Oracle® Developer Studio 12.6: dbxtool Tutorial

June 2017

Part No: E77794

- “Introduction” on page 2
- “Example Program” on page 2
- “Configuring dbxtool” on page 3
- “Diagnosing a Core Dump” on page 9
- “Using Breakpoints and Stepping” on page 13
- “Using Advanced Breakpoint Techniques” on page 22
- “Using Breakpoint Scripts to Patch Your Code” on page 39

Introduction

This tutorial uses a “buggy” example program to demonstrate how to use `dbxtool`, the stand-alone graphical user interface (GUI) for the `dbx` debugger, effectively. It starts with the basics and then moves on to more advanced features.

Example Program

This tutorial uses a simplified and somewhat artificial simulation of the `dbx` debugger. The source code for this C++ program is available in the sample applications zip file on the Oracle Developer Studio 12.6 downloads web page at `http://www.oracle.com/technetwork/server-storage/developerstudio/downloads/index.html`.

After accepting the license and downloading, you can extract the zip file in a directory of your choice.

1. If you have not already done so, download the sample applications zip file, and unpack the file in a location of your choice. The `debug_tutorial` application is located in the `Debugger` subdirectory of the `OracleDeveloperStudio12.5-Samples` directory.
2. Build the program.

```
$ make
CC -g   -c  main.cc
CC -g   -c  interp.cc
CC -g   -c  cmd.cc
CC -g   -c  debugger.cc
CC -g   -c  cmds.cc
CC -g   main.o interp.o cmd.o debugger.o cmds.o -o a.out
```

The program is made up of the following modules:

`cmd.h`	`cmd.cc`	Class `Cmd`, a base for implementing debugger commands
`interp.h`	`interp.cc`	Class `Interp`, a simple command interpreter
`debugger.h`	`debugger.cc`	Class `Debugger`, mimics the main semantics of a debugger
`cmds.h`	`cmds.cc`	Implementations of various debugging commands
`main.h`	`main.cc`	The `main()` function and error handling. Sets up an `Interp`, creates various commands and assigns them to the `Interp`. Runs the `Interp`.

Run the program and try a few `dbx` commands.

```
$ a.out
```

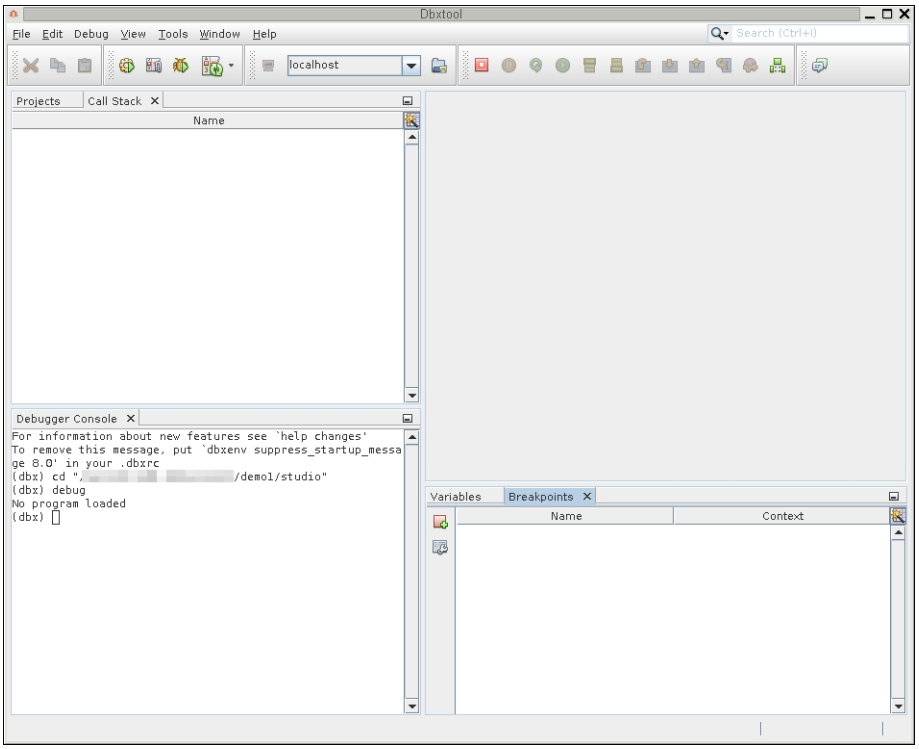
```
> display var
will display 'var'
> stop in X
> run running ...
stopped in X
var = {
        a = '100'
        b = '101'
        c = '<error>'
        d = '102'
        e = '103'
        f = '104'
}
> quit
Goodby
$
```

Configuring `dbxtool`

Start `dbxtool` by typing:

install-dir`/bin/dbxtool`

The first time you start dbxtool, the window probably looks like the following:



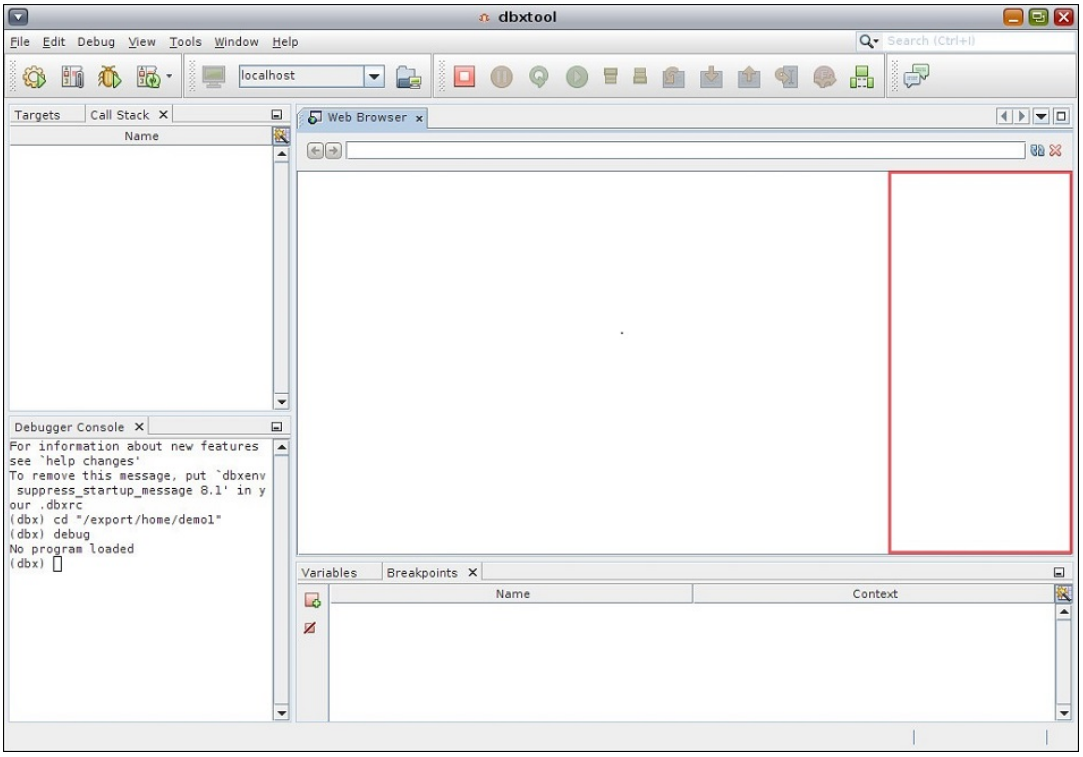
Note - Figures in this tutorial might differ from what you see when you use dbxtool.

If you need more room for other applications like a web browser, you might want to customize `dbxtool` to take up less space.

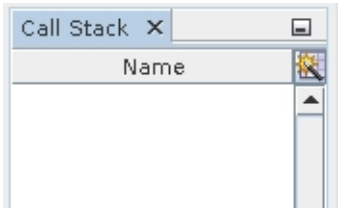
The following are examples of the various ways you can customize `dbxtool`.

- **Make the toolbar icons smaller.**
  - Right-click anywhere in the toolbar and choose Small Toolbar Icons.
- **Move the Call Stack window out of the way.**

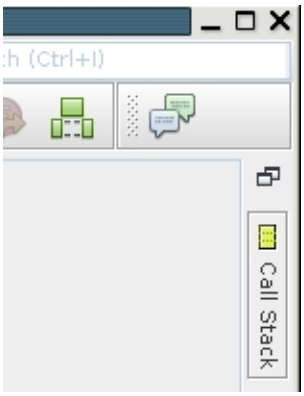
1. Click the header of the Call Stack window and drag the window downward and to the right. Let it go when the red outline is in the position in the following illustration:



2. Click the minimize button in the header of the Call Stack window.

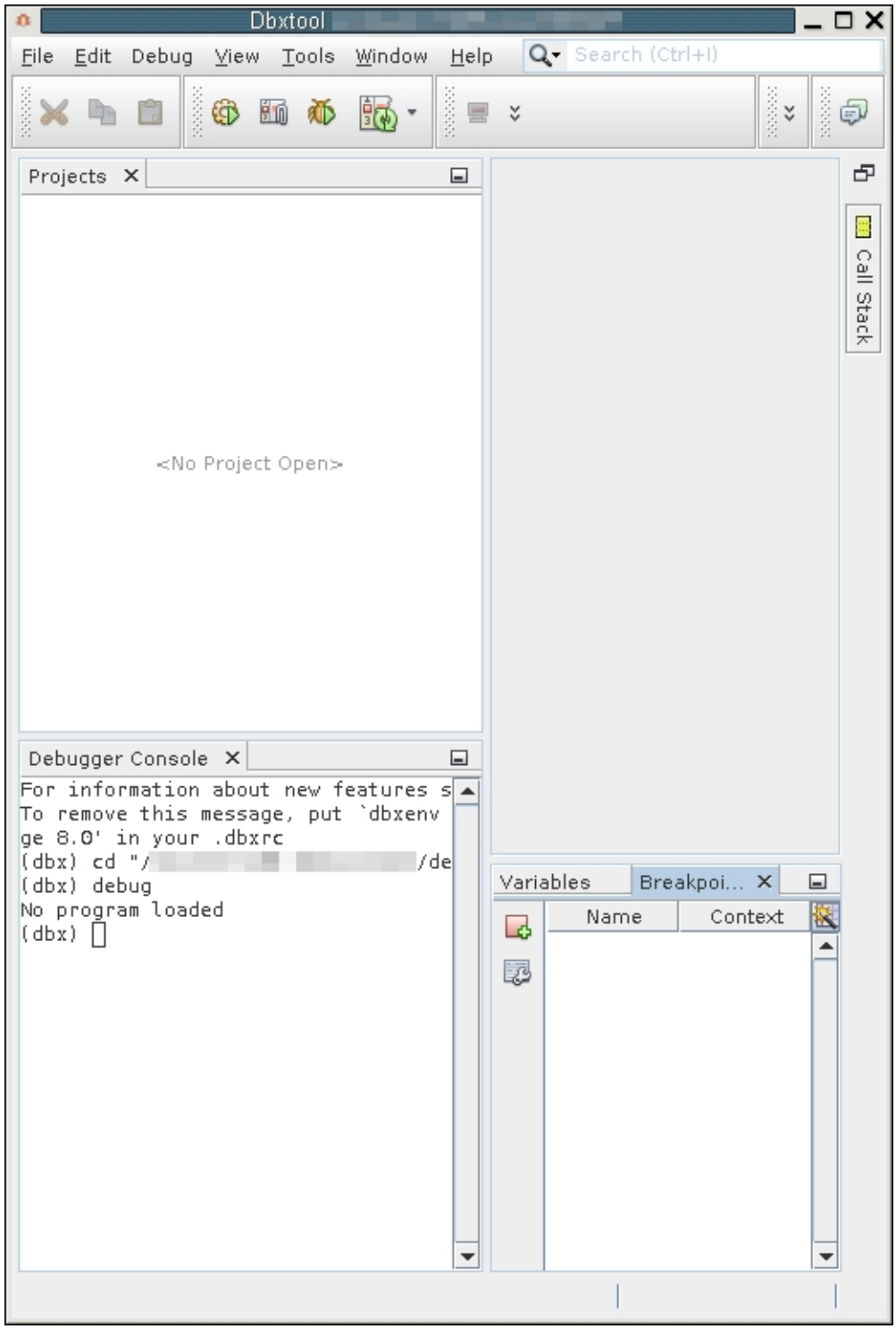


The Call Stack window is minimized in the right margin.



If you hold the cursor over the minimized Call Stack icon, the Call Stack window is maximized until you transfer focus to another window. If you click the minimized Call Stack icon, the Call Stack window is maximized until you click the icon again.

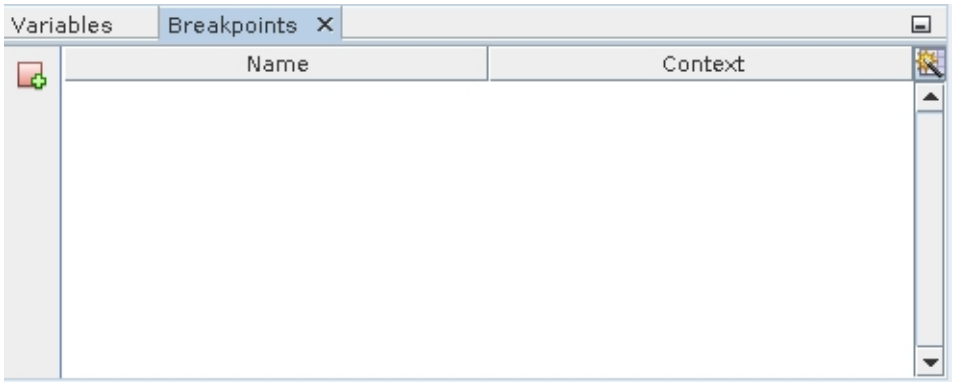
3. Narrow the main window to half-screen:



- **Minimize the windows grouping:**

  dbxtool can group windows together. You can perform actions on groups of windows in addition to individual windows. Each window belongs to a group that you can minimize/restore, drag to a new location, float in a separate window, or dock back into the IDE window.

For example, if you click the Breakpoints tab and then click the minimize window button, the entire window group minimizes.



- **Undock the Output window so you can easily interact with the input and output of programs you are debugging while having easy access to the other tabs in the dbxtool window.**
  - If you do not see the Output window, click Window → Output or press Ctrl - 4.
  - Click and hold on the header of the Output window, drag the window outside of the dbxtool window, and drop it onto your desktop.

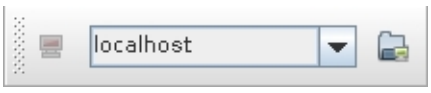
    To re-dock the Output window in the dbxtool window, right-click in the Output window and choose Dock Group.
- **Set the font size in the editor.** After you have some source code displayed in the Editor window, do the following to set the font size:

  1. Choose Tools → Options.
  2. In the Options window, select the Fonts & Colors category.
  3. On the Syntax tab, make sure All Languages is selected from the Languages drop-down list.
  4. Click the Browse (...) button next to the Font text box.
  5. In the Font Chooser dialog box, set the font, style, and size, and click OK.
  6. Click OK in the Options window.
- **Set the font size in the terminal windows.** The Debugger Console and Output windows are ANSI terminal emulators.

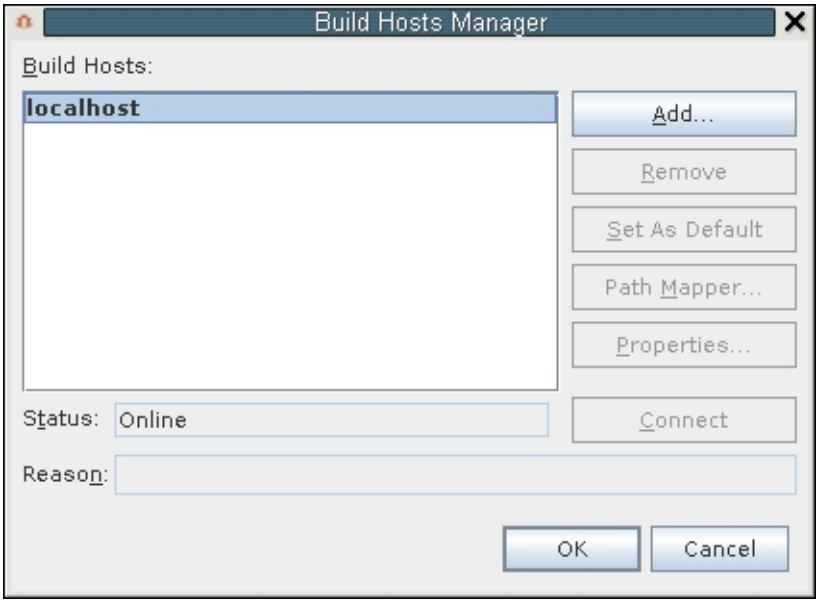
  1. Choose Tools → Options.
  2. In the Options window, select the Miscellaneous category.
  3. Click the Terminal tab.
  4. Select settings like Font Size and Click To Type.
  5. Click OK.
- **Add a remote host to run the debugger on.** dbxtool enables you to access remote servers to run dbxtool on, as well as accessing remote files.

  To add a remote host to dbxtool:

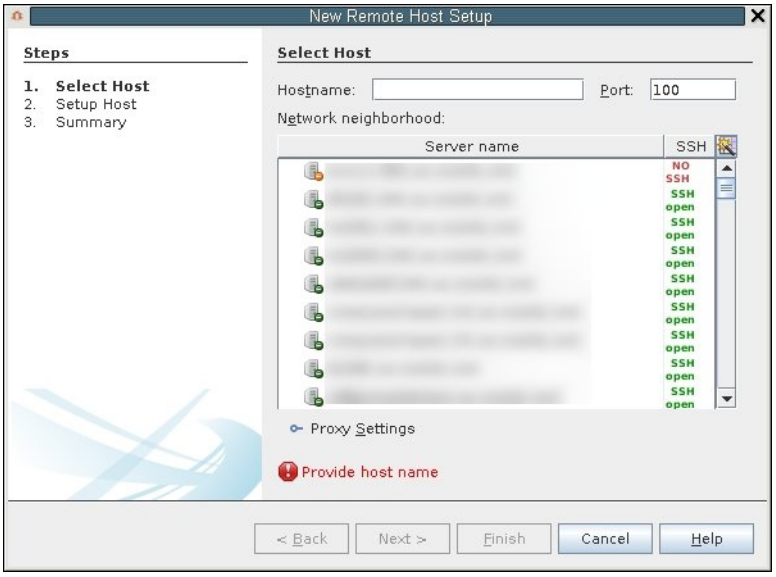
  1. In the Remote tool bar, click the down-arrow of the host drop-down list and choose Manage Hosts.



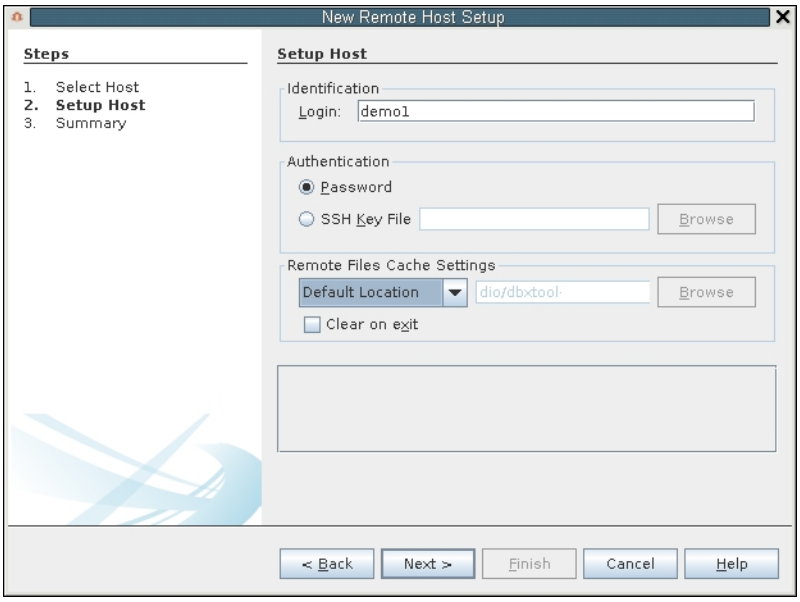
2. The Build Hosts Manager opens. Click Add to add a new server.



3. In the New Remote Host Setup wizard, choose an available server from the Network neighborhood list and click Next.



4. Enter your login information, choose an authentication method and click Next. If you chose Password, enter your password when prompted.



5. When your host is connected, the summary page shows your connection status. You can choose this remote host from the Remote toolbar while you are working.

For more information about remote hosts, see the online help in dbxtool, under the Remote Debugging topic.

Exit dbxtool once you are finished customizing. dbxtool remembers your preferences the next time you run it.

Diagnosing a Core Dump

To find bugs, run the example program again, and press Return without entering a command.

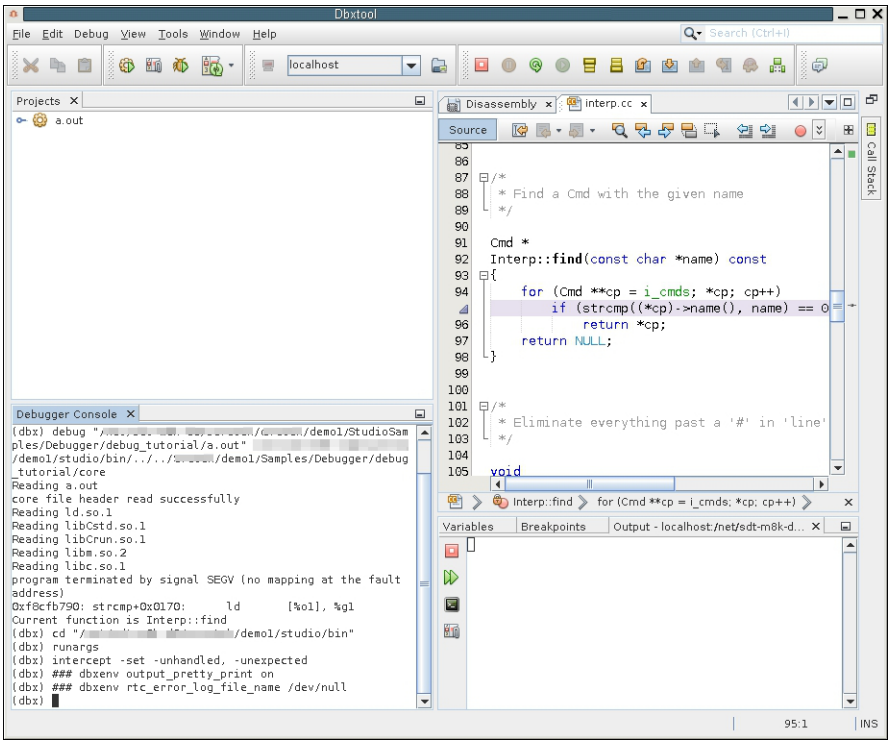
```
$ a.out
> display var
will display 'var'
>
Segmentation Fault (core dumped)
$
```

Start `dbxtool` with the executable and the core file.

```
$ dbxtool a.out core
```

Notice that the `dbxtool` command accepts the same arguments as the `dbx` command.

`dbxtool` displays output like the following example.

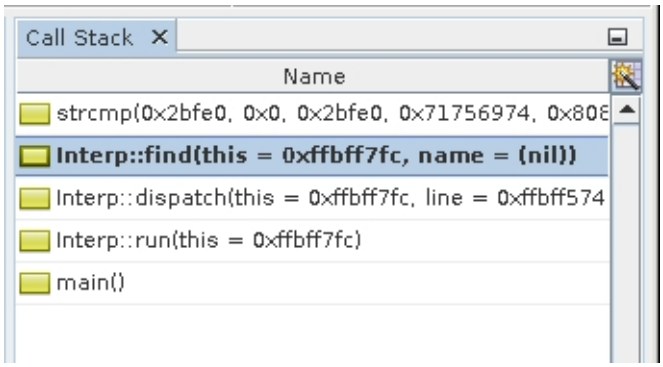


Note the following:

- In the Debugger Console window, you see a message like the following example:

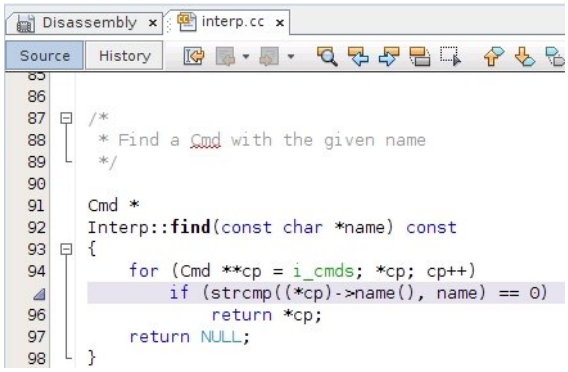
```
program terminated by signal SEGV (no mapping at fault address)
0xf8cfb790: strcmp+0x0170:      ld       [%ol], %gl
Current function is Interp::find
```

- Even though the `SEGV` happened in the `strcmp()` function, `dbx` automatically shows the first frame with a function that has debugging information. See how the stack trace in the Call Stack window has a border around the icon for the current frame.



Note that the Call Stack window shows the parameter names and values. In this example, the second parameter passed to `strcmp()` is 0x0 and that the value of `name` is NULL.

- In the Editor window, the lavender stripe and a triangle on line 95 instead of a green stripe and arrow signify the location of the call to `strcmp()` rather than the actual location of the error.

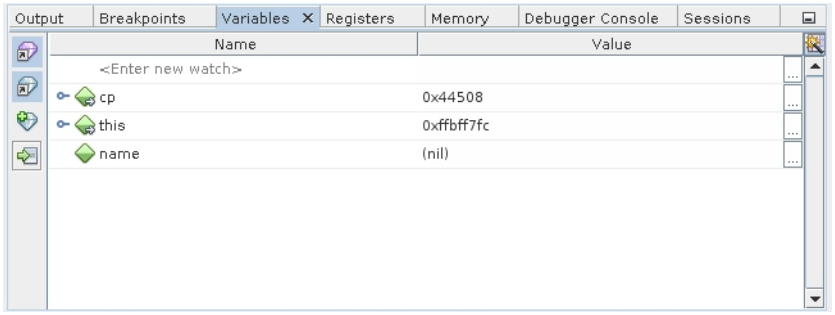


If you do not see parameter values, check that the `dbxenv` variable `stack_verbose` is set to `on` in your `.dbxrc` file. You can also set verbose mode in the Call Stack window by right-clicking in the window and selecting the Verbose option. For more information about `dbxenv` variables and your `.dbxrc`, see Chapter 3, "Customizing dbx" in *Oracle Developer Studio 12.6: Debugging a Program with dbx*.

Functions usually fail when they are passed bad values as parameters. To check the values passed to `strcmp()`:

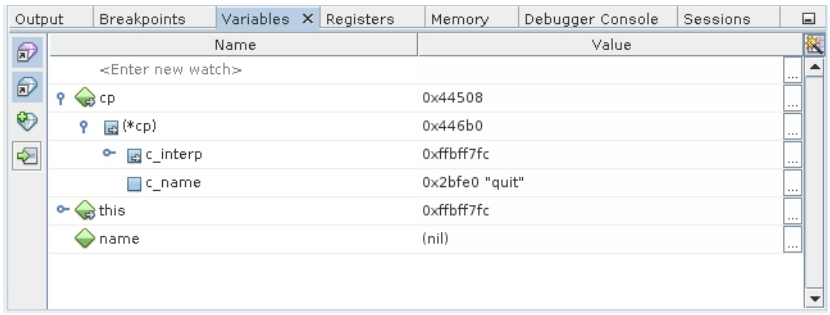
- The Variables window displays all local variables automatically. Check the values of the parameters in the Variables window.

  1. Click the Variables tab.



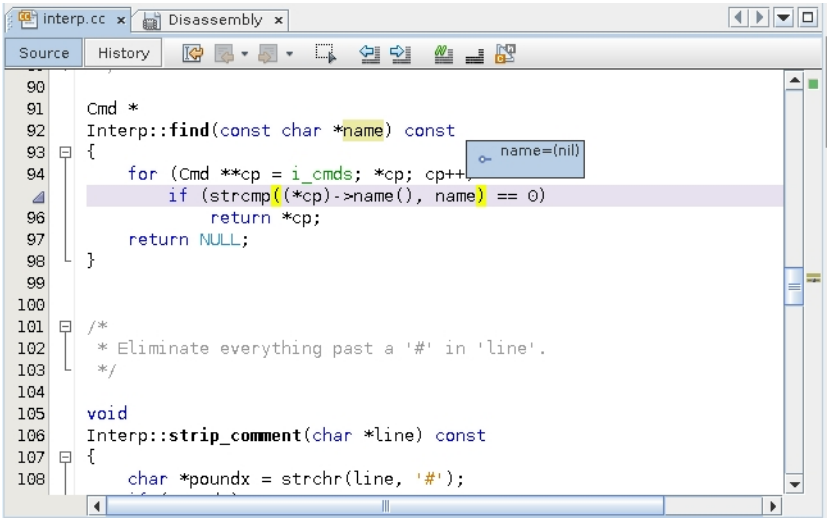
  Note that the value of `name` is NULL. That value is quite likely to be the cause of the SEGV, but check the value of the other parameter, `(*cp)->name()`.

  2. In the Variables window, expand the `cp` node and then expand the `(cp*)` node. The `name` in question is "quit", which is valid.



If expanding the `*cp` node does not show additional variables, check that the `dbx` environment variable `output_inherited_members` in your `.dbxrc` file is set to on. You can also turn on the display of inherited members by right-clicking in the window and selecting the Inherited Members check box to add a check mark.

- Use Balloon Evaluation to confirm the value of a parameter. Click into the Editor window, then hover the cursor over the `name` variable being passed to `strcmp()`. A tip is displayed showing the value of `name` as NULL.

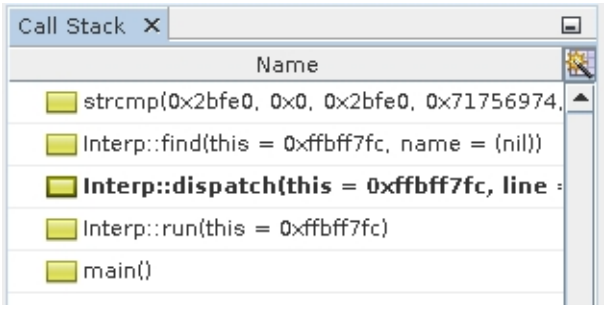


Using balloon evaluation, you can also place the cursor over an expression like `(*cp)->name()`. However, balloon evaluation of expressions with function calls is disabled because:

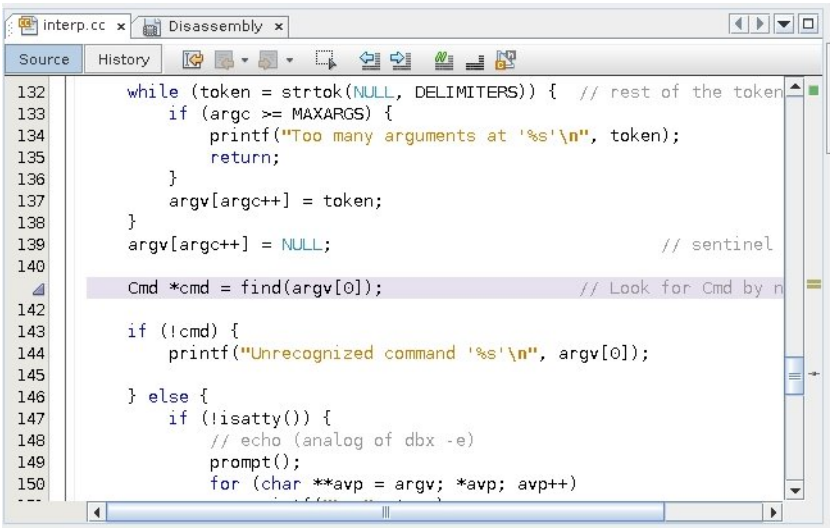
- You are debugging a core file.
- Function calls might have side effects that could occur as a result of casual hovering in the Editor window.

Because the value of `name` should not be NULL, you need to discover which code passed this bad value to `Interp::find()`. To find out:

1. Move up the call stack by choosing Debug → Stack → Make Caller Current or click the Make Caller Current button (Alt - Page Down)  on the toolbar.



2. In the Call Stack window, double-click the frame corresponding to `Interp::dispatch()`.
   The Editor window now highlights the corresponding code:



   This code is unfamiliar and does not provide any clues other than that the value of `argv[0]` is NULL.

Debugging this problem might be easier by dynamically using breakpoints and stepping.

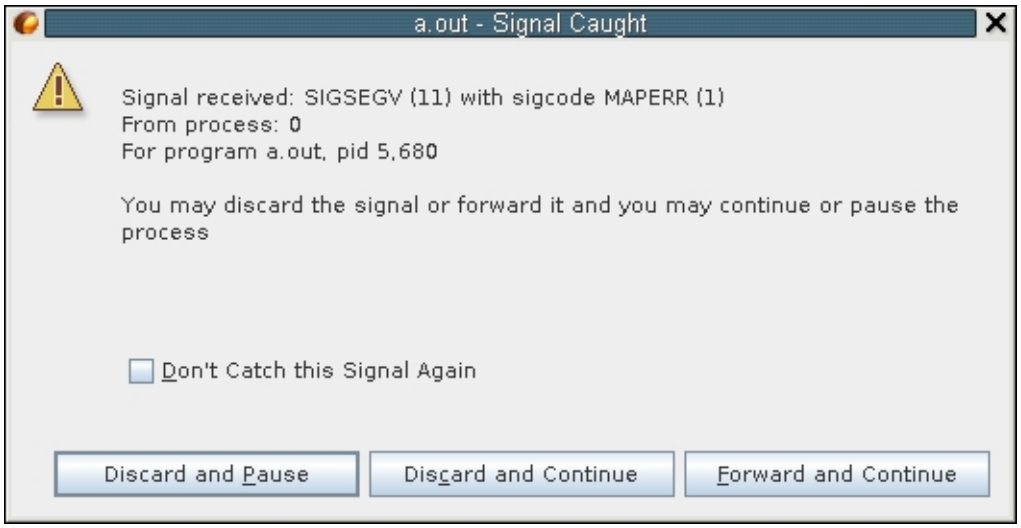
Using Breakpoints and Stepping

Breakpoints enable you to stop a program before the manifestation of a bug and step through the code in the hope of discovering what went wrong.

If you have not already done so, undock the Output window.

You ran the program from the command line earlier. Reproduce the bug by running the program in `dbxtool`.

1. Click the Restart button  on the toolbar or type `run` in the Debugger Console window.
2. Press Return in the Debugger Console window.
   An alert box provides information about the SEGV.



3. In the alert box, click Discard and Pause.

   The Editor window once again highlights the call to `strcmp()` in `Interp::find()`.

4. Click the Make Caller Current button in the toolbar to go to the unfamiliar code you saw earlier in `Interp::dispatch()`.

5. In the next section, you will set a breakpoint a bit before the call to `find()` so you can step through the code to learn why things went wrong.

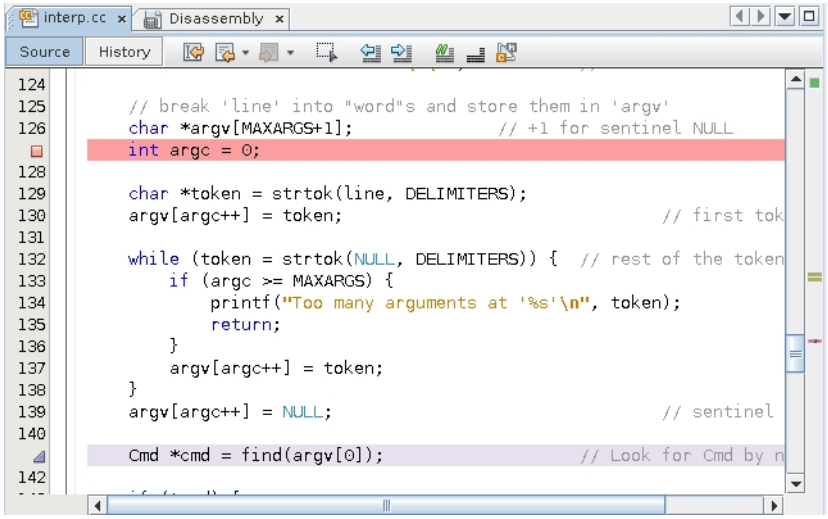
Setting Breakpoints

You can set a breakpoint in several ways, such as a line breakpoint or a function breakpoint. The following list explains the several ways to create a breakpoint.

Note - If the line numbers are not showing, enable line numbers in the editor by right-clicking in the left margin and selecting the Show Line Numbers option.

- **Setting a Line Breakpoint**

  Toggle a line breakpoint by clicking in the left margin next to line 127.

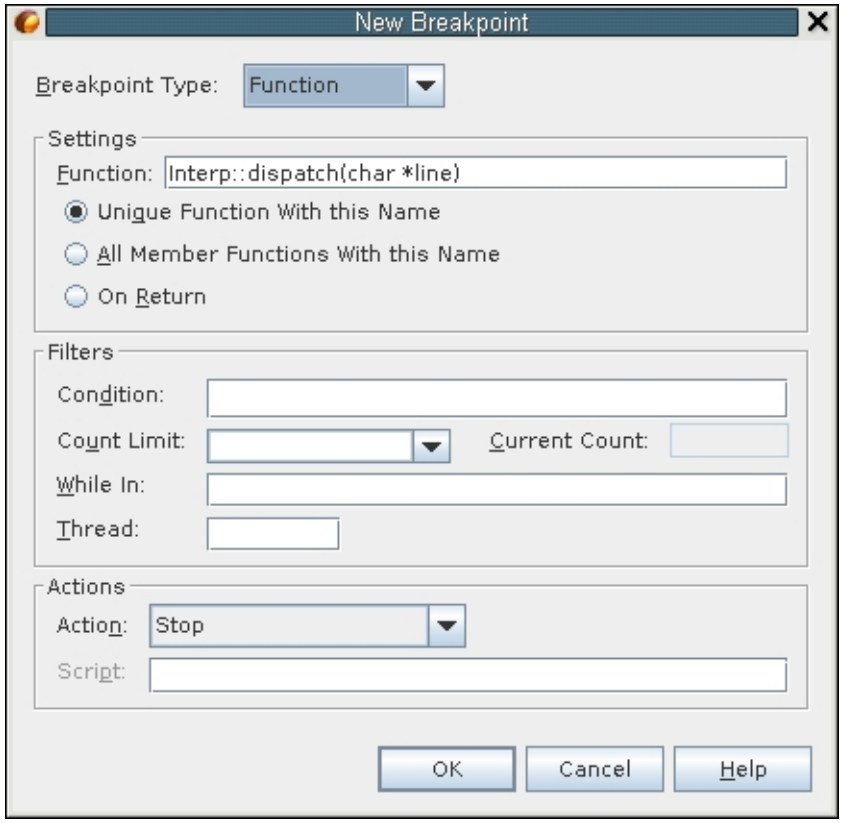


- **Setting a Function breakpoint**

  Set a function breakpoint.

  1. Select `Interp::dispatch` in the Editor window.
  2. Choose Debug → New Breakpoint or right-click and choose New Breakpoint.

     The New Breakpoint dialog box appears.



Notice that the Function field is seeded with the selected function name.

3. Click OK.

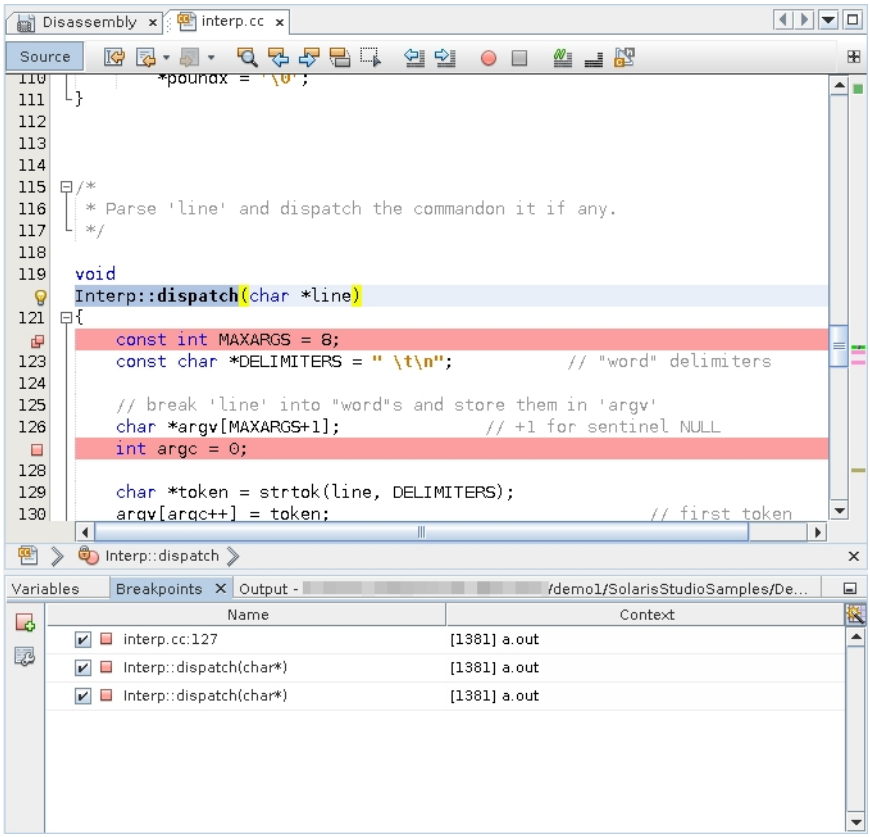
- **Setting a Breakpoint from the Command Line**

  The easiest method to set a function breakpoint is from the `dbx` command line. Type the `stop in` command in the Debugger Console window:

```
(dbx) stop in dispatch
(4) stop in Interp::dispatch(char*)
(dbx)
```

  Notice that you did not have to type `Interp::dispatch`. Just the function name sufficed.

Your breakpoints window and Editor probably look like the following:



To avoid clutter in the Editor, use the Breakpoints window.

1. Click the Breakpoints tab (or maximize it if you minimized it earlier).
2. Select the line breakpoint and one of the function breakpoints, right-click, and choose Delete.

For more information about breakpoints, see Chapter 6, “Setting Breakpoints and Traces” in *Oracle Developer Studio 12.6: Debugging a Program with dbx*.

Advantages of Function Breakpoints

Setting a line breakpoint by toggling in the editor might be intuitive. However, many `dbx` users prefer function breakpoints for the following reasons:

- Typing `si dispatch` in the Debugger Console window means you do not have to open a file in the editor and scroll to a line just to place a breakpoint.
- Because you can create function breakpoints by selecting any text in the editor, you can set a breakpoint on a function from its call site instead of opening a file.

Tip - `si` is an alias for `stop in`. Most `dbx` users define many aliases and put them in the `dbx` configuration file `~/.dbxrc`. Some common examples are:

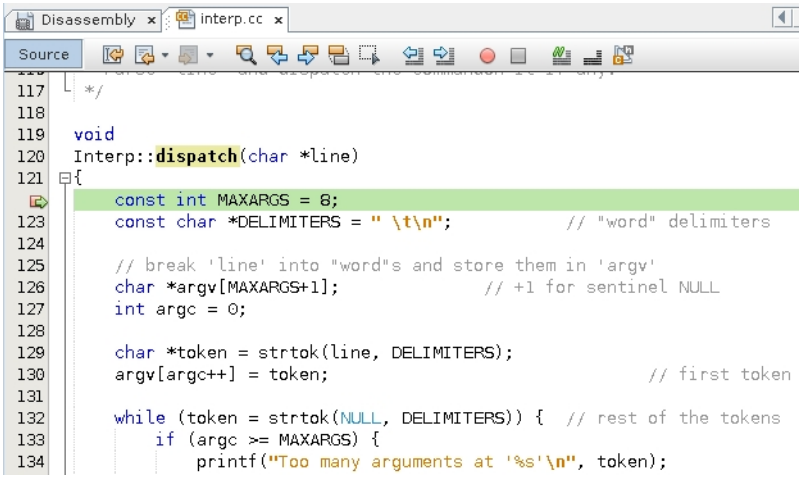
```
alias si stop in
alias sa stop at
alias s step
alias n next
alias r run
```

For more information about customizing your `.dbxrc` file and dbxenv variables, see "Setting dbxenv Variables" in *Oracle Developer Studio 12.6: Debugging a Program with dbx*.

- The name of a function breakpoint is descriptive in the Breakpoints window. The name of a line breakpoint is not descriptive, although you can find what is at line 127 by right-clicking the line breakpoint in the Breakpoints window and choosing Go To Source, or by double-clicking the breakpoint.
- Function breakpoints persist better. Because `dbxtool` persists breakpoints, line breakpoints might easily become skewed if you edit code or do a source code control merge. Function names are less sensitive to edits.

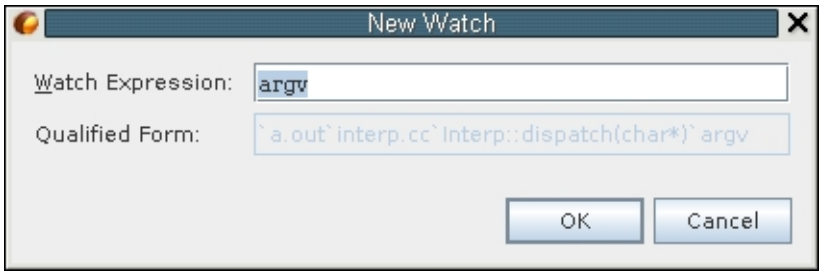
Using Watches and Stepping

Now that you have a single breakpoint at `Interp::dispatch()`, if you click Restart  again and press Return in the Debugger Console window, the program stops at the first line of the `dispatch()` function that contains executable code.

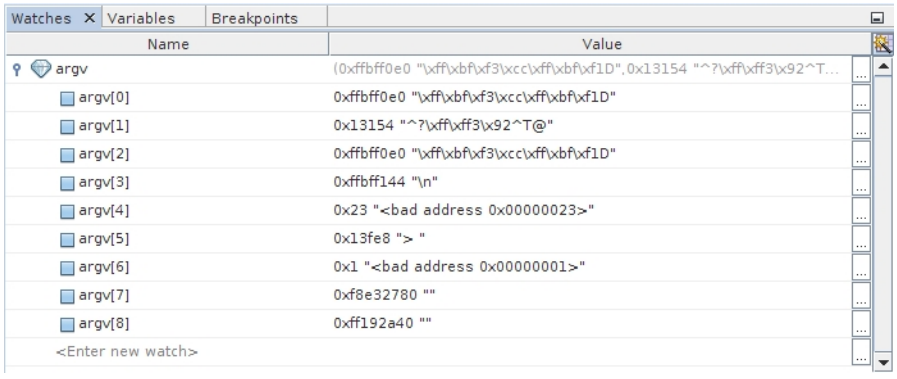


Because you have identified the problem of the `argv[0]` being passed to `find()` use a watch on `argv`:

1. Select an instance of `argv` in the Editor window.
2. Right-click and choose New Watch. The New Watch dialog box appears seeded with the selected text:



3. Click OK.
4. To open the Watches window, choose Window → Watches (Alt + Shift 2).
5. In the Watches window, expand `argv`.

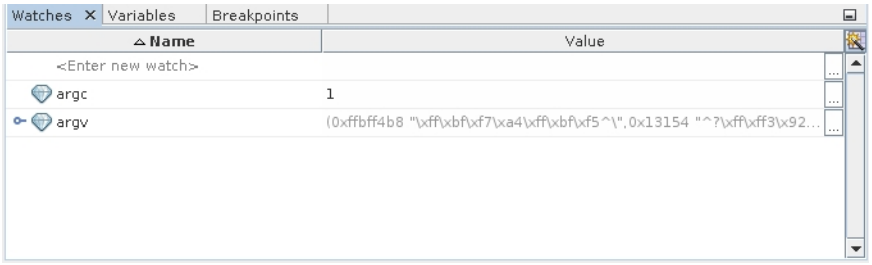


Note that `argv` is uninitialized and because it is a local variable, `argv` might "inherit" random values left on the stack from previous calls. Could this be the cause of problems?

Note - Watch variables can be viewed in the Variables window as well as the Watches window.

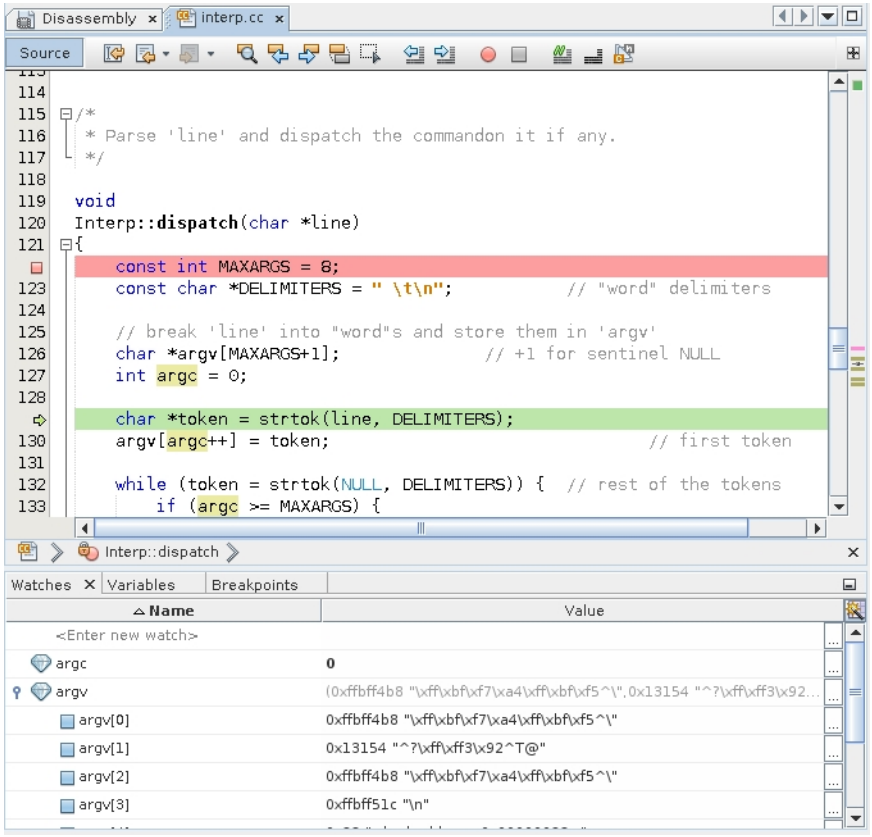
6. Click Step Over (F8)  twice until the green PC arrow points to `int argc = 0;`.
7. Because `argc` is going to be an index into `argv`, create a watch for `argc` also. Note that `argc` is also currently uninitialized and might contain unwanted values.

   Because you created the watch for `argc` after the watch for `argv`, it appears second in the Watches window.
8. To alphabetize the watch names, click the Name column header to sort the column. Note the sort triangle in the following illustration.:



9. Click Step Over (F8)  .

   `argc` now shows its initialized value of 0 and is displayed in bold to signify that the value just changed.



The application is going to call `strtok()`.

10. Click Step Over to step over the function, and observe, for example, by using balloon evaluation, that `token` is NULL.

    The `strtok()` function helps divide a string, for example, into tokens delimited by one of the `DELIMITERS`. For more information, see the `strtok`(3) man page.

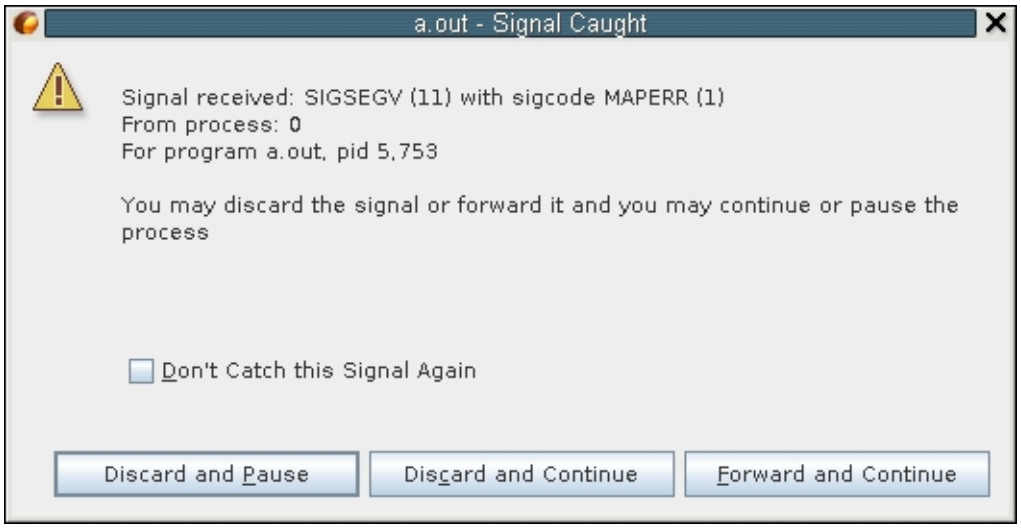
11. Click Step Over again to assign the token to `argv`. Then there is a call to `strtok()` in a loop.

    As you step over, you do not enter the loop (there are no more tokens) and instead a NULL is assigned.

12. Step over that assignment too, to reach the threshold of the call to find where the sample program crashed.

13. To double check that the program crashes at this point, step over the call to `find()`.

    The Signal Caught alert box is displayed again.



14. Click Discard and Pause as before.

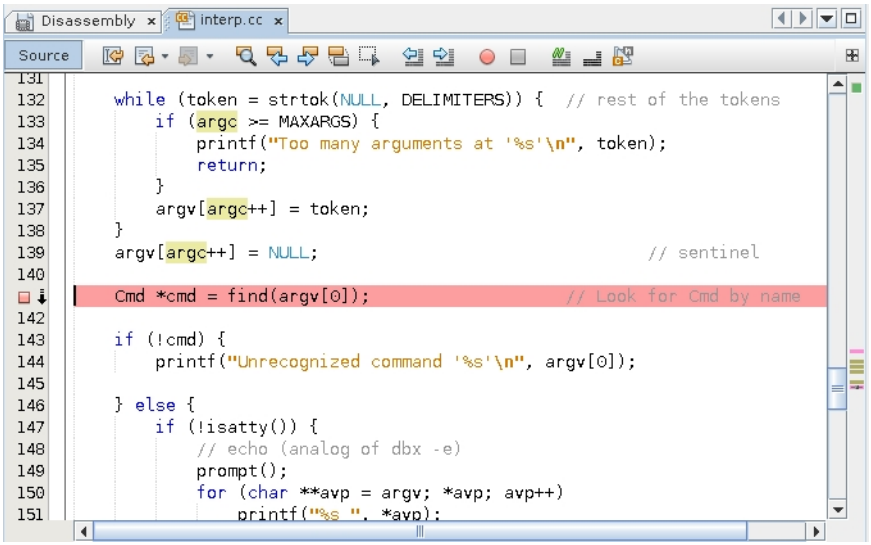
    The first call to `find()` after stopping in `Interp::dispatch()` is indeed where things go wrong. You can quickly get back to where you originally called `find()`.

    a. Click Make Caller Current  .

    b. Toggle a line breakpoint at the call site of `find()`.

    c. Open the Breakpoints window and disable the `Interp::dispatch()` function breakpoint.

    `dbxtool` should look like the following illustration:



    d. The downward arrow indicates that two breakpoints are set on line 141 and that one of them is disabled.

15. Click Restart  and press Return in the Debugger Console window.

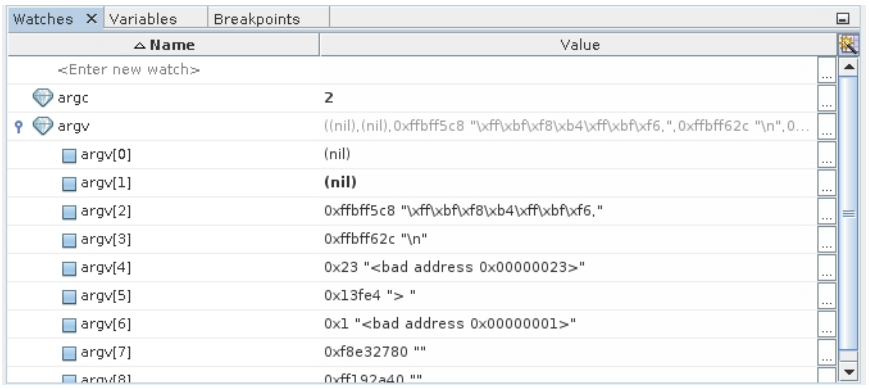
The program returns in front of the call to `find()`. Note that the Restart button evokes restarting. When debugging, you restart much more often than initially starting.)

Tip - If you rebuild your program, for example after discovering and fixing bugs, you need not exit `dbxtool` and restart it. When you click the Restart button, `dbx` detects that the program (or any of its constituents) has been recompiled, and reloads it.

Therefore, consider keeping `dbxtool` on your desktop, perhaps minimized, and ready to use on your debugging problems.

---

16. Where is the bug? Look at the watches again:



Note that `argv[0]` is NULL because the first call to `strtok()` returns NULL because the line was empty and had no tokens.

Fix this bug before proceeding with the remainder of this tutorial, if you like.

If you will be running the program under the debugger, you can patch the code in the debugger, as described in "Using Breakpoint Scripts to Patch Your Code" on page 39.

The developer of the example code should probably have tested for this condition and bypassed the rest of `Interp::dispatch()`.

Discussion

The example illustrates the most common debugging pattern, where you stop the misbehaving program at some point before things have gone wrong and then step through the code comparing the intent of the code with the way the code actually behaves.

The next section describes some advanced techniques for using breakpoints to avoid some of the stepping and watches that you used in this example.

Using Advanced Breakpoint Techniques

This section demonstrates some advanced techniques for using breakpoints:

- Using breakpoint counts

- Using bounded breakpoints
- Picking a useful breakpoint count
- Watchpoints
- Using breakpoint conditions
- Micro replay using pop
- Using fix and continue

This section, and the example program, are inspired by an actual bug discovered in `dbx` using much the same sequence described in this section.

Note - To get the correct output as shown in this section, the example program must still be "buggy". If you fixed the bug, re-download the `OracleDeveloperStudio12.5-Samples` directory from "Example Program" on page 2.

The source code includes a sample input file named `in`, which triggers a bug in the example program. `in` contains the following code:

```
display nonexistent_var # should yield an error
display var
stop in X # will cause one "stopped" message and display
stop in Y # will cause second "stopped" message and display
run
cont
cont
run
cont
cont
```

When you run the program with the input file, the output is as follows:

```
$ a.out < in
> display nonexistent_var
error: Don't know about 'nonexistent_var'
> display var
will display 'var'
> stop in X
> stop in Y
> run
running ...
stopped in X
var = {
      a = '100'
      b = '101
      c = '<error>'
      d = '102
      e = '103'
      f = '104'
      }
> cont
stopped in Y
var = {
      a = '105'
      b = '106'
      c = '<error>'
      d = '107'
      e = '108'
      f = '109'
      }
> cont
```

```
exited
> run
running ...
stopped in X
var = {
      a = '110'
      b = '111'

      c = '<error>'
      d = '112'
      e = '113'
      f = '114'
      }
> cont
stopped in Y
var = {
      a = '115'
      b = '116'

      c = '<error>'
      d = '117'
      e = '118'
      f = '119'
      }
> cont
exited
> quit
Goodby
```

This output might seem voluminous but the point of this example is to illustrate techniques to be used with long running, complex programs where stepping through code or tracing just are not practical.

Notice that when showing the value of field `c`, you get a value of `<error>`. Such a situation might occur if the field contains a bad address.

The Problem

Notice that when you ran the program a second time, you received additional error messages that you did not get on the first run:

```
error: cannot get value of 'var.c'
```

The `error()` function uses a variable, `err_silent`, to silence error messages in certain circumstances. For example, in the case of the display command, instead of displaying an error message, problems are displayed as `c = '<error>'`.

Step 1: Repeatability

The first step is to set up a debug target and configure the target so the bug can easily be repeated by clicking Restart  .

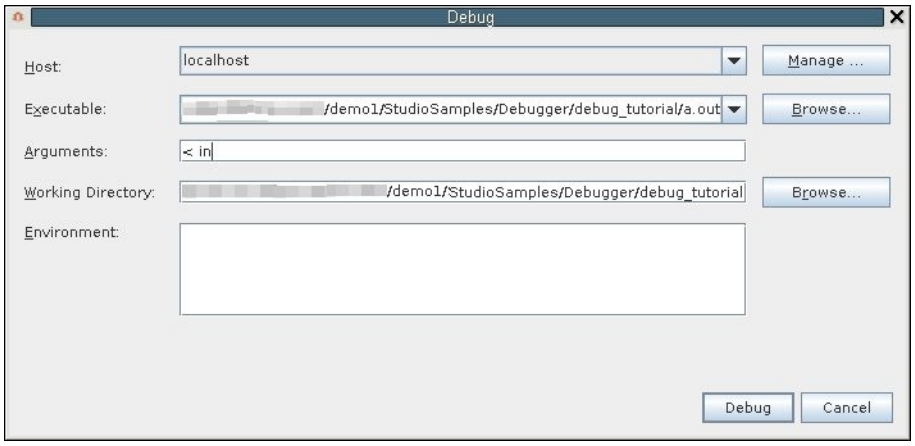
Start debugging the program as follows:

1. If you have not yet compiled the example program, do so by following the instructions in “Example Program” on page 2.

2. Choose Debug → Debug Executable.
3. In the Debug Executable dialog box, browse for or type the path to the executable.
4. In the Arguments field, type:

   ```
   < in
   ```

   The directory portion of the executable path is displayed in the Working Directory field.
5. Click Debug.



In a real world situation, you might want to populate the Environment field as well.

When debugging a program, `dbxtool` creates a debug target. You can use the same debugging configuration by choosing Debug → Debug Recent and then choosing the desired executable.

You can set many of these properties from the `dbx` command line. They will be stored in the debug target configuration.

The following techniques help sustain easy repeatability. As you add breakpoints, you can quickly go to a location of interest by clicking Restart without having to click Continue on various intermediate breakpoints.

Step 2: First Breakpoint

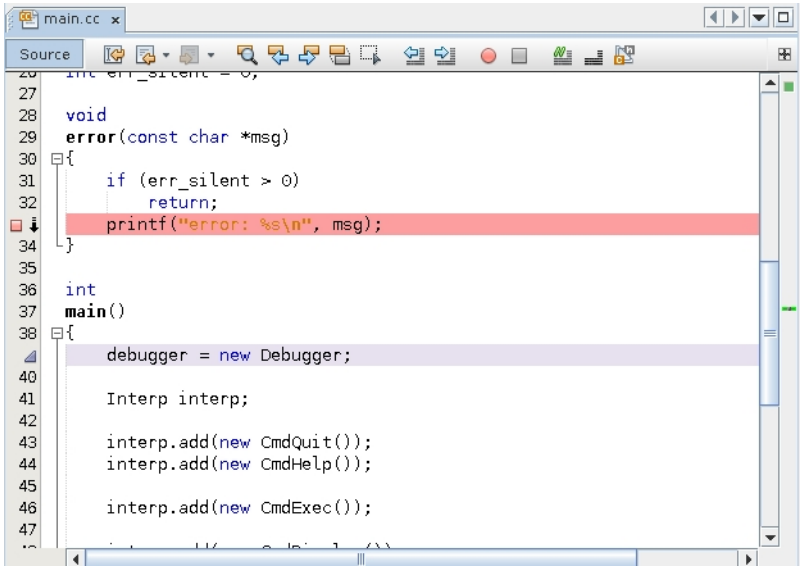
Put the first breakpoint inside the `error()` function in the case where it prints an error message. This breakpoint will be a line breakpoint on line 33.

In a larger program, you can easily change the current function in the Editor window by typing the following, for example, in the Debugger Console window:

```
(dbx) func error
```

The lavender stripe indicates the match found by the `func` command.

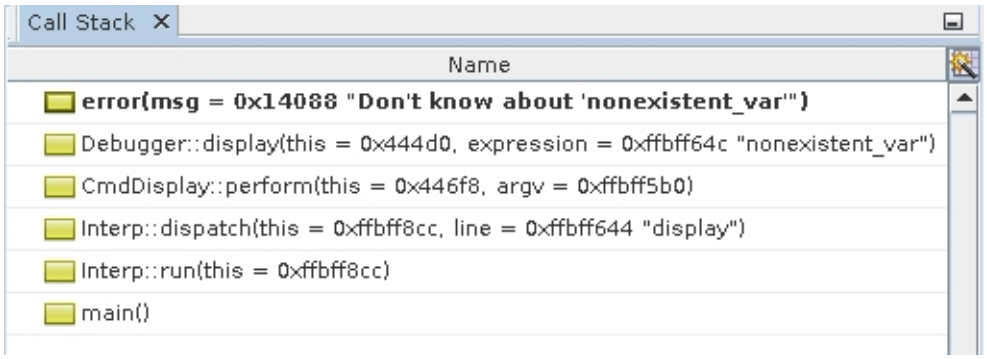
1. Create the line breakpoint by clicking in the left margin of the Editor window on top of the number 33.



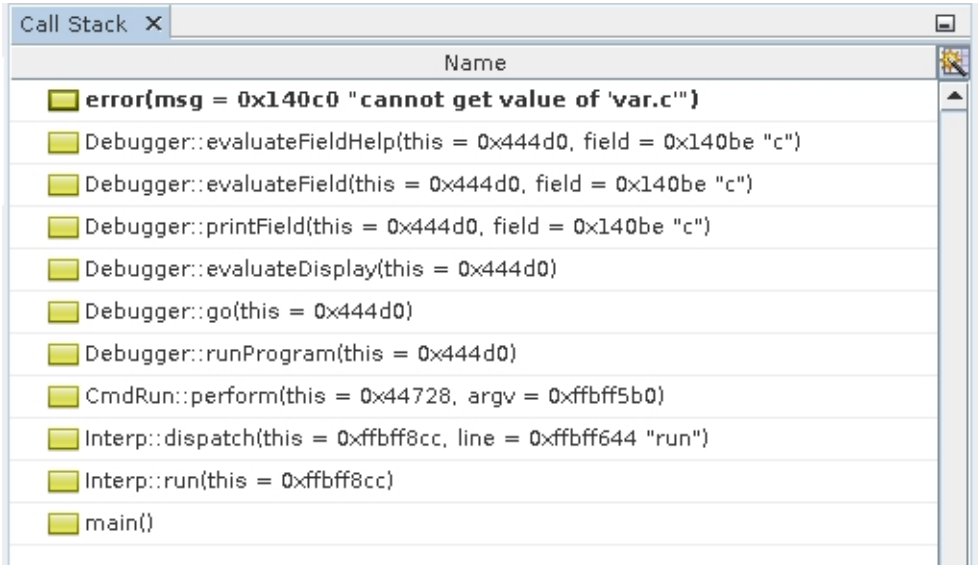
2. Click Restart  to run the program and upon hitting the breakpoint, the stack trace shows the error message that is generated due to the simulated command in the `in` file:

```
> display var # should yield an error
```

The call to `error()` is expected behavior.



3. Click Continue  to continue the process and hit the breakpoint again.
An unexpected error message appears.



Step 3: Breakpoint Counts

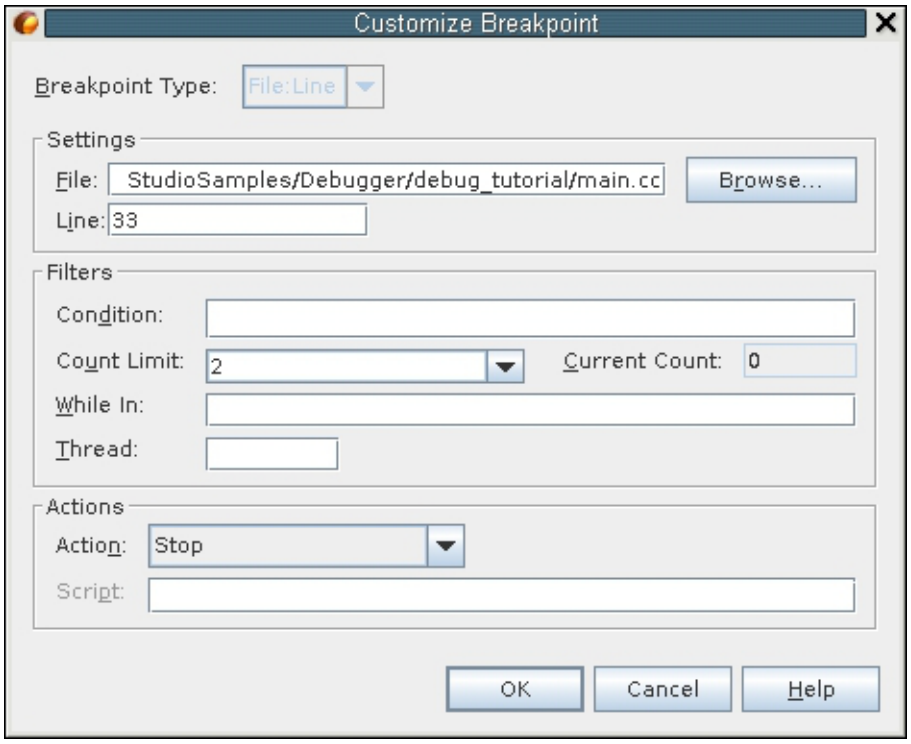
It would be better to arrive at this location repeatedly on each run without having to click Continue after the first hit of the breakpoint due to the command:

```
> display var # should yield an error
```

You can edit the program or input script and eliminate the first troublesome display command. However, the specific input sequence you are working with might be a key to reproducing this bug so you do not want to alter the input.

Because you are interested in the second time you reach this breakpoint, set its count to 2.

1. In the Breakpoints window, right-click the breakpoint and choose Customize.
2. In the Customize Breakpoint dialog box, type 2 in the Count Limit field.
3. Click OK.



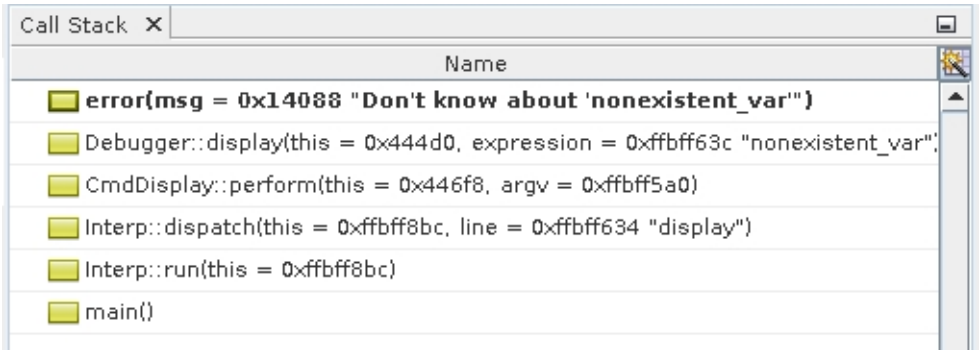
Now you can repeatedly arrive at the location of interest.

In this case,choosing a count of 2 was trivial. However, sometimes a place of interest is called many times. See “Step 7: Determining the Count Value” on page 33to easily choose a good count value. But for now, you will explore another way of stopping in `error()` only in the invocation you are interested in.

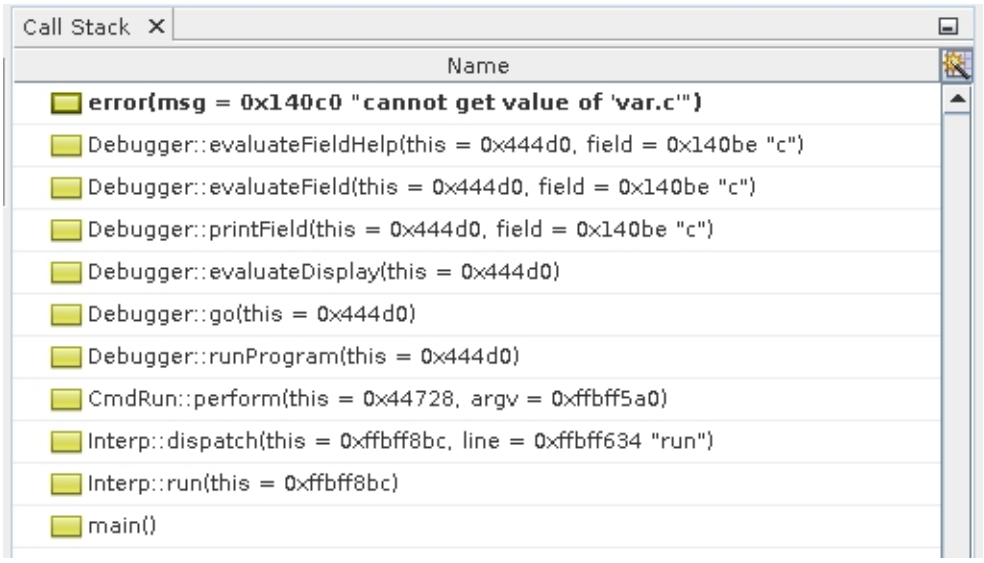
Step 4: Bounded Breakpoints

1. Open the Customize Breakpoint dialog box for the breakpoint inside `error()` and disable breakpoint counts by selecting Always Stop from the drop-down list for the Count Limit.
2. Rerun the program.

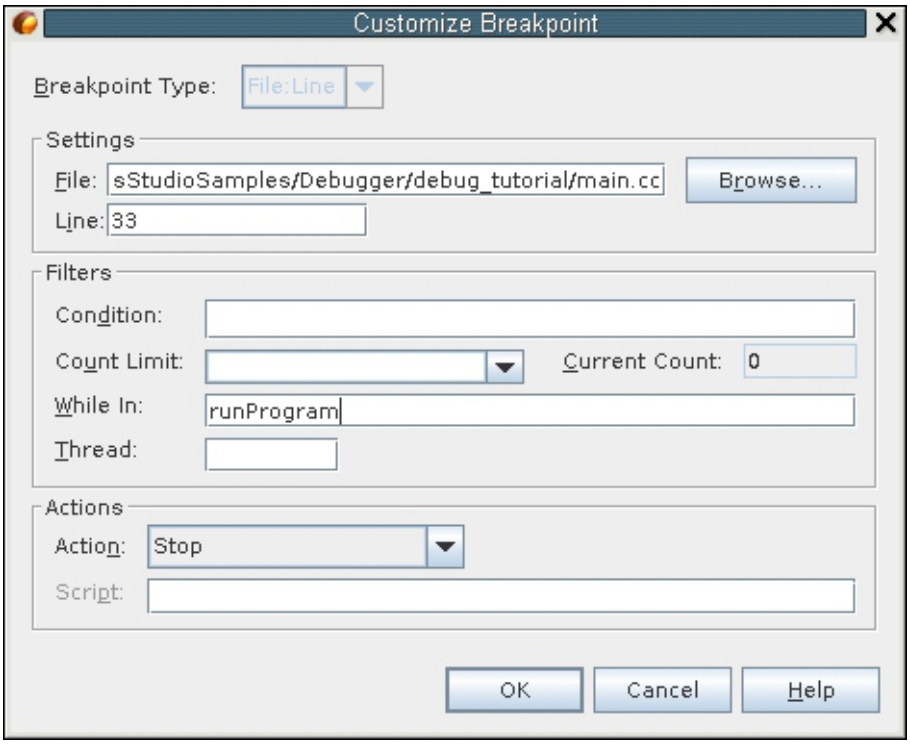
   Pay attention to the stack trace the two times you stop in `error()`. The first time, the stop in `error()` looks like the following screen:



The second time, the stop in `error()` looks like the following screen:



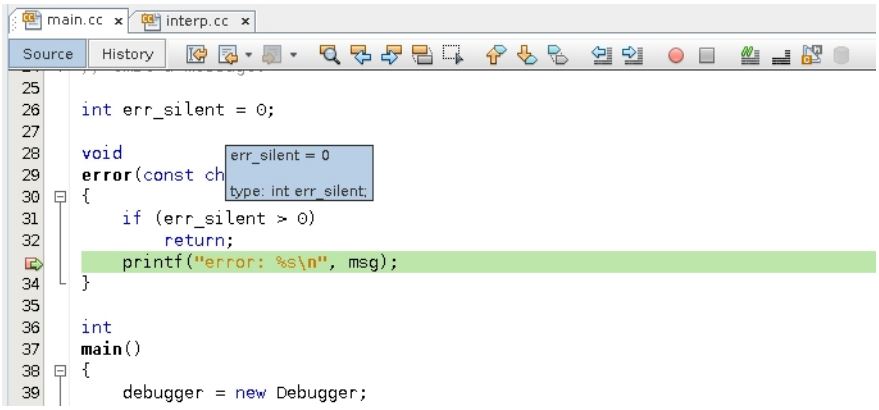
To arrange to stop at this breakpoint when it is called from `runProgram` (frame [7]), open the Customize Breakpoint dialog box again and set the While In field to runProgram.



Step 5: Looking for a Cause

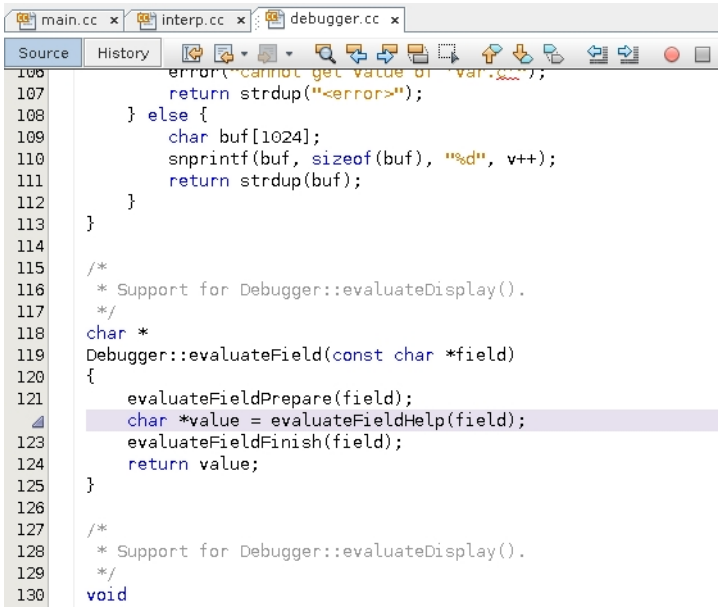
The unwanted error message is issued because `err_silent` is not > 0. Take a look at the value of `err_silent` with balloon evaluation.

1. Put your cursor over `err_silent` in line 31 and wait for its value to be displayed.

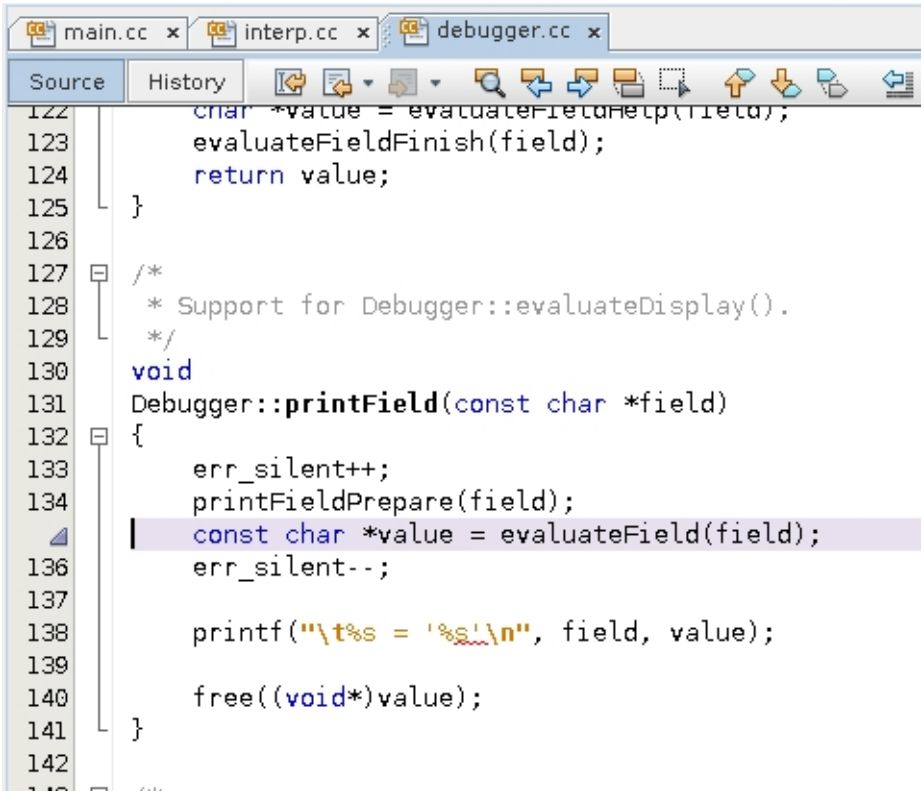


   Follow the stack to see where `err_silent` was set.

2. Click Make Caller Current  twice to `evaluateField()`, which has already called `evaluateFieldPrepare()` simulating a complex function that might be manipulating `err_silent`.



3. Click Make Caller Current again to get to `printField()`, where `err_silent` is being incremented. `printField()` has also already called `printFieldPrepare()`, also simulating a complex function that might be manipulating `err_silent`.



Notice how `err_silent++` and `err_silent--` bracket some code.

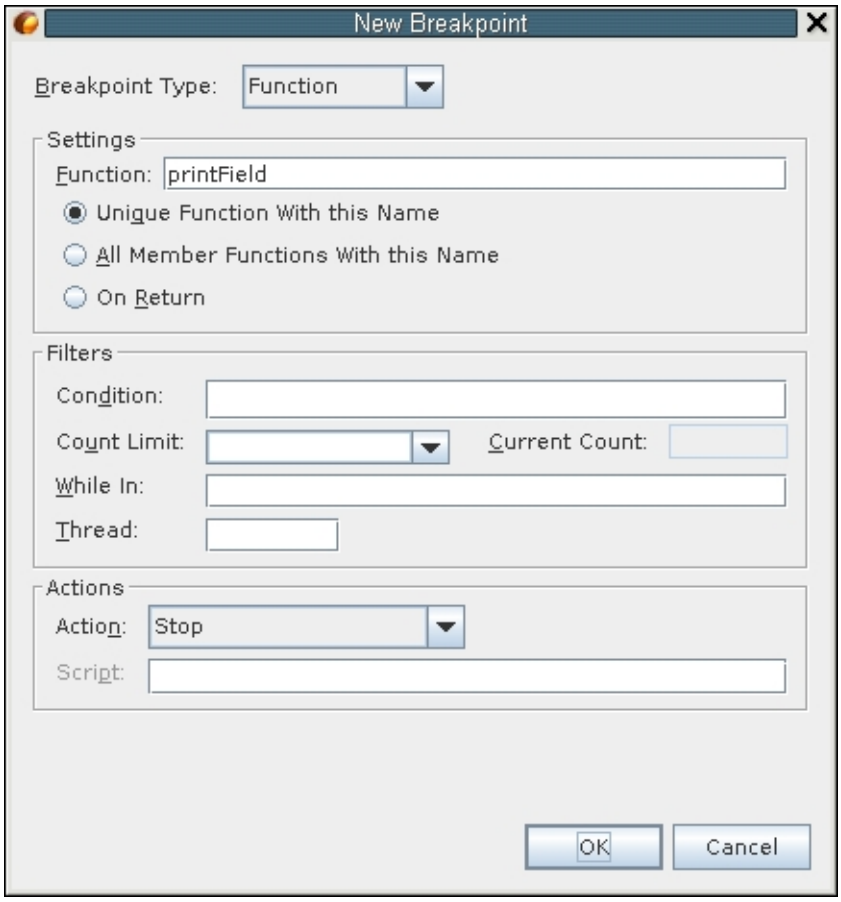
`err_silent` could go wrong in either `printFieldPrepare()` or `evaluateFieldPrepare()`, or it might already be wrong when control gets to `printField()`.

Step 6: More Breakpoint Counts

To find out whether `err_silent` was wrong before or after the call to `printField()`, put a breakpoint in `printField()`.

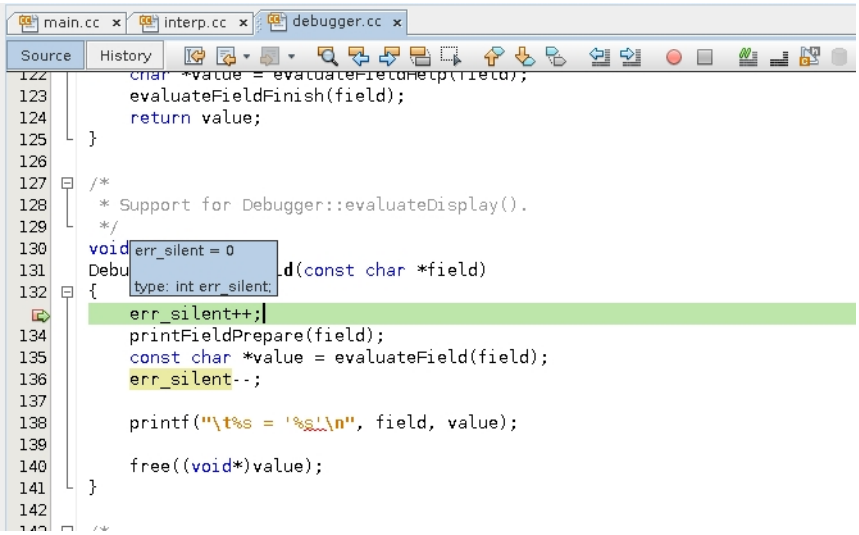
1. Select `printField()`, right-click, and choose New Breakpoint.

   The New breakpoint type is pre-selected and the Function field is pre-populated with `printfield`.
2. Click OK.



3. Click Restart  .

   The first time you hit the breakpoint is during the first run, on the first stop, and on the first field, `var.a`. `err_silent` is 0, which is OK.



4. Click Continue.

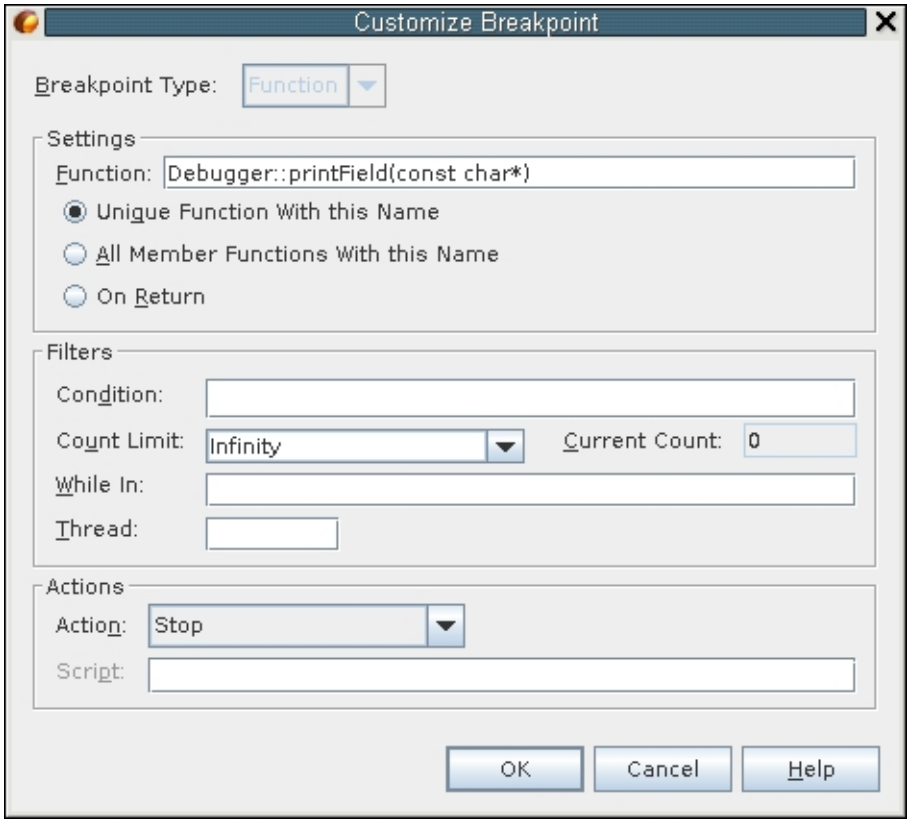
   `err_silent` is still OK.
5. Click Continue again.

   `err_silent` is still OK.

Reaching the particular call to `printField()` that resulted in the unwanted error message might take a while. You need to use a breakpoint count on the `printField` breakpoint. But what shall the count be set to? In this simple example, you could attempt to count the runs and the stops and the fields being displayed, but in practice the process might be more difficult. There is a way to determine the count semi-automatically.

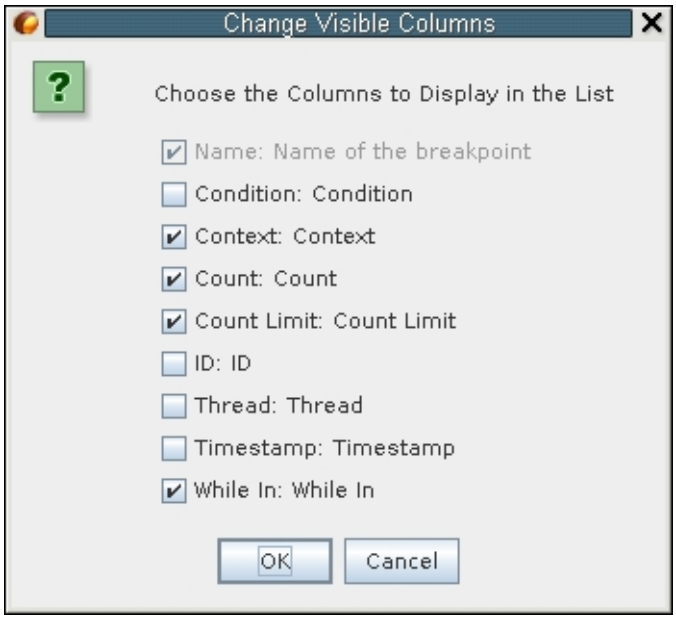
Step 7: Determining the Count Value

1. Open the Customize Breakpoint dialog box for the breakpoint on `printField()` and set the Count Limit field to infinity.

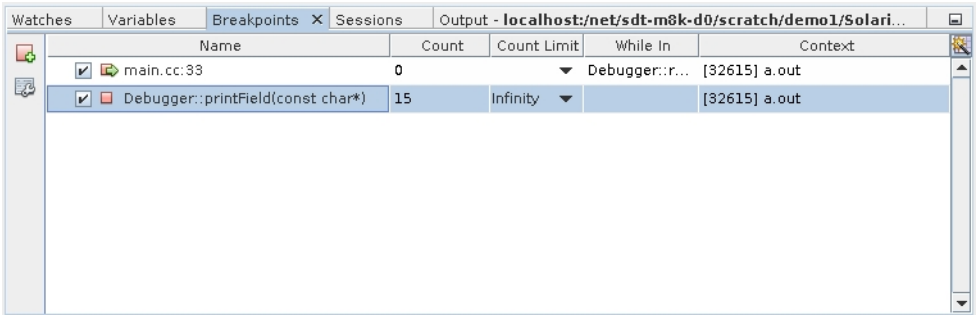


   This setting means that you will never stop at this breakpoint. However, it will still be counting.
2. Set the Breakpoints window to show more properties, such as counts.

   a. Click the Change Visible Columns button  at the top right corner of the Breakpoints window.
   b. Select Count Limit, Count, and While In.
   c. Click OK.



3. Run the program again. You will hit the breakpoint inside `error()`; the one bounded by `runProgram()`.
4. Look at the count for the breakpoint on `printField()`.



The count is 15.

5. In the Customize Breakpoint window again, click the drop-down list in the Count Limit column and select Use current Count value to transfer the current count to the count limit, and click OK.

Now when you run the program, you will stop in `printField()` the last time it is called before the unwanted error message.

Step 8: Narrowing Down the Cause

Use balloon evaluation to inspect `err_silent` again. Now it is -1. The most likely cause is one `err_silent--` too many, or one `err_silent++` too few, being executed before you got to `printField()`.

You can locate this mismatched pair of `err_silents` in a small program like this example by careful code inspection. However, a large program might contain numerous pairings of the following:

```
err_silent++;
```

```
err_silent--;
```

A quicker way to locate the mismatched pair is by using watchpoints.

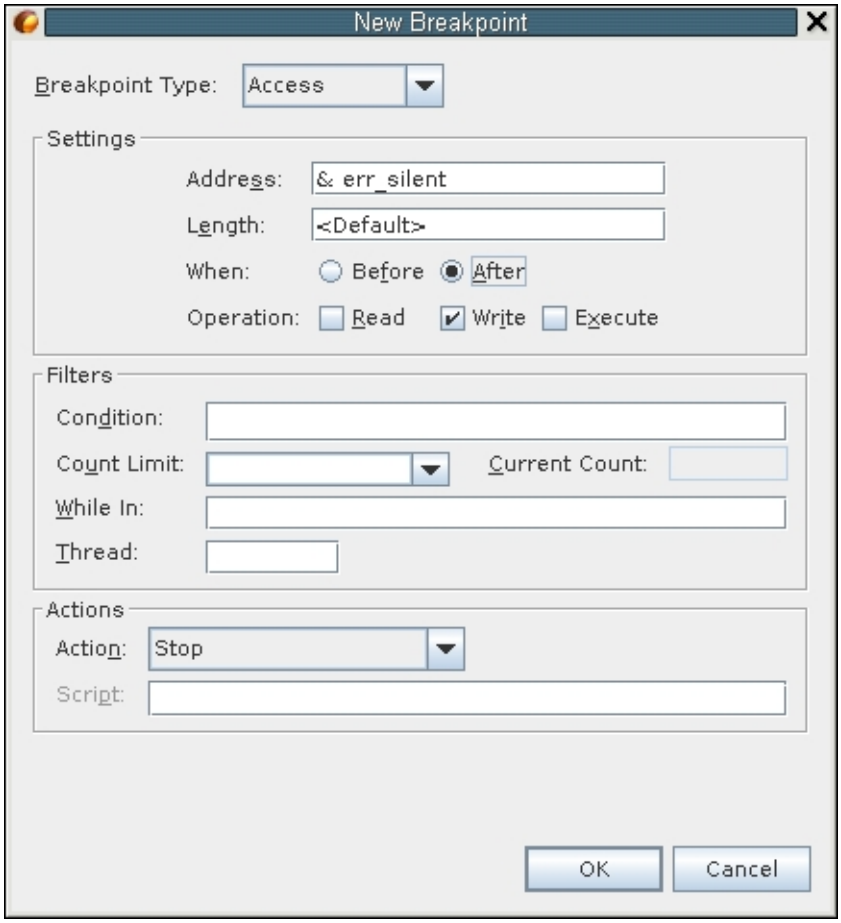
The cause of the error might not be a mismatched set of `err_silent++;` and `err_silent--;` at all, but a rogue pointer overwriting the contents of `err_silent`. Watchpoints would be more effective in catching such a problem.

Step 9: Using Watchpoints

To create a watchpoint on `err_silent`:

1. Select the `err_silent` variable, right-click, and choose New Breakpoint.
2. Set Breakpoint Type to Access.

   Note how the Settings section changes and how the Address field is `& err_silent`.
3. Select After in the When field.
4. Select Write in the Operation field.
5. Click OK.



6. Run the program.

   You stop in `init()`. `err_silent` was incremented to 1 and execution stopped after that.

7. Click Continue.

   You stop in `init()` again.
8. Click Continue again.

   You stop in `init()` again.
9. Click Continue again.

   You stop in `init()` again.
10. Click Continue again.

    Now you stop in `stopIn()`. Things look OK here too, with no -1s.

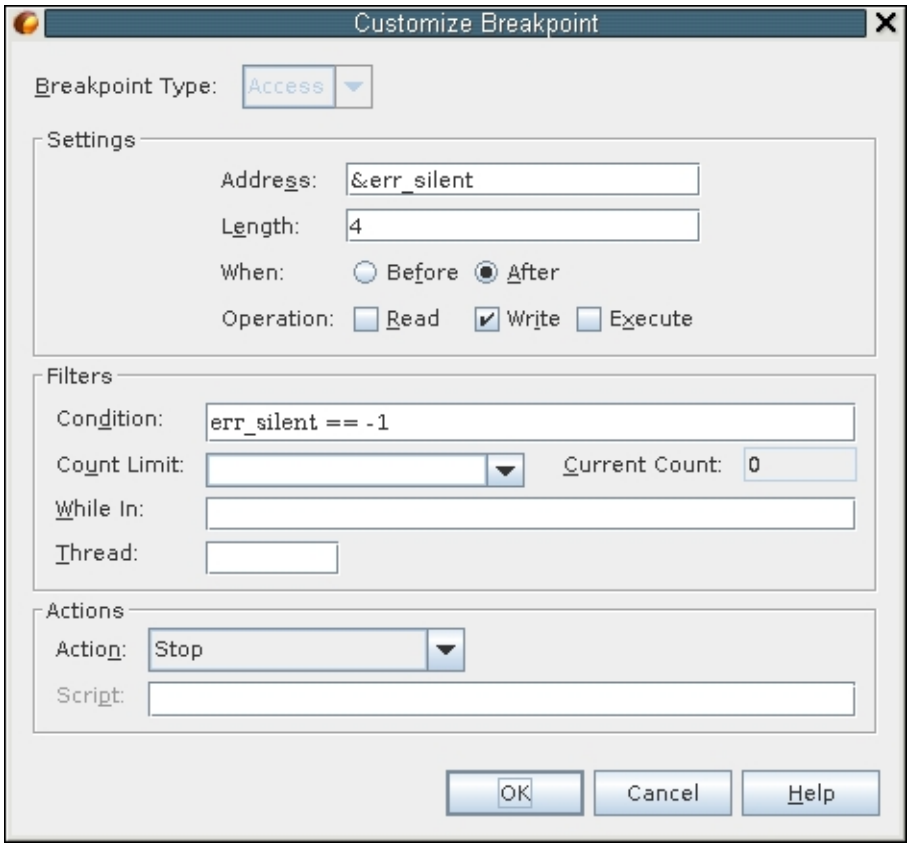
Instead of clicking Continue over and over until `err_silent` is set to -1, you can set a breakpoint condition.

Step 10: Breakpoint Conditions

To add a condition to your watchpoint:

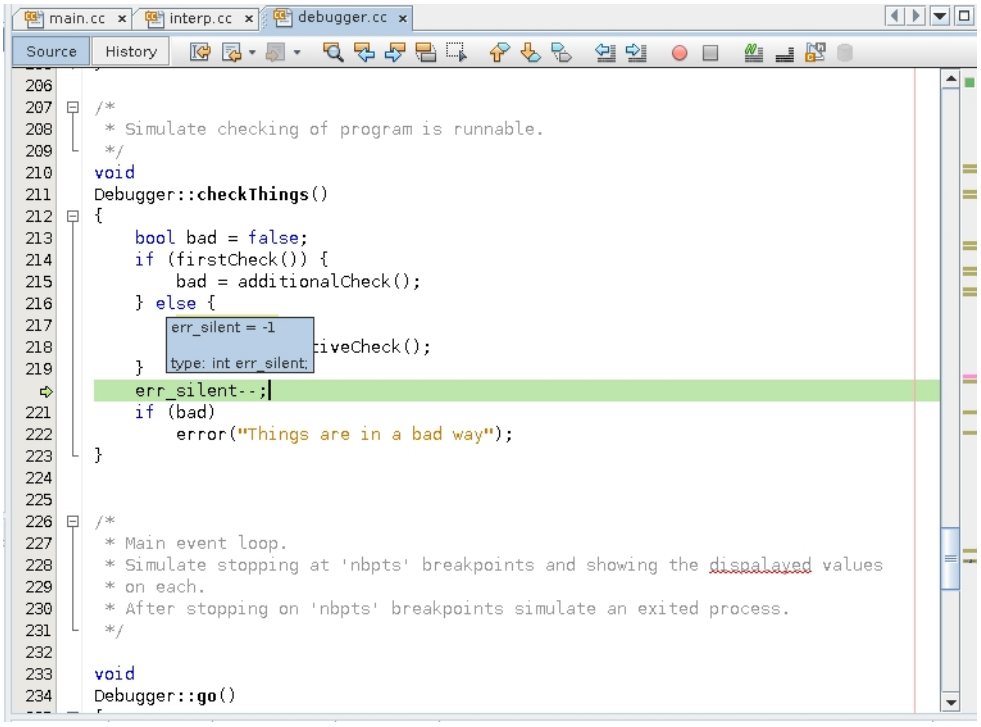
1. In the Breakpoints window, right-click the After Write breakpoint and choose Customize.
2. Verify that After is selected in the When field.

   Selecting After enables you to see what the value of `err_silent` was changed to.
3. Set the Condition field to `err_silent == -1`.
4. Click OK.



5. Run the program again.

You stop in `checkThings()`, which is the first time `err_silent` is set to -1. As you look for the matching `err_silent++` you see what looks like a bug: `err_silent` is incremented only in the `else` portion of the function.



Could this be the bug you've been looking for?

Step 11: Verifying the Diagnosis by Popping the Stack

One way to double-check that you indeed went through the `else` block of the function would be to set a breakpoint on `checkThings()` and run the program. But `checkThings()` might be called many times. You can use breakpoint counts or bounded breakpoints to get to the right invocation of `checkThings()`, but a quicker way to replay what was recently executed is to pop the stack.

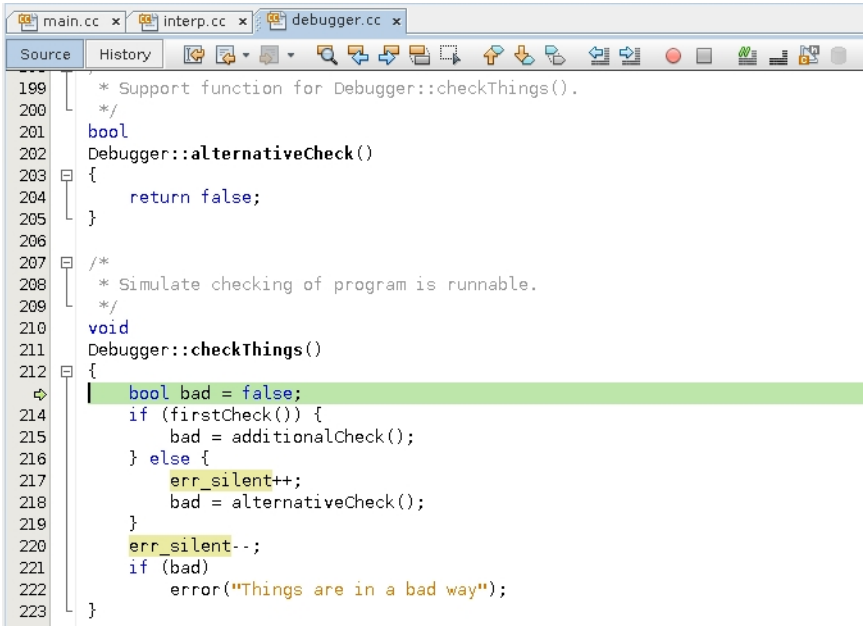
1. Choose Debug → Stack → Pop Topmost Call.

   Notice the Pop Topmost Call does not undo everything. In particular, the value of `err_silent` is already wrong because you are switching from data debugging to control flow debugging.

   The process state reverts to the beginning of the line containing the call to `checkThings()`.

2. Click Step Into  . and observe as `checkThings()` is called again.

   As you step through `checkThings()`, you can verify that the process executes the `if` block where `err_silent` is not incremented and then is decremented to -1.

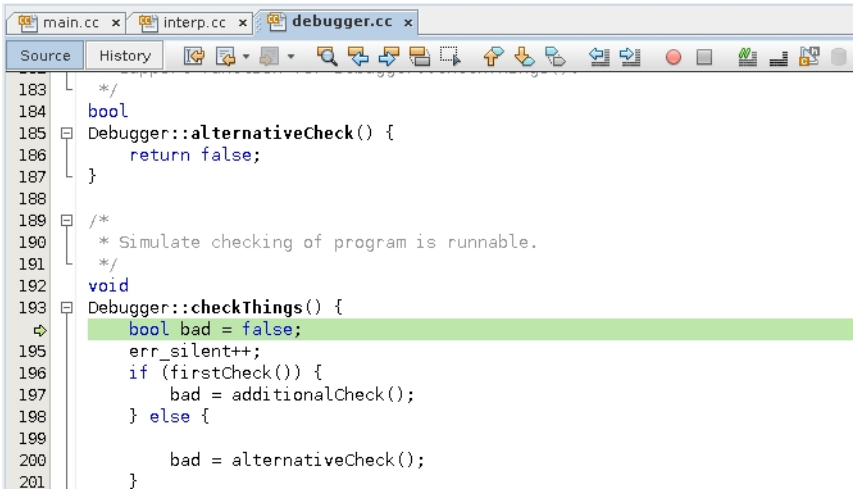


Although you appear to have found the programming error, you might want to triple check it.

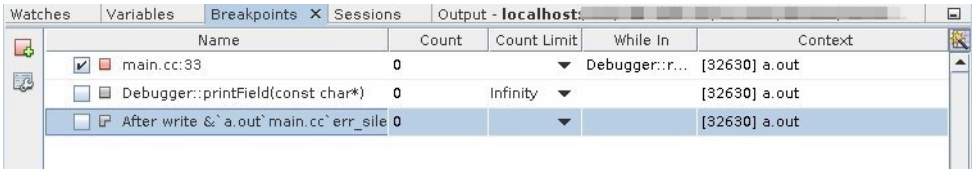
Step 12: Using Fix to Further Verify The Diagnosis

Fix the code in place and verify that the bug has indeed gone away.

1. Fix the code by putting the `err_silent++` above the `if` statement.

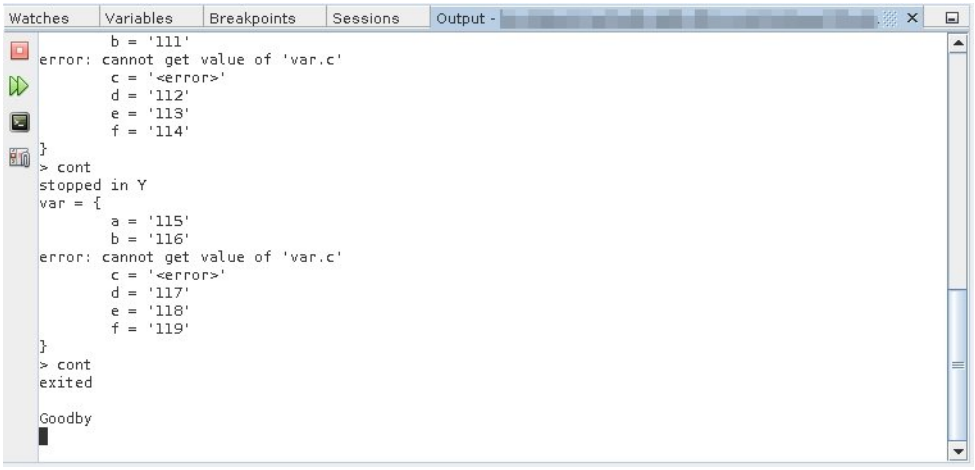


2. Choose Debug > Apply Code Changes or press the Apply Code Changes button  .
3. Disable the `printField` breakpoint and the watchpoint but leave the breakpoint in `error()` enabled.



4. Run the program again.

Note that the program completes without hitting the breakpoint in `error()` and its output is as expected.



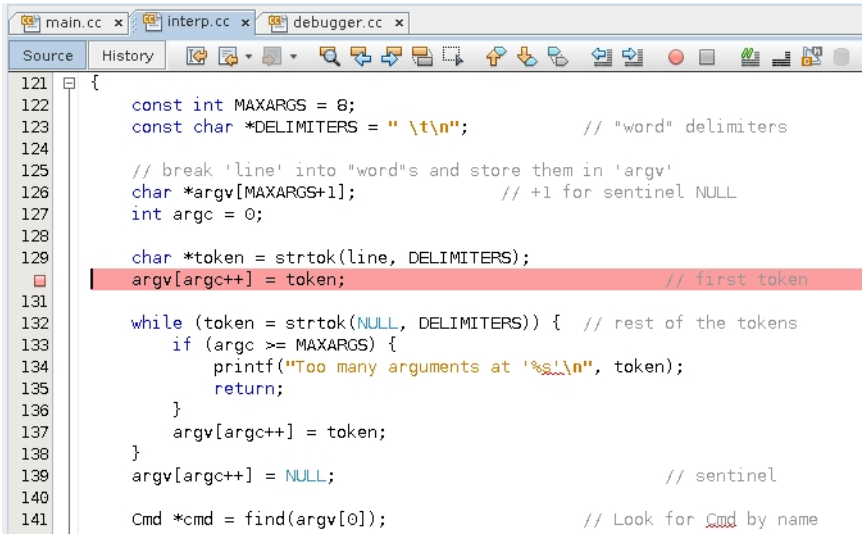
Discussion

This example illustrates the same pattern as discussed at the end of “Using Breakpoints and Stepping” on page 13, that is, you stop the misbehaving program at some point before things have gone wrong and then steps through the code comparing the intent of the code with the way the code actually behaves. The main difference is that finding the point before things have gone wrong is a bit more involved.

Using Breakpoint Scripts to Patch Your Code

In “Using Breakpoints and Stepping” on page 13, you discovered a bug where an empty line yields a NULL first token and causes a SEGV. You can use a workaround to avoid the error.

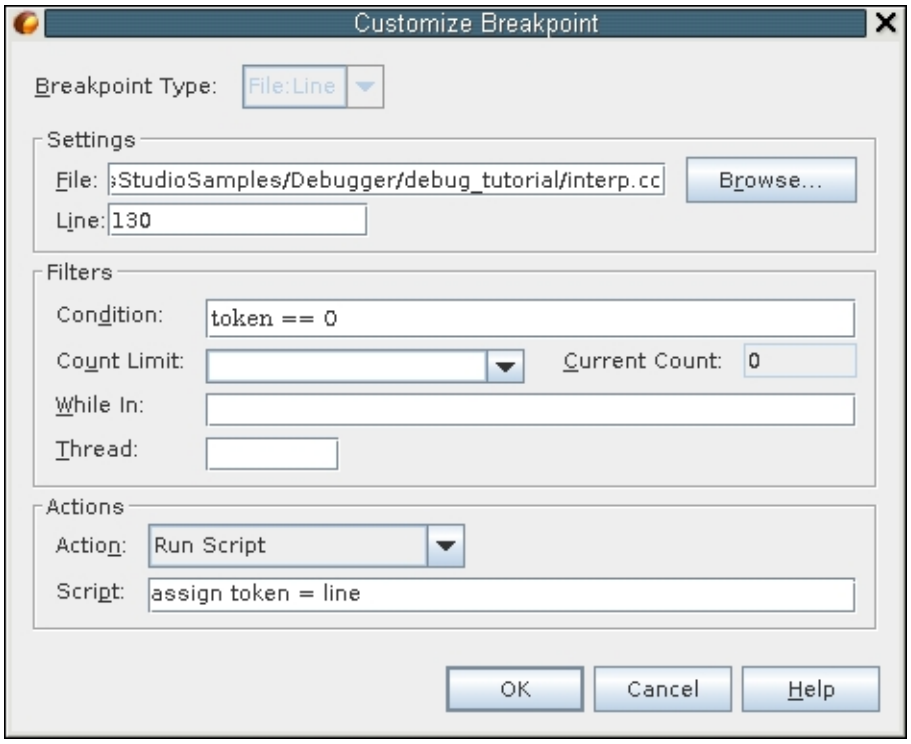
1. Delete all of the breakpoints you created previously. You can do this quickly by right-clicking in the Breakpoints window and selecting Delete All.
2. Delete the `<in` argument in the Debug Executable dialog box.
3. Toggle a line breakpoint at line 130 in `interp.cc`.



4. In the Breakpoints window, right-click the breakpoint you just created and choose Customize.
5. In the Customize Breakpoint dialog box, type `token == 0` in the Condition field.
6. Select Run Script from the Action drop-down list.
7. In the Script field, type `assign token = line`.

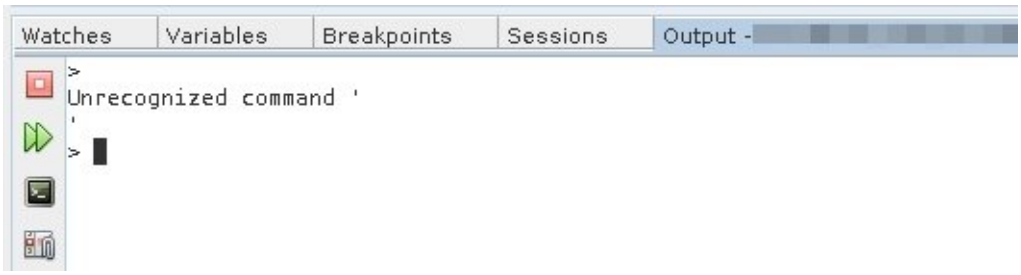
Note - You cannot `assign token = “dummy”` because `dbx` cannot allocate the `dummy` string in the debugged process. On the other hand, `line` is known to be equal to `""`.

The dialog box should look like the following screen.



8. Click OK.

Now if you run the program and enter an empty line, instead of crashing, it will warn you, as shown in the following screen.



This workaround might be clearer if you look at the command that `dbxtool` sent to `dbx`.

```
when at "interp.cc":130 -if token == 0 { assign token = line; }
```

Conclusion

Oracle Developer Studio `dbxtool` enables you to pinpoint the problem area that causes your program to crash, while using a convenient GUI format. `dbxtool` enables you to simply debug your code by creating breakpoints and stepping through your code. `dbxtool` also enables you use advanced breakpoint techniques with features like watchpoints, breakpoint conditions, and pop stacking, to identify bugs in your code and enable you to fix these problems.

Oracle Developer Studio 12.6: dbxtool Tutorial

Part No: E77794

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Référence: E77794

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` ou le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` si vous êtes malentendant.