

# Oracle Solaris Studio 12.2 dbxtool 教程

2010年9月

- 第 2 页中的“简介”
- 第 2 页中的“示例程序”
- 第 3 页中的“配置 dbxtool”
- 第 6 页中的“诊断核心转储”
- 第 13 页中的“使用断点和步进”
- 第 21 页中的“使用高级断点技巧”
- 第 40 页中的“使用断点脚本修补代码”

## 简介

本教程使用一个“有错误”的示例程序来演示如何有效地使用 dbxtool。dbxtool 是 dbx 调试器的独立图形用户界面 (graphical user interface, GUI)。首先从基础功能开始，然后是较为高级的功能。

## 示例程序

本教程模拟了一个简化的、有些虚拟的 dbx 调试器。在已安装的 Oracle Solaris Studio 12.2 软件的 examples/debugger/debug\_tutorial 目录下提供了此 C++ 程序的源代码。

1. 将该目录复制到您自己的专用工作区。例如：

```
cp -r /opt/solstudio12.2/examples/debugger/debug_tutorial ~/debug_tutorial
```

2. 生成程序：

```
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
```

程序由下列模块组成：

cmd.h	cmd.cc	类 Cmd，用于实现调试器命令的基类
interp.h	interp.cc	类 Interp，一个简单的命令解释程序
debugger.h	debugger.cc	类 Debugger，用于模拟调试器的主要语义
cmds.h	cmds.cc	各种调试命令的实现
main.h	main.cc	main() 函数和出错处理。设置 Interp，创建各种命令并将其分配给 Interp。运行 Interp。

运行程序并尝试几个 dbx 命令：

```
$ a.out
> exec date
Sun Jun 21 16:13:06 PDT 2009
> 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'
```

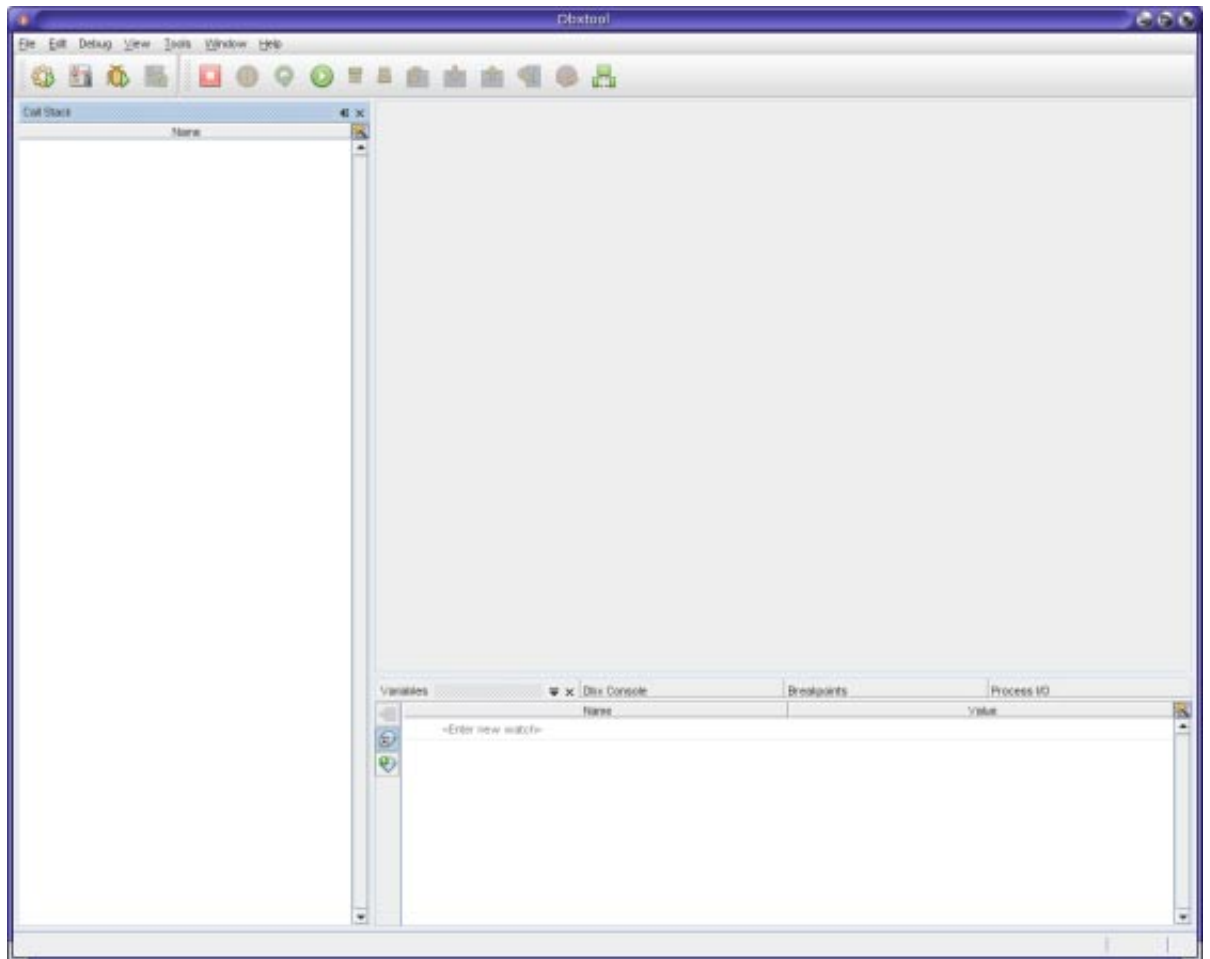
```
}  
> quit  
Goodby  
$
```

## 配置 dbxtool

通过键入以下内容启动 dbxtool :

```
installation_directory/bin/dbxtool
```

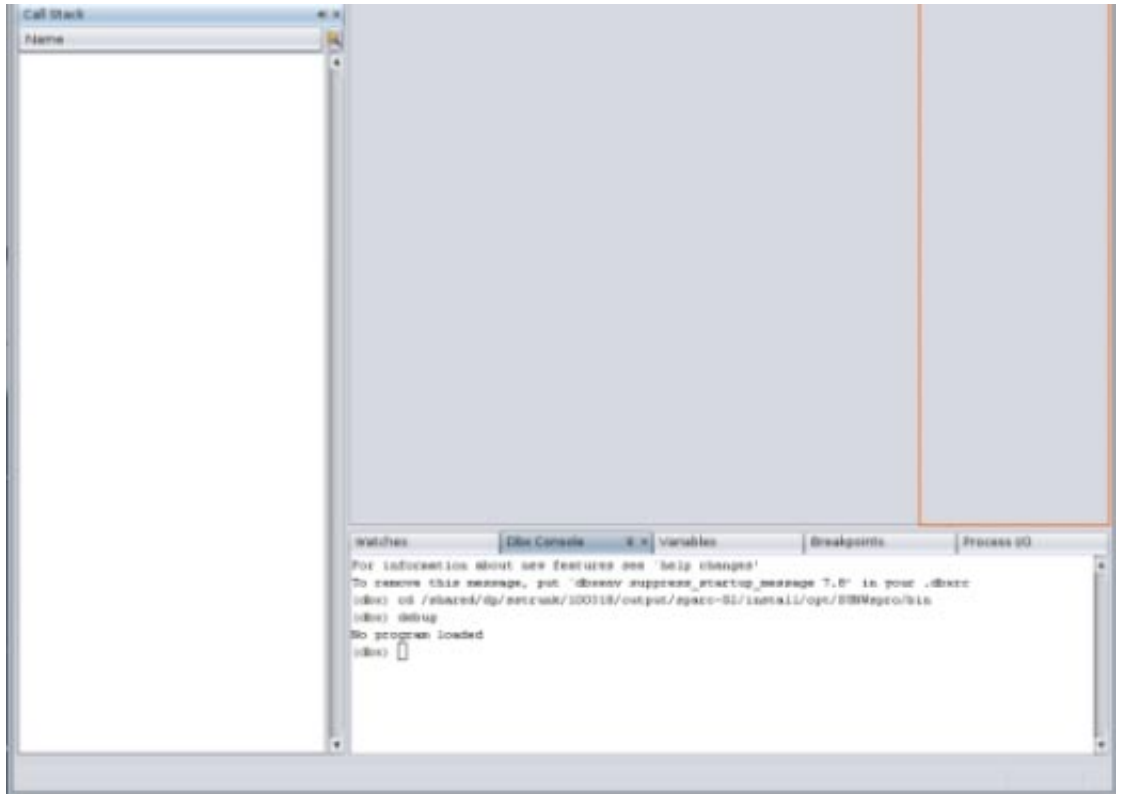
第一次启动 dbxtool 时，窗口的外观如下所示：



如果正在 Web 浏览器中阅读本教程，该窗口很可能会占据半个屏幕，此时将 dbxtool 自定义为一个半屏幕应用程序会很有用。

下面是自定义 dbxtool 的各种方法的示例。

- 使工具栏图标变小：
  - 右键单击工具栏中的任意位置并选择 "Small Toolbar Icons"（小工具栏图标）。
- 将 "Call Stack"（调用栈）窗口移开：
  1. 单击 "Call Stack"（调用栈）窗口的标题，并向下向右拖动该窗口。当红色轮廓处于此位置时松开鼠标：



2. 现在单击 "Call Stack"（调用栈）窗口标题中向右的箭头：

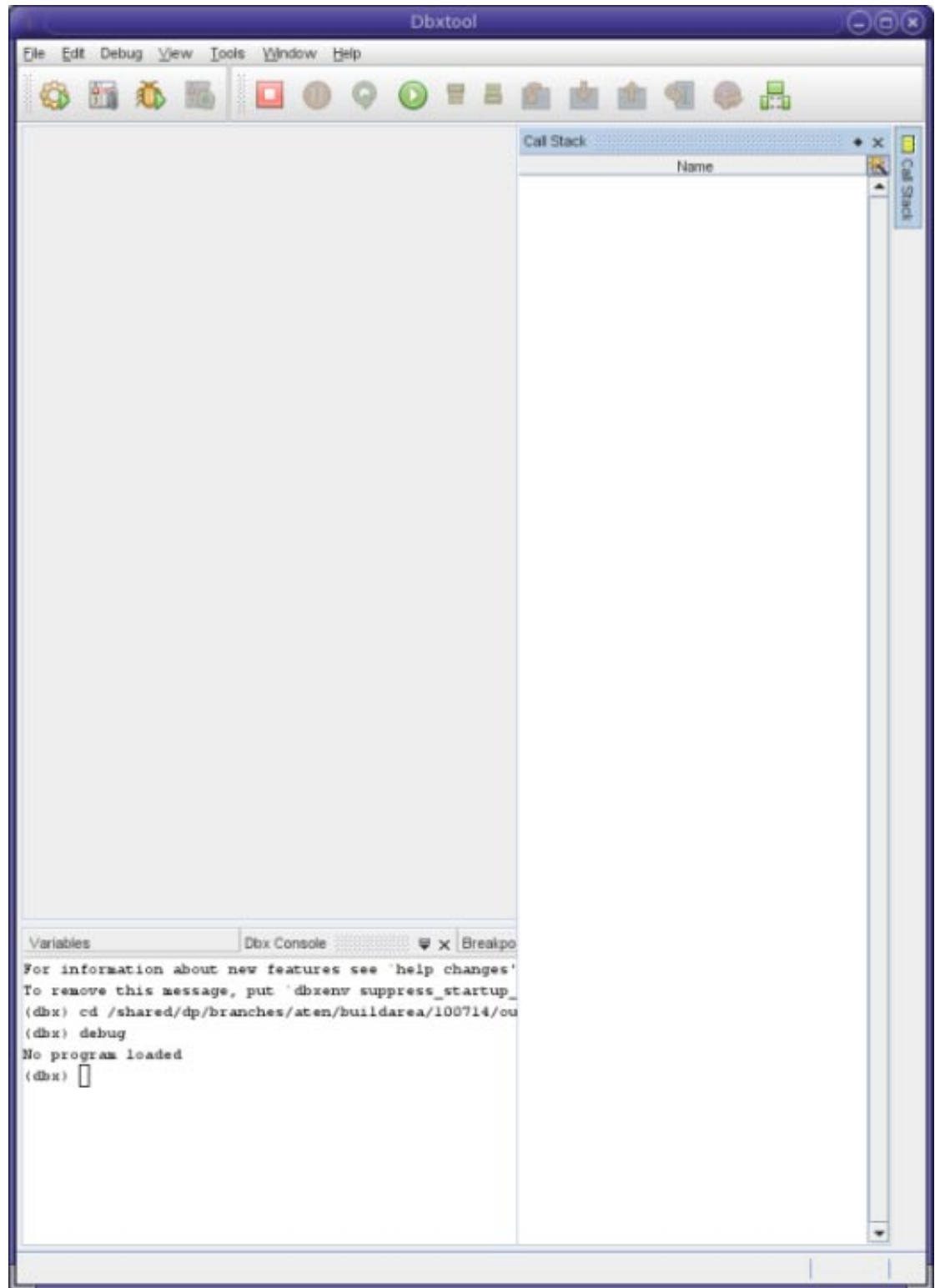


"Call Stack"（调用栈）窗口最小化在右边界中：



3. 如果将光标悬停在最小化的 "Call Stack"（调用栈）图标上方，"Call Stack"（调用栈）窗口会最大化，直到将焦点转移到另一个窗口。如果单击最小化的 "Call Stack"（调用栈）图标，"Call Stack"（调用栈）窗口会最大化，直到再次单击该图标。

4. 现在应该能够将主窗口缩小至半屏幕：



■ 最小化 "Breakpoints" (断点) 窗口：

1. 单击 "Breakpoints" (断点) 选项卡。
2. 单击选项卡上的向下箭头以最小化 "Breakpoints" (断点) 窗口。



- 取消固定 "Process I/O" (进程 I/O) 窗口：
  1. 单击并按住 "Process I/O" (进程 I/O) 窗口的标题，将该窗口拖到 dbxtool 窗口之外，然后放到桌面上。现在您可以轻松与您正在调试的程序的输入和输出进行交互，同时可以轻松访问 dbxtool 窗口中的其他选项卡。
  2. 要在 dbxtool 窗口中重新固定 "Process I/O" (进程 I/O) 窗口，请在 "Process I/O" (进程 I/O) 窗口中右键单击，然后选择 "Dock window" (固定窗口)。
- 在编辑器中设置字体大小。在将某些源代码显示在 "Editor" (编辑器) 窗口中之后：
  1. 选择 "Tools" (工具) > "Options" (选项)。
  2. 在 "Options" (选项) 窗口中，选择 "Fonts & Colors" (字体和颜色) 类别。
  3. 在 "Syntax" (语法) 选项卡上，确保从 "Languages" (语言) 下拉式列表中选择 "All Languages" (所有语言)。
  4. 单击 "Font" (字体) 文本框旁边的浏览按钮。
  5. 在 "Font Chooser" (字体选择器) 对话框中，设置字体、样式和大小，然后单击 "OK" (确定)。
  6. 在 "Options" (选项) 窗口中单击 "OK" (确定)。
- 在终端窗口中设置字体大小。"Dbx Console" (Dbx 控制台) 窗口和 "Process I/O" (进程 I/O) 窗口是 ANSI 终端仿真器。
  1. 选择 "Tools" (工具) > "Options" (选项)。
  2. 在 "Options" (选项) 窗口中，选择 "Miscellaneous" (其他) 类别。
  3. 单击 "Terminal" (终端) 选项卡。
  4. 选择 "Font Size" (字体大小) 和 "Click To Type" (单击以键入) 等设置。
  5. 单击 "OK" (确定)。

## 诊断核心转储

既然您已根据自己的喜好配置了 dbxtool，让我们来找一些错误。

再次运行示例程序，但这次要按 Return 键而不是输入命令：

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

现在启动包含可执行文件和核心转储文件的 dbxtool：

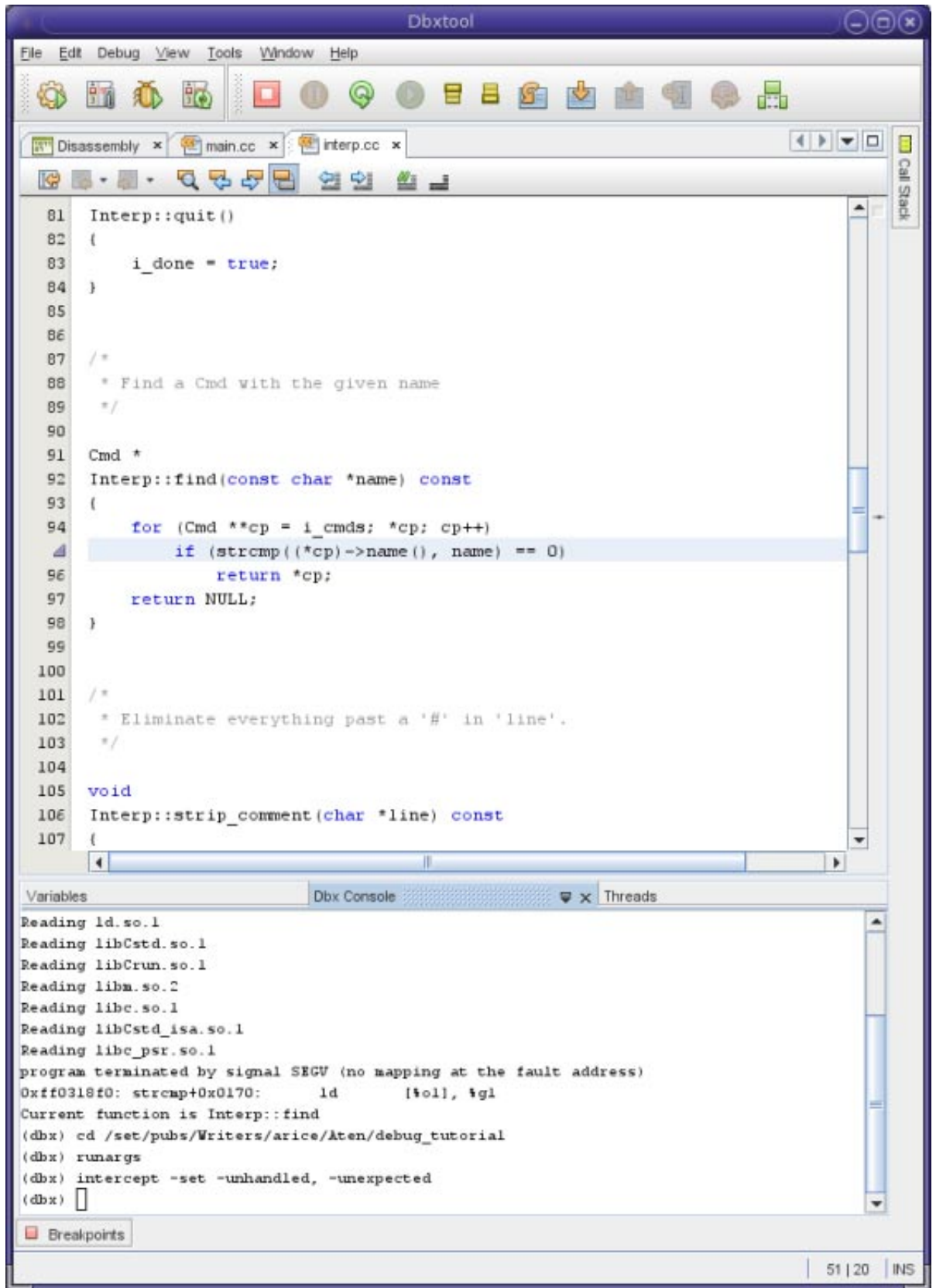
```
$ dbxtool a.out core
```

---

提示 - 请注意，dbxtool 命令接受的参数与 dbx 命令相同。

---

dbxtool 显示如下内容：

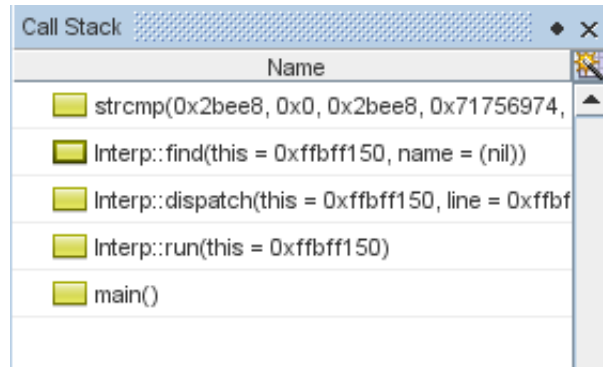


下面是一些注意事项：

- 在"Dbx Console"（Dbx控制台）窗口中，将看到一条类似以下内容的消息：

```
program terminated by signal SEGV (no mapping at fault address)
0xff0318f0: strcmp+0x0170:    ld      [%o1], %g1
Current function is Interp::find
```

- 即使 `strcmp()` 函数中发生了 SEGV，dbx 也会自动在第一个帧中显示包含调试信息的函数。请注意，"Call Stack"（调用栈）窗口中的栈跟踪有一个边框围绕着当前帧的图标：



"Call Stack"（调用栈）窗口显示参数名称和值。您可以看到传递给 `strcmp()` 的第二个参数为 `0x0`，名称值为 `NULL`。

- 在 "Editor"（编辑器）窗口中，淡紫色长条和一个三角形（而不是绿色条和箭头）表示调用 `strcmp()` 的位置，而不是错误的实际位置：

---

**提示** – 如果您没有看到参数值，请检查 `.dbxrc` 文件中 `dbx` 环境变量 `stack_verbose` 是否设置为打开。通过在 "Call Stack"（调用栈）窗口中右键单击并选择 "Verbose"（详细）复选框来添加复选标记，您还可以在该窗口中打开详细模式。

---



```
Disassembly x main.cc x interp.cc x
Interp::quit()
{
    i_done = true;
}

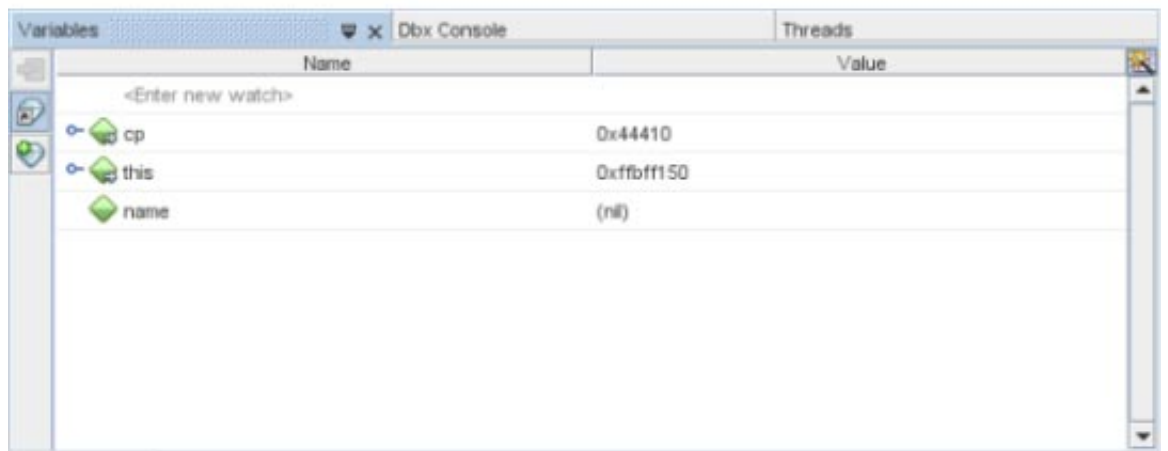
/*
 * Find a Cmd with the given name
 */
Cmd *
Interp::find(const char *name) const
{
    for (Cmd **cp = i_cmds; *cp; cp++)
        if (strcmp((*cp)->name(), name) == 0)
            return *cp;
    return NULL;
}

/*
 * Eliminate everything past a '#' in 'line'.
 */
void
Interp::strip_comment(char *line) const
{

```

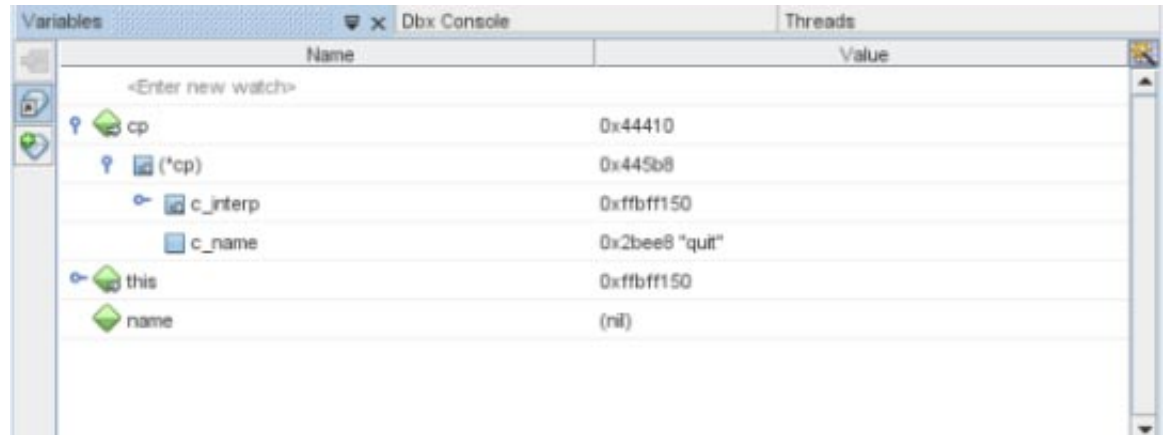
向函数传递错误值作为参数时，函数通常会失败。下面是一些检查传递给 `strcmp()` 的值的方法：

- 检查 "Variables" (变量) 窗口中参数的值。  
单击 "Variables" (变量) 选项卡。



请注意，名称值为 NULL。该值很可能是导致 SEGV 的原因，但我们还是来检查一下另一个参数 (\*cp) ->name() 的值。

在 "Variables" (变量) 窗口中，展开 cp 节点，然后展开 (cp\*) 节点。此处所涉及的名称为 "quit"，该名称可以接受：



---

提示 - 如果展开 \*cp 节点不显示其他变量，请检查 .dbxrc 文件中 dbx 环境变量 output\_inherited\_members 是否设置为打开。通过在窗口中右键单击并选择 "Inherited Members" (继承的成员) 复选框来添加复选标记，您还可以显示继承的成员。

---

- 使用气球表达式求值确认参数的值。在 "Editor" (编辑器) 窗口中，将光标放置到正传递给 strcmp() 的名称变量上方。将显示一个提示，名称值显示为 NULL。

```

82 {
83     i_done = true;
84 }
85
86
87 /*
88  * Find a Cmd with the given name
89  */
90
91 Cmd *
92 Interp::find(const char *name) const name = (nil)
93 {
94     for (Cmd **cp = i_cmds; *cp; cp++) type: const char *name;
95     if (strcmp((*cp)->name(), name) == 0)
96         return *cp;
97     return NULL;
98 }
99
100
101 /*
102  * Eliminate everything past a '#' in 'line'.
103  */
104
105 void
106 Interp::strip_comment(char *line) const
107 {
108     char *poundx = strchr(line, '#');

```

使用气球表达式求值，您还可以将光标放置到一个表达式的上方，例如 `(*cp)->name()`。但在此情况下，您不能这样做，因为表达式包含函数调用。

---


**提示 -**

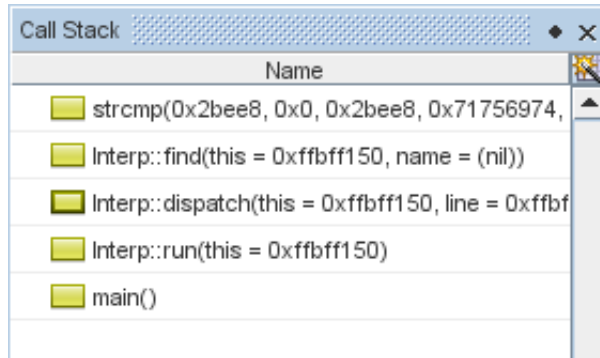
包含函数调用的表达式的气球表达式求值会禁用，因为：

- 您正在调试一个核心转储文件。
  - 函数调用可能会有副作用，您不会希望无意中在 "Editor"（编辑器）窗口中悬停光标时发生函数调用。
- 

现在我们清楚了，名称的值不应为 NULL。那么是哪个代码将此错误值传递给 `Interp::find()` 的呢？要弄清这个问题，可以执行以下操作：

- 通过选择 "Debug"（调试） > "Stack"（栈） > "Make Caller Current"（使调用方成为当前调用

方）或单击工具栏上的 "Make Caller Current"（使调用方成为当前调用方）按钮 ，向上移动调用栈：



- 在 "Call Stack" (调用栈) 窗口中，双击与 `Interp::dispatch()` 相对应的帧。现在 "Editor" (编辑器) 窗口中突出显示了相应的代码：

```

128
129     char *token = strtok(line, DELIMITERS);
130     argv[argc++] = token;                                // first token
131
132     while (token = strtok(NULL, DELIMITERS)) { // rest of the tokens
133         if (argc >= MAXARGS) {
134             printf("Too many arguments at '%s'\n", token);
135             return;
136         }
137         argv[argc++] = token;
138     }
139     argv[argc++] = NULL;                                // sentinel
140
141     Cmd *cmd = find(argv[0]);                            // Look for Cmd by name
142
143     if (!cmd) {
144         printf("Unrecognized command '%s'\n", argv[0]);
145     } else {
146         if (!isatty()) {
147             // echo (analog of dbx -e)
148             prompt();
149             for (char **avp = argv; *avp; avp++)
150                 printf("%s ", *avp);
151             printf("\n");
152         }
153     }
154

```


此代码是未知代码。除了知道 `argv[0]` 的值为 `NULL` 之外，一眼看不出任何线索。使用断点和步进以动态方式调试此问题可能会容易一些。

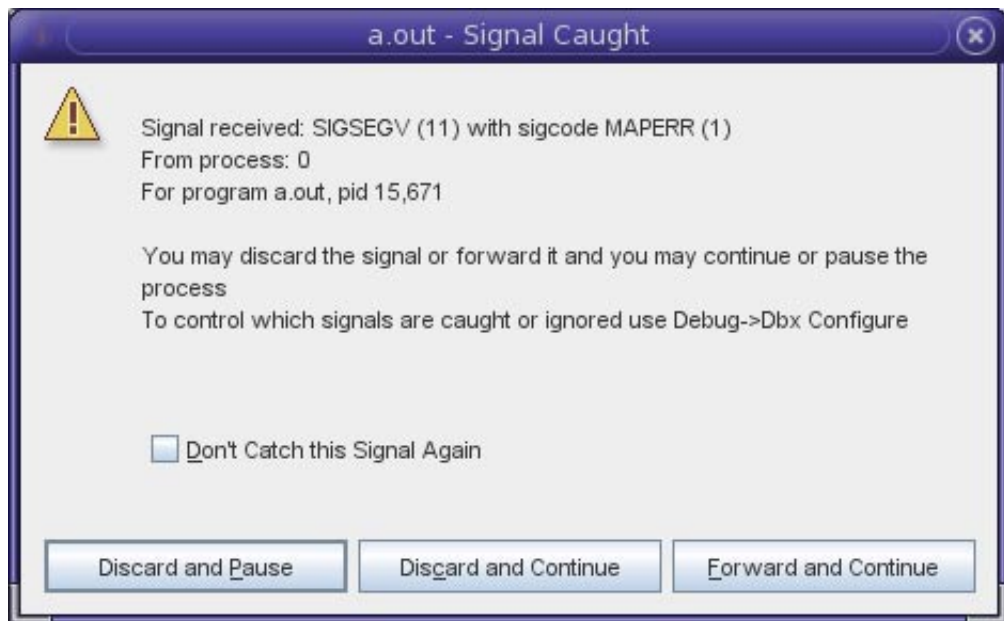
## 使用断点和步进


使用断点，您可以在错误出现之前的片刻停止程序，并单步执行代码查找何处出错。

如果尚未这样做，现在可能是取消固定 "Process I/O"（进程 I/O）窗口的好机会。

您早先是从命令行运行的程序。现在通过在 `dbxtool` 中运行程序来重现错误：

1. 单击工具栏上的 "Run"（运行）按钮  或在 "Dbx Console"（Dbx 控制台）窗口中键入 `run`。
2. 在 "Process I/O"（进程 I/O）窗口中按 `Return` 键。此时会出现一个警报框，通知您有关 `SEGV` 的信息：



3. 在警报框中，单击 "Discard and Pause"（放弃并暂停）。"Editor"（编辑器）窗口会再次在 `Interp::find()` 中突出显示对 `strcmp()` 的调用。
4. 单击工具栏中的 "Make Caller Current"（使调用方成为当前调用方）按钮 ，转到您先前在 `Interp::dispatch()` 中看到的未知代码。现在您可以在调用 `find()` 之前的片刻设置一个断点。稍后您可以单步执行代码以查找出错的原因。

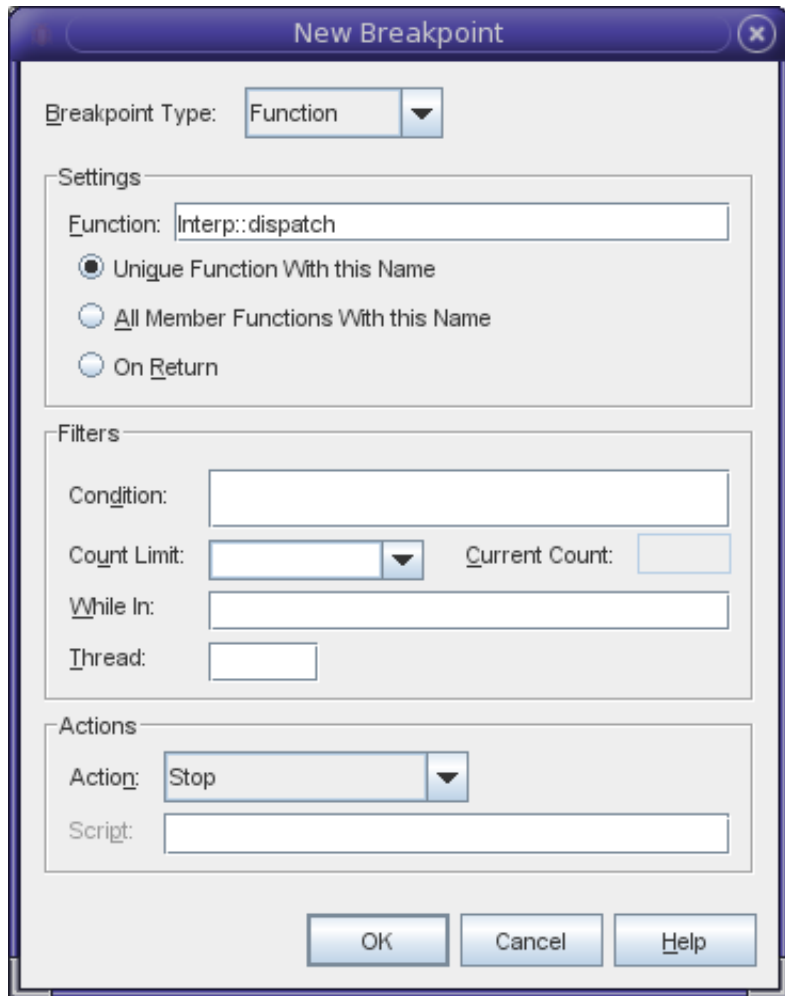
## 设置断点

设置断点有几种方法。首先，如果未显示行号，请通过在左边界中右键单击并选中 "Show Line Numbers"（显示行号）复选框，在编辑器中启用行号。

- 通过在第 127 行旁边的左边界中单击，开启/关闭行断点。

```
116  * Parse 'line' and dispatch the command it if any.
117  */
118
119  void
120  Interp::dispatch(char *line)
121  {
122      const int MAXARGS = 8;
123      const char *DELIMITERS = " \\t\\n";          // "word" delimiters
124
125      // break 'line' into "word"s and store them in 'argv'
126      char *argv[MAXARGS+1];                      // +1 for sentinel NULL
127      int argc = 0;
128
129      char *token = strtok(line, DELIMITERS);
130      argv[argc++] = token;                        // first token
```

- 通过执行下列操作设置函数断点：
  1. 在"Editor"（编辑器）窗口中选择 Interp::dispatch。
  2. 选择"Debug"（调试）>"New Breakpoint"（新建断点）或右键单击并选择"New Breakpoint"（新建断点）。将打开"New Breakpoint"（新建断点）对话框。



请注意，"Function"（函数）字段随选定的函数名称进行填充。

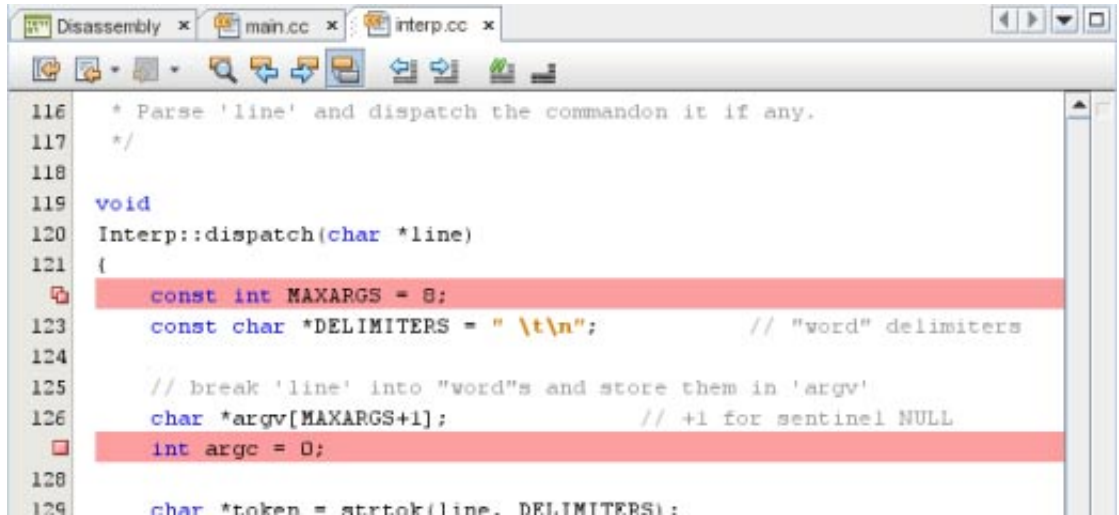
3. 单击"OK"（确定）。

- 通过 dbx 命令行设置函数断点最为容易。要执行此操作，请在 "Dbx Console"（Dbx 控制台）窗口中键入 stop in 命令：

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

请注意，不必键入 Interp::dispatch。只需键入函数名称即可。

现在编辑器开始变得很杂乱：



```
116  * Parse 'line' and dispatch the command it if any.
117  */
118
119  void
120  Interp::dispatch(char *line)
121  {
122      const int MAXARGS = 8;
123      const char *DELIMITERS = " \\t\\n"; // "word" delimiters
124
125      // break 'line' into "word"s and store them in 'argv'
126      char *argv[MAXARGS+1]; // +1 for sentinel NULL
127      int argc = 0;
128
129      char *token = strtok(line, DELIMITERS);
```

您可以使用 "Breakpoints" (断点) 窗口清理这种杂乱情形。

1. 单击 "Breakpoints" (断点) 选项卡 (如果先前最小化了该选项卡, 请先将其最大化)。
2. 选择行断点和其中一个函数断点, 右键单击, 然后选择 "Delete" (删除)。

## 函数断点的优点

通过在编辑器中切换来设置行断点可能比较直观。但是, 许多 dbx 用户喜欢使用函数断点是由于以下原因:

- 仅在 "Dbx Console" (Dbx 控制台) 窗口中键入 `si dispatch` 通常是最容易的。这样可免去在编辑器中打开文件并滚动到某个行以放置断点的麻烦。
- 您可以通过在编辑器中选择任何文本来创建函数断点。因此您可以从函数的调用点在函数上设置断点, 而不必打开文件。


---

提示 - `si` 是 `stop in` 的别名。大多数 dbx 用户都会定义许多别名, 并将这些别名放置在 dbx 配置文件 `~/.dbxrc` 中。一些常见的示例有:

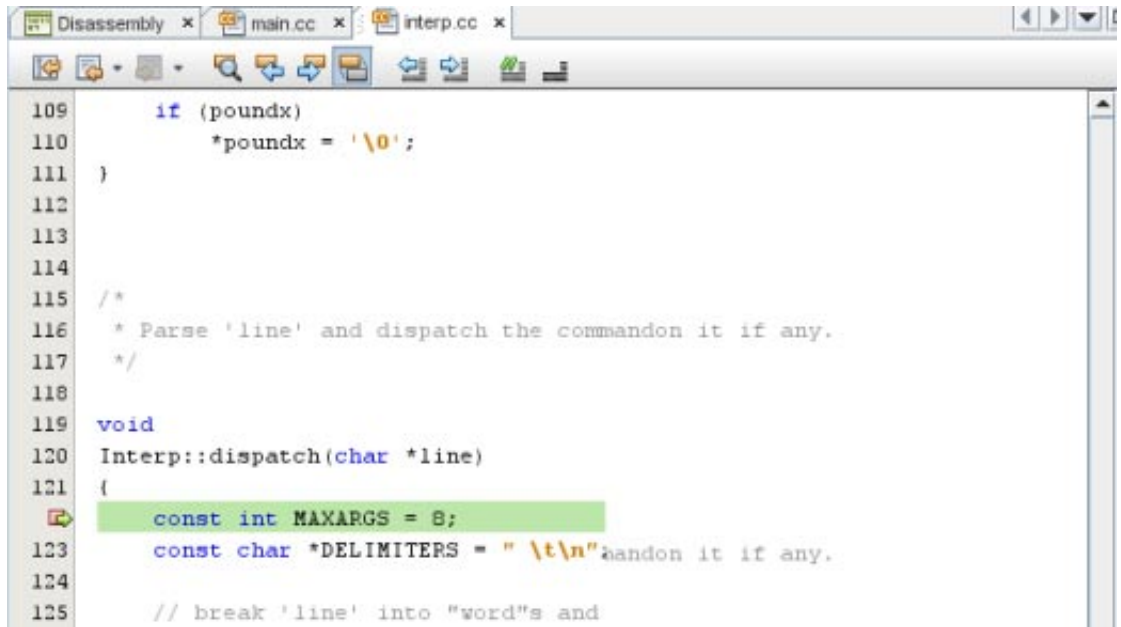
```
alias si stop in
alias sa stop at
alias s step
alias n next
alias r run
```

- 在 "Breakpoints" (断点) 窗口中, 函数断点的名称是描述性的。行断点的名称不是描述性的, 那么如何知道 `interp.cc:127` 处的内容呢? (实际上, 您可以通过在 "Breakpoints" (断点) 窗口中右键单击行断点并选择 "Go To Source" (转至源), 或者通过双击断点, 来了解第 127 行处的内容。)
- 函数断点更为稳定。由于 `dbxtool` 一直存在断点, 如果您编辑代码或执行源代码控制合并, 行号断点可能容易发生偏移。而函数名称则不太容易受到编辑操作的影响。

## 使用监视和步进

现在您在 `Interp::dispatch()` 处有一个断点。如果您再次单击 "Run" (运行)  并在 "Process I/O" (进程 I/O) 窗口中按 Return 键, 程序会停止在包含可执行代码的 `dispatch()` 函数的第一行。

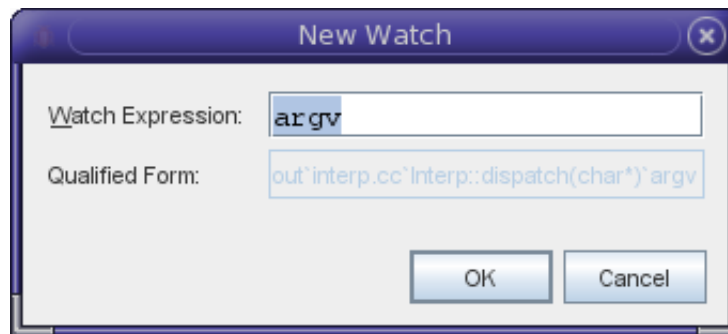




```
109     if (poundx)
110         *poundx = '\\0';
111     }
112
113
114
115     /*
116     * Parse 'line' and dispatch the command on it if any.
117     */
118
119     void
120     Interp::dispatch(char *line)
121     {
122         const int MAXARGS = 8;
123         const char *DELIMITERS = "\\t\\n";
124
125         // break 'line' into "word"s and
```

您已经知道原因是传递给 find() 的 argv[0]，因此请使用监视留意 argv：

1. 在 "Editor"（编辑器）窗口中选择 argv 的一个实例。
2. 选择 "Debug"（调试）> "New Watch"（新建监视）或右键单击并选择 "New Watch"（新建监视）。"New Watch"（新建监视）对话框将打开，并随选定的文本进行填充：



3. 单击 "OK"（确定）。
4. 通过选择 "Window"（窗口）> "Watches"（监视）打开 "Watches"（监视）窗口。
5. 展开 argv。

Watches		Variables	Dbx Console	Threads
Name	Value			
argv	(0xffbfeb0 "x\xff\xbf\xee\xa8\xff\xbf\xec\$,0x13074 "?\xff\xff;\x92^T@",0xff...			
argv[0]	0xffbfeb0 "x\xff\xbf\xee\xa8\xff\xbf\xec\$"			
argv[1]	0x13074 "?\xff\xff;\x92^T@"			
argv[2]	0xffbfeb0 "x\xff\xbf\xee\xa8\xff\xbf\xec\$"			
argv[3]	0xffbfec24 "in"			
argv[4]	0x23 "<bad address 0x00000023>"			
argv[5]	0xff0f7870 ""			
argv[6]	(nil)			
argv[7]	0xff0f3700 ""			
argv[8]	0xff1f2a00 ""			

所有这些无用信息是什么？请注意，argv 没有初始化，而且因为它是一个局部变量，可能会从先前的调用中“继承”栈上留下的随机值。这会不会是问题的原因？不要着急，让我们按部就班地进行。



- 单击 "Step Over" (步过) 两次，直到绿色 PC 箭头指向 `int argc = 0;`。
- 很显然，argc 将成为 argv 中的一个索引，因此也要留意，并为其创建一个监视。请注意，它当前也未初始化，并且可能包含无用值。
- 由于您为 argc 创建了监视，因此它会显示在 "Watches" (监视) 窗口中的 argv 下面。如果显示在窗口的第一行也可以。您可以删除这些监视，然后按期望的顺序重新输入。但是，在这种情况下，可以使用一些小技巧。单击 "Name" (名称) 列标题可对该列进行排序。单击该列标题，直到获得类似以下的内容为止 (请注意排序三角形)：

Watches		Variables	Dbx Console	Threads
Name	Value			
<Enter new watch>				
argc	1024			
argv	(0xffbfeb0 "x\xff\xbf\xee\xa8\xff\xbf\xec\$,0x13074 "?\xff\xff;\x92^T@",0xff...			
argv[0]	0xffbfeb0 "x\xff\xbf\xee\xa8\xff\xbf\xec\$"			
argv[1]	0x13074 "?\xff\xff;\x92^T@"			
argv[2]	0xffbfeb0 "x\xff\xbf\xee\xa8\xff\xbf\xec\$"			
argv[3]	0xffbfec24 "in"			
argv[4]	0x23 "<bad address 0x00000023>"			
argv[5]	0xff0f7870 ""			
argv[6]	(nil)			



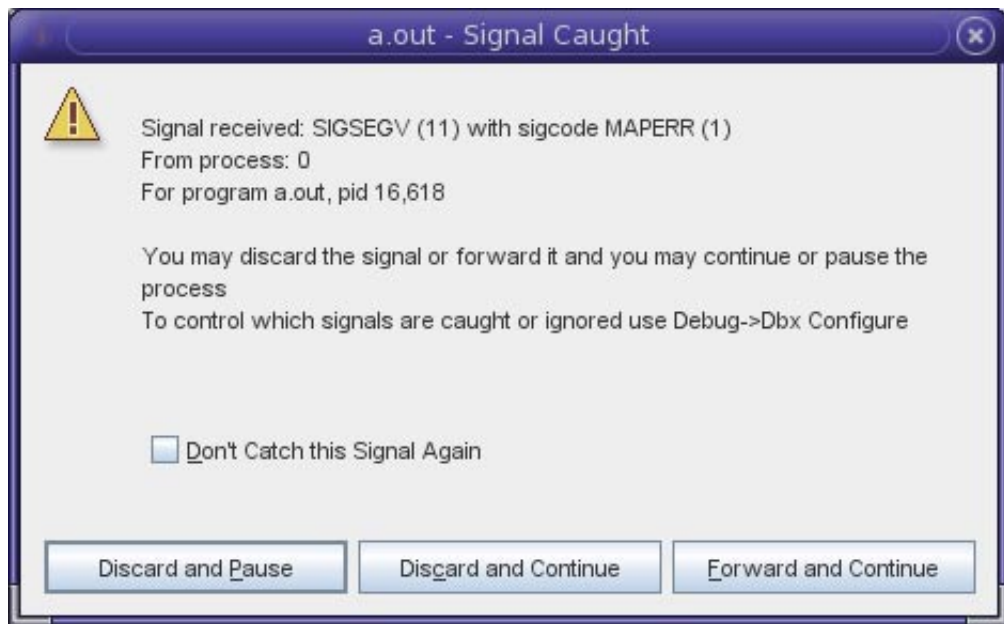
- 单击 "Step Over" (步过)。请注意，argc 现在显示其初始化值 0。该值以粗体显示，表明刚刚更改过。

Watches		Variables	Dbx Console	Threads
Name		Value		
<Enter new watch>				
argc		0		
argv		(0xffbfebcd "\xff\xbf\xee\xa8\xff\xbf\xec\$ ,0x13074 "?\xff\xff,\x92^T@",0xff...		
argv[0]		0xffbfebcd "\xff\xbf\xee\xa8\xff\xbf\xec\$"		
argv[1]		0x13074 "?\xff\xff,\x92^T@"		
argv[2]		0xffbfebcd "\xff\xbf\xee\xa8\xff\xbf\xec\$"		
argv[3]		0xffbfebcd "\n"		
argv[4]		0x23 "<bad address 0x00000023>"		
argv[5]		0xff0f7870 ""		
argv[6]		/nil		

10. 我们的应用程序将调用 `strtok()`。单击 "Step Over"（步过）步过该函数，并观察标记是否为 NULL（例如，通过使用气球表达式求值）。

提示 - `strtok()` 的功能是什么？您可以阅读 `strtok(3)` 手册页，但简言之，它可帮助将一个字符串（例如，`line`）打断成由其中一个 `DELIMITERS/` 分隔的标记

11. 再次单击 "Step Over"（步过）会将该标记分配给 `argv`，然后在循环中将会存在一个对 `strtok()` 的调用。在步过时，您不会进入循环（不再有标记），而是会分配一个 NULL。同样步过该赋值，您将到达 `find` 调用的阈值。如果您再调用，程序将在此处崩溃。
12. 通过单步执行对 `find()` 的调用，仔细检查程序是否在此处崩溃。果然，再次显示了 "Signal Caught"（捕获的信号）警报框。

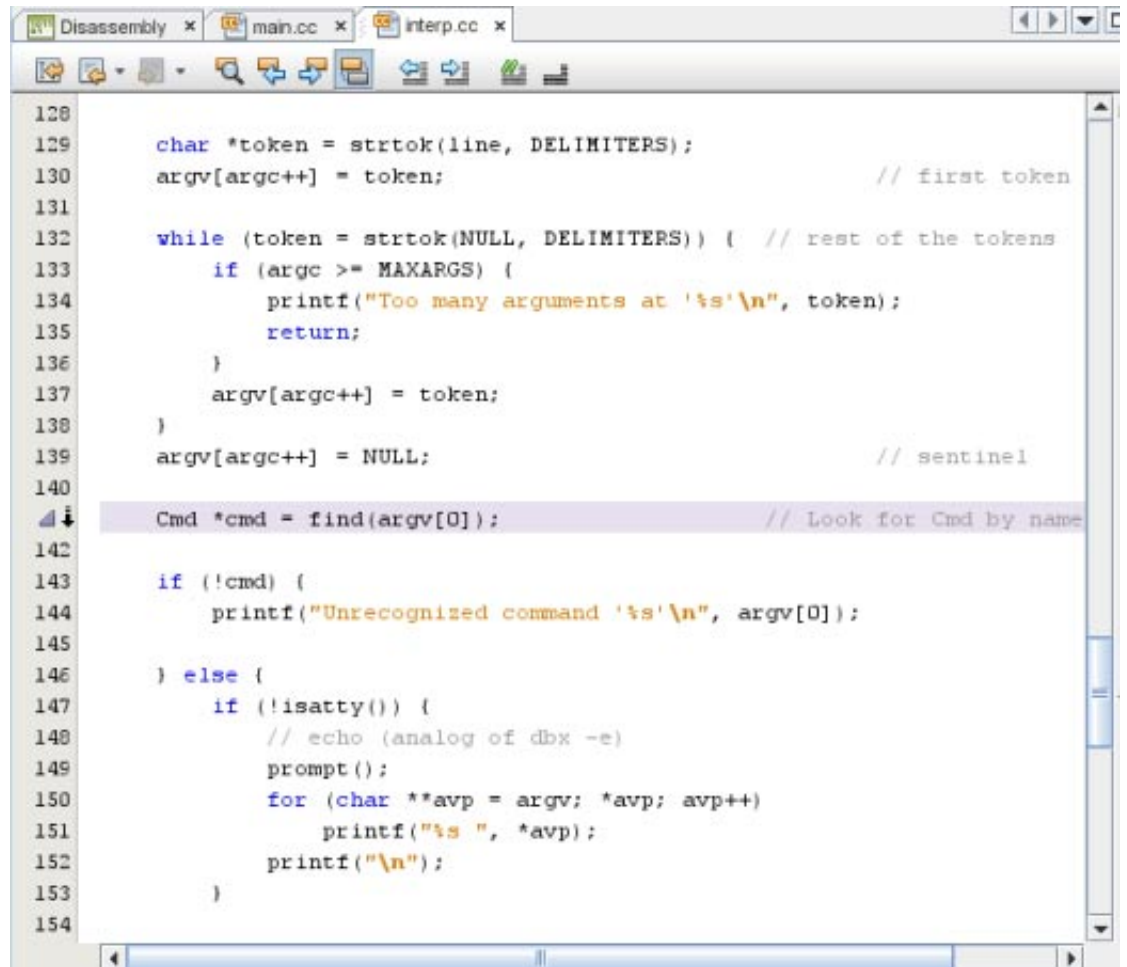


像以前一样单击 "Discard and Pause"（放弃并暂停）。

13. 因此，在 `Interp::dispatch` 中停止之后第一次调用 `find()` 实际上就是出错的位置。这可能已经很明显，但主要是为了说明您可以快速返回到您曾经所处的位置。下面是操作方法：


- a. 单击 "Make Caller Current"（使调用方成为当前调用方） 。

- b. 在 find() 的调用点处开启/关闭行断点。
- c. 打开 "Breakpoints" (断点) 窗口并禁用 Interp::dispatch() 函数断点。  
dbxtool 应如下所示：



```
128
129     char *token = strtok(line, DELIMITERS);
130     argv[argc++] = token;                                // first token
131
132     while (token = strtok(NULL, DELIMITERS)) { // rest of the tokens
133         if (argc >= MAXARGS) {
134             printf("Too many arguments at '%s'\n", token);
135             return;
136         }
137         argv[argc++] = token;
138     }
139     argv[argc++] = NULL;                                // sentinel
140
141     Cmd *cmd = find(argv[0]);                            // Look for Cmd by name
142
143     if (!cmd) {
144         printf("Unrecognized command '%s'\n", argv[0]);
145     } else {
146         if (!isatty()) {
147             // echo (analog of dbx -e)
148             prompt();
149             for (char **avp = argv; *avp; avp++)
150                 printf("%s ", *avp);
151             printf("\n");
152         }
153     }
154
```

- d. 向下的箭头表示在第 141 行上设置了两个断点，其中一个断点已禁用。

14. 单击 "Run" (运行)  并在 "Process I/O" (进程 I/O) 窗口中按 Return 键，程序将恰好在调用 find() 之前结束。(请注意 "Run" (运行) 按钮是如何导致重新启动的。在调试时，重新启动的次数要比刚启动时频繁得多。)

---

**提示** - 如果您重新生成程序 (例如，在搜索和修复错误之后)，不需要退出 dbxtool 并重新启动。单击 "Run" (运行) 按钮时，dbx 检测到程序 (或其任何成员) 已重新编译，因此会将其重新装入。

因此，只需保持 dbxtool 一直位于您的桌面上 (也许是最小化) 并随时可供调试问题之用，将会更加高效。

---

15. 那么，错误在哪里呢？请再次查看监视：

Watches		Variables	Dbx Console	Threads
Name		Value		
<Enter new watch>				
argc	2			
argv	((nil),(nil),0xffbfec0 "\xff\xbf\xee\xa8\xff\xbf\xec\$",0xffbfec24 "\n",0x23 "<b...			
argv[0]	(nil)			
argv[1]	(nil)			
argv[2]	0xffbfec0 "\xff\xbf\xee\xa8\xff\xbf\xec\$"			
argv[3]	0xffbfec24 "\n"			
argv[4]	0x23 "<bad address 0x00000023>"			
argv[5]	0xff0f7870 ""			
argv[6]	(nil)			

此处您可以实现巨大的直观跳跃，即 `argv[0]` 为 `NULL`，因为第一次调用 `strtok()` 会返回 `NULL`，这是因为这一行是空的而且没有任何标记。

**提示** – 可以在继续本教程的剩余部分之前修复此错误吗？

可以。您还可以选择记住不要按 `Return` 键并创建空行。

或者，如果您将主要在调试器下运行程序，您可以在调试器中对其进行修补，如第 40 页中的“使用断点脚本修补代码”中所述。

该示例代码的开发人员很可能已测试此条件并绕过 `Interp::dispatch()` 的剩余部分。

## 讨论

上面的示例说明了最常见的调试模式，在该模式下，用户在出错之前的某个点停止错误的程序，然后单步执行代码，将代码的本意与代码的实际行为相比较。

您是否本来能够更加直接地发现错误，而不需要任何步进和监视？事实上的，但您首先需要了解使用断点的更多技巧。

## 使用高级断点技巧

本节演示了使用断点的一些高级技巧：

- 使用断点计数
- 使用受限制的断点
- 拾取有用的断点计数
- 监视点
- 使用断点条件
- 使用弹出进行微重播
- 使用修复并继续

本节和示例程序受到了在 `dbx` 中发现的一个实际错误的启发，错误的发现顺序与本节中所述的顺序相同。

源代码包括一个名称为 `in` 的样例输入文件，该文件会在我们的示例程序中触发一个错误。该文件包含以下内容：

```
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
```

请注意，此处不存在任何空行，因此不会触发在前一节中发现的错误。

使用该输入文件运行程序时，输出如下所示：

```
$ 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'
error: cannot get value of 'var.c'
  c = '<error>'
  d = '112'
  e = '113'
  f = '114'
}
> cont
stopped in Y
var = {
  a = '115'
  b = '116'
error: cannot get value of 'var.c'
  c = '<error>'
  d = '117'
  e = '118'
  f = '119'
}
> cont
exited
```

Goodby

此输出可能看起来很长，但此示例的重点是介绍要用于长期运行的复杂程序的技术，在这些程序中，单步执行代码或跟踪是不切实际的。

请注意，显示字段 c 的值时，您将获取 <error> 的值。如果该字段包含错误地址，可能会发生这样的情形。


## 问题

请注意，当您第二次运行程序时，您收到了在第一次运行时没有收到的其他错误消息：

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

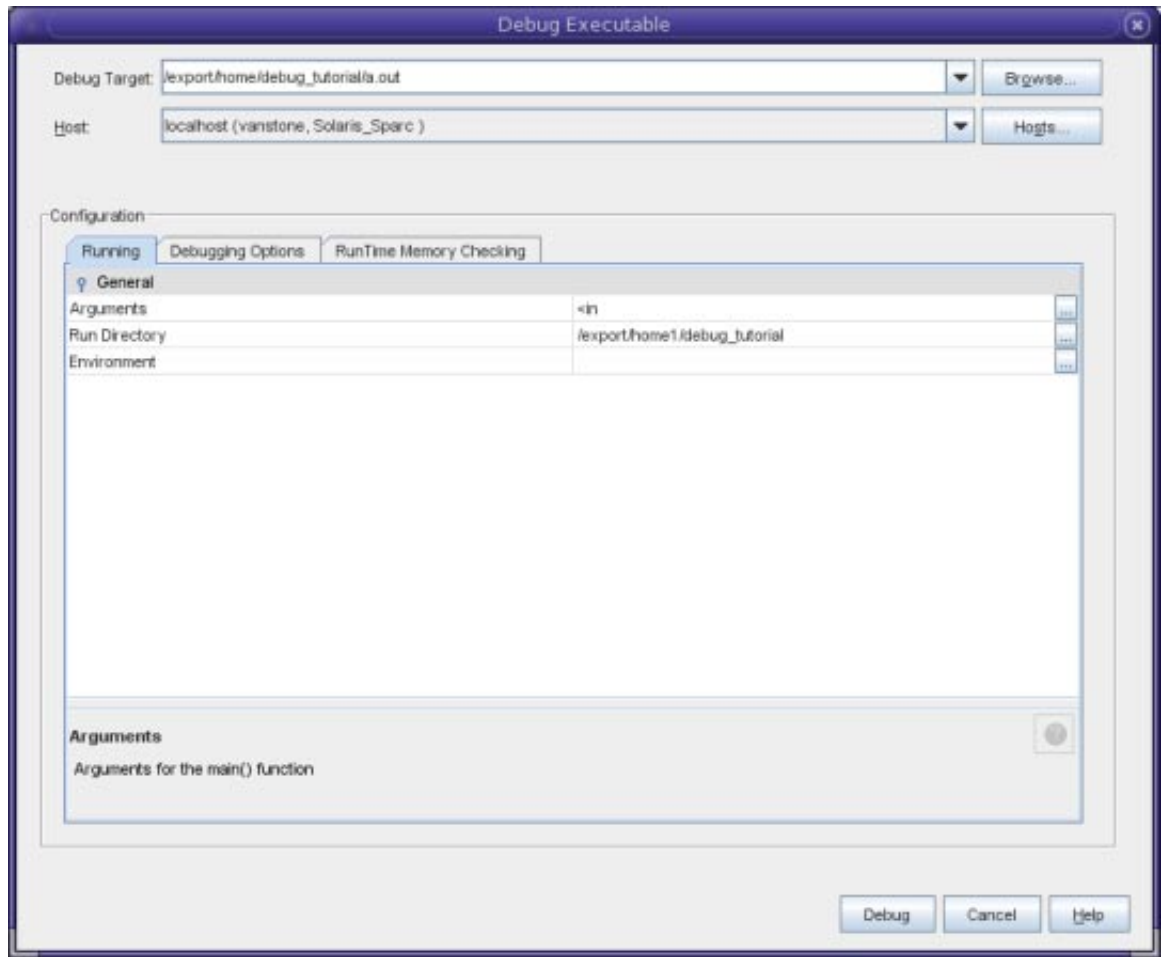
`error()` 函数使用变量 `err_silent` 使错误消息在某些情况下不再出现。例如，对于 `display` 命令，不会显示错误消息，而会将问题显示为 `c = '<error>'`。

## 步骤 1：反复性

第一步是设置一个调试目标并配置该目标，以便可通过单击 "Run"（运行） 轻松地重复该错误。

按如下方式开始调试程序：

1. 如果您尚未编译示例程序，请按照第 2 页中的“示例程序”中的说明进行编译。
2. 选择 "Debug"（调试）> "Debug Executable"（调试可执行文件）。
3. 在 "Debug Executable"（调试可执行文件）对话框中，浏览可执行文件或键入可执行文件的路径。
4. 在 "Arguments"（参数）字段中，键入：  
`< in`
5. 将可执行文件路径的目录部分复制并粘贴到 "Run Directory"（运行目录）字段中。
6. 单击 "Debug"（调试）。



在真实情形中，您可能还希望填充 "Arguments"（参数）字段或 "Environment"（环境）字段。

您可以通过选择 "Debug"（调试）> "Configure Current Session"（配置当前会话）来更改该配置的属性。

调试程序时，dbxtool 会创建一个调试目标。您可以通过选择 "Debug"（调试）> "Debug Recent"（调试最近的目标）然后选择所需的可执行文件，始终使用同一调试配置。

有时，通过 dbx 命令行设置其中多个属性会更为容易。这些属性将存储在调试目标配置中。

后面的许多步骤是为了在添加断点时保持轻松的反复性，这样始终可以通过单击 "Run"（运行）（而不必在出现各种中间断点时单击 "Continue"（继续））转到某个感兴趣的位置。

## 步骤 2：第一个断点

在 `error()` 函数输出一条错误消息时，我们在该函数内放置一个断点。此断点将成为第 33 行上的一个行断点。

在更大的程序中，通过键入以下内容（例如，在 "Dbx Console"（Dbx 控制台）窗口中），用户可以轻松地在 "Editor"（编辑器）窗口中更改当前函数：

```
(dbx) func error
```

淡紫色条表示 `func` 命令所找到的匹配项。


1. 通过在 "Editor"（编辑器）窗口左边界中数字 33 的上面单击，创建行断点。



```

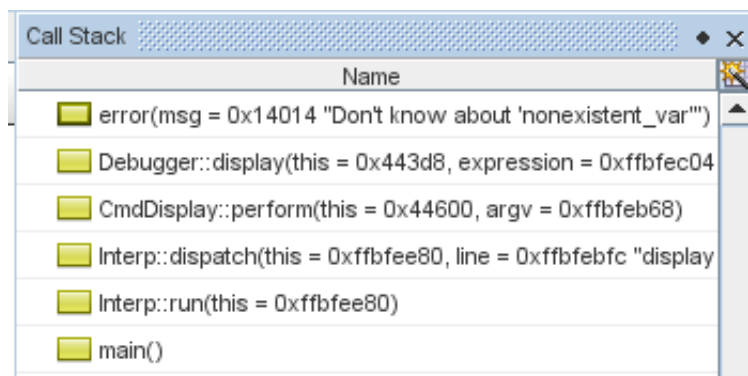
24 // emit a message.
25
26 int err_silent = 0;
27
28 void
29 error(const char *msg)
30 {
31     if (err_silent > 0)
32         return;
33     printf("error: %s\n", msg);
34 }
35
36 int
37 main()
38 {
39     debugger = new Debugger;
40
41     Interp interp;
42
43     interp.add(new CmdQuit());
44     interp.add(new CmdHelp());
45
46     interp.add(new CmdExec());
47
48     interp.add(new CmdDisplay());
49     interp.add(new CmdStop());
50     interp.add(new CmdRun());

```

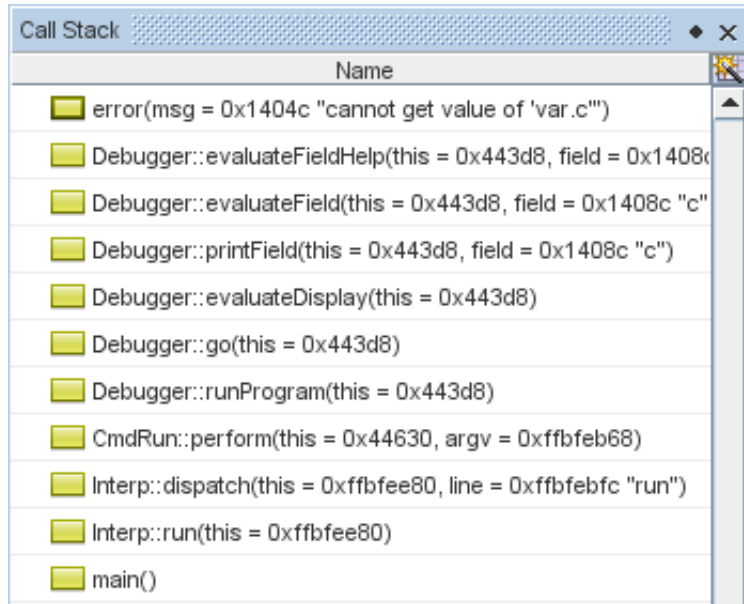
- 单击 "Run" (运行)  运行程序，并在命中断点时查看栈跟踪。系统会显示一个错误消息，该消息是由于 in 文件中的模拟命令而导致的。

> display var # should yield an error

调用 error() 是预期行为。



- 单击 "Continue" (继续)  继续该进程并再次命中该断点。此次您会收到意外的错误消息。



### 步骤 3：断点计数

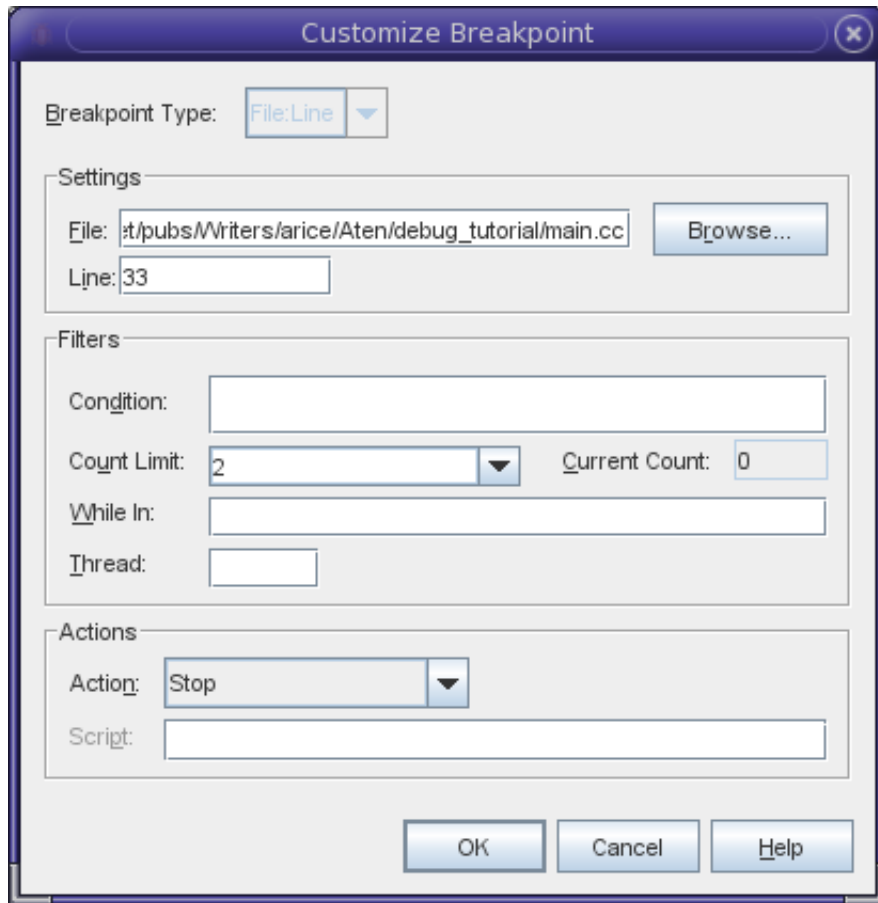
最好是在每次运行时都能反复到达此位置，而不必在第一次命中断点之后由于以下命令而单击 "Continue"（继续）：

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

您可以编辑程序或输入脚本，然后消除第一个麻烦的 `display` 命令。但是，您正在处理的特定输入顺序可能是重现此错误的键，因此我们不要去扰乱这一情形。

因为您对第二次到达此断点感兴趣，我们将其计数设置为 2：

1. 在 "Breakpoints"（断点）窗口中，右键单击该断点，然后选择 "Customize"（自定义）。
2. 在 "Customize Breakpoint"（自定义断点）对话框中，在 "Count Limit"（计数限制）字段中键入 2。
3. 单击 "OK"（确定）。



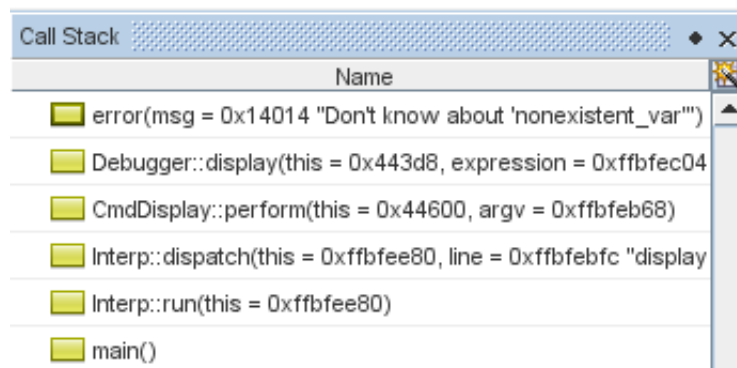
现在您可以反复到达您感兴趣的位置了。

#### 步骤 4：受限制的断点

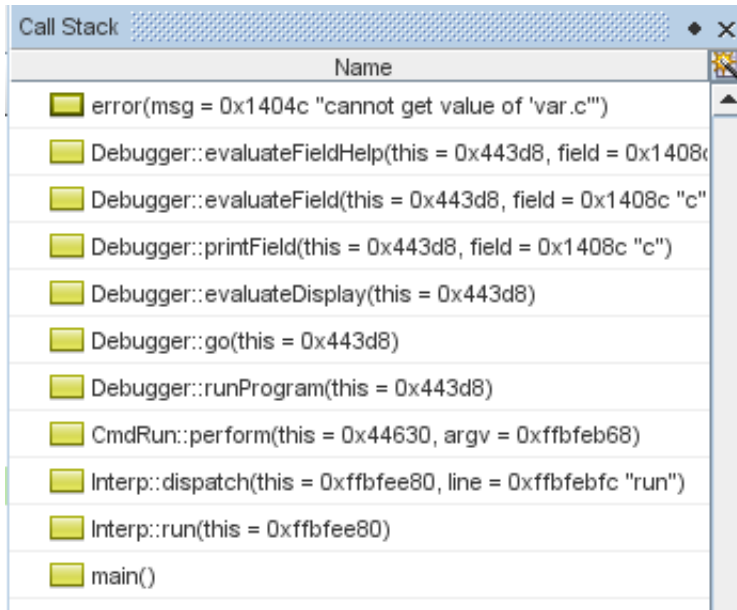
在这种情况下，选择计数 2 并不重要。但有时，您需要停止的位置会多次调用。稍后您将看到如何轻松地选择一个好的计数值。但现在我们先了解一下仅在您感兴趣的调用中在 `error()` 中停止的另一种方式。

如同以前对于 `error()` 内部的断点一样，打开 "Customize Breakpoint"（自定义断点）对话框，然后通过从 "Count Limit"（计数限制）的下拉式列表中选择 "Always stop"（总是停止）禁用断点计数。

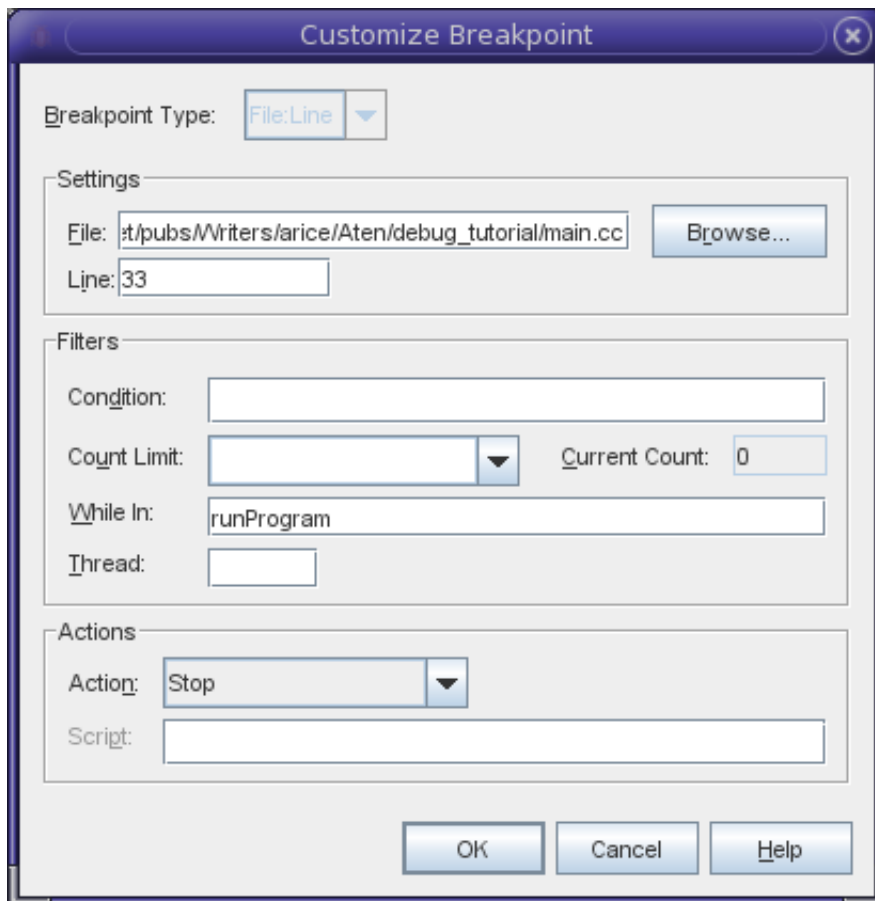
现在单击 "Run"（运行），并注意两次在 `error()` 中停止的栈跟踪。第一次的情形如下：



第二次的情形如下：



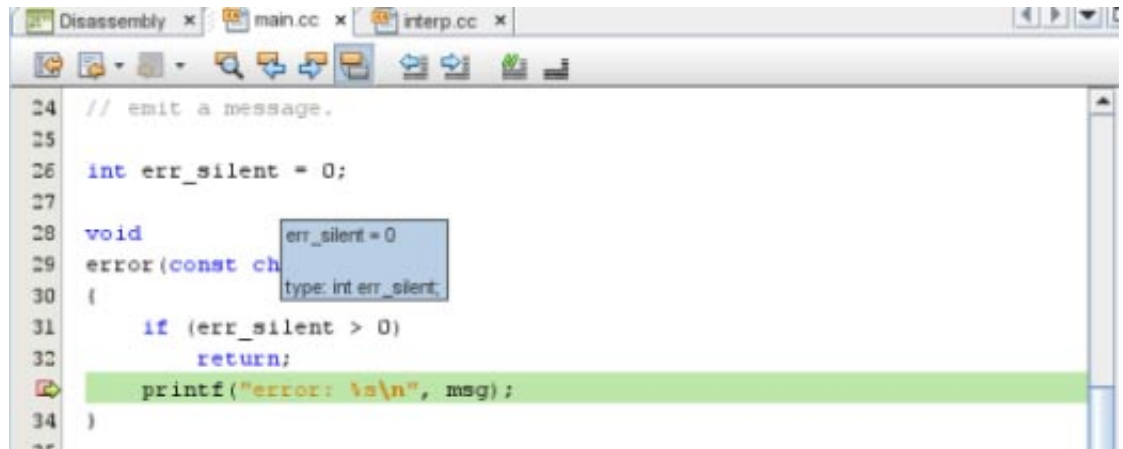
如果始终从 runProgram（第 [7] 帧）调用此断点，您会安排在此断点处停止。要这样做，请再次打开 "Customize Breakpoint"（自定义断点）对话框并将 "While In"（满足条件）字段设置为 runProgram。



现在，您又可以反复地轻松到达感兴趣的位置了。


## 步骤 5：查找原因

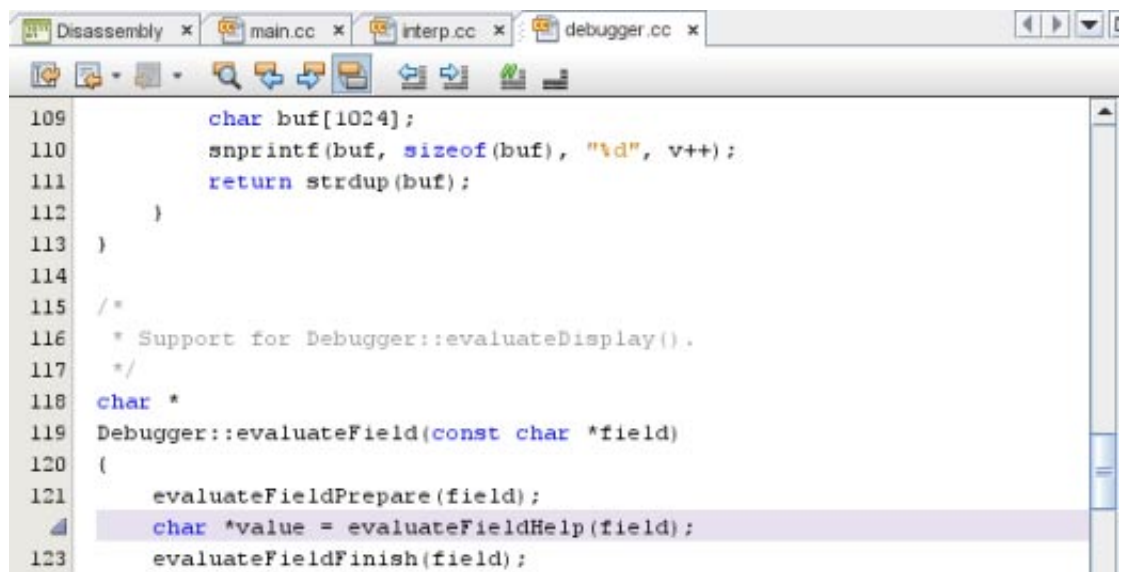
为什么发出了不需要的错误消息？很显然，这是因为 `err_silent` 不大于 0。让我们通过气球表达式求值看一下 `err_silent` 的值。将光标放在第 31 行中 `err_silent` 的上方，然后等待其值显示出来。



```
24 // emit a message.
25
26 int err_silent = 0;
27
28 void
29 error(const ch
30 {
31     if (err_silent > 0)
32         return;
33     printf("error: %s\n", msg);
34 }
```

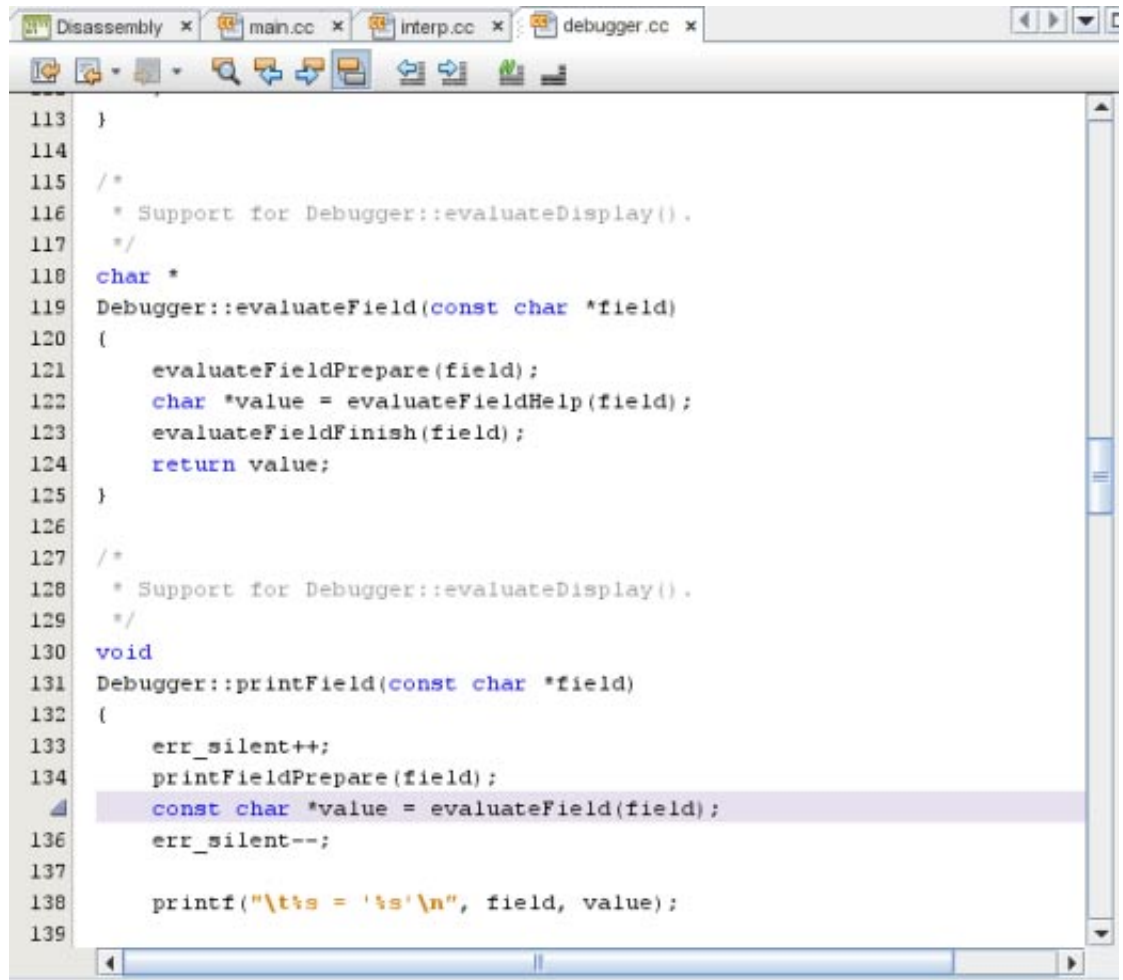
让我们跟随栈看一下设置 `err_silent` 的位置。两次单击 "Make Caller Current"（使调用方成为当前

调用方） 可到达 `evaluateField()`，`evaluateField` 已调用 `evaluateFieldPrepare()`，`evaluateFieldPrepare` 模拟一个可能正在处理 `err_silent` 的复杂函数。



```
109     char buf[1024];
110     snprintf(buf, sizeof(buf), "%d", v++);
111     return strdup(buf);
112 }
113 )
114
115 /*
116  * Support for Debugger::evaluateDisplay().
117  */
118 char *
119 Debugger::evaluateField(const char *field)
120 {
121     evaluateFieldPrepare(field);
122     char *value = evaluateFieldHelp(field);
123     evaluateFieldFinish(field);
124 }
```

再次单击 "Make Caller Current"（使调用方成为当前调用方）可到达 `printField()`。此处 `err_silent` 正在递增。`printField()` 也已调用 `printFieldPrepare()`，`printFieldPrepare` 也模拟一个可能正在处理 `err_silent` 的复杂函数。



```
113 }
114
115 /*
116  * Support for Debugger::evaluateDisplay().
117  */
118 char *
119 Debugger::evaluateField(const char *field)
120 {
121     evaluateFieldPrepare(field);
122     char *value = evaluateFieldHelp(field);
123     evaluateFieldFinish(field);
124     return value;
125 }
126
127 /*
128  * Support for Debugger::evaluateDisplay().
129  */
130 void
131 Debugger::printField(const char *field)
132 {
133     err_silent++;
134     printFieldPrepare(field);
135     const char *value = evaluateField(field);
136     err_silent--;
137
138     printf("\t%s = '%s'\n", field, value);
139 }
```

请注意 `err_silent++` 和 `err_silent--` 如何将某些代码包围起来。

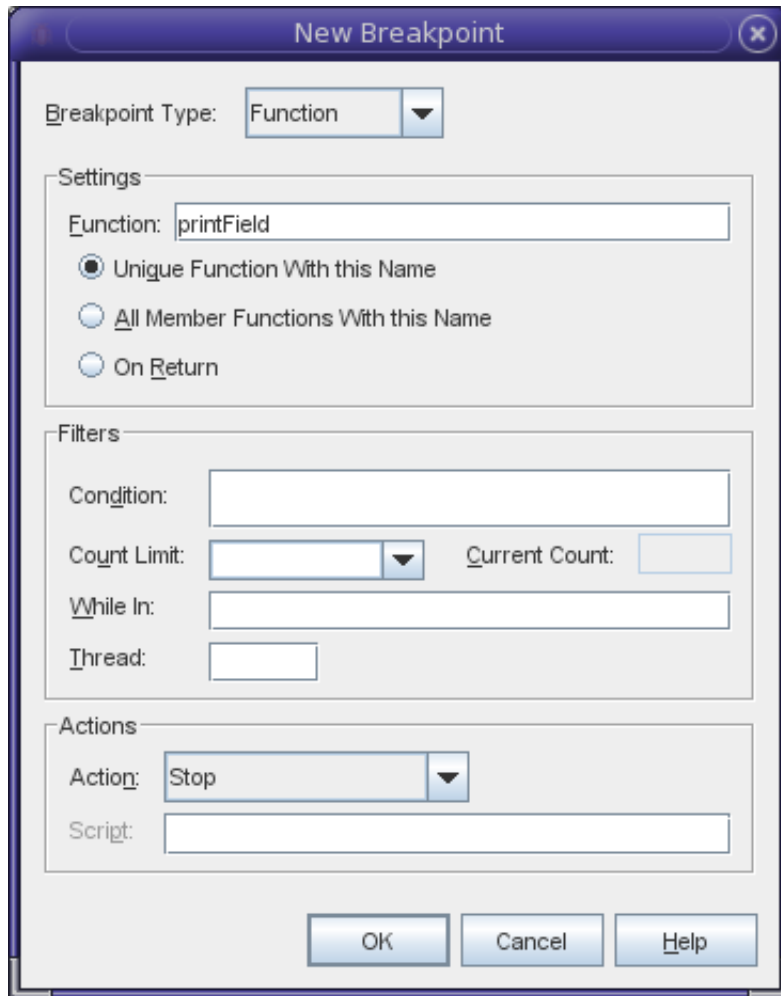
因此有可能 `err_silent` 在 `printFieldPrepare()` 或 `evaluateFieldPrepare()` 中出错，或者当控制到达 `printField()` 时 `err_silent` 已经出错。


让我们通过在 `printField()` 中放置断点，查明是在调用 `printField()` 之前还是之后出错。

## 步骤 6：更多断点计数

在 `printField()` 中设置断点。

1. 选择 `printField()`，右键单击，然后选择 "New Breakpoint"（新建断点）。
2. 新的断点类型已预先选择，且 "Function"（函数）字段已使用 `printfield` 预先填充。
3. 单击 "OK"（确定）。



4. 单击 "Run" (运行) 。第一次命中断点是在第一次运行期间，在第一次停止时，且在第一个字段 var.a 上。err\_silent 为 0，该值可以接受。

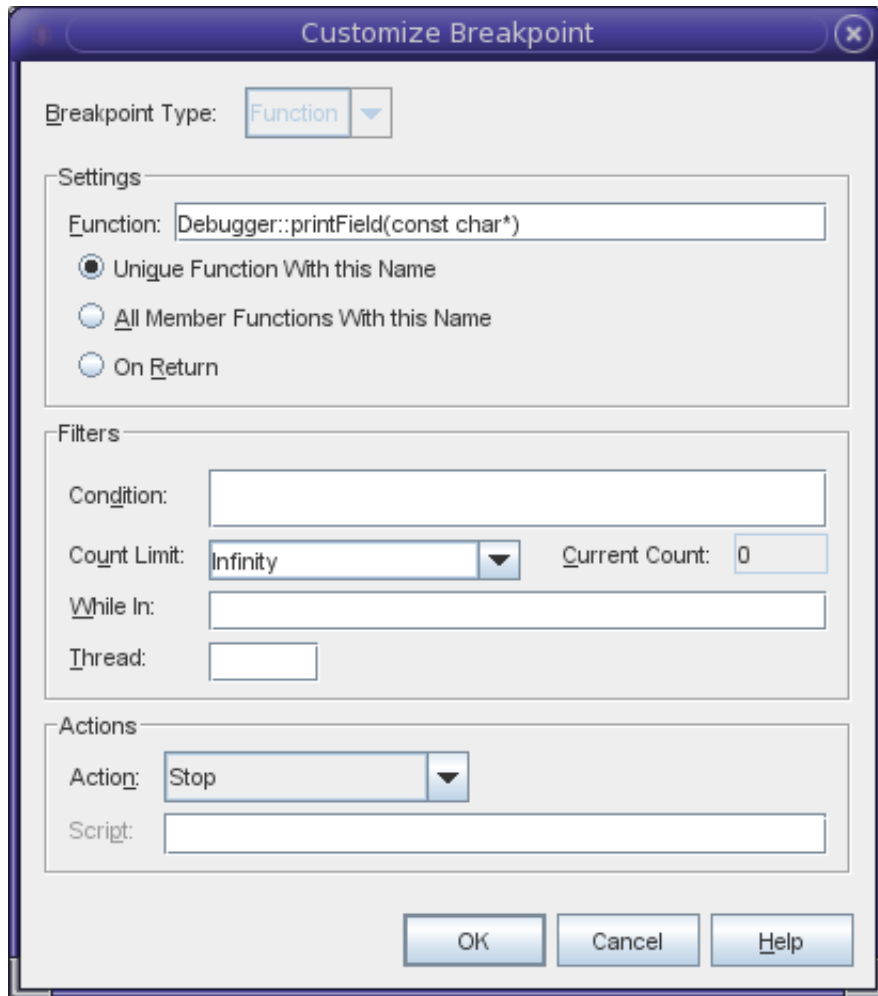
```
113 }
114
115 /*
116  * Support for Debugger::evaluateDisplay().
117  */
118 char *
119 Debugger::evaluateField(const char *field)
120 {
121     evaluateFieldPrepare(field);
122     char *value = evaluateFieldHelp(field);
123     evaluateFieldFinish(field);
124     return value;
125 }
126
127 /*
128  * Support for Debugger::evaluateDisplay().
129  */
130 void
131 Debugger::printField(int err_silent, const char *field)
132 {
133     err_silent++;
134     printFieldPrepare(field);
135     const char *value = evaluateField(field);
136     err_silent--;
```

5. 单击 "Continue" (继续)。err\_silent 仍可以接受。
6. 再次单击 "Continue" (继续)。err\_silent 仍可以接受。


在到达对 printField() 的特定调用需要一段时间，该调用导致了不需要的错误消息。您需要在 printField 断点上使用断点计数。可是应该将计数设置成什么呢？在这个简单示例中，用户可以尝试对运行、停止和要显示的字段计数，但现实中可能不会像预测的那样。但是，有一种方法可以半自动地计算出该计数应该是什么。

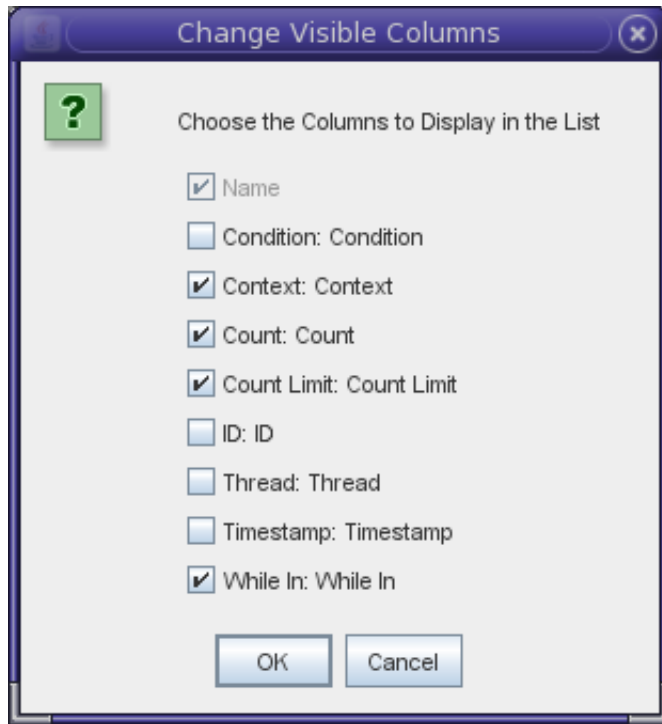
1. 打开 "Customize Breakpoint" (自定义断点) 对话框，找到 printField() 上的断点，然后将 "Count Limit" (计数限制) 字段设置为无限大。



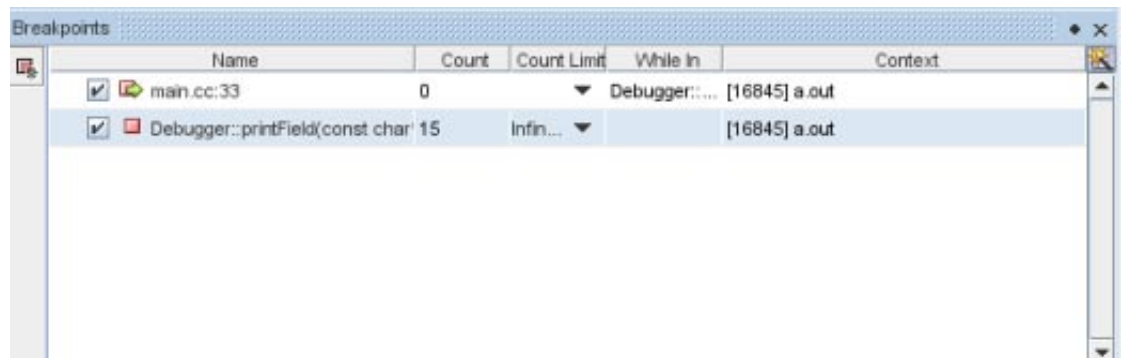


此设置意味着您将永远不会在此断点处停止。但是，仍将进行计数。

2. 此时，让 "Breakpoints"（断点）窗口显示更多属性（如计数）将很有帮助。
  - a. 单击位于 "Breakpoints"（断点）窗口右上角的 "Change Visible Columns"（更改可见列）按钮 。
  - b. 选择 "Count Limit"（计数限制）、"Count"（计数）和 "While In"（满足条件）。
  - c. 单击 "OK"（确定）。



3. 重新运行程序。您将命中 `error()` 内部的断点，也就是受 `runProgram()` 限制的断点。
4. 现在看一下 `printField()` 上断点的计数。



计数为 15，正是您想要的计数。

5. 单击 "Count Limit"（计数限制）列中的下拉式列表，选择 "Use current Count value"（使用当前计数值）将当前计数传输给计数限制，然后按 `Return` 键。

现在如果运行程序，将在最后一次在不需要的错误消息之前调用 `printField()` 时在该函数中停止！

## 步骤 6：确定具体原因

再次使用气球表达式求值检查 `err_silent`。现在的值 -1。最有可能的是，在您达到 `printField()` 之前，一个 `err_silent--` 执行得太多，或者一个 `err_silent++` 执行得太少。

如何找到这个不匹配的 `err_silent` 对呢？在像此示例这样的小程序中，仔细检查代码就可以找到。但在可能存在数量巨大的以下 `err_silent` 对的大型程序中

```
err_silent++;
err_silent--;
```

更为快捷地找到不匹配的 `err_silent` 对的方法是使用监视点。

---

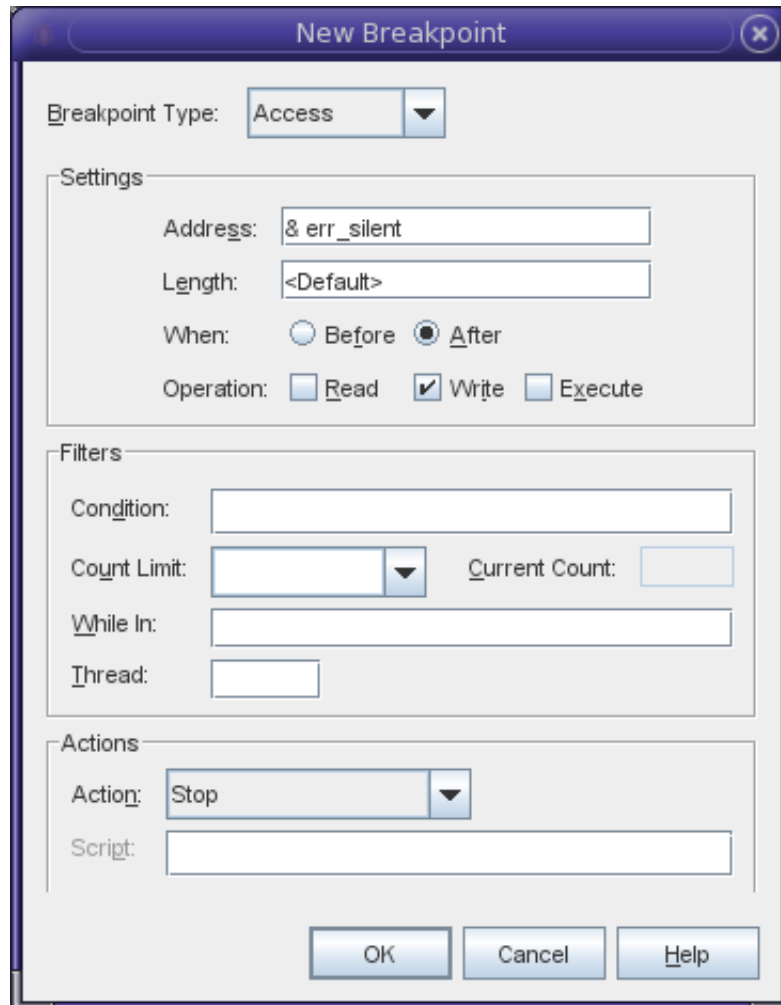
提示 - 还有一种可能是，这根本不是不匹配的 `err_silent++`; 和 `err_silent--`; 对，而是覆盖了 `err_silent` 内容的一个异常指针。在捕获此类问题时，监视点会比较有效。

---

## 步骤 7：使用监视点

在 `err_silent` 上创建监视点：

1. 选择 `err_silent` 变量，右键单击，然后选择 "New Breakpoint" (新建断点)。
2. 将 "Breakpoint Type" (断点类型) 设置为 "Access" (访问)。请注意 "Settings" (设置) 部分是如何改变以及 "Address" (地址) 字段是如何成为 `&err_silent` 的。
3. 在 "When" (时间) 字段中选择 "After" (之后)。
4. 在 "Operation" (操作) 字段中选择 "Write" (写入)。
5. 单击 "OK" (确定)。



6. 现在运行程序。您将在 `init()` 中停止。此时看起来一切正常，也就是说 `err_silent` 递增到 1，且执行在此后停止。
7. 单击 "Continue" (继续)。您再次在 `init()` 中停止。
8. 再次单击 "Continue" (继续)。您再次在 `init()` 中停止。
9. 再次单击 "Continue" (继续)。您再次在 `init()` 中停止。

10. 再次单击 "Continue" (继续)。现在您将在 `stopIn()` 中停止。此时看起来也是一切正常，也就是说没有出现 -1。

`err_silent` 设置为 -1 可能还需要一段时间，您又不想一直辛苦地单击 "Continue" (继续) 以免不留神错过了变为 -1 的时刻。但有一个更好的方法。

## 步骤 8：断点条件

为您的监视点添加一个条件：

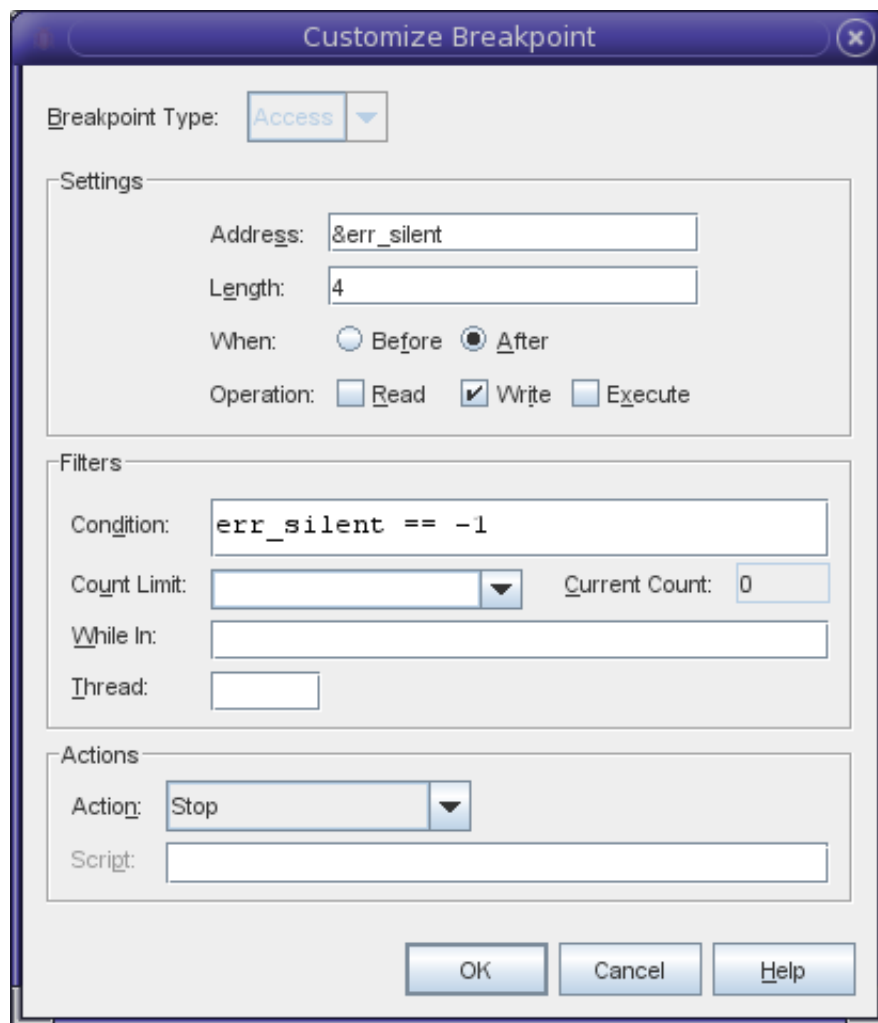
1. 在 "Breakpoints" (断点) 窗口中，右键单击 "After" (之后) 写入断点，然后选择 "Customize" (自定义)。
2. 确保在 "When" (时间) 字段中选中 "After" (之后)。

---

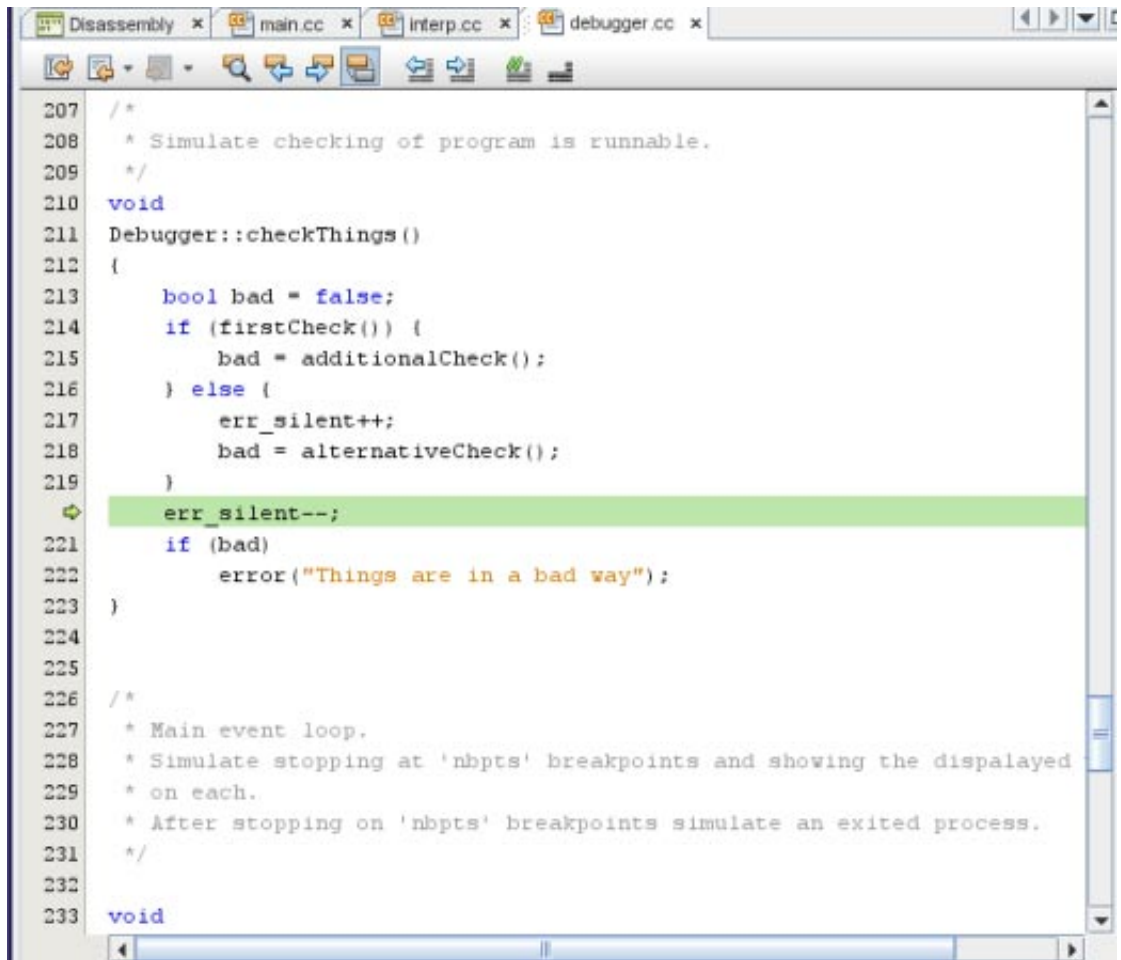
提示 - 选择 "After" (之后) 是很重要的，因为您想知道 `err_silent` 的值变成了什么。

---

3. 将 "Condition" (条件) 字段设置为 `err_silent == -1`。
4. 单击 "OK" (确定)。



现在重新运行程序。您将在 `checkThings()` 中停止。这是第一次将 `err_silent` 设置为 -1。在您查找匹配的 `err_silent++` 时，您会看清错误：`err_silent` 仅在该函数的 `else` 部分中递增。



```
207 /*
208  * Simulate checking of program is runnable.
209  */
210 void
211 Debugger::checkThings()
212 {
213     bool bad = false;
214     if (firstCheck()) {
215         bad = additionalCheck();
216     } else {
217         err_silent++;
218         bad = alternativeCheck();
219     }
220     err_silent--;
221     if (bad)
222         error("Things are in a bad way");
223 }
224
225
226 /*
227  * Main event loop.
228  * Simulate stopping at 'nbpts' breakpoints and showing the displayed
229  * on each.
230  * After stopping on 'nbpts' breakpoints simulate an exited process.
231  */
232
233 void
```

这是您所要寻找的错误吗？

## 步骤9：通过弹出栈来验证诊断

让我们仔细检查一下确实完成了该函数的 else 块。

一种检查方法是，在 checkThings() 上设置断点，然后运行程序。但 checkThings() 可能会被多次调用。您可以使用断点计数或受限制的断点来实现对 checkThings() 的正确调用，但有一种更快的方法来重播最近执行的内容。


- 选择 "Debug" (调试) > "Stack" (栈) > "Pop Topmost Call" (弹出最上面的调用)。

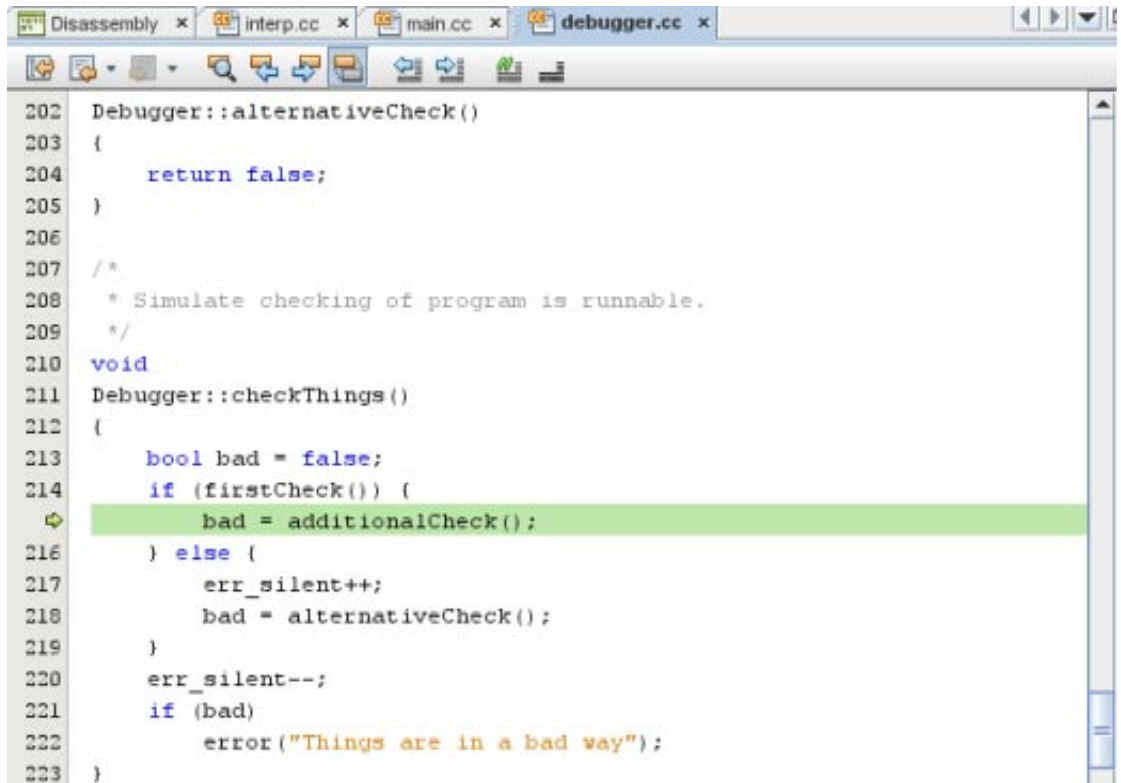
---

提示 - 请注意 "Pop Topmost Call" (弹出最上面的调用) 不会撤消任何内容。特别是，err\_silent 的值已经错了。但这应该可以接受，因为您正从数据调试切换到控制流调试。

---

您正处于调用 checkThings() 的位置。事实上，进程状态已经恢复到包含对 checkThings() 的调用的行的开始处。

现在您可以单击 "Step Into" (步入) ，并在再次调用 checkThings() 时进行观察。在单步执行 checkThings() 时，您可以验证该进程确实执行了 if 块（此处 err\_silent 没有递增，并且接着会递减至 -1）。



```
202 Debugger::alternativeCheck()
203 {
204     return false;
205 }
206
207 /*
208  * Simulate checking of program is runnable.
209  */
210 void
211 Debugger::checkThings()
212 {
213     bool bad = false;
214     if (firstCheck()) {
215         bad = additionalCheck();
216     } else {
217         err_silent++;
218         bad = alternativeCheck();
219     }
220     err_silent--;
221     if (bad)
222         error("Things are in a bad way");
223 }
```

看起来您已经发现了编程错误。但让我们进行第三次检查。

### 步骤 10：使用修复进一步验证我们的诊断

让我们修复代码并验证错误确实已经消除。

1. 通过将 `err_silent++` 置于 `if` 语句的上方来修复代码，如下所示：

```
202 Debugger::alternativeCheck()
203 {
204     return false;
205 }
206
207 /*
208  * Simulate checking of program is runnable.
209  */
210 void
211 Debugger::checkThings()
212 {
213     bool bad = false;
214     err_silent++;
215     if (firstCheck()) {
216         bad = additionalCheck();
217     } else {
218
219         bad = alternativeCheck();
220     }
221     err_silent--;
222     if (bad)
223         error("Things are in a bad way");
224 }
```

2. 选择 "Debug" (调试) > "Apply Code Changes" (应用代码更改)。
3. 禁用 printField 断点和监视点，但保留 error() 中断点的启用状态。

Name	Count	Count Limit	While In	Context
<input checked="" type="checkbox"/> main.cc:33	0	▼	Debugger:...	a.out
<input type="checkbox"/> Debugger::printField(const char 0	Infin...	▼		a.out
<input type="checkbox"/> After write &'a.out' main.cc'err_0'		▼		a.out

4. 重新运行程序。

您将注意到，程序完成但没有命中 error() 中的断点，其输出正如预期。

```
Process I/O
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'
```

## 讨论

以上内容仍说明了与第 13 页中的“使用断点和步进”结尾处所讨论的模式相同的模式，即，用户在出错之前在某个点停止行为异常的程序，然后单步执行代码，将代码的本意与代码实际的行为相比较。主要差异在于，查找出错之前的点的过程要复杂一些。

## 使用断点脚本修补代码

在第 13 页中的“使用断点和步进”中，您发现了一个错误，一个空行产生了一个 NULL 首标记并导致一个 SEGV。以下是快速解决此错误的一种方法：

1. 删除在前几部分中创建的所有断点。
2. 在 "Debug Executable"（调试可执行文件）对话框中删除 <in 参数。
3. 在 interp.cc 中的第 130 行处开启/关闭一个行断点：

```
Disassembly x  interp.cc x  main.cc x  debugger.cc x
120  Interp::dispatch(char *line)
121  {
122      const int MAXARGS = 8;
123      const char *DELIMITERS = " \\t\\n";           // "word" delimiters
124
125      // break 'line' into "word"s and store them in 'argv'
126      char *argv[MAXARGS+1];                       // +1 for sentinel NULL
127      int argc = 0;
128
129      char *token = strtok(line, DELIMITERS);
130      argv[argc++] = token;                         // first token
131
132      while (token = strtok(NULL, DELIMITERS)) { // rest of the tokens
133          if (argc >= MAXARGS) {
134              printf("Too many arguments at '%s'\\n", token);
135              return;
```

4. 在 "Breakpoints"（断点）窗口中，右键单击刚刚创建的断点（较新的断点添加在底部），然后选择 "Customize"（自定义）。



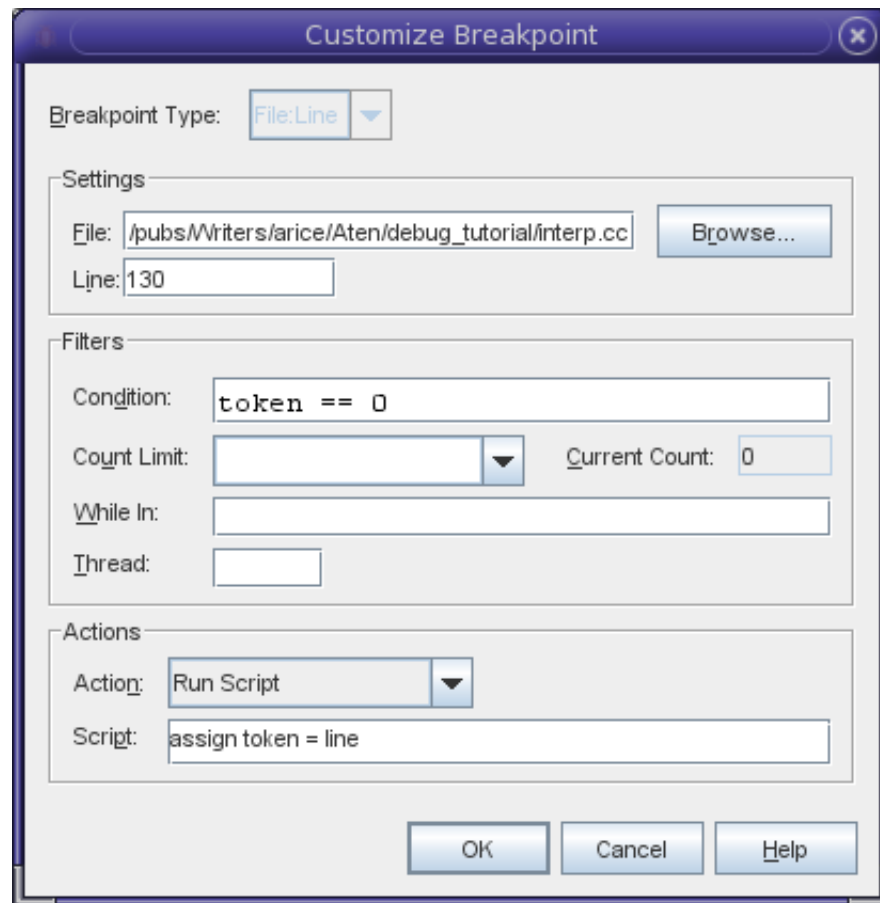
5. 在 "Customize Breakpoint" (自定义断点) 对话框中, 在 "Condition" (条件) 字段中键入 `token == 0`。
6. 从 "Action" (操作) 下拉式列表中选择 "Run Script" (运行脚本)。
7. 在 "Script" (脚本) 字段中, 键入 `assign token = line`。

---

提示 - 为什么不能输入 `assign token = "dummy"`? 因为 dbx 无法在已调试进程中分配 `dummy` 字符串。另一方面, 已知 `line` 等于 ""。

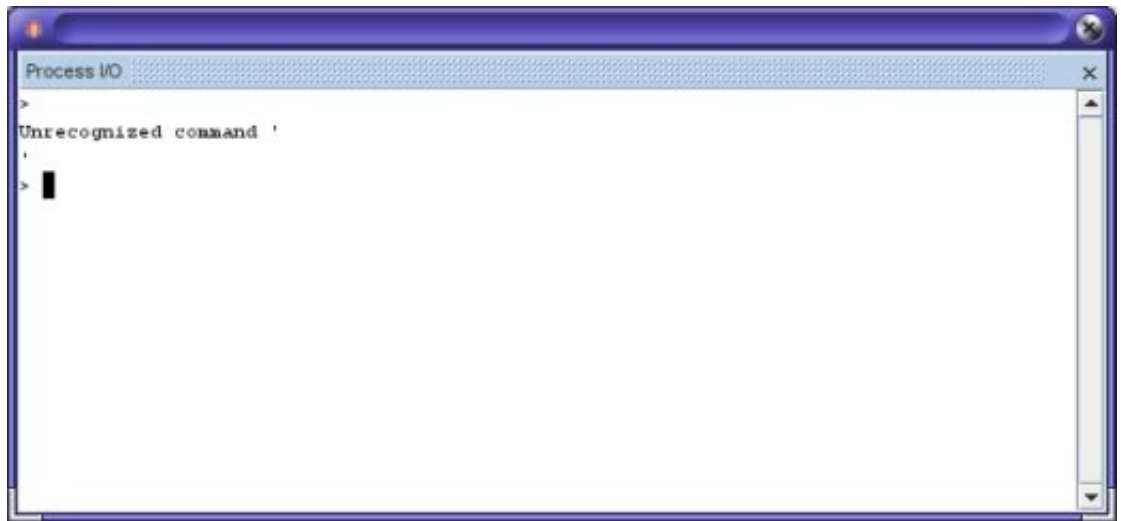
---

对话框应如下所示:



8. 单击 "OK" (确定)。

现在, 如果您运行程序并输入一个空行, 不会发生崩溃, 其行为将如下所示:



如果您查看 dbxtool 发送给 dbx 的命令，可以更加清楚地了解该窗口的工作原理。

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

版权所有 ©2010 本软件和相关文档是根据许可证协议提供的，该许可证协议中规定了关于使用和公开本软件和相关文档的各种限制，并受知识产权法的保护。除非在许可证协议中明确许可或适用法律明确授权，否则不得以任何形式、任何方式使用、拷贝、复制、翻译、广播、修改、授权、传播、分发、展示、执行、发布或显示本软件和相关文档的任何部分。除非法律要求实现互操作，否则严禁对本软件进行逆向工程设计、反汇编或反编译。

此文档所含信息可能随时被修改，恕不另行通知，我们不保证该信息没有错误。如果贵方发现任何问题，请书面通知我们。

如果将本软件或相关文档交付给美国政府，或者交付给以美国政府名义获得许可证的任何机构，必须符合以下规定：

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

本软件或硬件是为了在各种信息管理应用领域内的一般使用而开发的。它不应被应用于任何存在危险或潜在危险的应用领域，也不是为此而开发的，其中包括可能会产生人身伤害的应用领域。如果在危险应用领域内使用本软件或硬件，贵方应负责采取所有适当的防范措施，包括备份、冗余和其它确保安全使用本软件或硬件的措施。对于因在危险应用领域内使用本软件或硬件所造成的一切损失或损害，Oracle Corporation 及其附属公司概不负责。

Oracle 和 Java 是 Oracle 和/或其附属公司的注册商标。其他名称可能是各自所有者的商标。

AMD、Opteron、AMD 徽标以及 AMD Opteron 徽标是 Advanced Micro Devices 的商标或注册商标。Intel 和 Intel Xeon 是 Intel Corporation 的商标或注册商标。所有 SPARC 商标均是 SPARC International, Inc 的商标或注册商标，并按许可证的规定使用。UNIX 是通过 X/Open Company, Ltd 授权的注册商标。

本软件或硬件以及文档可能提供了访问第三方内容、产品和服务的方式或有关这些内容、产品和服务的信息。对于第三方内容、产品和服务，Oracle Corporation 及其附属公司明确表示不承担任何种类的担保，亦不对其承担任何责任。对于因访问或使用第三方内容、产品或服务所造成的任何损失、成本或损害，Oracle Corporation 及其附属公司概不负责。

821-2517

Oracle Corporation 500 Oracle Parkway, Redwood City, CA 94065 U.S.A.

