Oracle Solaris Studio 12.3: dbxtool チュートリアル

2011 年 12 月

ORACLE

概要

このチュートリアルでは、「バグを含んだ」プログラム例を使用して、dbx デバッガ用のスタンドアロングラフィカルユーザーインタフェース (GUI) である dbxtool の効果的な使用方法について説明します。最初に基本的な機能について説明し、その後、より詳細な機能について説明していきます。

プログラム例

このチュートリアルでは、dbx デバッガの単純で、やや擬似的なシミュレーションを使用します。この C++ プログラムのソースコードは、「Oracle Solaris Studio 12.3 Sample Applications」の Web ページ (http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-samples-1408618.html) で、サンプルアプリケーションの zip ファイルから入手できます。

1. まだ行なっていない場合、サンプルアプリケーションの zip ファイルをダウンロードし、選択した場所にファイルを展開します。debug_tutorial アプリケーションは、SolarisStudioSampleApplications ディレクトリの Debugger サブディレクトリにあります。
2. プログラムを構築します。

```
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
```

プログラムは次のモジュールから構成されます。

cmd.h	cmd.cc	Cmd クラス、デバッガコマンドを実装するためのベース
interp.h	interp.cc	Interp クラス、簡単なコマンドインタプリタ
debugger.h	debugger.cc	Debugger クラス、デバッガの主要なセマンティクスの模倣
cmds.h	cmds.cc	さまざまなデバッグコマンドの実装
main.h	main.cc	main() 関数とエラー処理Interp をセットアップし、さまざまなコマンドを作成して、それらのコマンドを Interp に割り当てます。Interp を実行します。

プログラムを実行して、dbx コマンドをいくつか試します。

```
$ a.out
> display var
will display 'var'
```

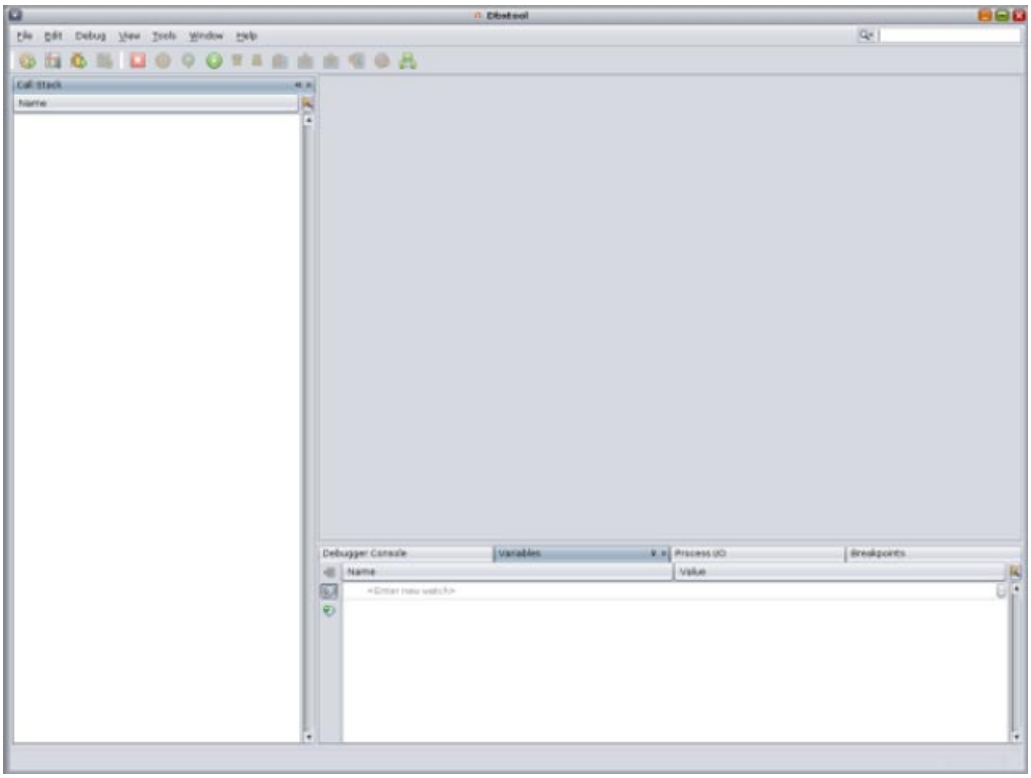
```
> stop in X
> run running ...
stopped in X
var = {
        a = '100'
        b = '101'
        c = '<error>'
        d = '102'
        e = '103'
        f = '104'
}
> quit
Goodby
$
```

dbxtoolの設定

次のように入力して、dbxtoolを起動します。

installation_directory/bin/dbxtool

最初にdbxtoolを起動したときに、ウィンドウは次のように表示されます。

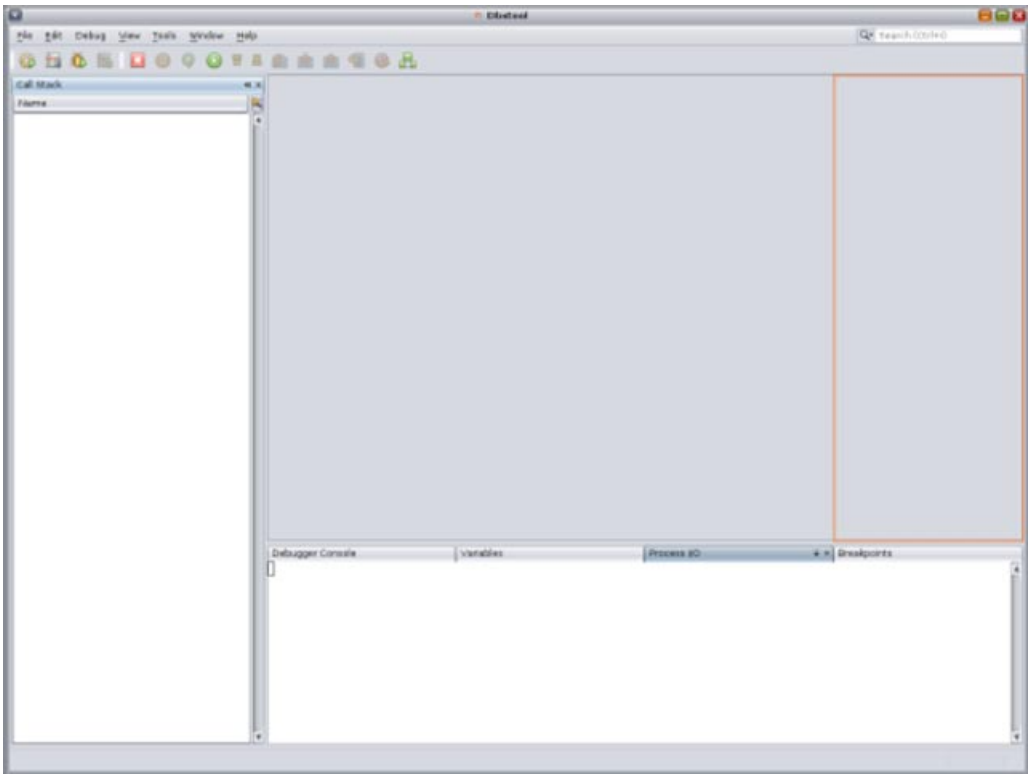


このチュートリアルをWebブラウザで参照している場合は、おそらくそれだけで画面の半分が使用されるので、dbxtoolのアプリケーションのサイズを画面の半分になるようにカスタマイズすると便利です。

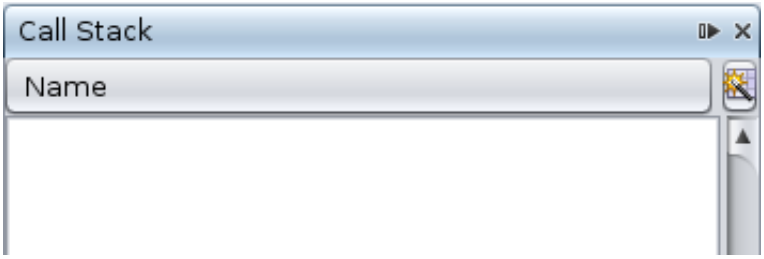
次に、dbxtool のさまざまなカスタマイズ例を示します。

- ツールバーアイコンを小さくする:
  - ツールバーの任意の場所を右クリックして、「小さいツールバーアイコン」を選択します。
- 「呼び出しスタック」ウィンドウを隠す:

1. 「呼び出しスタック」ウィンドウのヘッダーをクリックして、ウィンドウを下および右へドラッグします。赤色のアウトラインがこの位置にあっても気にしないでください。

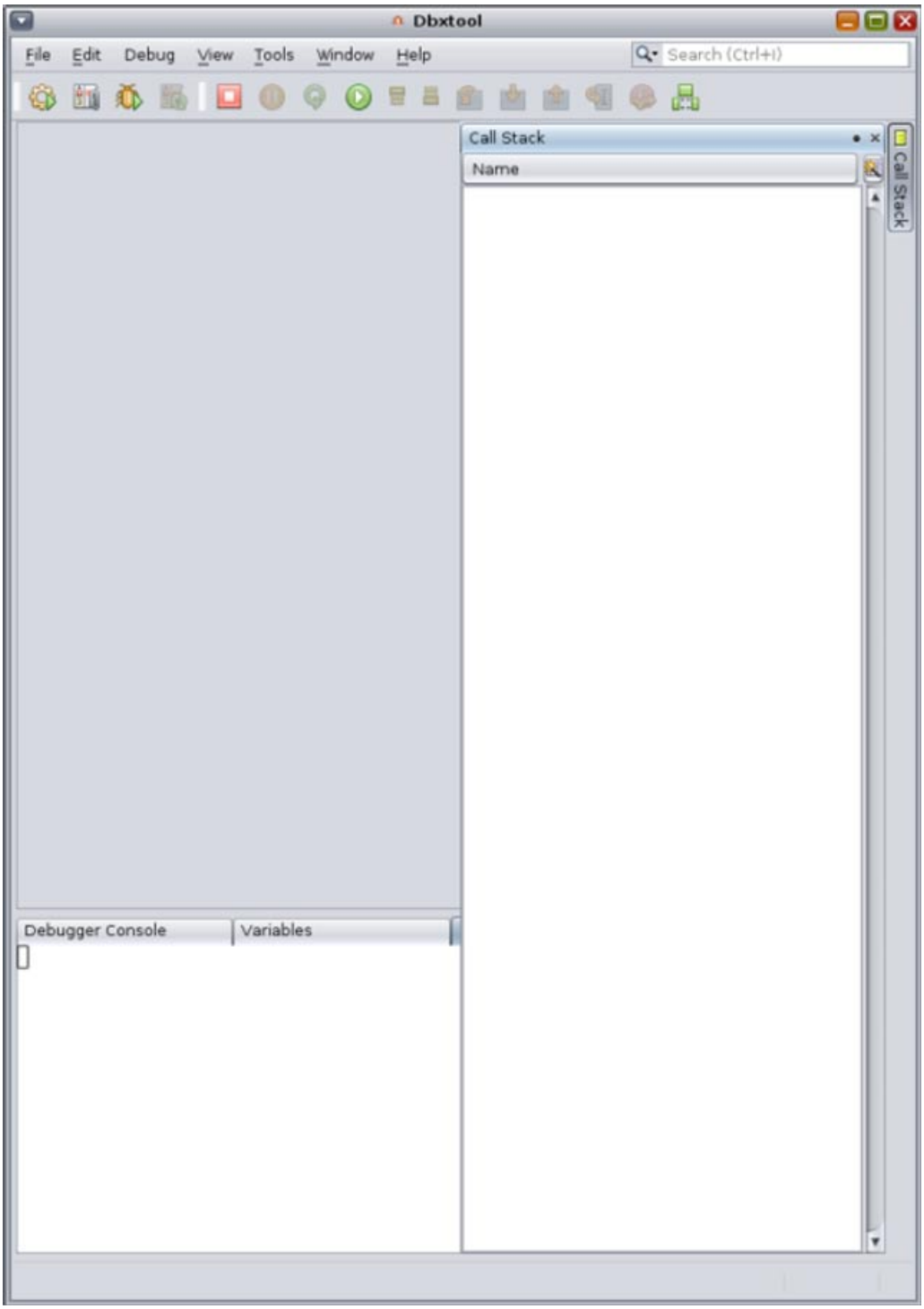


2. ここで、「呼び出しスタック」ウィンドウのヘッダーの右側の矢印をクリックします。

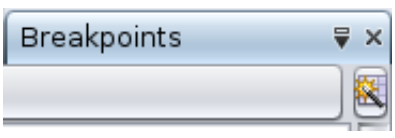


「呼び出しスタック」ウィンドウは右マージンに最小化されます。

3. 最小化された「呼び出しスタック」アイコンにカーソルを合わせると「呼び出しスタック」ウィンドウが最大化され、カーソルを別のウィンドウに移すと最小化に戻ります。最小化された「呼び出しスタック」アイコンをクリックすると「呼び出しスタック」ウィンドウが最大化され、もう一度クリックするとまた最小化されます。
4. これでメインウィンドウを画面の半分に縮小できるはずです。



- 「ブレークポイント」ウィンドウを最小化する:
  1. 「ブレークポイント」タブをクリックします。
  2. タブの下矢印をクリックして、「ブレークポイント」ウィンドウを最小化します。



- 「プロセス入出力」ウィンドウの合体を解除する:
  1. 「プロセス入出力」ウィンドウのヘッダーをクリックしたままにして、そのウィンドウをdbxtoolウィンドウの外側にドラッグして、デスクトップ上にドロップします。これでdbxtoolウィンドウの他のタブに簡単にアクセスしながら、デバッグしているプログラムの入出力を簡単に相互作用させることができます。
  2. dbxtoolウィンドウで「プロセス入出力」ウィンドウを再度合体させるには、「プロセス入出力」ウィンドウを右クリックして、「ウィンドウを合体」を選択します。
- エディタ内のフォントサイズを設定する。「エディタ」ウィンドウにソースコードが表示されたあと、次を実行します。
  1. 「ツール」>「オプション」を選択します。
  2. 「オプション」ウィンドウで、「フォントと色」カテゴリを選択します。
  3. 「構文」タブで、「言語」ドロップダウンリストからすべての言語が選択されていることを確認します。
  4. 「フォント」テキストボックスの横の参照ボタンをクリックします。
  5. 「フォント選択」ダイアログボックスで、フォント、スタイル、およびサイズを設定し、「OK」をクリックします。
  6. 「オプション」ウィンドウで、「OK」をクリックします。
- 端末ウィンドウ内のフォントサイズを設定する。「デバッガコンソール」ウィンドウと「プロセス入出力」ウィンドウはANSI端末エミュレータです。
  1. 「ツール」>「オプション」を選択します。
  2. 「オプション」ウィンドウで、「その他各種」カテゴリを選択します。
  3. 「端末」タブをクリックします。
  4. 「フォントサイズ」などの設定を選択して、「タイプへ」をクリックします。
  5. 「OK」をクリックします。

コアダンプの診断

使用状況に合うように`dbxtool`を設定しました。次に、バグをいくつか見つけてみましょう。

プログラム例をもう一度実行します。ただし、今回はコマンドを入力しないで改行キーを押します。

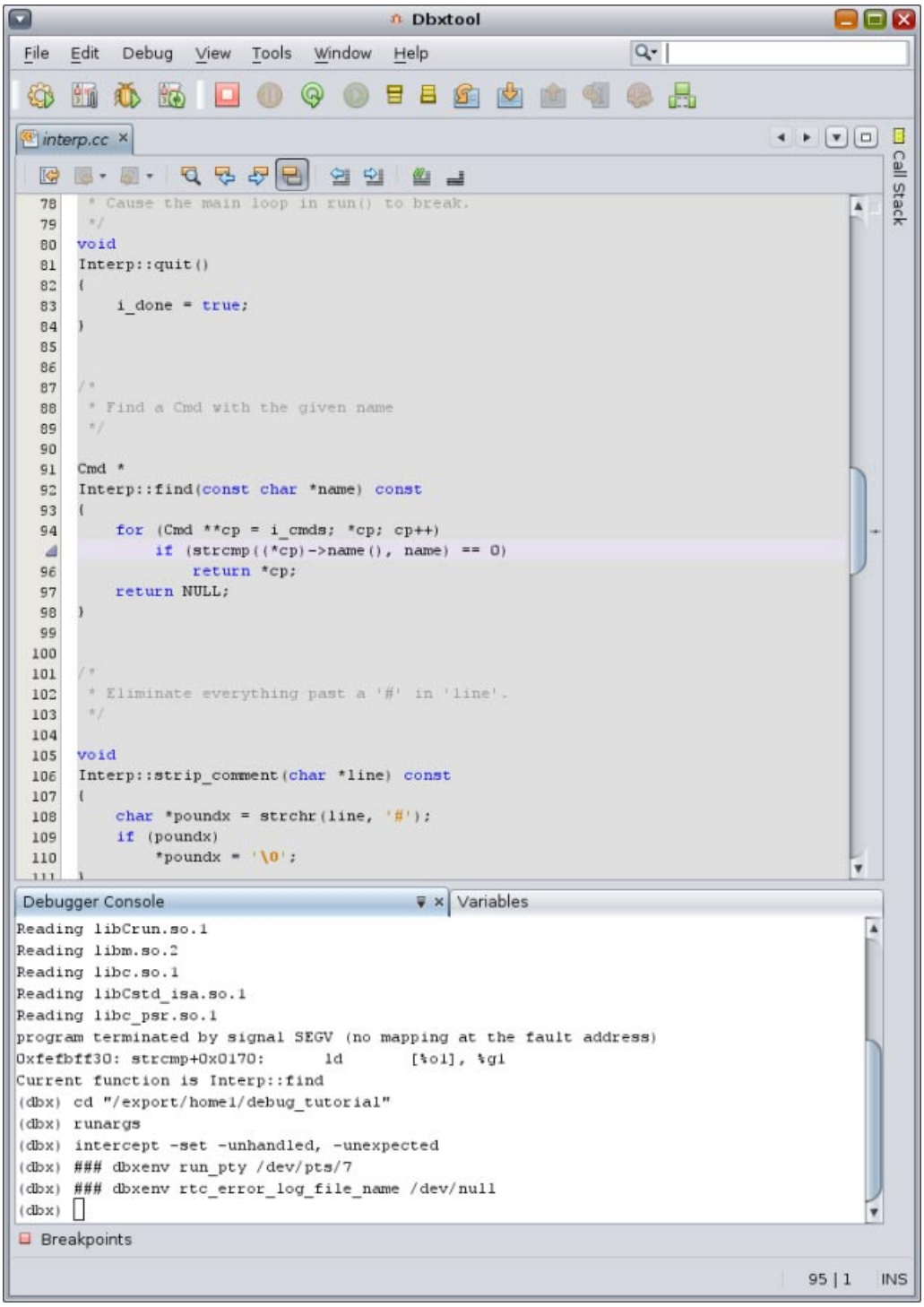
```
$ a.out
> display var
will display 'var'
>
Segmentation Fault (core dumped)
$
```

ここで、実行可能ファイルおよびコアファイルを指定して`dbxtool`を起動します。

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

ヒント – `dbxtool`コマンドは`dbx`コマンドと同じ引数を受け入れていることに注目してください。

`dbxtool`は、次のような内容を表示します。

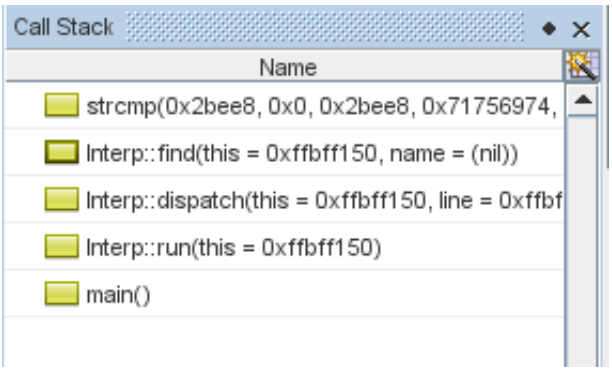


ここでいくつかの注意事項を挙げます。

- 「デバッガコンソール」ウィンドウに、次のようなメッセージが表示されます。

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

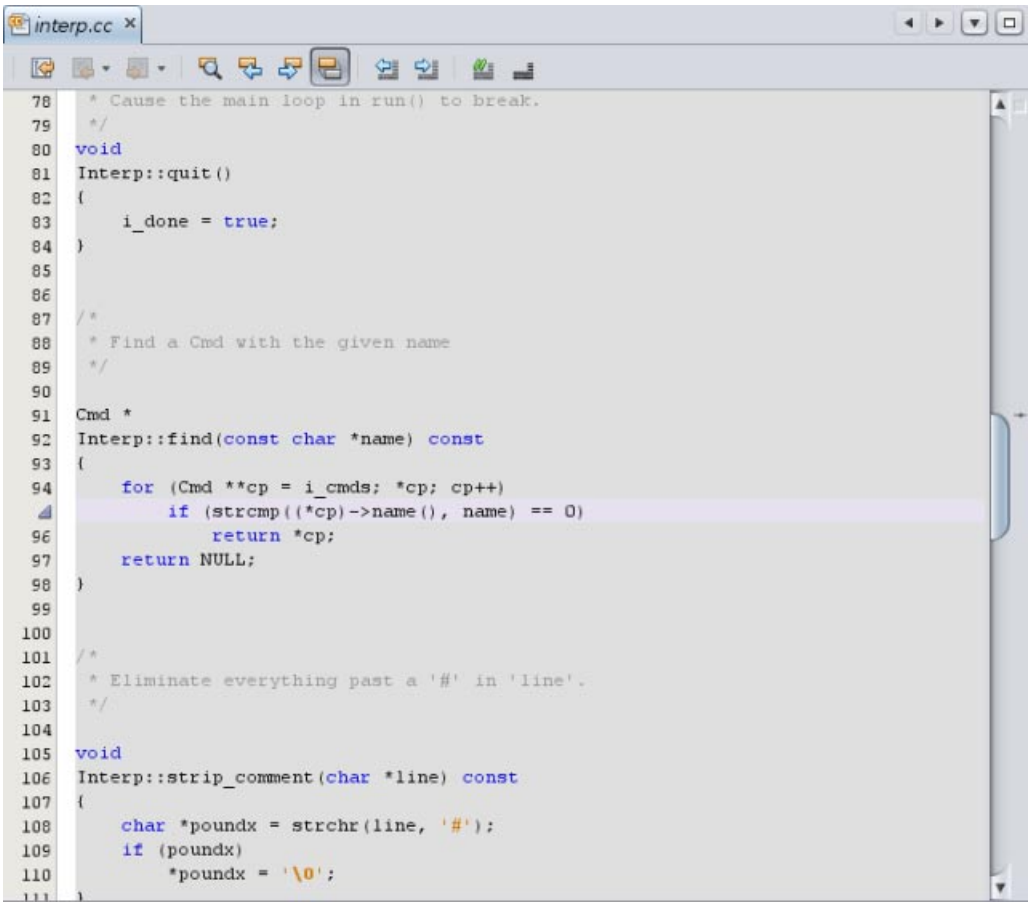
- `strcmp()` 関数で `SEGV` が発生していても、`dbx` はデバッグ情報のある最初の関数フレームを自動的に表示します。下図のように、「呼び出しスタック」ウィンドウのスタックトレースでは、アイコンの周囲を強調表示して現在のフレームを示します。



「呼び出しスタック」ウィンドウにパラメータ名と値が表示されます。これで `strcmp()` に渡された 2 番目のパラメータが 0x0 で、name の値は NULL であることがわかります。

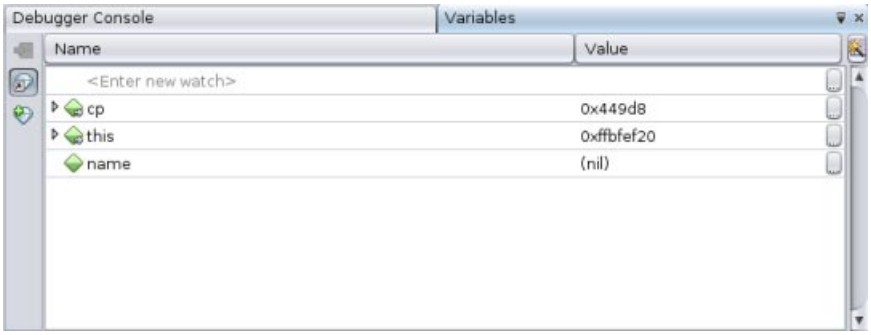
- 「エディタ」ウィンドウ内のラベンダー色のストライプと三角印は、`strcmp()` を呼び出している場所を示しています (実際にエラーの発生した場所は、緑のストライプと矢印で示されます)。

ヒント – パラメータの値が表示されない場合は、`dbx` 環境変数 `stack_verbose` が `.dbxrc` ファイルでオンに設定されていることを確認してください。「呼び出しスタック」ウィンドウ内を右クリックして、「詳細」チェックボックスを選択してチェックマークを追加し、ウィンドウの詳細モードをオンにすることもできます。



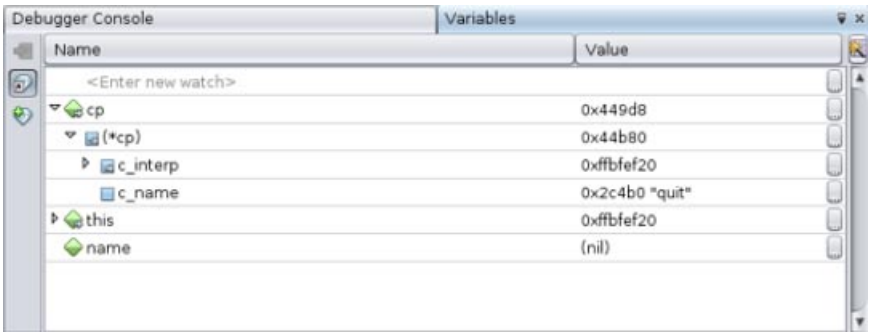
関数は、パラメータとして不正な値を渡される場合は通常失敗します。ここに、`strcmp()` に渡された値を確認するためのいくつかの方法があります。

- 「変数」ウィンドウのパラメータの値を確認します。
  「変数」タブをクリックします。



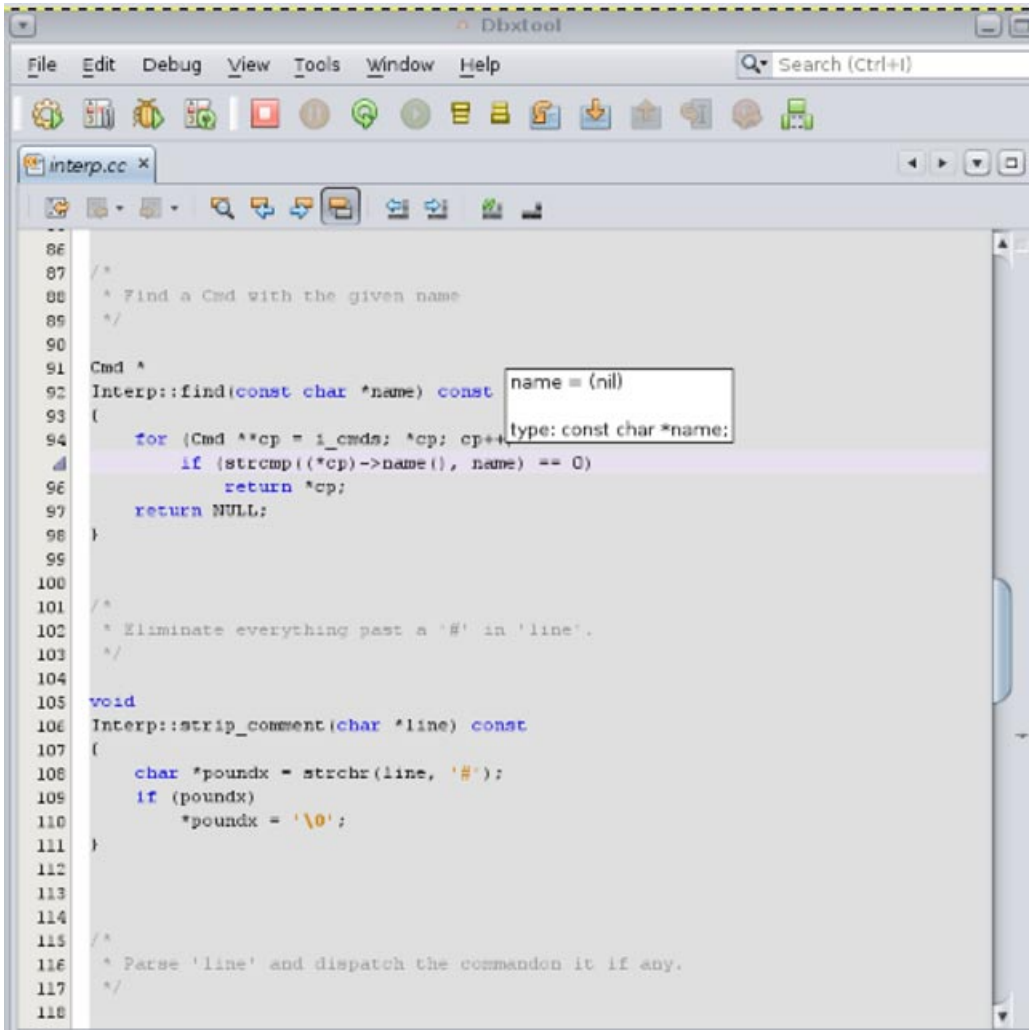
name の値が NULL であることに注意してください。その値が SEGV の原因である可能性が高いのですが、もう一方のパラメータ `(*cp)->name()` の値を確認しましょう。

「変数」ウィンドウで、`cp` ノードを展開し、次に `(cp*)` ノードも展開します。この中の `name` は "quit" になっており、これは問題ありません。



ヒント – `*cp` ノードを展開して追加の変数が表示されない場合は、`.dbxrc` ファイルの `dbx` 環境変数 `output_inherited_members` がオンに設定されていることを確認します。ウィンドウを右クリックして、「継承されたメンバー」チェックボックスをオンにしてチェックマークを追加して、継承されたメンバーの表示をオンにすることもできます。

- バルーン評価を使用して、パラメータの値を確認します。「エディタ」ウィンドウで、カーソルを `strcmp()` に渡されている変数 name の上に置きます。ヒントが表示され、name の値が NULL であるとわかります。



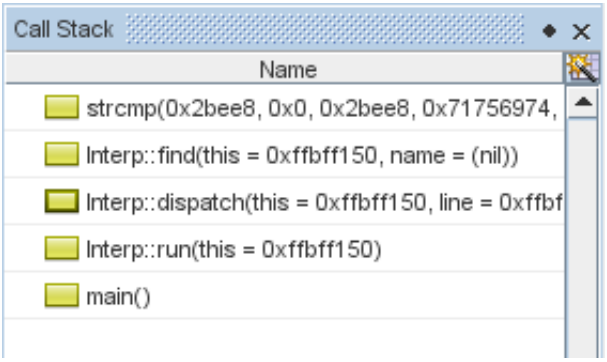
バルーン評価を使用すると、`(*cp)->name()` のような式上にカーソルを置くこともできます。ただし、ここでは式に関数呼び出しが含まれているため、カーソルを置くことはできません。

ヒント –
関数呼び出しを含む式のバルーン評価が無効になるのは、次の理由からです。

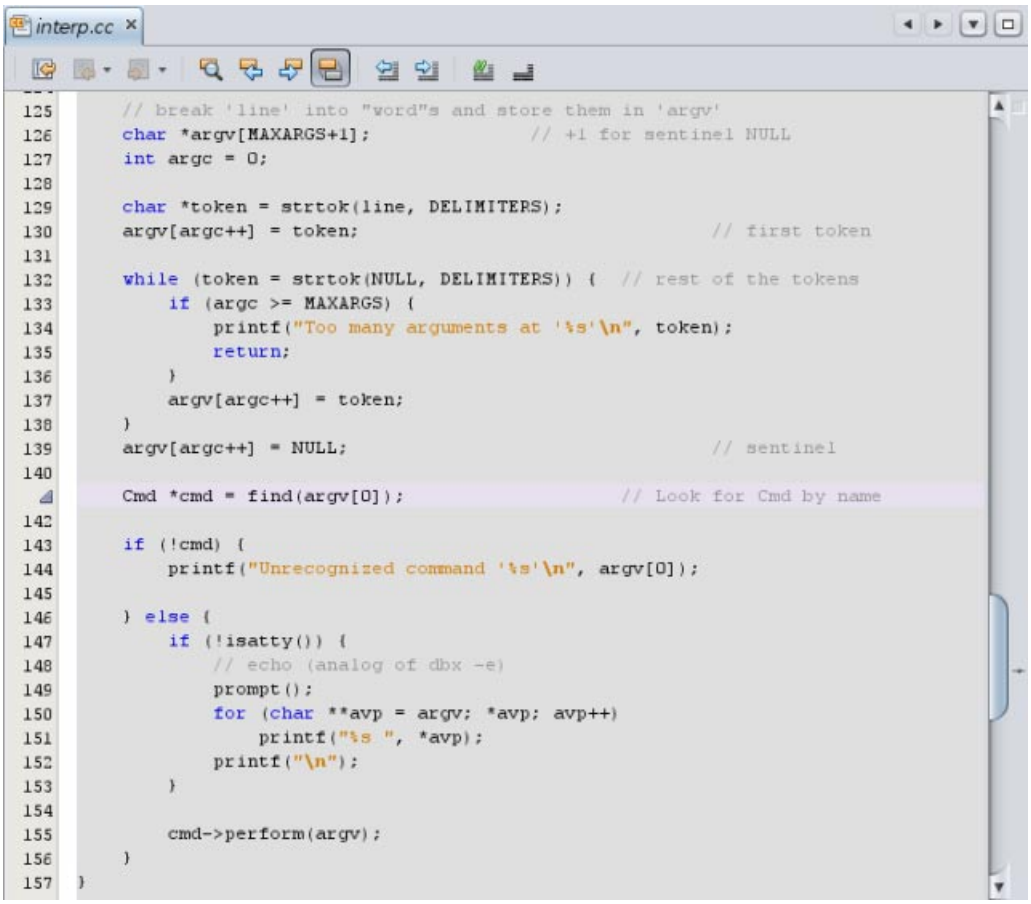
- デバッグしているのがコアファイルである。
- 関数呼び出しには副作用のある場合があり、それが「エディタ」ウィンドウにカーソルを置いてしまうことにより発生するのは困る。

ここで、name の値が NULL であるべきではないことは非常に明白です。しかし、どのコードがこの不正な値を `Interp::find()` に渡したのでしょうか。これを調べるには、次の操作を行います。

- 「デバッグ」>「スタック」>「呼び出し元を現在に設定」を選択するか、またはツールバー上にある「呼び出し元を現在に設定」ボタン  をクリックして、呼び出しスタックを上に移動させます。



- 「呼び出しスタック」ウィンドウで、`Interp::dispatch()` に対応するフレームをダブルクリックします。「エディタ」ウィンドウで、対応するコードが強調表示されます。



このコードは未知です。また、少し見ただけでは、`argv[0]` の値が NULL であること以外はわかりません。

ブレークポイントおよびステップ動作を使用して、この問題を動的にデバッグする方がよい場合があります。

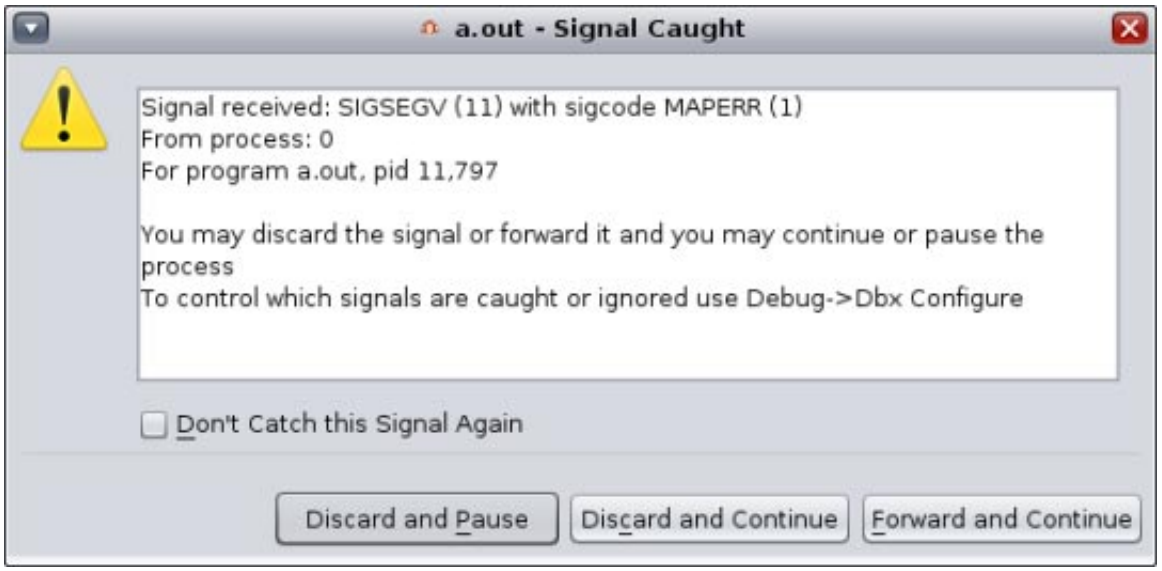
ブレークポイントとステップ動作の使用法

ブレークポイントにより、バグを示す少し前でプログラムを停止したり、何が間違っているかを検出するためにコードをステップスルーすることができます。

「プロセス入出力」ウィンドウの合体をまだ解除していなければ、このタイミングで行うとよいでしょう。

以前はこのプログラムをコマンド行から実行しました。ここで、`dbxtool` でプログラムを実行して、バグを再現します。

1. ツールバーで「実行」ボタン  をクリックするか、「デバッガコンソール」ウィンドウで `run` と入力します。
2. 「プロセス入出力」ウィンドウで改行キーを押します。このとき、警告ボックスが SEGV について知らせます。

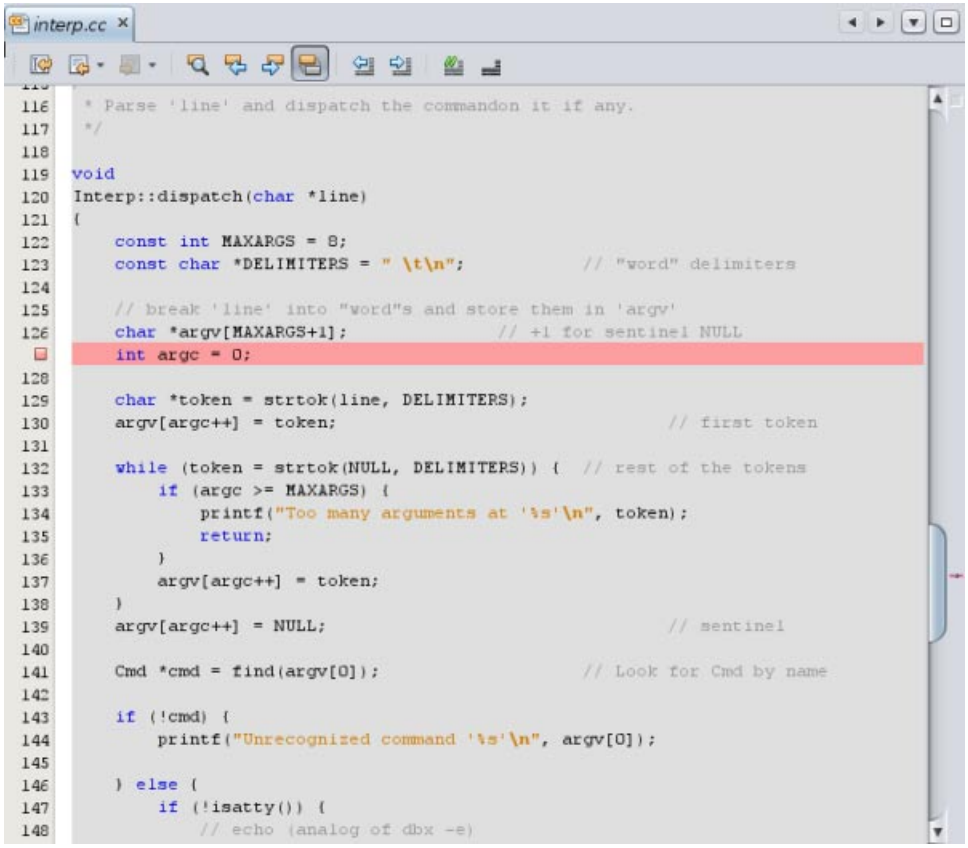


3. 警告ボックスで、「破棄して一時停止」をクリックします。エディタ (Editor) ウィンドウで、`Interp::find()` の `strcmp()` への呼び出しが再度強調表示されます。
4. ツールバーの「呼び出し元を現在に設定」ボタン  をクリックして、`Interp::dispatch()` に以前に表示された不明なコードに移動します。ここで、`find()` への呼び出しの少し前にブレークポイントを設定できます。後で、間違っている理由を調べるため、コードをステップスルーできます。

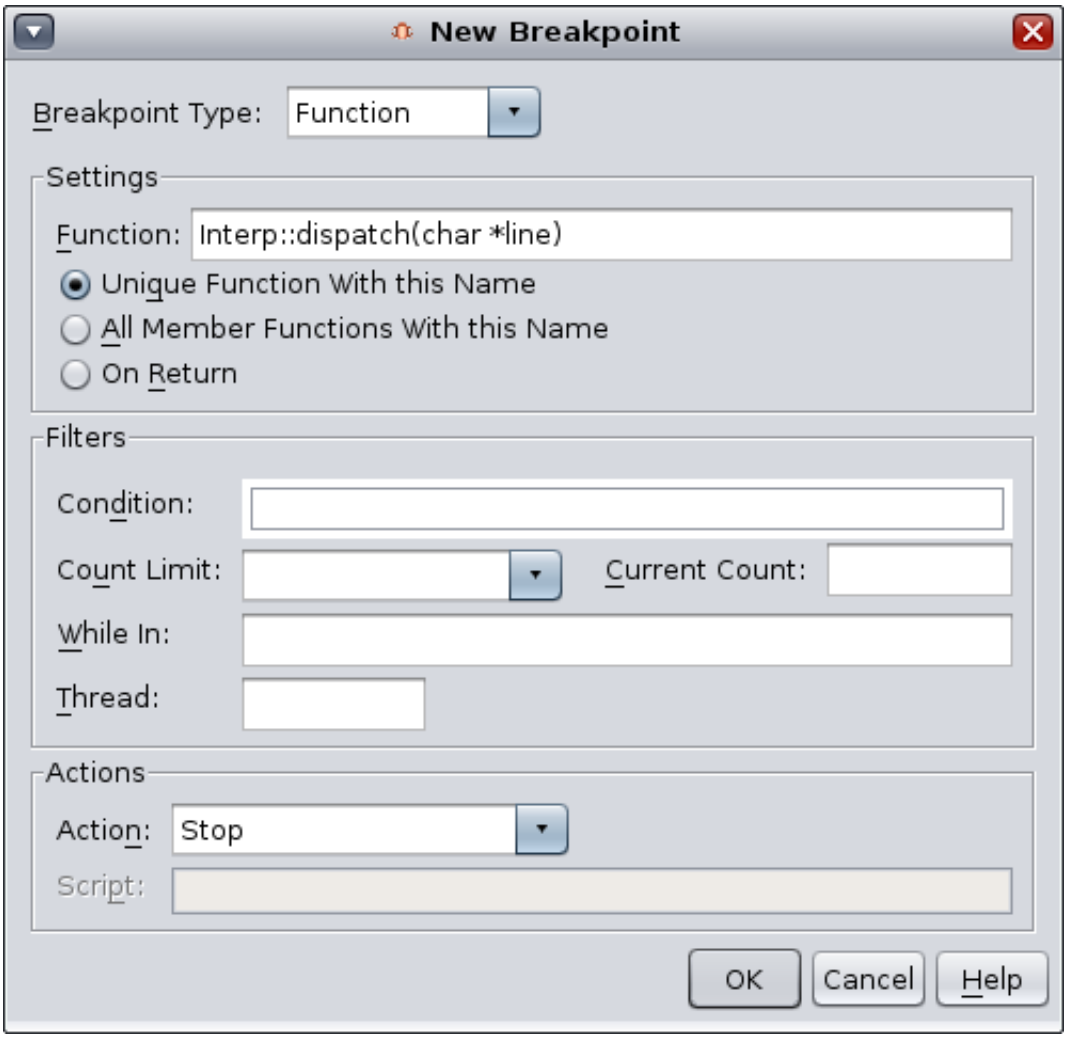
ブレークポイントを設定する

ブレークポイントを設定するには、いくつかの方法があります。行番号が表示されていなければ、最初に左マージン内を右クリックし、「行番号を表示」チェックボックスをオンにして、エディタで行番号を有効にします。

- 127 行目の横の左マージン内をクリックして、行ブレークポイントを切り替えます。



- 次の操作を実行して、関数ブレークポイントを設定します。
  1. 「エディタ」ウィンドウで、「`Interp::dispatch`」を選択します。
  2. 「デバッグ」>「新規ブレークポイント」を選択するか、または右クリックして「新規ブレークポイント」を選択します。「新規ブレークポイント」ダイアログボックスが表示されます。



選択した関数の名前が「関数」フィールドに表示されています。

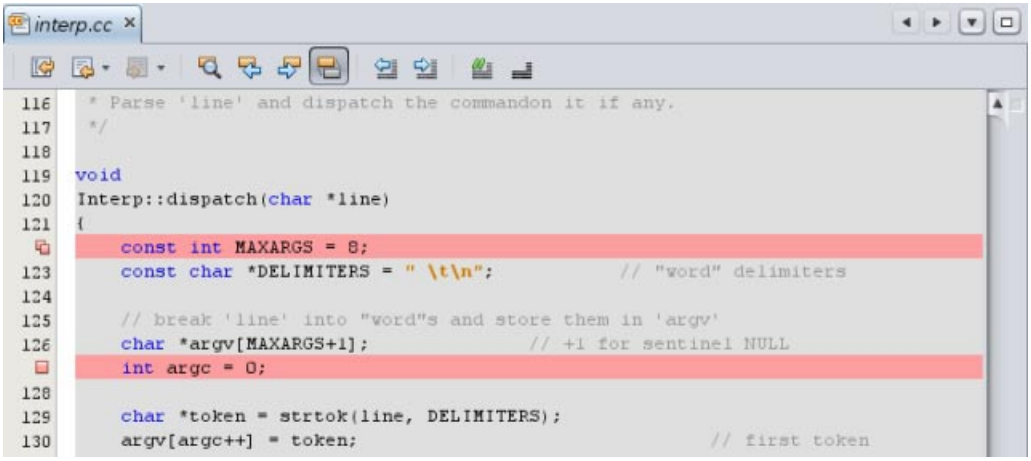
3. 「OK」をクリックします。

- dbx コマンド行から関数ブレークポイントを設定するのが一番簡単です。これを実行するには、「デバッガコンソール」ウィンドウで stop in コマンドを入力します。

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

「`Interp::dispatch`」と入力する必要がないことがわかります。関数名を指定するだけで十分です。

現在は、エディタが乱雑になっています。



「ブレークポイント」ウィンドウを使用して、この乱雑さをクリーンアップできます。

1. 「ブレークポイント」タブをクリックします (または前に最小化している場合は最大化します)。
2. 行ブレークポイントおよび関数ブレークポイントの 1 つを選択して、右クリックして「削除」を選択します。

関数ブレークポイントの利点

エディタで切り替えて、行ブレークポイントを設定することは直感的である場合があります。ただし、多くの dbx ユーザーは、次の理由で関数ブレークポイントの方を好みます。

- 多くの場合、「デバッガコンソール」ウィンドウで「si dispatch」と入力するのがもっとも簡単です。これは、エディタでファイルを開き、ブレークポイントを配置する行までスクロールする手間を省きます。
- エディタで任意のテキストを選択して、関数ブレークポイントを作成できます。したがって、ファイルを開くのではなく、呼び出しサイトから関数上にブレークポイントを設定できます。

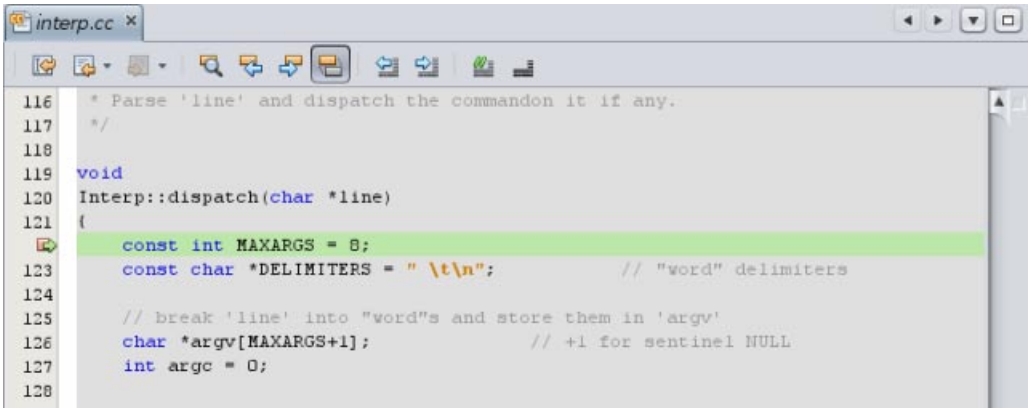
ヒント – si は、stop in の別名です。ほとんどの dbx ユーザーは多数の別名を定義し、その定義内容を dbx の設定ファイル ~/.dbxrc に置きます。次に、一般的な例を示します。

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

- 関数ブレークポイントの名前は、「ブレークポイント」ウィンドウに表示されています。行ブレークポイントの名前は表示されていません。したがって、たとえば interp.cc:127 に何が記述されているかはわかりません。(実際には、「ブレークポイント」ウィンドウの行ブレークポイントを右クリックし、「ソースに移動」を選択するか、またはそのブレークポイントをダブルクリックすると、127 行目の内容を検索できます。)
- 関数ブレークポイントの方が、より持続します。dbxtool はブレークポイントを持続させるため、コードを編集したりソースコード制御のマージを行なったりすると、行番号ブレークポイントは簡単にずれてしまうことがあります。関数名の方が編集に耐えられます。

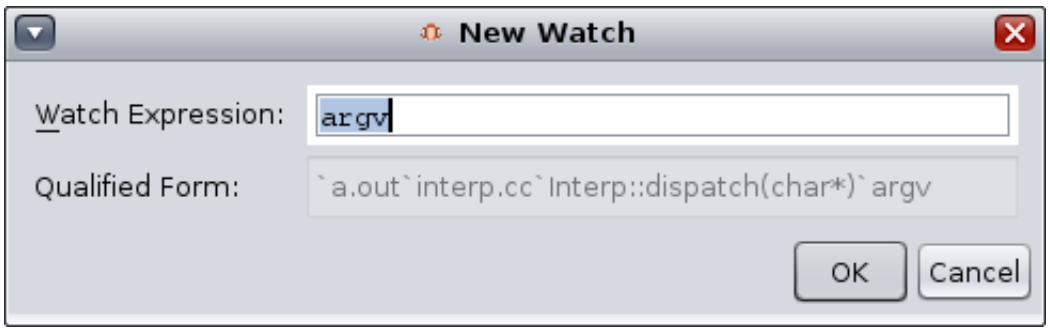
ウォッチポイントとステップ動作の使用法

したがって、現在、Interp::dispatch() に単一のブレークポイントがあります。ふたたび「実行」  をクリックし、「プロセス入出力」ウィンドウの改行キーを押すと、プログラムは実行可能コードを含む dispatch() 関数の最初の行で停止します。

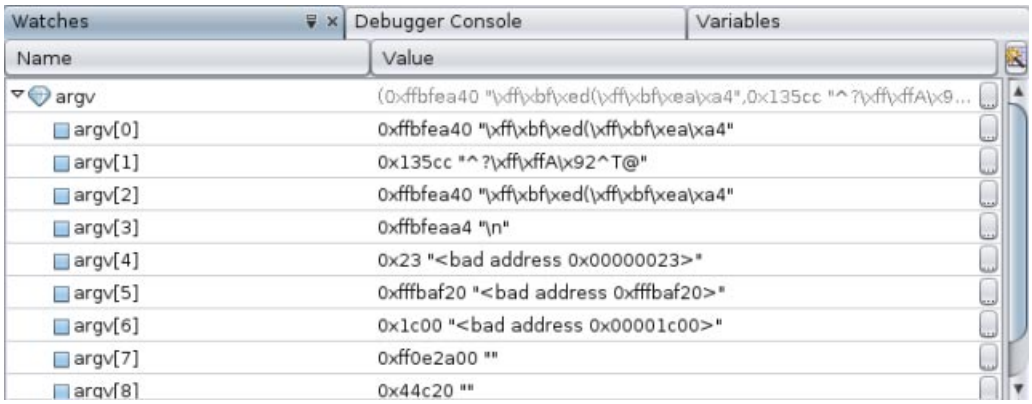


culprit が `find()` へ渡されている `argv[0]` であることがすでにわかっているため、ウォッチポイントを使用して `argv` を監視します。

1. 「エディタ」ウィンドウで `argv` のインスタンスを選択します。
2. 右クリックして「新規ウォッチポイント」を選択します。「新規ウォッチポイント」ダイアログボックスが選択したテキストで表示されます。

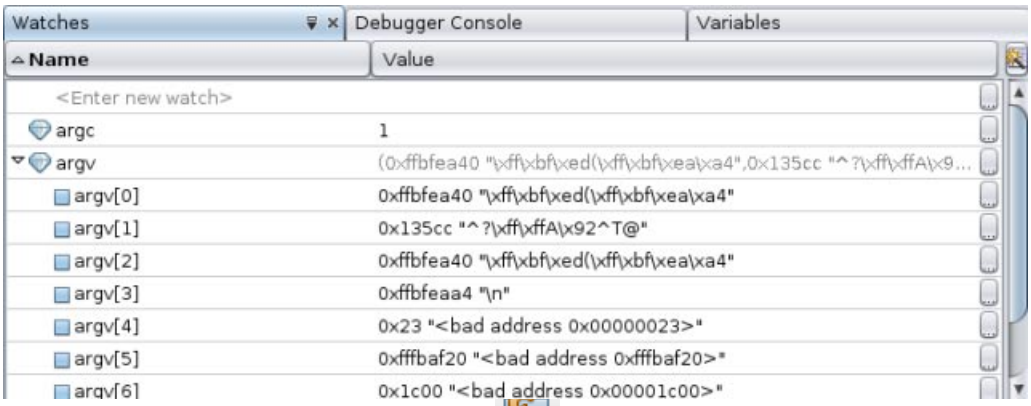


3. 「OK」をクリックします。
4. 「ウィンドウ」>「ウォッチポイント」を選択して、「ウォッチポイント」ウィンドウを開きます。
5. `argv` を展開します。

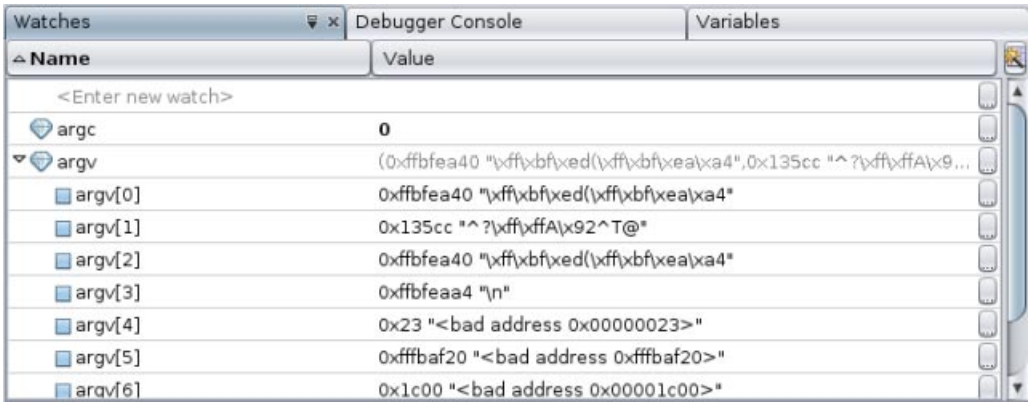


このガベージはいったい何でしょう。argv が初期化されておらず、ローカル変数であるため、前の呼び出しからスタック上に残っているランダムな値を「継承」していることに注意してください。これは問題の原因でしょうか。先走りせずに系統的に進めましょう。

6. 「ステップオーバー」  を 2 回、緑色の PC の矢印が `int argc = 0;` を示すまでクリックします。
7. argc が argv の索引であることは明白であるため、同様にそれに注意して、そのウォッチポイントも作成します。また、今は初期化されておらず、ガベージ値が含まれている可能性もあるので注意してください。
8. argc 秒にウォッチポイントを作成したため、「ウォッチポイント」ウィンドウの argv 下に表示されます。ウィンドウの最初の行に表示されれば問題ありません。ウォッチポイントを削除して、希望の順序で再入力できます。ただし、この場合、使用可能なクイックトリックがあります。「名前」列ヘッダーをクリックすると、列がソートされます。次のような内容が取得されるまでクリックします (ソートトライアングルに注意)。



9. 「ステップオーバー」をクリックします  。argc が、その初期化された値の 0 をどのように表示するか注目してください。太字は、値がちょうど変更されたことを示しています。

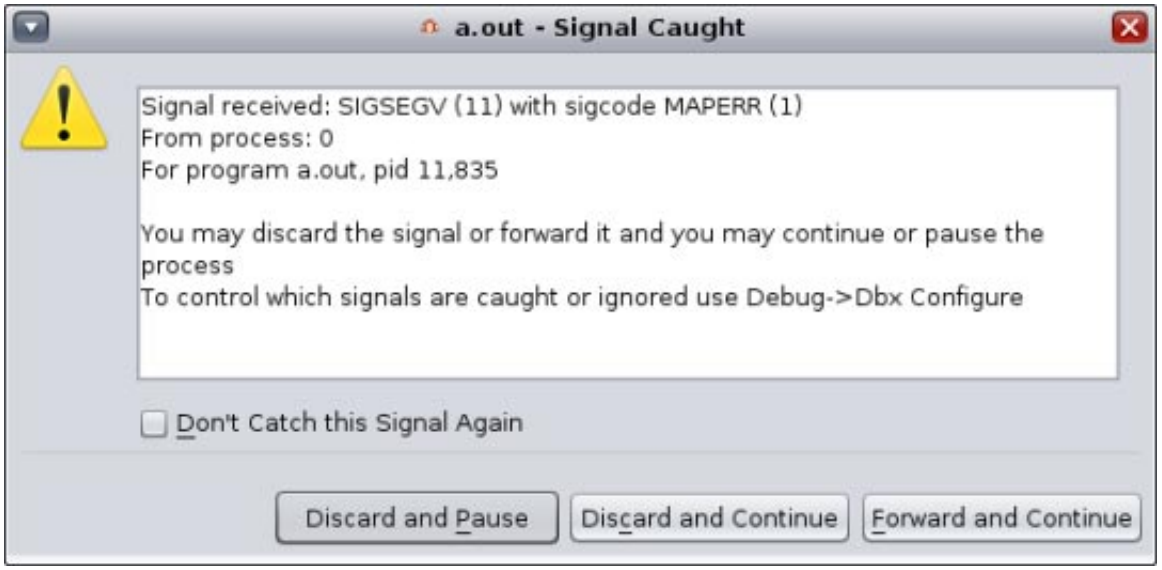


10. アプリケーションが `strtok()` を呼び出します。「ステップオーバー」をクリックして、関数をステップオーバーし、たとえば、バルーン式を使用して、トークンが NULL であることを監視します。

ヒント – `strtok()` は何を行いますか。`strtok` (3) マニュアルページを参照できますが、簡単に言えば、`DELIMETERS/` の 1 つによって区切られたトークンに line などの文字列を分割するのに役立ちます。

---

11. 「ステップオーバー」をもう一度クリックすると、argv にトークンを割り当て、次にループに strtok() への呼び出しがあります。ステップオーバーするにつれて、ループには入らずに (これ以上トークンがない)、逆に NULL が割り当てられます。その割り当てもステップオーバーすると、検出するための呼び出しのしきい値となります。覚えていますか、これはプログラムがクラッシュした場所です。
12. find() に呼び出しをステップオーバーすることによって、プログラムがここでクラッシュすることをダブルチェックします。実際には、「シグナル捕獲」警告ボックスがふたたび表示されます。



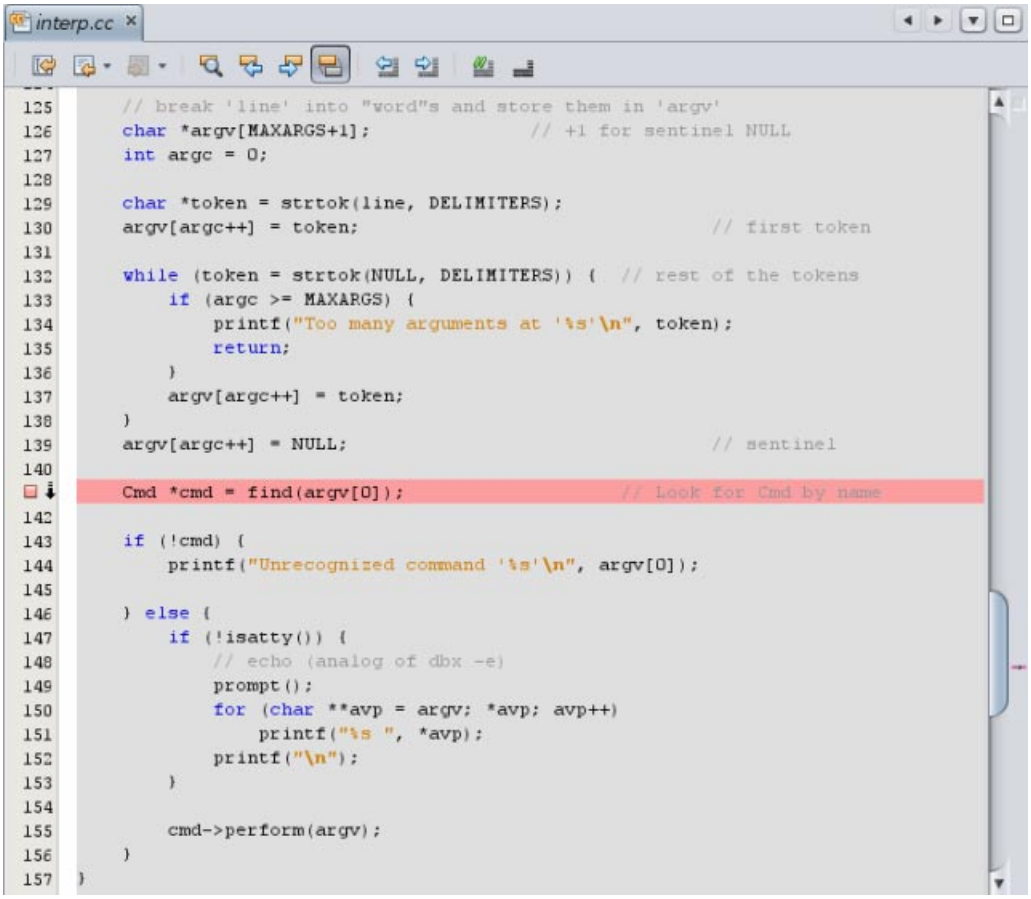
以前のように「破棄して一時停止」をクリックします。

13. したがって、Interp::dispatch で停止した後で、find() への最初の呼び出しは、実際は不正な場所です。

    これは明白であったかもしれませんが、重要な点は、元の場所にすばやく戻れることを示すことです。ここで、方法を説明しましょう。

    a. 「呼び出し元を現在に設定」をクリックします  。
    b. find() の呼び出しサイトで行ブレークポイントを切り替えます。
    c. 「ブレークポイント」ウィンドウを開いて、Interp::dispatch () 関数ブレークポイントを無効にします。

    dbxtool は、このように見えるはずです。



d. 下矢印は、2 つブレークポイントが 141 行に設定され、それらの 1 つが無効であることを示しています。

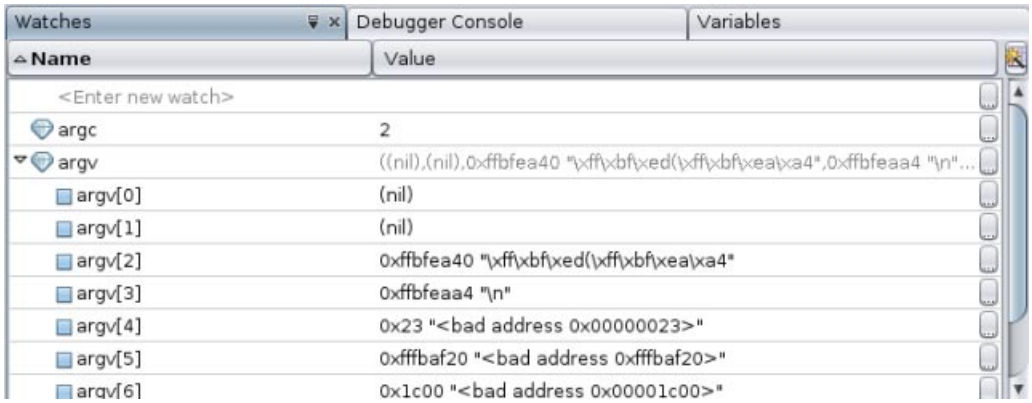
14. 「実行」をクリック  して、「プロセス入出力」ウィンドウの改行キーを押すと、プログラムが `find()` への呼び出しの前に戻って終了します。(「実行」ボタンがいかに再起動するかに注意してください。デバッグ時には、さらに頻繁に再起動します。)

ヒント – たとえば、プログラムを再構築する場合、バグを検出して修正した後で、`dbxtool` を終了して、再起動する必要はありません。「実行」ボタンをクリックすると、`dbx` がプログラム (またはその構成要素) が再コンパイルされていることを検出し、再ロードします。

したがって、デバッグに関する問題で使用しやすいように、デスクトップ上で、おそらく最小化して `dbxtool` を簡単に保持するのがより効率的です。

---

15. それではバグはどこですか。ふたたびウォッチポイントをみてみましょう。



ここで、最初の `strtok()` へ呼び出しが NULL を返し、つまり行が空でありトークンを持っていなかったため、`argv[0]` が NULL であるすばらしい直感的なリープを作成できます。

ヒント – このチュートリアルの残りに進む前に、このバグを修正できますか。

できます。また、改行キーを押さずに、空の行を作成しないことも選択できます。

または、主に、デバッガ下でプログラムを実行している場合、39 ページの「ブレークポイントスクリプトを使用してコードにパッチを適用する」 で説明されるように、デバッガで「パッチを適用」することができます。

コード例の開発者は、この条件に対しておそらくテストし、`Interp::dispatch()` の残りを省略しているはずです。

ディスカッション

上記の例では、調子が悪くなる前の時点で誤動作しているプログラムを停止し、コードが実際に動作している方法でコードの意図を比較してコードをステップスルーする最も一般的なデバッグパターンを示しています。

あらゆるステップ動作と監視なしで、より直接的にバグを検出することができていましたか。実際はできていましたが、最初にブレークポイントを使用するための技術をもっと学習する必要があるでしょう。

高度なブレークポイント技術の使用法

この節では、ブレークポイントを使用するためのいくつかの高度な技術について説明します。

- ブレークポイントカウントの使用法
- 境界ブレークポイントの使用法
- 役立つブレークポイントカウントのピッキング
- ウォッチポイント
- ブレークポイント条件の使用法
- ポップを使用したマイクロ再実行
- 修正と継続機能の使用法

この節、およびプログラム例は、ここで紹介する手順とほぼ同じものを使用して `dbx` で実際に検出されたバグを基にしています。

ソースコードには `in` というサンプル入力ファイルが含まれており、これを使いプログラム例でバグを発生させます。in の内容は次のとおりです。

```
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
```

前の節で検出したものと同じバグが出てしまわないように、ここでは空の行を除いてあります。

この入力ファイルでプログラムを実行すると、出力は次のようになります。

```
$ 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
> quit
Goodby
```

この出力を見て、多いなと思われるかもしれませんが、この例では、プログラムが長大で複雑なためコードを 1 行ずつ見ては追跡することが困難な場合に使用すべき手法を解説することを主眼にしています。

c フィールドの値を表示する箇所で、値が <error> になっていることに注目してください。フィールドに不正なアドレスが含まれていると、このような状況が発生することがあります。

問題

プログラムを 2 度実行した場合に、最初の実行時に取得しなかった追加のエラーメッセージを取得していることに注目してください。

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

error() 関数は、特定の状況におけるサイレントエラーメッセージに変数 err_silent を使用します。たとえば、エラーメッセージを表示するのではなく、表示コマンドの場合に、問題が c = '<error>' として表示されます。

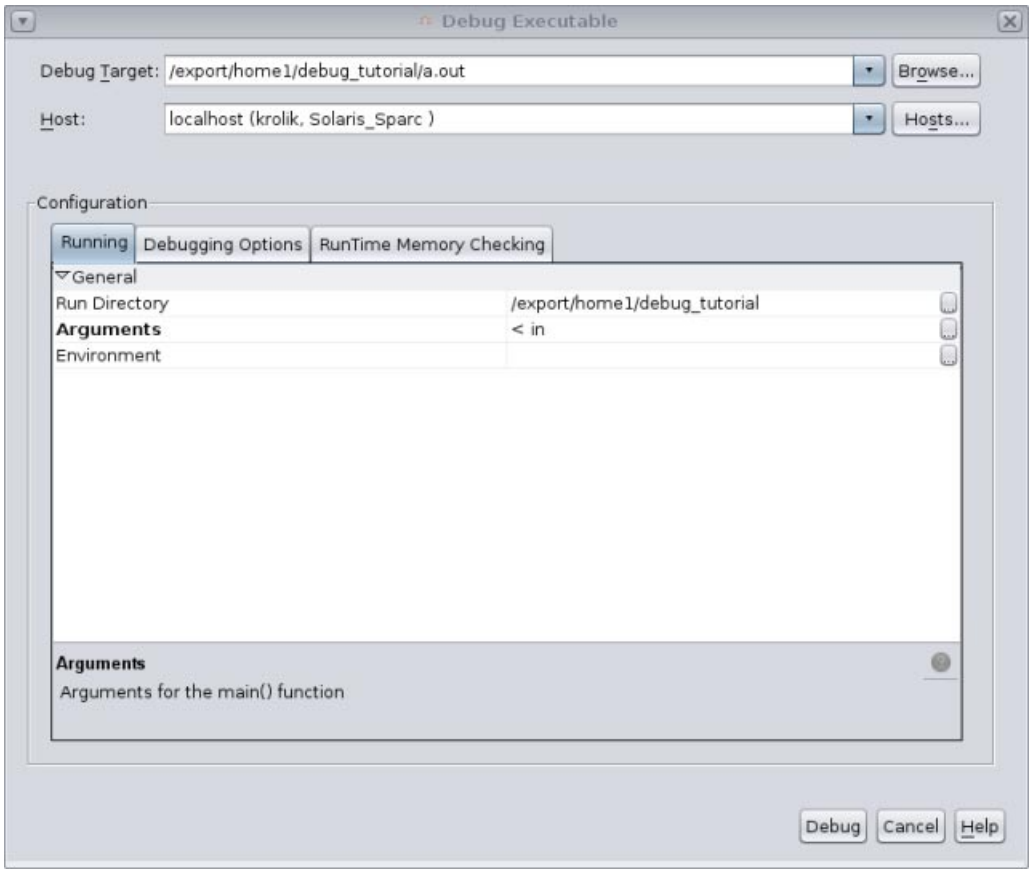
ステップ 1: 再現性

最初のステップは、デバッグターゲットをセットアップし、「実行」をクリックしてバグを簡単に繰り返すことができるように設定することです  。

次のようにプログラムのデバッグを開始します。

1. プログラム例をコンパイルしていない場合は、2 ページの「プログラム例」 の説明に従って実行してください。
2. 「デバッグ」 >「実行可能ファイルをデバッグします」を選択します。
3. 「実行可能ファイルをデバッグします」ダイアログボックスで、実行可能ファイルへのパスを参照するか、入力します。
4. 「引数」フィールドで、次のものを入力します。

   ```
   < in
   ```
5. 実行可能ファイルのパスのディレクトリ部分が「実行ディレクトリ」フィールドに表示されます。
6. 「デバッグ」をクリックします。



現実の状況で、「環境」フィールドに同様に入力したい場合があります。

「デバッグ」>「現在のセッションを設定」を選択して、設定のプロパティーをすべて変更できます。

プログラムをデバッグするときに、`dbxtool` がデバッグターゲットを作成します。「デバッグ」>「最近行なったデバッグ」を選択し、次に目的の実行可能ファイルを選択して、常に同じデバッグ設定を使用できます。

`dbx` コマンドラインからこれらのプロパティーの多くを設定する方が簡単な場合があります。これらはデバッグターゲット設定に格納されます。

次の内容の多くの目的は、さまざまな中間ブレークポイントで「継続」をクリックする必要なく、「実行」をクリックすることによって、興味の場所に常に移動できるように、ブレークポイントを追加するにつれて容易な再現性を維持することです。

ステップ 2: 最初のブレークポイント

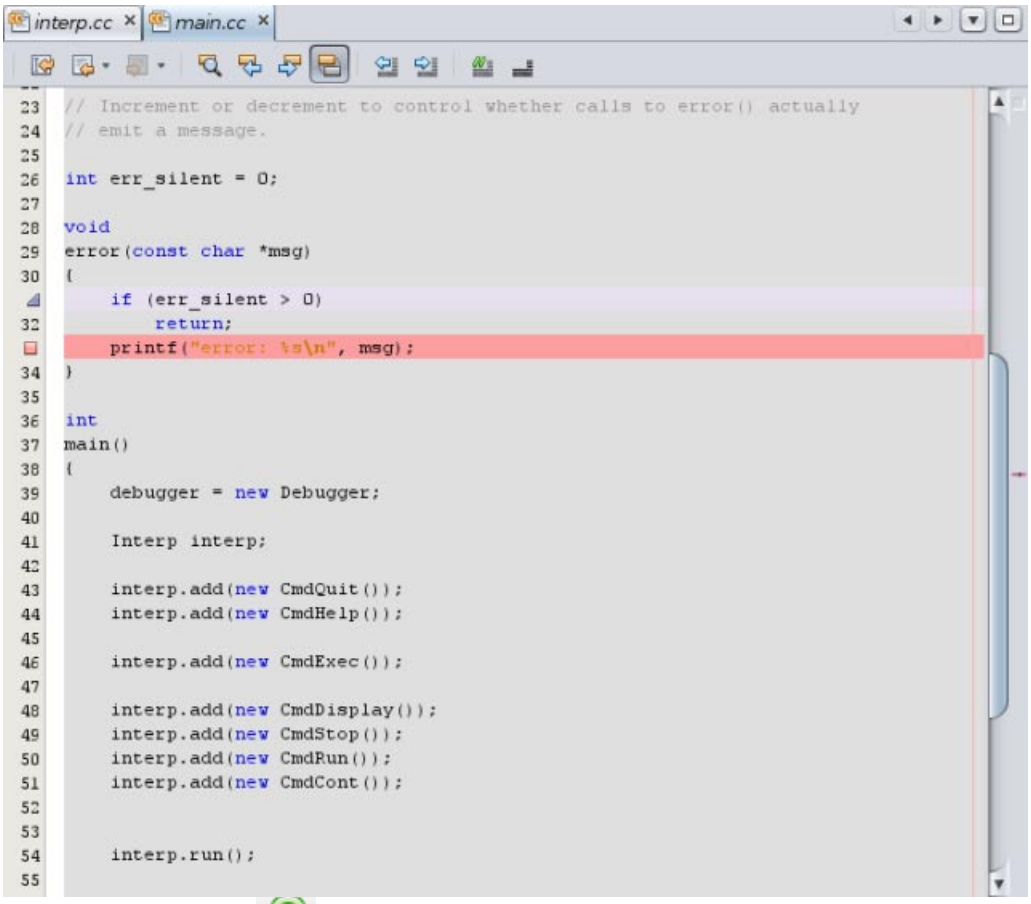
エラーメッセージを出力する場合には、`error()` 関数内にブレークポイントを置きましょう。このブレークポイントは、33 行目の行ブレークポイントとなります。

より大きなプログラムでは、たとえば「デバッガコンソール」ウィンドウで、次のように入力して「エディタ」ウィンドウの現在の関数を簡単に変更できます。

```
(dbx) func error
```

ラベンダーストライプに、`func` コマンドで検出された一致が示されます。

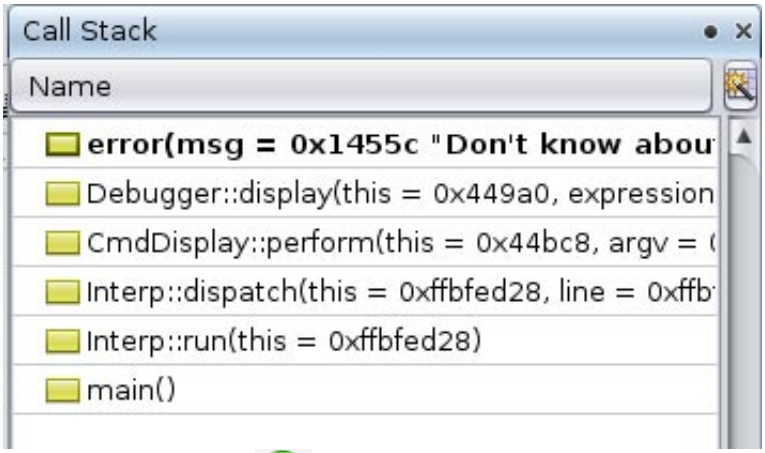
1. 33 番目の上の「エディタ」ウィンドウの左マージンをクリックして、行ブレークポイントを作成します。



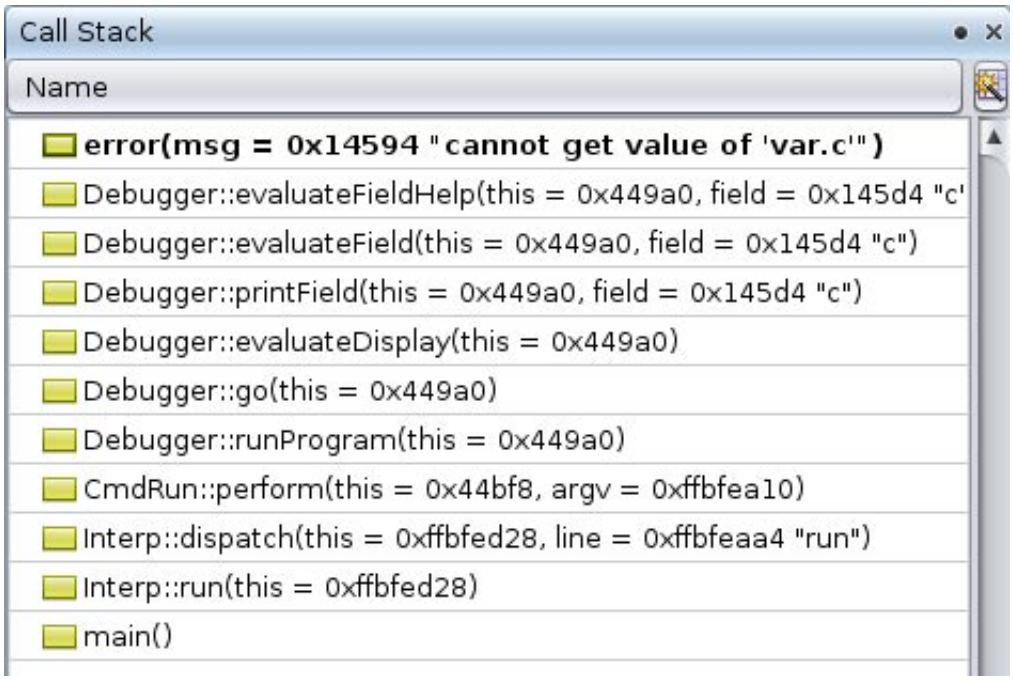
2. 「実行」をクリック  してプログラムを実行し、ブレークポイントをヒットすると、スタックトレースが表示されます。 `in` ファイルのシミュレートされたコマンドによって発行されているエラーメッセージが表示されます。

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

`error()` への呼び出しは想定された動作です。



3. 「継続」をクリック  して、プロセスを継続し、ふたたびブレークポイントをヒットします。今回は予期しないエラーメッセージを受け取ります。



ステップ 3: ブレークポイントカウント

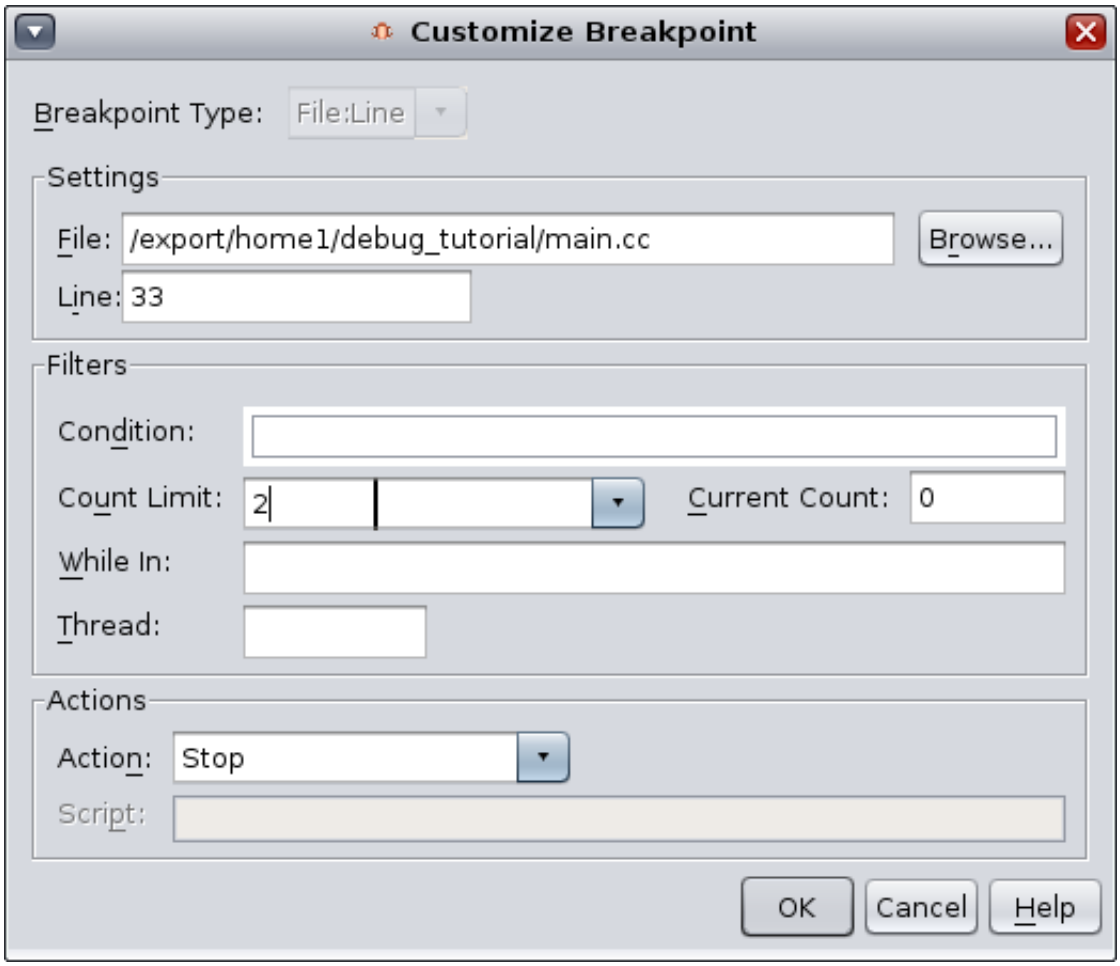
コマンドによるブレークポイントの最初のヒットの後で、「継続」をクリックする必要なく、実行ごとに繰り返してこの場所に到着する方がよいでしょう。

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

プログラムまたは入力スクリプトを編集して、最初の問題を引き起こす表示コマンドを削除できます。ただし、作業している特定の入力順序は、このバグを再現する鍵となる可能性があるため、状況を混乱させないようにしましょう。

このブレークポイントに到着する 2 回目に興味があるため、カウントを 2 に設定しましょう。

1. 「ブレークポイント」ウィンドウで、ブレークポイントを右クリックして、「カスタマイズ」を選択します。
2. 「ブレークポイントをカスタマイズ」で、「カウンタの上限値」フィールドに「2」を入力します。
3. 「OK」をクリックします。

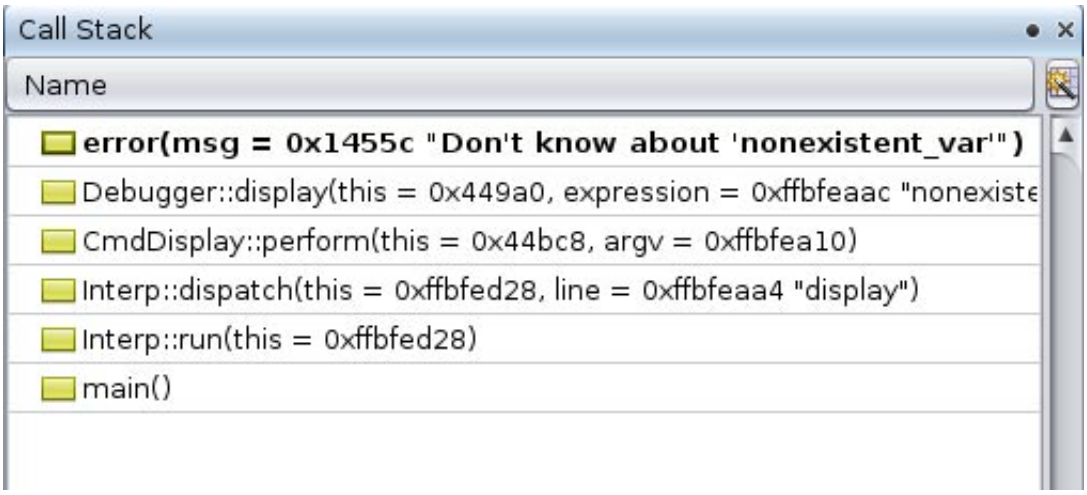


これで、興味のある場所に繰り返し到着できます。

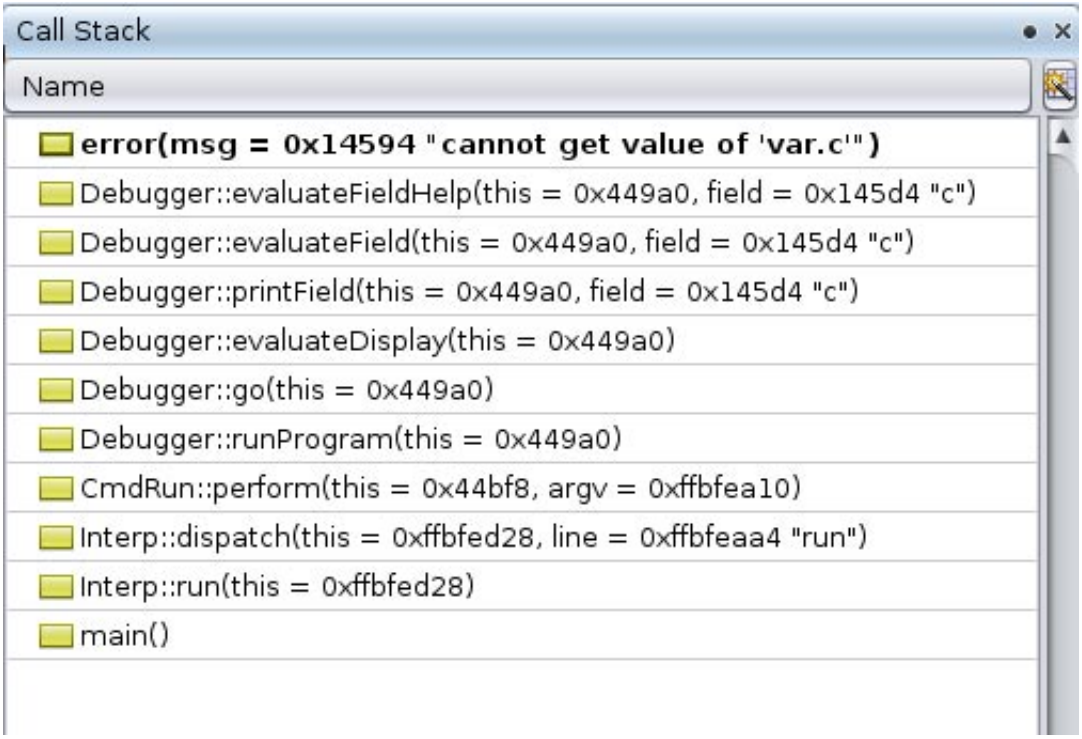
ステップ 4:境界ブレークポイント

この場合、カウント 2 を選択することは簡単でした。しかし、停止に興味のある場所は何度も呼び出される場合があります。後で、適切なカウント値をどのように簡単に選択できるかわかるでしょう。しかし現在、興味のある呼び出しでのみ、`error()` で停止する別の方法を調査しましょう。

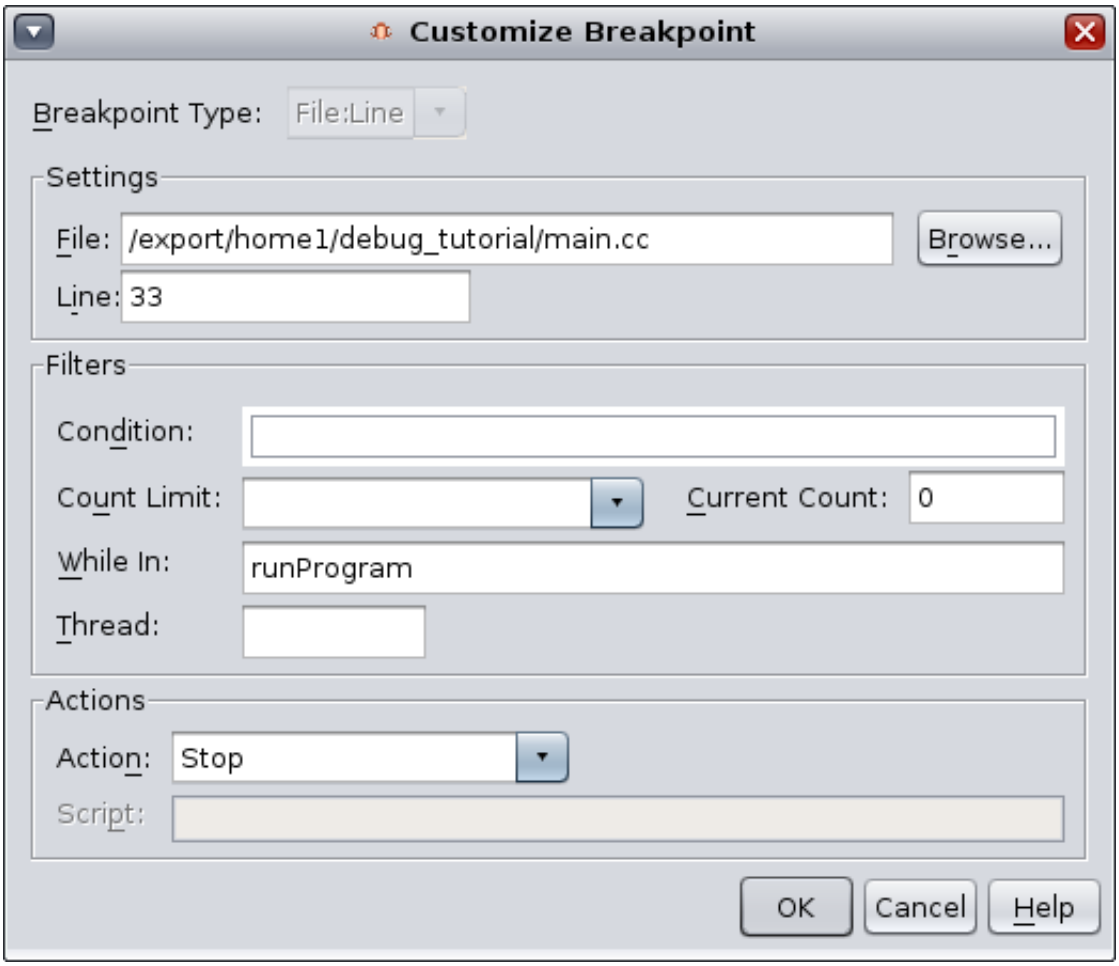
1. `error()` 内のブレークポイントの前のように「ブレークポイントをカスタマイズ」を開いて、「カウンタの上限値」のドロップダウンリストから「常に停止」を選択して、ブレークポイントカウントを無効にします。
2. ここで、「実行」をクリックして、`error()` で 2 回停止するときに、スタックトレースに注意を払います。1 回目は次のように表示されます。



2 回目は次のように表示されます。



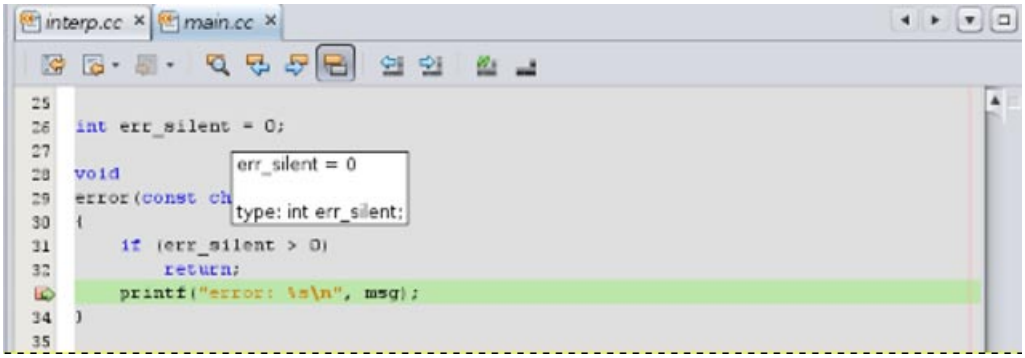
`runProgram` (フレーム [7]) から最後と呼ばれるときに、このブレークポイントで停止するように調整します。これを行うには、ふたたび「ブレークポイントをカスタマイズ」ダイアログボックスを開いて、「指定関数内」フィールドを runProgram に設定します。



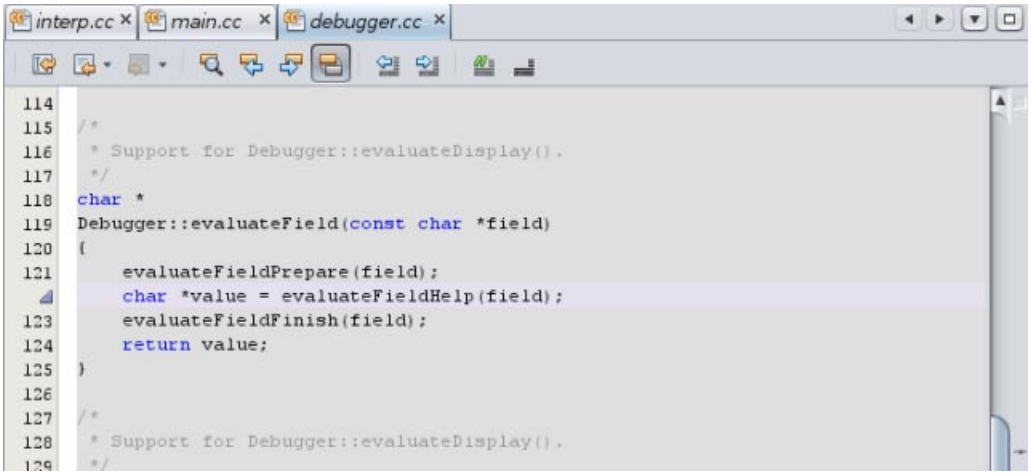
ここで、再び、興味のあるポイントに繰り返し、普通に到着できます。

ステップ 5: 原因の調査

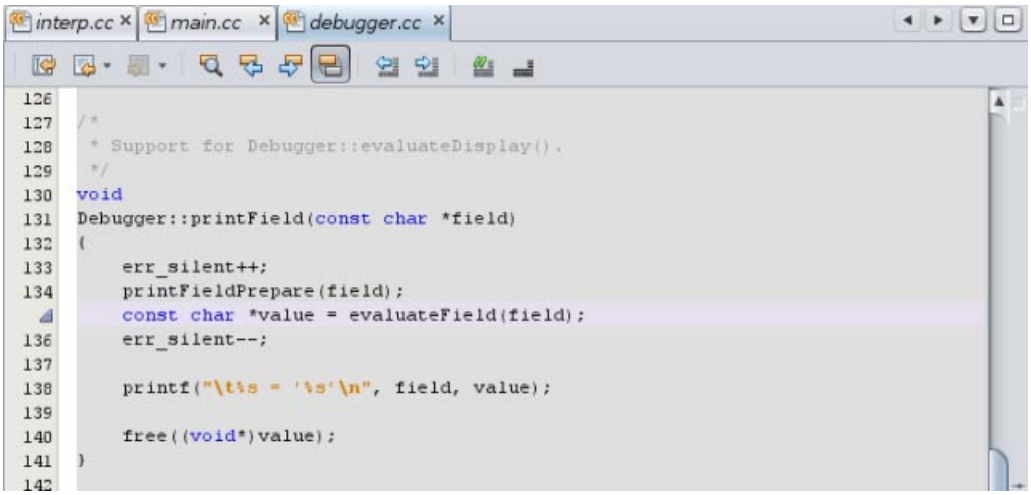
不要なエラーメッセージが発行されているのは何故ですか。明らかに、 `err_silent` が > 0 であるためです。バルーン評価を使用して `err_silent` の値を見てみましょう。カーソルを 31 行目の `err_silent` に置いて、表示される値を待機します。



`err_silent` が設定される場所を確認するために、スタックに従いましょう。「呼び出し元を現在に設定」  を 2 回クリックすると `evaluateField()` に到着し、すでに `evaluateFieldPrepare()` を呼び出していて、`err_silent` を操作している可能性のある複雑な関数をシミュレートします。



「呼び出し元を現在に設定」をもう一度クリックすると、printField() に到着します。ここで err_silent は増やされています。printField() はすでに printFieldPrepare() を呼び出していて、err_silent を操作している可能性のある複雑な関数もシミュレートします。



err_silent++ および err_silent-- がどのようにいくつかのコードを囲むかに注意してください。

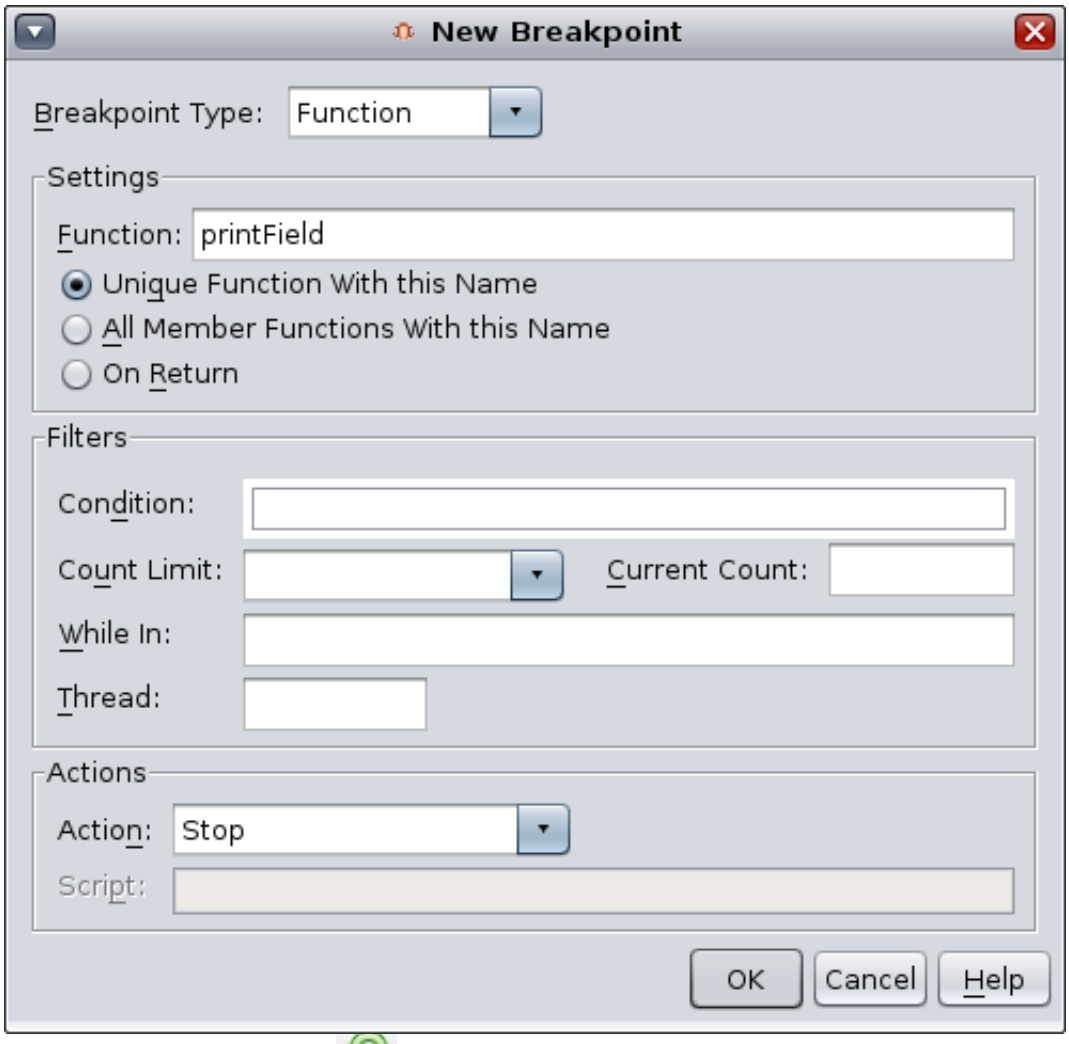
printFieldPrepare() または evaluateFieldPrepare() のいずれかで、err_silent が間違っているか、またはコントロールが printField() に到着するときにすでに間違っていることがあります。

printField () にブレークポイントを置くことによって printField() への呼び出し前後が間違っているかどうかがわかるでしょう。

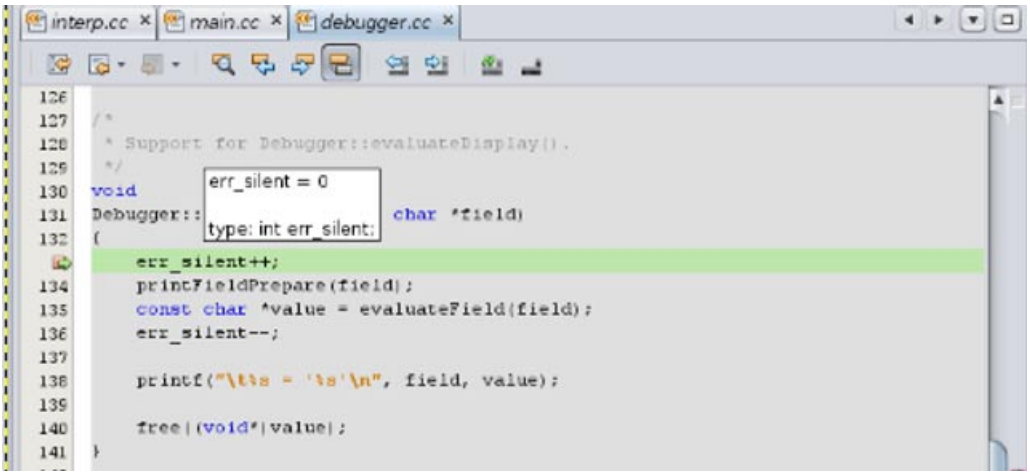
ステップ 6: より多くのブレークポイントカウント

printField() にブレークポイントを設定します。

1. printField() を選択し、「新規ブレークポイント」を右クリックして、選択します。
2. 新しいブレークポイントタイプが事前に選択され、「関数」フィールドに printfield で事前入力されています。
3. 「OK」をクリックします。



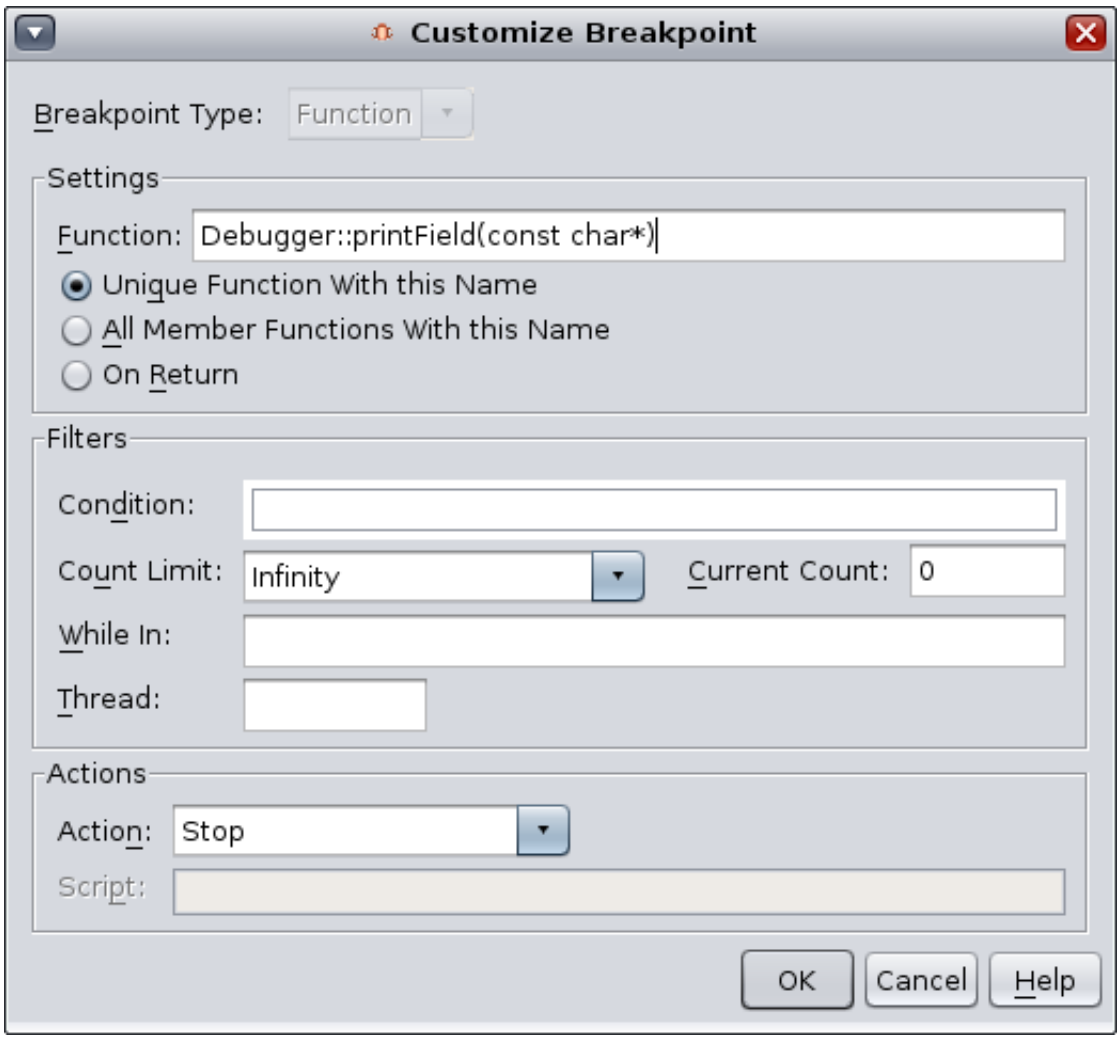
4. 「実行」をクリックします  。ブレークポイントをヒットする初回は、最初の実行時、最初の停止時、および最初のフィールド `var.a` 上です。`err_silent` は 0 で、これは問題ありません。



5. 「継続」をクリックします。`err_silent` は依然として問題ありません。
6. ふたたび「継続」をクリックします。`err_silent` は依然として問題ありません。

特定の printField() への呼び出しに達し、その結果、不要なエラーメッセージが発生するまでしばらくかかります。printField ブレークポイントでブレークポイントカウントを使用する必要があります。しかしカウントをどのように設定したらよいでしょうか。この簡単な例では、表示されている実行、停止、およびフィールドを数えようとしますが、実際には非常に予測可能ではない可能性があります。ただし、カウントが半自動的であるかを推測する方法があります。

1. printField() 上のブレークポイントの「ブレークポイントをカスタマイズ」を開いて、「カウンタの上限値」フィールドを infinity に設定します。



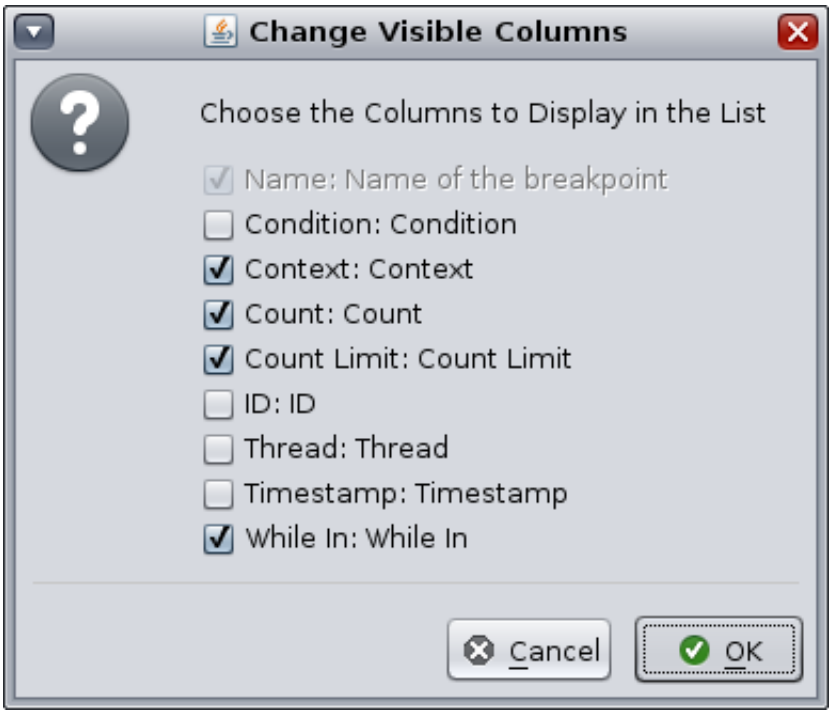
この設定は、このブレークポイントで停止しないことを意味します。ただし、依然として数えています。

2. この時点で、「ブレークポイント」ウィンドウに、カウントなどのより多くのプロパティーを表示することが望ましいでしょう。

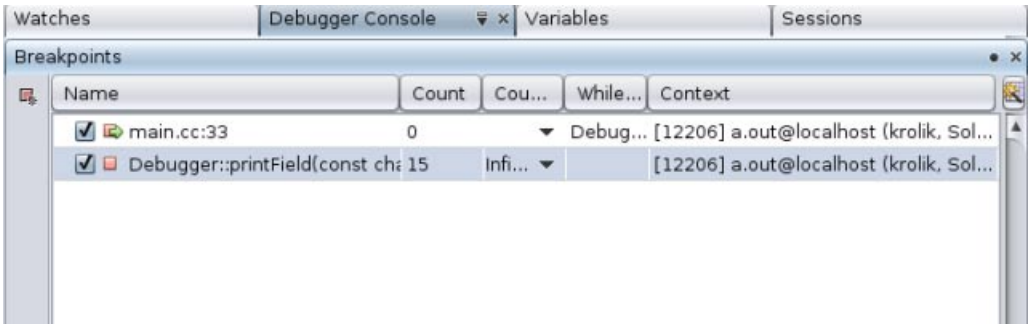
   a. 「ブレークポイント」ウィンドウの右上角にある「表示項目を変更」ボタン をクリックします。

   b. 「カウント」、「制限」、および「指定関数内」を選択します。

   c. 「OK」をクリックします。



3. プログラムを再実行します。`error()` 内のブレークポイント、`runProgram()` によって境界を付けられるブレークポイントをヒットします。
4. ここで、`printField()` 上のブレークポイントのカウントを見てみましょう。



15 で、すなわち、ほしいカウントです。

5. 「カウンタの上限値」列のドロップダウンリストをクリックして、「現在のカウント値を使用」を選択し、現在のカウントをカウンタの上限に転送して、改行キーを押します。

ここで、`printField()` で停止するプログラムを実行すると、最後が不要なエラーメッセージの前に呼び出されます。

ステップ 7: 原因の範囲を限定する

バルーン評価を使用して、ふたたび `err_silent` を検査します。現在は -1 です。`printField()` に到達する前に、1 つの `err_silent--` が必要以上に、あるいは 1 つの `err_silent++` が必要以下に実行されたという可能性は最も高いといえます。

`err_silent` のこのミスマッチのペアをどのように検索できますか。この例のような小さなプログラムでは、注意深いコード検査によって行うことができます。しかし、大きなプログラムでは、法外な数のペアがある可能性があります。

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

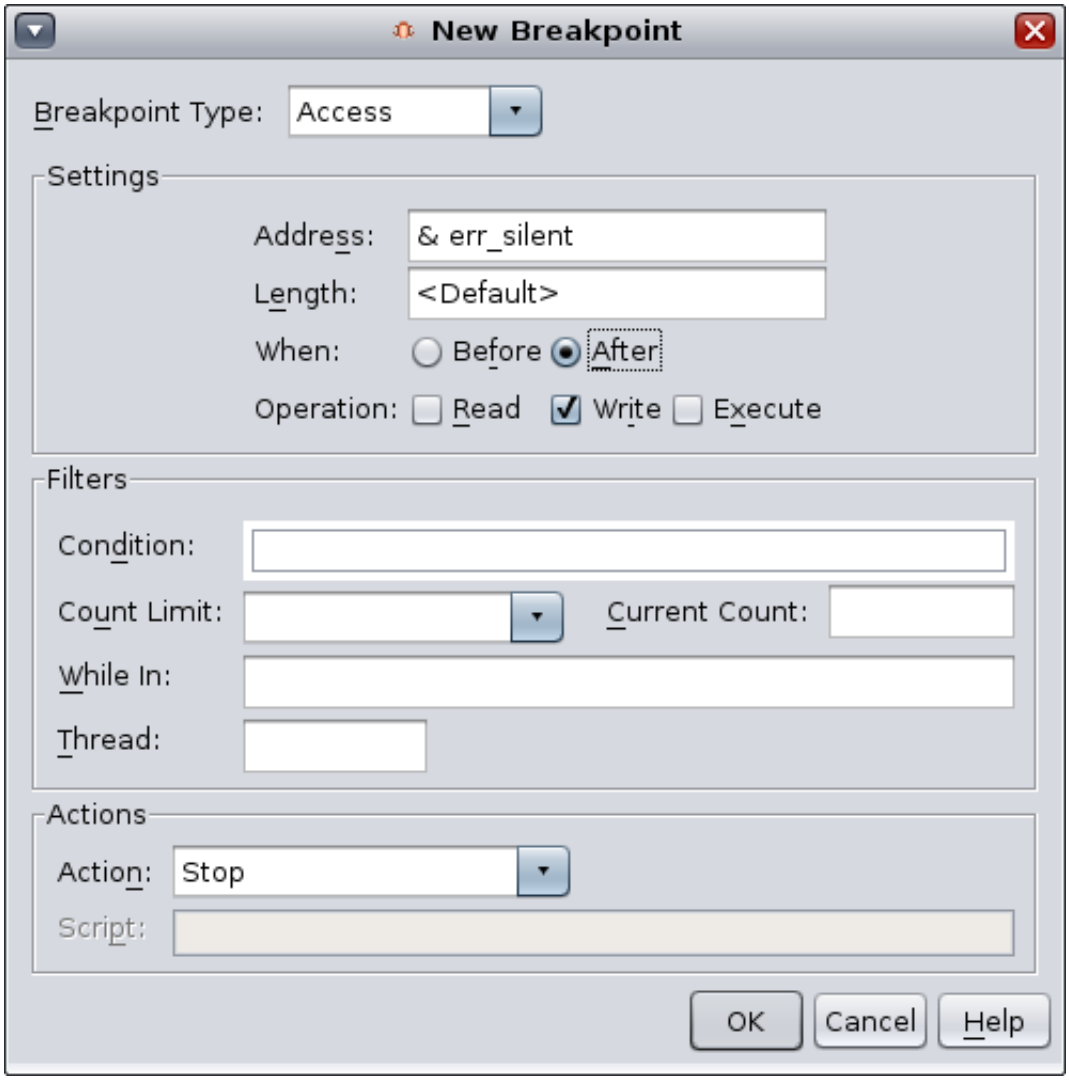
ミスマッチのペアをより速く検索する方法として、ウォッチポイントの利用があります。

ヒント – `err_silent++;` および `err_silent--;` のミスマッチのセットではなく、`err_silent` のコンテンツを上書きする不正ポインタである場合もあります。ウォッチポイントはそのような問題をとらえるのにより効果的でしょう。

ステップ 7: ウォッチポイントの使用法

`err_silent` でウォッチポイントを作成するには、次の手順に従います。

1. `err_silent` 変数を選択し、「新規ブレークポイント」を右クリックして、選択します。
2. アクセスするブレークポイントの種類を設定します。「設定」セクションがどのように変更され、「アドレス」フィールドがどのように `& err_silent` であるか注意してください。
3. 「いつ」フィールドの「後」を選択します。
4. 「演算」フィールドの「書き込み」を選択します。
5. 「OK」をクリックします。



6. ここでプログラムを実行します。`init()` で停止します。ここでは問題ないのようです。つまり、`err_silent` は 1 に増分され、その後実行が停止しました。
7. 「継続」をクリックします。ふたたび `init()` で停止します。

8. ふたたび「継続」をクリックします。ふたたび `init()` で停止します。
9. ふたたび「継続」をクリックします。ふたたび `init()` で停止します。
10. ふたたび「継続」をクリックします。ここで、`stopIn()` で停止します。ここでも問題ないのようで、つまり -1s ではありません。

`err_silent` が -1 に設定されるまでしばらく時間がかかることがありますが、視界が霞んで実際に -1 に変わった瞬間を見逃すまいと「継続」ボタンを何十回もクリックしたくもありません。しかしさらによい方法があります。

ステップ **8:**ブレークポイントの条件

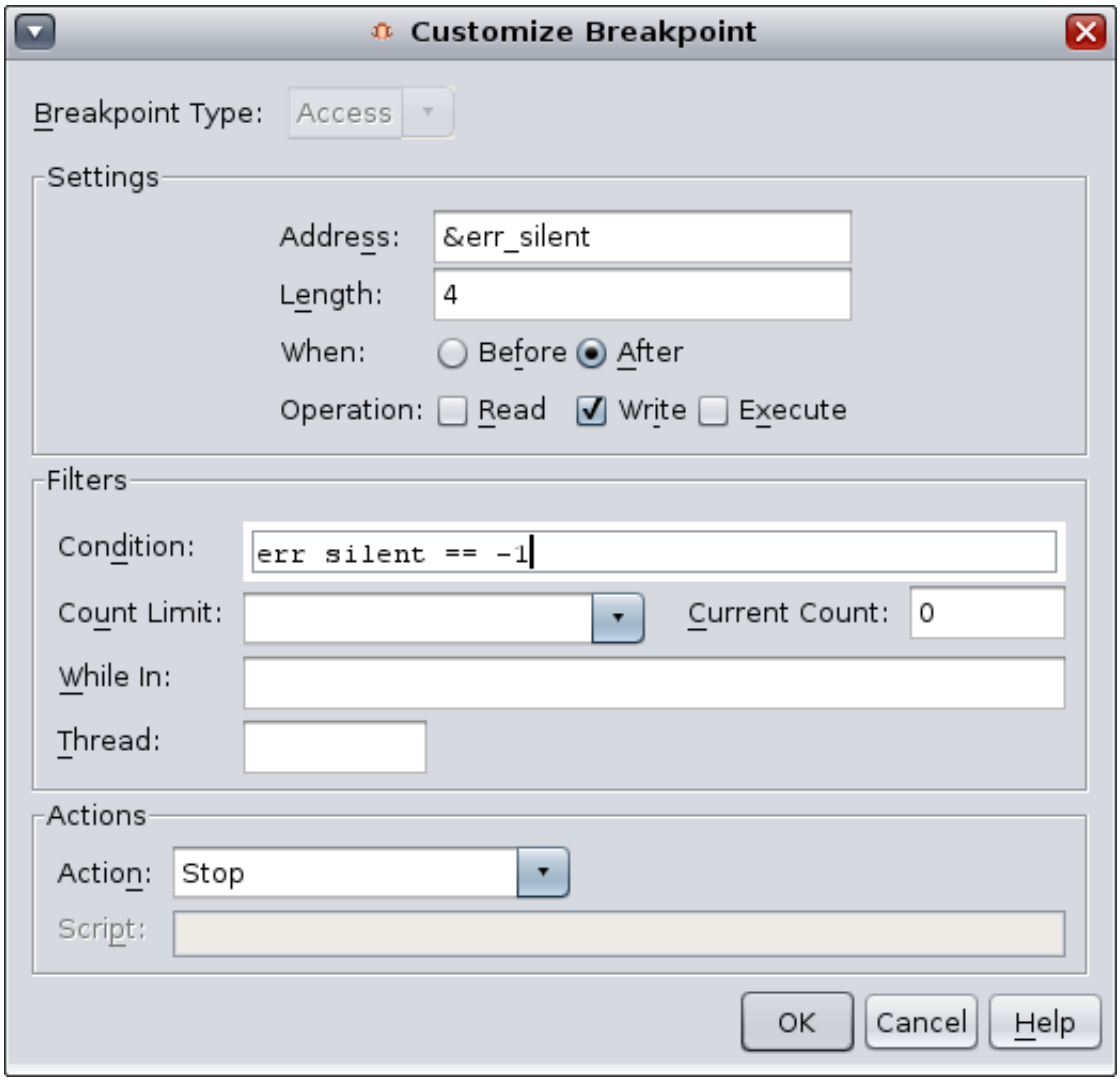
ウォッチポイントに条件を追加するには、次の手順に従います。

1. 「ブレークポイント」ウィンドウで、「後」の「書き込み」ブレークポイントを右クリックして、「カスタマイズ」を選択します。
2. 「後」が「とき」フィールドで選択されていることを確認します。

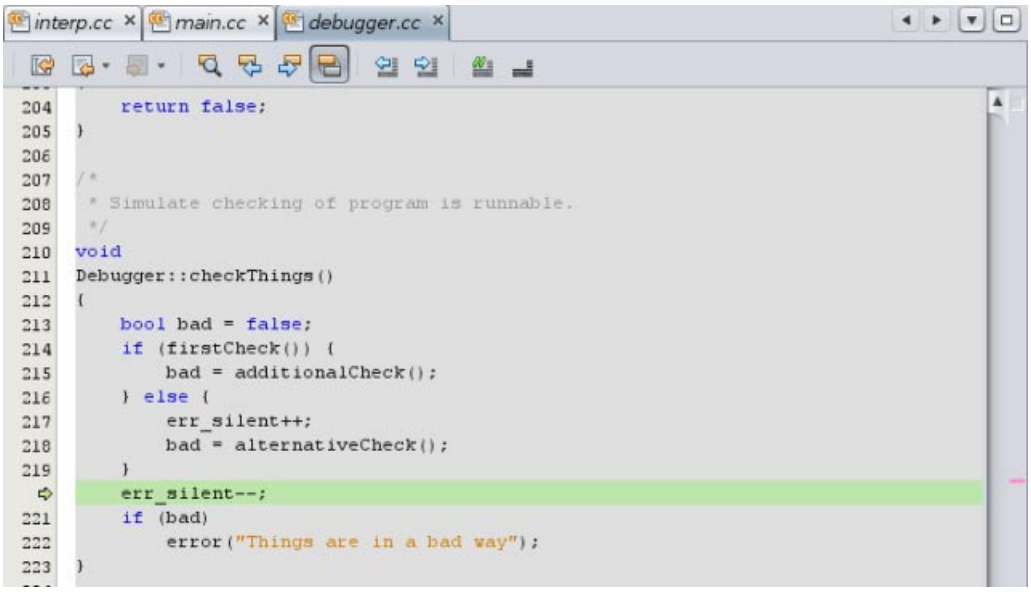
ヒント – `err_silent` の値が何に変更されたか知りたいため、「後」を選択することは重要です。

---

3. 「条件」フィールドを err_silent == -1 に設定します。
4. 「OK」をクリックします。



これでプログラムを再実行します。`checkThings()` で停止します。これは `err_silent` が `-1` に設定されている最初です。マッチングしている `err_silent++` を探すにつれて、バグのように見えるものを確認します。`err_silent` は関数の `else` 部分でのみ増分されます。



これはあなたが探しているバグでしょうか。

ステップ 9:スタックをポップすることによる診断の確認

関数の `else` ブロックから実際に進んでいることをダブルチェックしましょう。

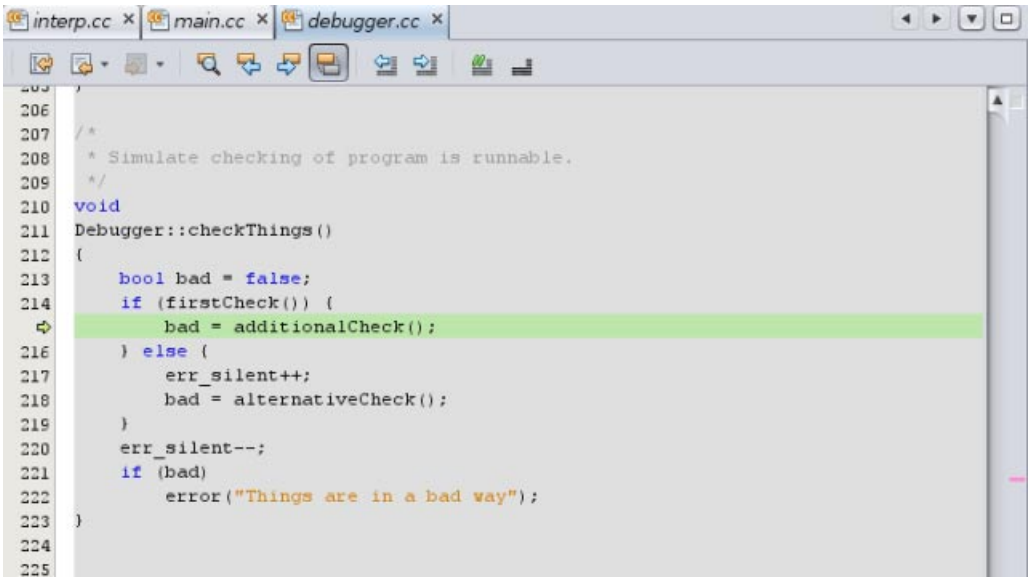
これを行う 1 つ方法は、`checkThings()` でブレークポイントを設定し、プログラムを実行することです。しかし、`checkThings()` は何度も呼び出される可能性があります。`checkThings()` の正しい呼び出しに達するにはブレークポイントカウントまたは境界ブレークポイントを使用できますが、最近実行されたものを再実行するすばやい方法があります。

- 「デバッグ」 >「スタック」 >「最上位の呼び出しをポップ」を選択します。

ヒント– 「最上位の呼び出しをポップ」がすべてを元に戻さないことに注意してください。特に、`err_silent` の値はすでに間違っています。しかし、これはデータデバッグからコントロールフローデバッグまで切り替えているため、問題ないはずです。

`checkThings()` への呼び出しに気付きます。実際、プロセスの状態は、`checkThings()` への呼び出しを含む行の最初に戻されています。

ここで、「ステップイン」  をクリックし、ふたたび `checkThings()` が呼び出されるたびに監視します。 `checkThings()()` をステップスルーするにつれて、実際にプロセスが `if` ブロックを実行することを確認できます。`err_silent` は増分されず、`-1` に減らされます。

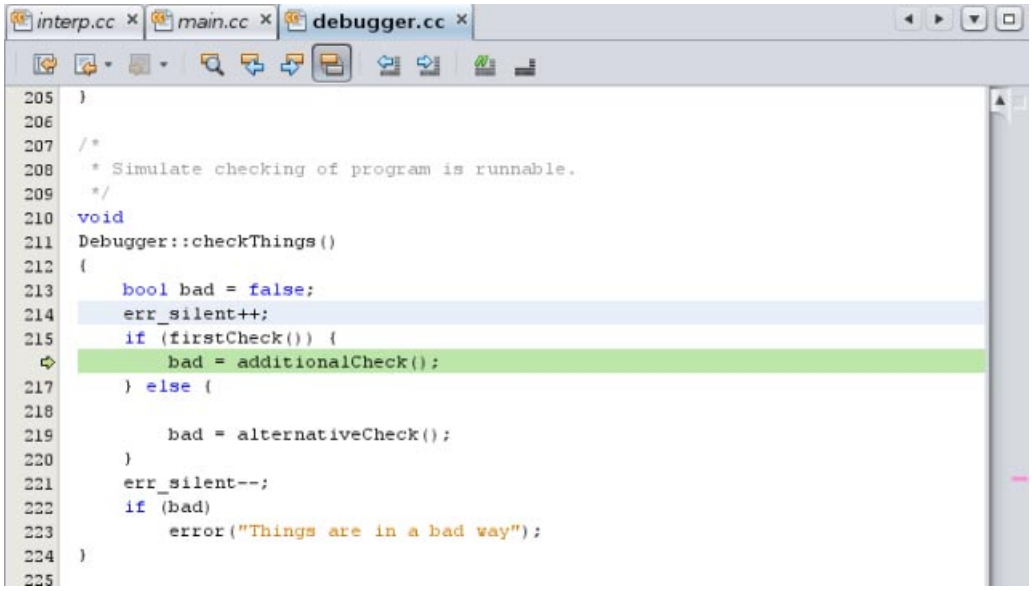


プログラミングエラーを検出しているように見えます。しかし、それをトリプルチェックしましょう。

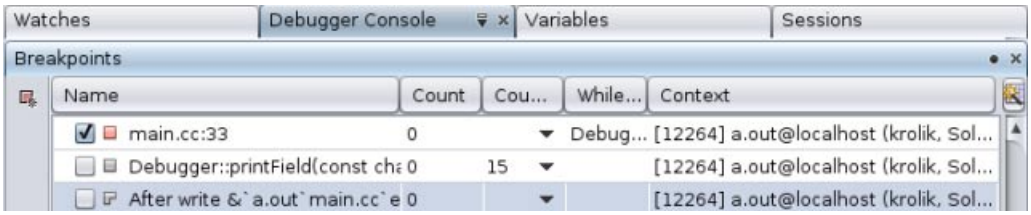
ステップ 10:診断をさらに確認するための修正の使用法

コードを適切に修正し、バグが実際に解決されたことを確認します。

1. `err_silent++` が `if` 文の上に来るようにコードを修正します。結果は次のようになります。

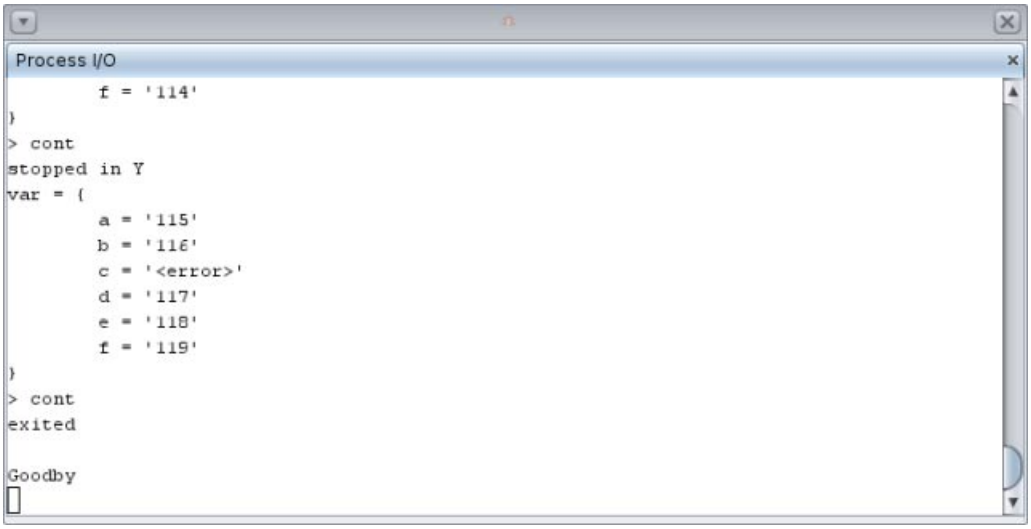


2. 「デバッグ」>「コード変更を適用」を選択します。
3. `printField` ブレークポイントおよびウォッチポイントを無効にしますが、`error()` のブレークポイントは有効なままにします。



4. プログラムを再実行します。

`error()` でブレークポイントをヒットすることなくプログラムが完了し、出力が想定どおりであることに注意しましょう。



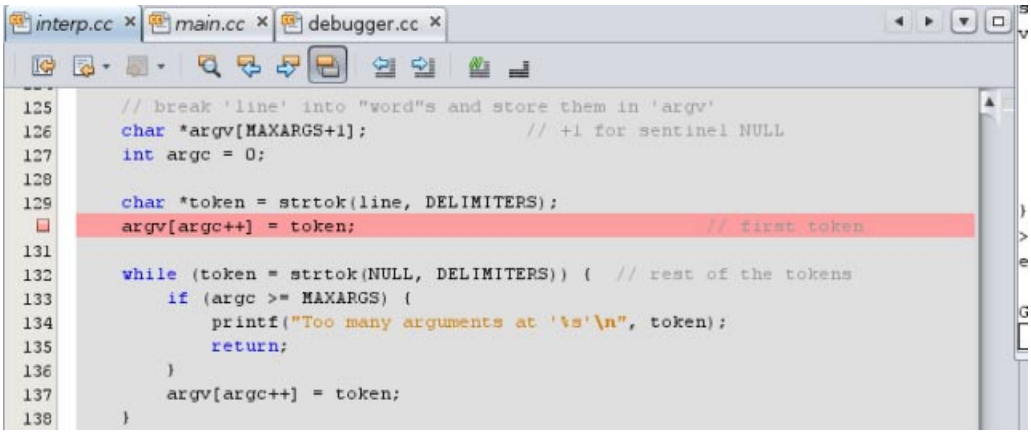
ディスカッション

上記は、以前として13 ページの「ブレークポイントとステップ動作の使用法」の終わりで説明したのと同じパターンを示しており、つまり、調子が悪くなる前の時点の誤動作プログラムを停止し、実際にコードが動作する方法でコードの意図を比較してコードをステップスルーします。主な相違は、調子が悪くなる前にポイントを検出することが少し関連しています。

ブレークポイントスクリプトを使用してコードにパッチを適用する

13 ページの「ブレークポイントとステップ動作の使用法」 で、空の行が NULL の最初のトークンを生成し、SEGV の原因となるバグを検出しました。ここにすばやく解決する方法があります。

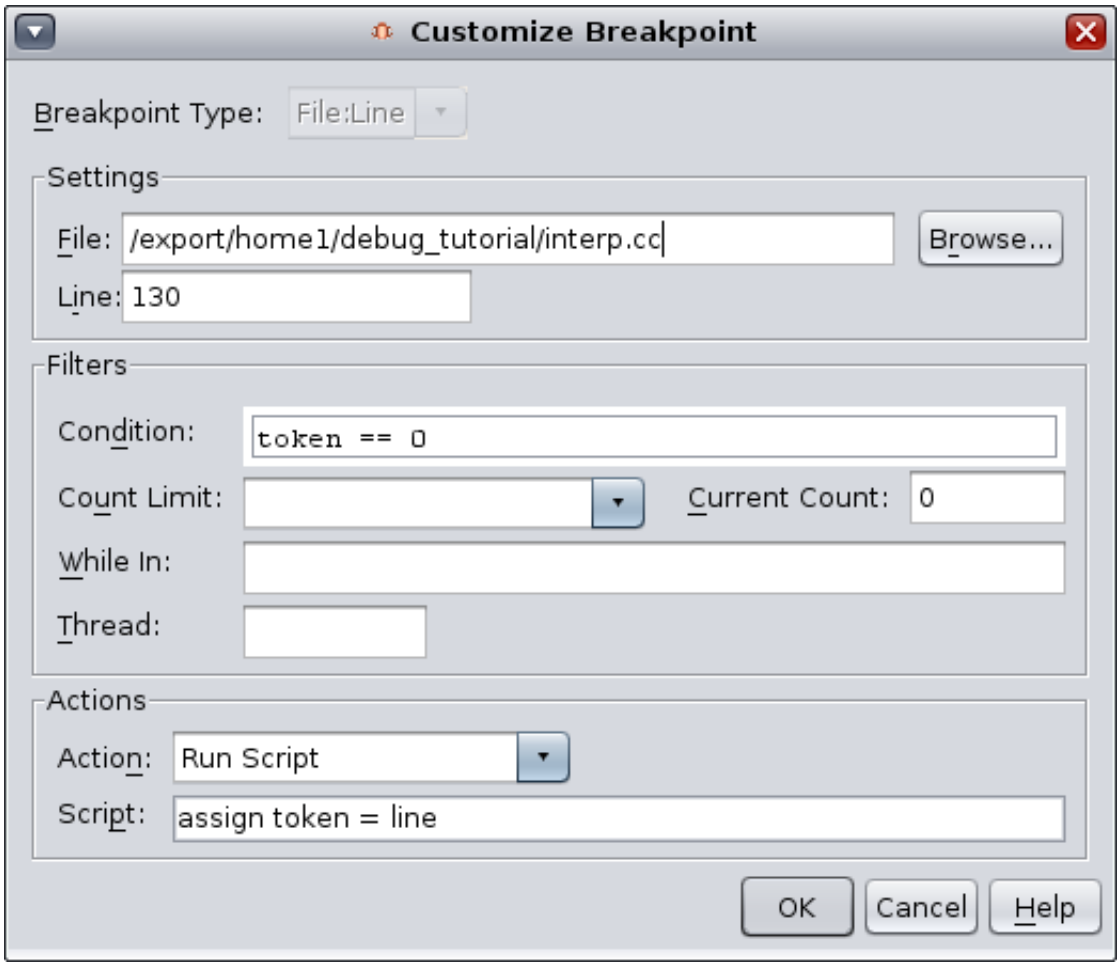
1. 前の節で作成したブレークポイントのすべてを削除します。
2. 「実行可能ファイルをデバッグします」ダイアログボックスの `<in` 引数を削除します。
3. `interp.cc` の 130 行目の行ブレークポイントを切り替えます。



4. 「ブレークポイント」ウィンドウで、ちょうど作成したブレークポイントを右クリックし (より新しいブレークポイントは一番下に追加される)、「カスタマイズ」を選択します。
5. 「ブレークポイントをカスタマイズ」ダイアログボックスの、「条件」フィールドで `token == 0` と入力します。
6. 「アクション」ドロップダウンリストから「スクリプトの実行」を選択します。
7. 「スクリプト」フィールドで、`assign token = line` と入力します。

ヒント – `assign token = "dummy"` はどうですか。`dbx` はデバッグプロセスの `dummy` 文字列を割り当てることはできません。その一方で `line` は `""` に等しいことが知られています。

ダイアログボックスは次のように表示されるはずです。



8. 「OK」をクリックします。

ここで、プログラムを実行し、クラッシュではなく、空の行を入力すると、次のように動作します。

`dbxtool` が `dbx` に送られたコマンドで表示される場合はどのように明白に機能するでしょうか。

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

Copyright ©2010, 2011 このソフトウェアおよび関連ドキュメントの使用と開示は、ライセンス契約の制約条件に従うものとし、知的財産に関する法律により保護されています。ライセンス契約で明示的に許諾されている場合もしくは法律によって認められている場合を除き、形式、手段に関係なく、いかなる部分も使用、複写、複製、翻訳、放送、修正、ライセンス供与、送信、配布、発表、実行、公開または表示することはできません。このソフトウェアのリバース・エンジニアリング、逆アセンブル、逆コンパイルは互換性のために法律によって規定されている場合を除き、禁止されています。
ここに記載された情報は予告なしに変更される場合があります。また、誤りが無いことの保証はいたしかねます。誤りを見つけた場合は、オラクル社までご連絡ください。
このソフトウェアまたは関連ドキュメントを、米国政府機関もしくは米国政府機関に代わってこのソフトウェアまたは関連ドキュメントをライセンスされた者に提供する場合は、次の通知が適用されます。
U.S. GOVERNMENT END USERS:
Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are “commercial computer software” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.
このソフトウェアもしくはハードウェアは様々な情報管理アプリケーションでの一般的な使用のために開発されたものです。このソフトウェアもしくはハードウェアは、危険が伴うアプリケーション（人的傷害を発生させる可能性があるアプリケーションを含む）への用途を目的として開発されていません。このソフトウェアもしくはハードウェアを危険が伴うアプリケーションで使用する際、安全に使用するために、適切な安全装置、バックアップ、冗長性（redundancy）、その他の対策を講じることは使用者の責任となります。このソフトウェアもしくはハードウェアを危険が伴うアプリケーションで使用したことに起因して損害が発生しても、オラクル社およびその関連会社は一切の責任を負いかねます。
OracleおよびJavaはOracle Corporationおよびその関連企業の登録商標です。その他の名称は、それぞれの所有者の商標または登録商標です。
Intel、Intel Xeonは、Intel Corporationの商標または登録商標です。すべてのSPARCの商標はライセンスをもとに使用し、SPARC International, Inc.の商標または登録商標です。AMD、Opteron、AMDロゴ、AMD Opteronロゴは、Advanced Micro Devices, Inc.の商標または登録商標です。UNIXは、The Open Groupの登録商標です。
このソフトウェアまたはハードウェア、そしてドキュメントは、第三者のコンテンツ、製品、サービスへのアクセス、あるいはそれらに関する情報を提供することがあります。オラクル社およびその関連会社は、第三者のコンテンツ、製品、サービスに関して一切の責任を負わず、いかなる保証もいたしません。オラクル社およびその関連会社は、第三者のコンテンツ、製品、サービスへのアクセスまたは使用によって損失、費用、あるいは損害が発生しても一切の責任を負いかねます。
E26462

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

ORACLE