */
14       if _n_ eq 1 then do;
15           course_record = round_score;
16           record_holder=player;
17       end;
```

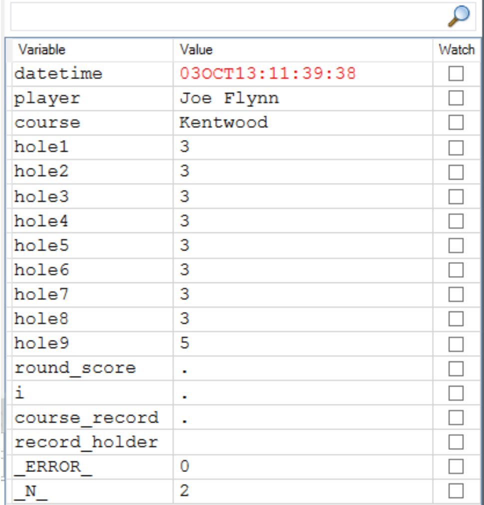
Figure 5. Breakpoint set in editor

Watch Window

The watch window provides a quick and easy way to see which variables are defined, watch their values, monitor them for any changes made, and even change values while debugging.

Viewing Variables

The watch window contains a searchable list of information found in the program data vector (PDV). As you continue program execution, these values are updated automatically. If the value of the variable has changed since the last time execution was suspended, the variable value is colored red. This provides a convenient indication of values of interest.



Variable	Value	Watch
datetime	03OCT13:11:39:38	☐
player	Joe Flynn	☐
course	Kentwood	☐
hole1	3	☐
hole2	3	☐
hole3	3	☐
hole4	3	☐
hole5	3	☐
hole6	3	☐
hole7	3	☐
hole8	3	☐
hole9	5	☐
round_score	.	☐
i	.	☐
course_record	.	☐
record_holder		☐
ERROR	0	☐
_N_	2	☐

Figure 6. Watch window

Automatic variables such as _N_ and _ERROR_ are included in the watch window. If you are using by group processing, then the FIRST. and LAST. are included as well.

Modifying Values

The watch window also lets you directly manipulate the value of a variable. With the value selected, press F2 (or double click the value) and an edit field is displayed. This allows you to manipulate values in real time.

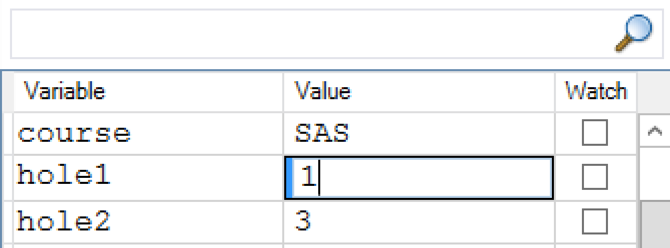


Figure 7. Modifying variable value

Watching Values

A watch checkbox is also placed next to each variable. If the checkbox is selected, the debugger 'watches' that variable for any changes. If the value of any variables being watched changes, execution is suspended. The old and new values are reported in the Debug Console.

Debug Console

The debug console provides a command line interface to debugging. It accepts most of the same commands handled by the Base SAS DATA step debugger and can be used to access more advanced debugging functionality. The debug console also provides a running log for your debug session. As you interact with the user interface, commands are submitted on your behalf. These commands are displayed in the debug console along with their output. Any errors or warnings encountered during your debug session are displayed in the debug console. Below is an example of the information displayed by the Debug Console when a watch value has changed:

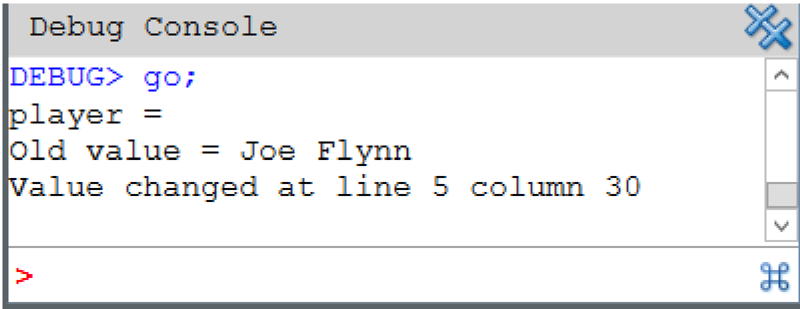


Figure 8. Debug Console displaying watch value changed

When pressing the ⌘ icon a list of the available debug commands is provided as shown in **Figure 9**. The basic syntax for each command is also provided for a quick reference. See the DATA Step Debugger Commands documentation for a detailed explanation of each command.

Note: The ENTER, HELP, SWAP, and TRACE commands do not work in the DATA Step Debugger in Enterprise Guide. Those commands are specific to the DATA Step Debugger in Base SAS.

```
Commands:
GO <line-number | label>
JUMP line-number | label
STEP <n>
CALC expression
DESCRIBE variable(s) | _ALL_
EXAMINE variable-1 <format-1> <...variable-n <format-n>>
SET variable=expression
BREAK location <AFTER count> <WHEN expression> <DO group >
DELETE BREAK location
DELETE WATCH variable(s) | _ALL_
LIST _ALL_ | BREAK | DATASETS | FILES | INFILES | WATCH
WATCH variable(s)
QUIT
```

Figure 9. Command syntax help

DATA STEP DEBUGGER IN ACTION

Now that you have been introduced to the DATA step debugger in SAS Enterprise Guide, let's walk through some examples.

The data being used in these examples was recorded while playing rounds of disc golf. If you are unfamiliar with disc golf, just imagine traditional golf but using a Frisbee. In the first example, the data is imported into SAS and subset only to rounds played at the course on SAS campus. Next, a course record will be calculated. Finally, the mental toughness of each player will be evaluated. Each example will run into a simple problem which can be tracked down using the DATA step debugger.

EXAMPLE 1: IMPORTING DATA

The data for this example is exported from a phone application into a fixed-width text document. The raw data contains columns for date-time, name, course, and a score for each hole played in the round:

```
03OCT13:11:39:38 Casey Smith          Kentwood            2 3 2 2 3 2 3 3 2
03OCT13:11:39:38 Joe Flynn            Kentwood            3 3 3 3 3 3 3 3 5
03OCT13:11:39:38 Tim Beese            Kentwood            2 3 3 3 3 2 3 3 2
04NOV13:11:34:05 David Henderson      SAS                 3 3 3 4 3 4 4 4 3
04NOV13:11:34:05 Tim Beese            SAS                 2 2 3 4 4 3 4 3 3
04NOV13:11:58:43 David Henderson      SAS                 3 3 3 3 4 3 4 4 4
04NOV13:11:58:43 Tim Beese            SAS                 3 4 3 3 4 3 3 4 3
11NOV13:11:34:51 Casey Smith          SAS                 4 3 3 2 3 2 3 4 3
```

Figure 10. Fixed width disc golf data

I have written some initial code to read in and subset the data. Again, we only want to look at rounds played at the course located on SAS campus. An initial revision of the program can be found below:

```
data disc_golf;
      format datetime datetime18.;
      infile 'c:\sgf\disc_golf.txt';
      input datetime datetime18. player $26. course $CHAR15. hole1-hole9;
      if course not eq 'SAS' then delete;
run;
```

The code runs successfully and there are no warnings or errors in the log. However, we know there is a problem because no observations were written to the output data set. We'll use the DATA step debugger to investigate the problem further.

To launch the debugger, press the bug icon in the program editor:

```
1  data disc_golf;
2      format datetin
3      infile 'c:\sgi
4      input @1 datet
5      if course not
6  run;
7
```

Figure 11. Debug indicator next to program

Once the DATA step debugger window is displayed, execution of the program is suspended on the first line. Now, you can STEP through the program using either the **F10** key or the button on the toolbar. Each time you STEP, the highlighted code is executed. Shown in **Figure 12**, we step over the input statement, which is located on line 4. The variables have been populated automatically in the watch window.

```
1  data disc_golf / ldebug;
2      format datetime datetime18.;
3      infile 'c:\sgf\disc_golf.txt';
4      input @1 datetime datetime18. @19 play
4  @76 hole5 1. @78 hole6 1. @80 hole7 1. @82
5      if course not eq 'SAS' then delete;
```

Variable	Value
datetime	03OCT13:11:39:38
player	Casey Smith
course	Kentwood

Figure 12. Execution suspended after input statement executes

The IF statement on line 5 compares the value of the course variable to 'SAS' and deletes the observation if they are not equal. Continue to step through the program until the fourth iteration of the DATA step. This is the first observation where course is SAS.

```
4  @76 hole5 1. @78 hole6 1. @80 hole7 1. @82
5      if course not eq 'SAS' then delete;
```

player	David Henderson
course	SAS

Figure 13. Course variable with leading space

The watch window shows the value of each variable and the value of 'course' has a leading space. The watch window uses a monospace font which makes detecting leading spaces much easier. This leading space can be observed for all values, and therefore the location of the course variable in the input statement must be incorrect. To fix this problem, modify the input statement to include one extra space for the 'player' variable, so that the 'course' variable will start at the proper location.

```
input datetime datetime18. player $27. course $CHAR15. hole1-hole9;
```

Running this modified program will show observations written to the dataset. Problem solved.

EXAMPLE 2: CALCULATING THE COURSE RECORD

Now that the data is being correctly read in and subset, it is time to calculate a course record. First you need to calculate a score for the entire round. Since the SAS course is only nine holes, this can be done by adding up the scores for holes 1 through 9. Next, the lowest round score and player name needs to be retained and compared during each subsequent iteration of the DATA step. The code in this example is integrated with the DATA step found in Example 1. I've added comments to the code specifying which goal each section of the code is accomplishing. Here is my first attempt:

```
data disc_golf (drop=i);
      format datetime datetime18.;
      infile 'c:\sgf\disc_golf.txt';
      input datetime datetime18. player $27. course $CHAR15. hole1-hole9;
      if course not eq "SAS" then delete;
      /* Goal 1: Compute Round Score */
      array holes {*} hole1-hole9;
      round_score = 0;
      do i = 1 to dim(holes);
            round_score = round_score + holes[i];
      end;
      /* Goal 2: Keep track of lowest round */
      /* Initialize the low */
      if _n_ eq 1 then do;
            course_record = round_score;
            record_holder = player;
      end;
      /* Goal 2: Keep track of lowest round */
      /* Update All Time Low */
      if (course_record > round_score) then do;
            course_record = round_score;
            record_holder = player;
      end;
      /* Goal 2: Keep track of lowest round */
retain course_record record_holder;
run;
```

After executing the program something appears to be wrong. Both the course_record and record_holder variables are missing in the resulting dataset. Again, no warnings or errors are listed in the SAS Log. Below is a screenshot of the output data:

hole9	round_score	course_record	record_holder
5	33	.	
5	34	.	
5	36	.	
4	37	.	
4	34	.	
4	37	.	

Figure 14. Resulting data after initial program execution

Launch the debugger to investigate why the course_record and record_holder variables are never set. You can use the **WATCH** command to track any changes made to the course_record variable. Setting a watch on course_record suspends program execution whenever the variable is modified. To set a watch on the course_record variable, locate it in the watch window, and check the **Watch** checkbox.

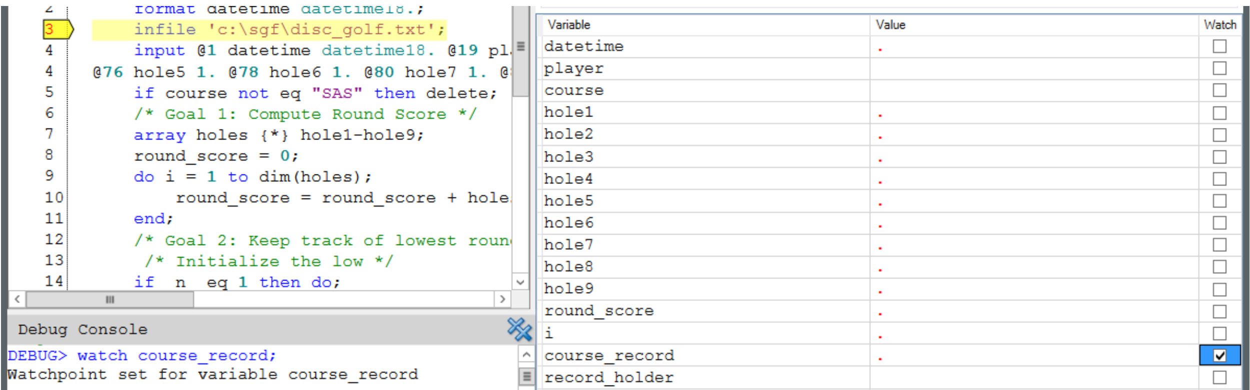


Figure 15. WATCH set on course_record variable

Press **F5** to continue execution until the value of course_record changes. Surprisingly, the debugger never suspends execution. The data step runs to completion. The debugger displays a message stating 'Debugging is now complete'.

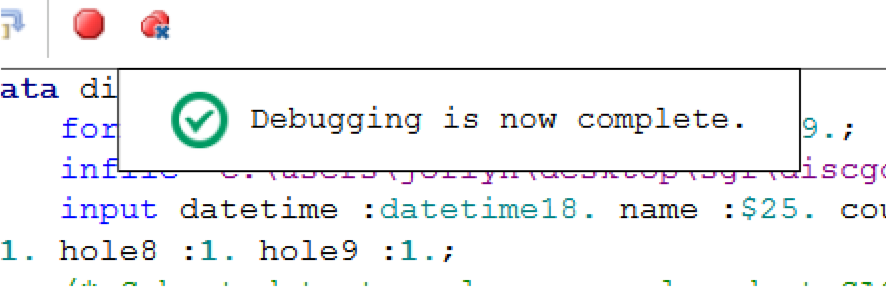


Figure 16. DATA step debugger execution complete

Again, this is an unexpected behavior. It is unclear why the course_record variable is missing. Press the **F5** key to restart debuggging. Instead of using the WATCH command, begin manually stepping through the program using the **F10** key or the step button on the toolbar. Stepping through I can see the input statement correctly populates all the variables. When hitting line 6 of the program source however, the value of course is not "SAS", so the delete statement is executed.

```
4   :1. hole8 :1. hole9 :1.;
5       /* Subset data to only games played
6       if course not eq "SAS" then delete;
```

Figure 17. DELETE statement executing

After stepping over the delete statement, program execution moves back up to line 3, and the next iteration of the DATA step begins.

Source line 14 is where _N_ is checked and initializes course_record. This is only done on the first iteration of the DATA step. Set a breakpoint on line 14 by clicking the line number in the editor. A red circle appears in the left margin, and the source on that line is highlighted red.

Now that the breakpoint is set, press **F5** to continue execution. Execution is stopped on source line 14 as we expect.

```
14    if _n_ eq 1 then do;
15        course_record = round_score;
16        record_holder = player;
17    end;
18    /* Goal 2: Keep track of lowest rour
19     /* Update All Time Low */
20    if (course_record > round_score) the
21        course_record = round_score;
```

hole9	3
round_score	31
i	10
course_record	.
record_holder	
ERROR	0
_N_	4

Figure 18. Bug in the program is located

Using the watch window, you notice that the first time the statement executes, _N_ is already set to 4. The delete statement is preventing the course_record score from being initialized. Since our variables are never initialized, each row is comparing its value to a missing number, and the course record is never set.

At this point, I'd like to see if initializing course_record and record_holder will fix the issue. The JUMP command can be used to force the program to move execution inside the IF block. This command can be accessed from the editor context menu or directly from the debug console by entering: jump 15.

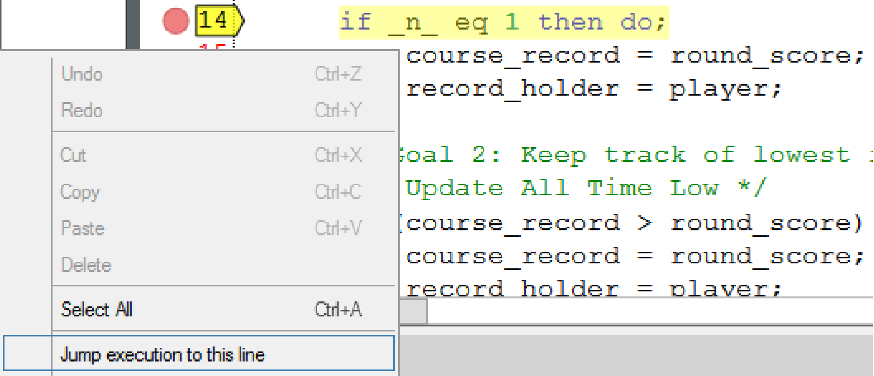


Figure 19. Jumping execution to specified source line

Line 15 is now highlighted. This allows the course_record and record_holder fields to be initialized. Stepping through the rest of the program, you can observe that the values are properly retained and updated. Now that the logic error in the program has been identified, the code can be fixed up to properly initialize these variables. This can be done by either replacing the _N_ check with one for a missing value, or by moving the delete command to the bottom of the program. Moving the delete to the bottom of the program does however incur the cost of calculating round scores for observations which will eventually be deleted.

EXAMPLE 3: MENTAL TOUGHNESS

The final analysis to be performed on the data will be to gauge "mental toughness". We will use how well a player scores after a good or bad hole as an indicator of mental toughness. Players' mental toughness will be determined by calculating their mean score on holes after scoring under par (which is known as a ' birdy' and is good) or over par (which is known as a 'bogie' and is bad). If the player performs worse on holes after a bad score on a hole, we will treat it as a sign of mental weakness. The player allows their mental state to be impacted by poor play. If the player performs consistently on holes following good and bad holes, we will treat it as a sign of mental strength. The player retains their composure, regardless of prior results.

To compute the average score after a birdie or bogie, we first sort the data by player and make use of by-group processing. A total count must be kept of the number of birdies and bogies, as well as a total score for each category. When the last observation of the player by-group is hit, an average is calculated using the count and total score from each category. A mental score is then given by subtracting birdie average

from the bogie average. If this score is greater than zero, we indicate that the player is mentally weak. The initial code to calculate mental toughness can be found below:

```
data mental_toughness(keep=player avg_score_after_birdie
avg_score_after_bogie mentally_weak);
      set work.disc_golf;
      by player;
      array holes {*} hole1-hole9;
      /* Initialize variables */
      if first.player then do;
            birdie_count = 0;
            bogie_count = 0;
            total_score_after_birdie = 0;
            total_score_after_bogie = 0;
      end;
      /* Loop through holes 1 - 8. */
      /* Hole 9 cannot factor into this calculation */
      /* because there is no hole after it */
      do i = 1 to dim(holes)-1;
            if holes[i] < 3 then do;
                  birdie_count = birdie_count + 1;
                  total_score_after_birdie = total_score_after_birdie
                                                + holes[i];
            end;
            if holes[i] > 3 then do;
                  bogie_count = bogie_count + 1;
                  total_score_after_bogie = total_score_after_bogie
                                               + holes[i+1];
            end;
      end;

      /* Calculate mental toughness score for player */
      if last.player then do;
            avg_score_after_birdie = total_score_after_birdie / birdie_count;
            avg_score_after_bogie = total_score_after_bogie / bogie_count;
            ms = score_after_bogie - score_after_birdie;
            if (ms) > 0 then mentally_weak = "Y";
            else mentally_weak = "N";
            output;
      end;

      /* retain totals for player */
      retain birdie_count bogie_count total_score_after_birdie
             total_score_after_bogie;
run;
```

Execute the program and examine the output dataset.

	player	avg_score_after_birdie	avg_score_after_bogie	mentally_weak
1	Andrew Clegg	2	3.96875	N
2	Casey Smith	2	2.9777777778	N
3	David Henders...	2	3.6598639456	N
4	Eric Hill	1.8	3.8217054264	N
5	Joe Flynn	2	3.4173228346	N
6	Tim Beese	1.995049505	3.0533807829	N

Figure 20. Initial mental toughness results

Although there are no errors or warnings in the log, the average score after birdie appears to be incorrect. According to the data, most players average a birdie after scoring under par. Some players even average

less than two. It is difficult to imagine someone throwing a hole in one regularly. There must be something incorrect in the code, so again we go back to the debugger to investigate.

Rather than stepping through the entire program, the area of most interest is when calculating scores after holes where the score was less than three. We will now use a more advanced breakpoint feature. Breakpoints allow a condition to be attached, so the program will only suspend execution when the source line is hit AND the condition is met. Since we know the problematic area of our program deals with the calculation for holes following a score under three, we can specifically target these points in the program execution. Locating the DO loop which iterates through the score for each hole looking for bogies and birdies, we can set a conditional break point. Issue the command in **Figure 21** to the debug console.

```
13        do i = 1 to dim(holes)-1;
14            if holes[i] < 3 then do;
15                birdie_count = birdie_
16                total_score_after_bird
17            end;

Debug Console
Debug has started
DEBUG> break 14 when holes[i] < 3;
Breakpoint 1 set at line 14
```

Figure 21. Conditional breakpoint set

The syntax is fairly straight forward. Break at source line 14 when holes[i] is less than 3. This breakpoint appears in the editor after it is issued in the debug console. Press the **F5** key to continue execution and the program runs until source line 14 is hit and the condition is met.

When the program suspends, there are a few values of interest. First, you can examine some variables at this point in the program. You can examine holes[i] to ensure that the current hole is in fact a birdie, holes[i+1] to see the score of the next hole, and the total_score_after_birdie. Since we're working with an array, you'll have to locate the value of i, then look at the corresponding hole variable associated with it in the watch window. Alternatively, the "examine" command can help make this process a bit easier since it displays one or more variable values. Submit the following command in the debug console:

examine holes[i] holes[i+1] total_score_after_birdie

Figure 22Error! Reference source not found. shows a screenshot of the current state of the program and the resulting output of the examine command. Notice that each variable you examine is returned on its own line in the debug console.

```
 9            total_score_after_birdie = 0;
10            total_score_after_bogie = 0;
11        end;
12        /* Loop through holes 1 - 8. Hole 9 cannot fac
13        do i = 1 to dim(holes)-1;
14            if holes[i] < 3 then do;
15                birdie_count = birdie_count + 1;
16                total_score_after_birdie = total_score
17            end;
```

Debug Console

```
Debug has started
DEBUG> break 14 when holes[i] < 3;
Breakpoint 1 set at line 14
DEBUG> go;
Break at line 14 column 3
DEBUG> examine holes[i] holes[i+1] total_score_after_birdie;
holes[4](hole4) = 2
holes[5](hole5) = 3
total_score_after_birdie = 0
```

hole3	3
hole4	2
hole5	3
hole6	3
hole7	3
hole8	5
hole9	3
round_score	30
course_record	.
record_holder	
FIRST.player	0
LAST.player	1
birdie_count	0
bogie_count	31
total_score_after_birdie	0
total_score_after_bogie	124
i	4
avg_score_after_birdie	.
avg_score_after_bogie	.

Figure 22. Examine command and watch windows

Examining the variables, you can see on hole four a score of two was recorded. The next hole, which is hole five, recorded a score of three. The value of total_score_after_birdie is currently zero. This means the expected value of total_score_after_birdie should be three after the calculation. Use F10 or the button on the toolbar to step through the do loop until the value of total_score_after_birdie is calculated.

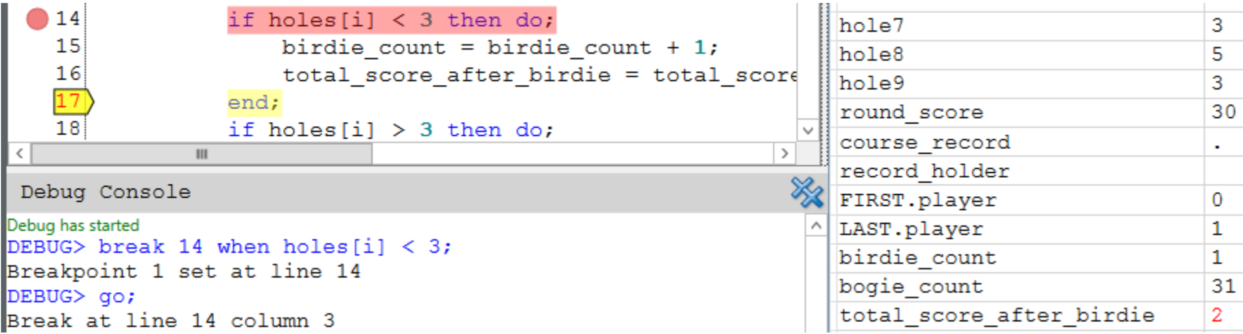


Figure 23. Watch window after incorrect calculation

After stepping over the calculation the value is two rather than three. This points to an incorrect calculation. Taking a closer look at the source code to calculate total_score_after_birdie:

```
total_player_score_after_birdie = total_player_score_after_birdie + holes[i];
```

The mistake in the code is using the score of the current hole in the calculation instead of the next hole. As a result, only the holes with scores less than 3 are ever added to the total. This explains the extremely low scores. Modify the source code to use holes[i+1] instead. After the program is executed again, the results look much more reasonable.

	player	avg_score_after_birdie	avg_score_after_bogie	mentally_weak
1	Andrew Clegg	3	3.96875	Y
2	Casey Smith	3.046875	2.9777777778	N
3	David Henders...	3.4285714286	3.6598639456	Y
4	Eric Hill	4	3.8217054264	N
5	Joe Flynn	3.2976190476	3.4173228346	Y
6	Tim Beese	3.0940594059	3.0533807829	N

Figure 24. Final results

CONCLUSION

Whether you are trying to locate a bug or understand existing legacy code, the DATA step debugger is a great feature for digging into your DATA step code. Hopefully you now have a better understanding of the capabilities of the DATA step debugger and will be able to put it to good use.

RECOMMENDED READING

- *Base SAS® Utilities: Reference*
- SAS Institute Inc. 2013. "DATA Step Debugger" chapters in *Base SAS(R) 9.4 Utilities: Reference.* Available at http://support.sas.com/documentation/cdl/en/lebaseutilref/64791/PDF/default/lebaseutilref.pdf.
- Hemedinger, Chris. "Using the DATA step debugger in SAS Enterprise Guide" in "The SAS Dummy: A SAS blog for the rest of us." Available at http://blogs.sas.com/content/sasdummy/2016/11/30/data-step-debugger-sas-eg/. Accessed on February 3, 2017.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Joe Flynn
SAS Institute Inc.
joe.flynn@sas.com
http://www.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 brand and product names are trademarks of their respective companies.