Oracle® Solaris Studio 12.4: dbxtool チュートリアル

2014 年 10 月

Part No: E57224

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 12.4 の「Downloads」Web ページ (http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/index.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。

ライセンスに同意してダウンロードしたあと、任意のディレクトリに zip ファイルを抽出できます。

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。

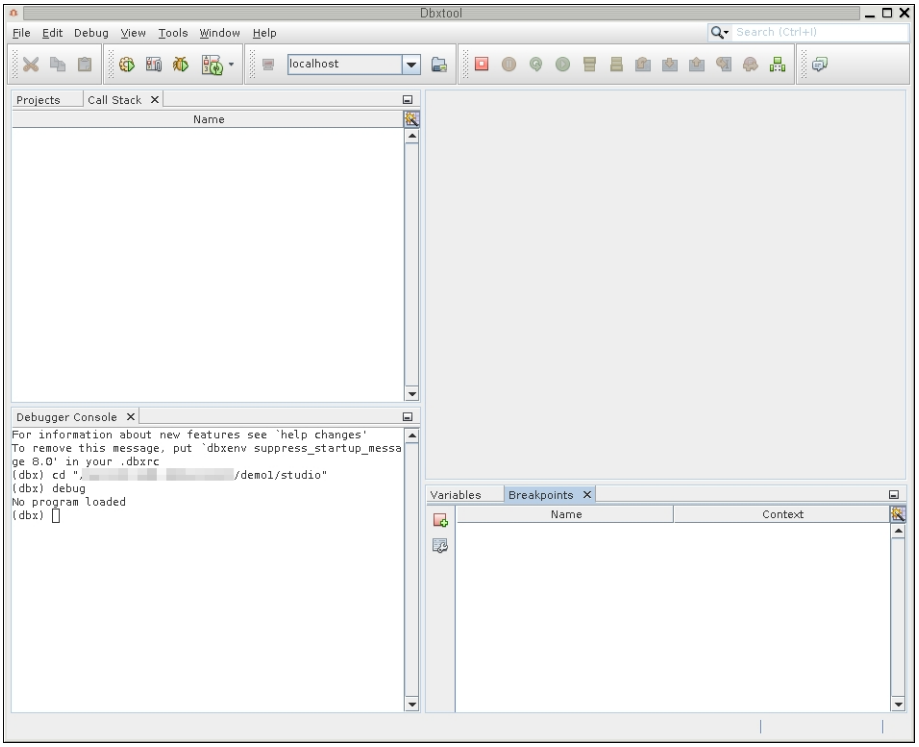
```
$ a.out
> display var
will display 'var'
> stop in X
> run running ...
stopped in X
var = {
        a = '100'
        b = '101'
        c = '<error>'
        d = '102'
        e = '103'
        f = '104'
}
> quit
Goodby
$
```

dbxtool の構成

次のように入力して、dbxtool を起動します。

install-dir/bin/dbxtool

最初に dbxtool を起動すると、通常はウィンドウが次のように表示されます。



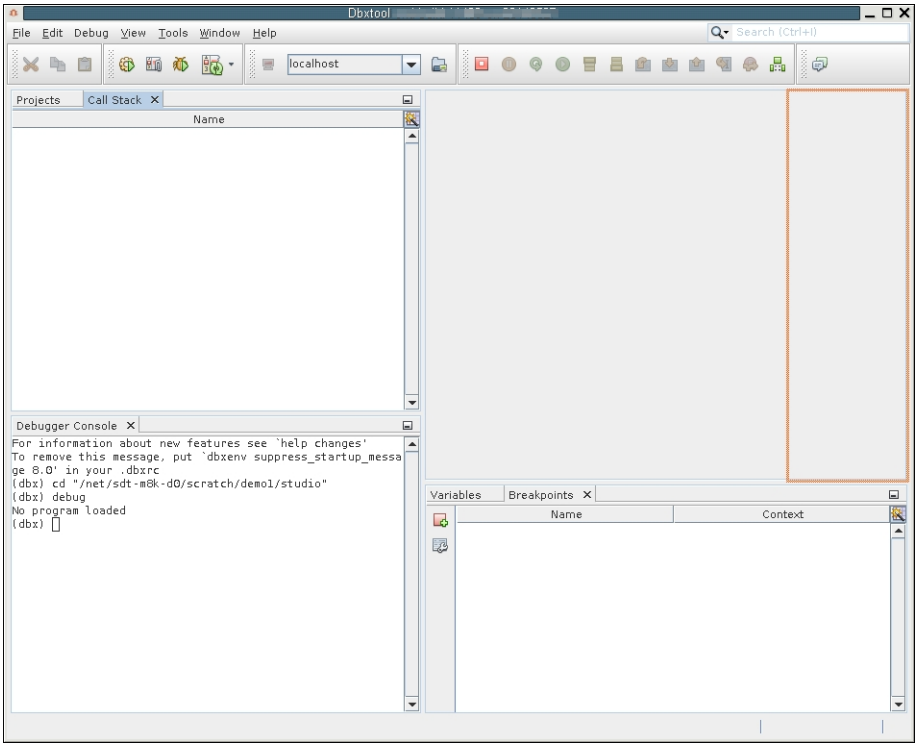
注記 - このチュートリアルの図は、dbxtool を実際に使用したときの表示と異なる場合があります。

Web ブラウザなどのほかのアプリケーション用により多くの領域が必要な場合は、`dbxtool` をカスタマイズして占有する領域を小さくすることもできます。

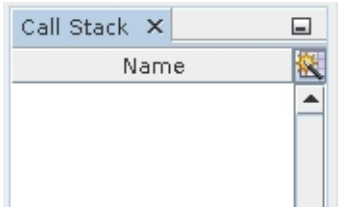
次に、`dbxtool` のさまざまなカスタマイズ例を示します。

- **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。**

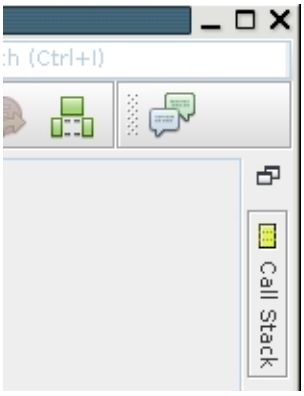
1. 「呼び出しスタック」ウィンドウのヘッダーをクリックして、ウィンドウを下および右へドラッグします。赤いアウトラインが次の図の位置に表示されたら、ウィンドウを放します。



2. 「呼び出しスタック」ウィンドウのヘッダーの最小化ボタンをクリックします。

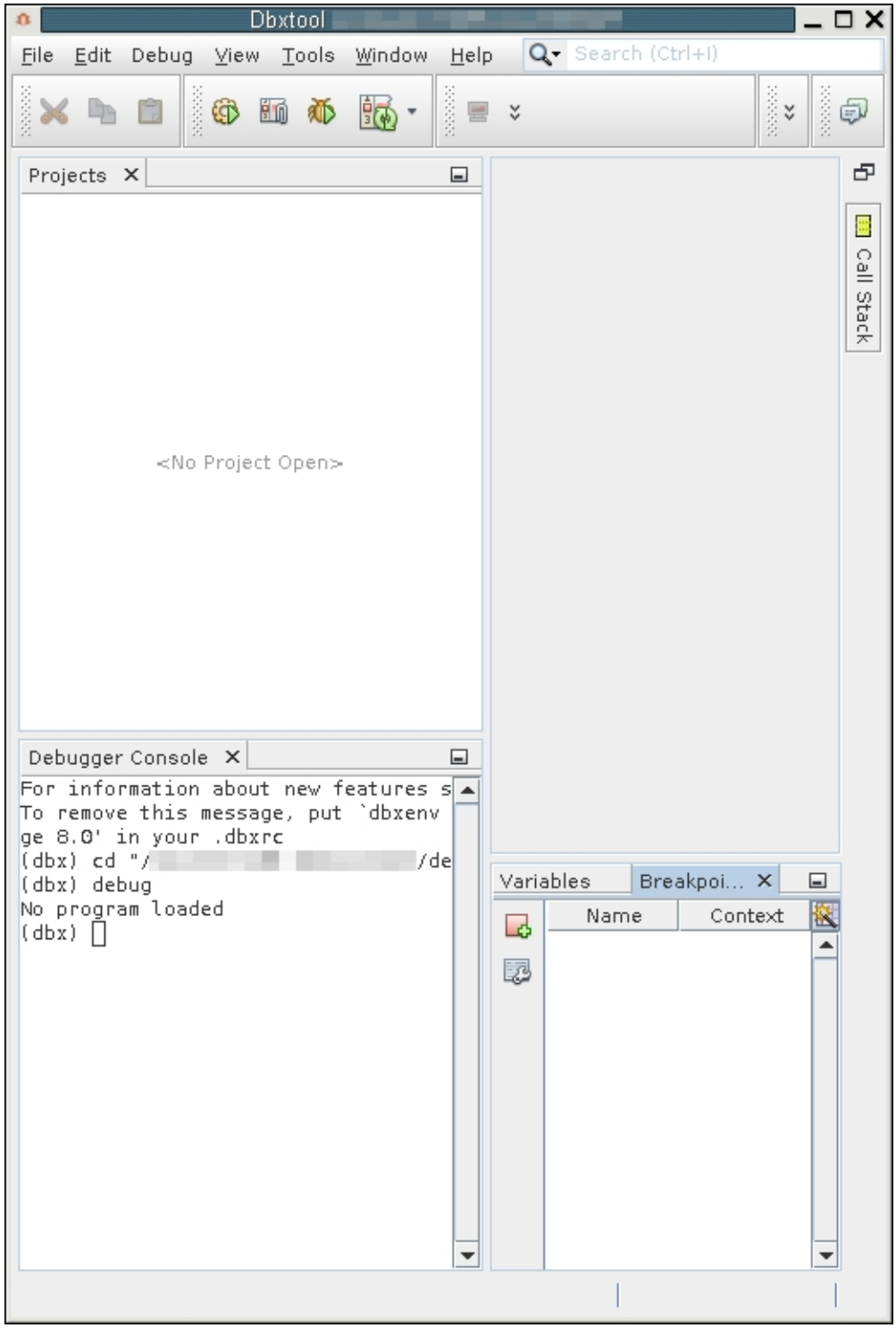


「呼び出しスタック」ウィンドウが右マージンに最小化されます。



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。

3. メインウィンドウを画面の半分に縮小します。

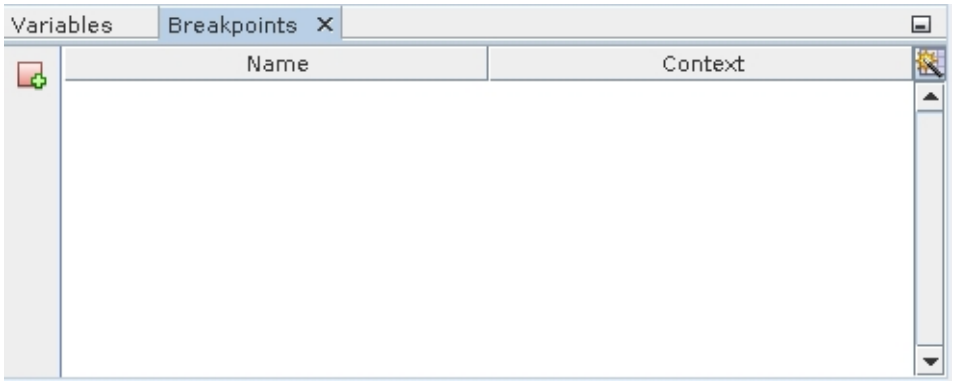


■ **ウィンドウのグループを最小化する。**

dbxtool では、ウィンドウをグループ化できます。個々のウィンドウに加えて、ウィンドウのグループに対してもアクションを実行できます。各ウィンドウは 1 つのグループに属し、このグループに対して、最小化と

復元、新しい場所への移動、別個のウィンドウでのフロート、または IDE ウィンドウへの再連結を行うことができます。

たとえば、「ブレークポイント」タブをクリックし、ウィンドウの最小化ボタンをクリックすると、ウィンドウのグループ全体が最小化します。



- **「出力」ウィンドウの連結を解除することで、dbxtool ウィンドウのほか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できるようにする。**
  - 「出力」ウィンドウが表示されていない場合は、「ウィンドウ」->「出力」をクリックするか、Ctrl+4 を押します。
  - 「出力」ウィンドウのヘッダーをクリックしたまま、そのウィンドウを dbxtool ウィンドウの外側にドラッグし、デスクトップ上にドロップします。

    「出力」ウィンドウを dbxtool ウィンドウに再連結するには、「出力」ウィンドウを右クリックして、「グループの連結」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ら、次を実行してフォントサイズを設定します。

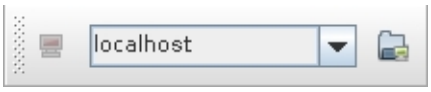
  1. 「ツール」->「オプション」を選択します。
  2. 「オプション」ウィンドウで、「フォントと色」カテゴリを選択します。
  3. 「構文」タブで、「言語」ドロップダウンリストからすべての言語が選択されていることを確認します。
  4. 「フォント」テキストボックスの横の「参照...」ボタンをクリックします。
  5. 「フォント選択」ダイアログボックスで、フォント、スタイル、およびサイズを設定し、「OK」をクリックします。
  6. 「オプション」ウィンドウで、「OK」をクリックします。
- **端末ウィンドウ内のフォントサイズを設定する。**「デバッガコンソール」ウィンドウと「出力」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。

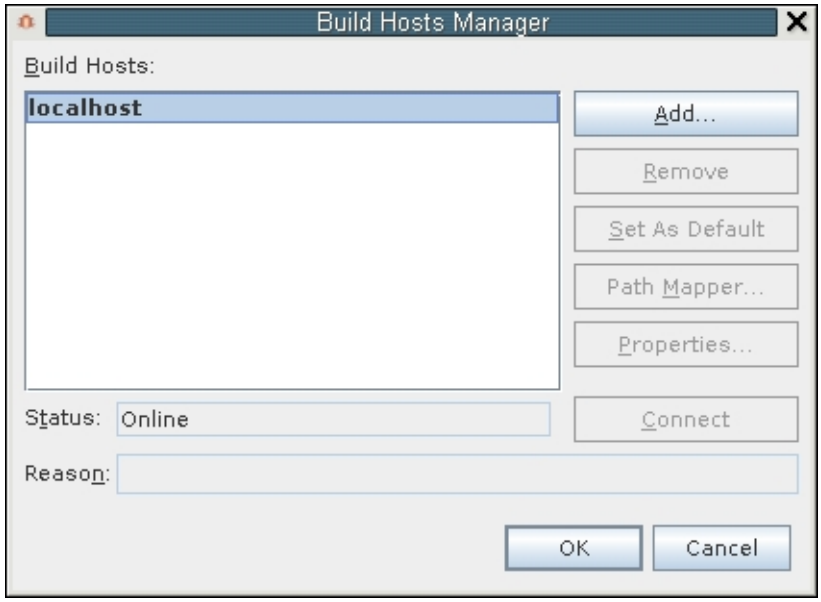
■ **デバッガを実行するリモートホストを追加する。**dbxtool では、リモートファイルにアクセスするだけでなく、dbxtool を実行するリモートサーバーにもアクセスできます。

dbxtool にリモートホストを追加するには:

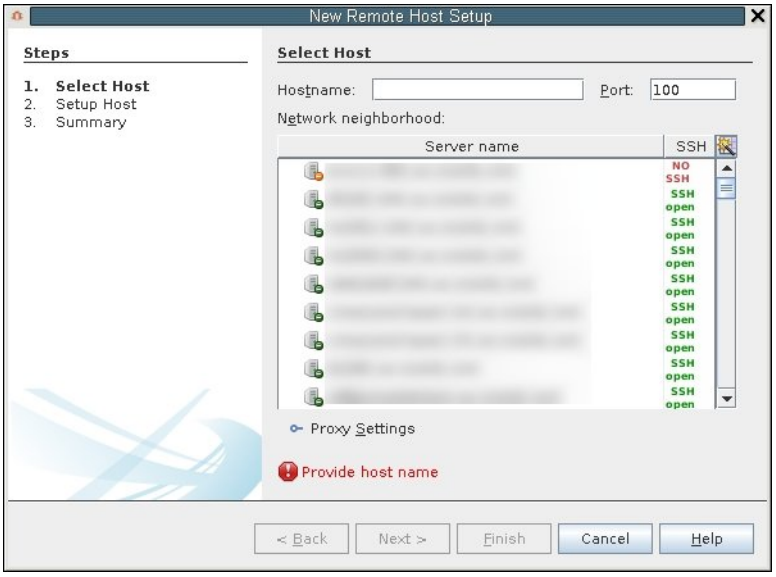
1. 「リモート」ツールバーで、ホストドロップダウンリストの下矢印をクリックして、「ホストの管理」を選択します。



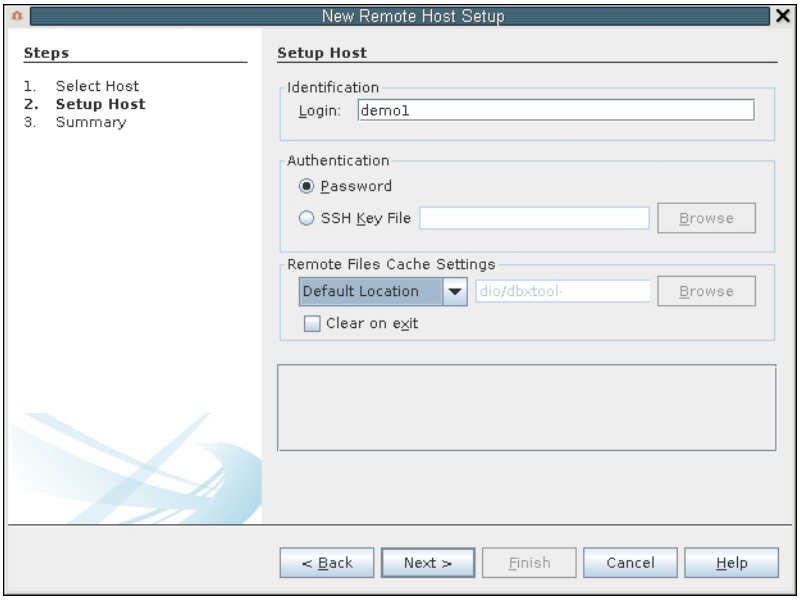
2. 「ビルド・ホスト・マネージャ」が開きます。「追加」をクリックして新しいサーバーを追加します。



3. 「新規リモート・ホストの設定」ウィザードで、「隣接ネットワーク」リストから使用可能なサーバーを選択し、「次へ」をクリックします。



4. ログイン情報を入力し、認証方法を選択して「次へ」をクリックします。「パスワード」を選択した場合は、入力を要求されたらパスワードを入力します。



5. ホストが接続されると、サマリーページに接続ステータスが表示されます。作業中にこのリモートホストを「リモート」ツールバーから選択できます。

リモートホストの詳細は、dbxtool のオンラインヘルプの「リモートデバッグ」を参照してください。

カスタマイズが完了したら、dbxtool を終了します。設定が dbxtool に保持され、次回実行時に使用されます。

コアダンプの診断

バグを見つけるには、プログラム例を再度実行して、コマンドを入力せずに Return キーを押します。

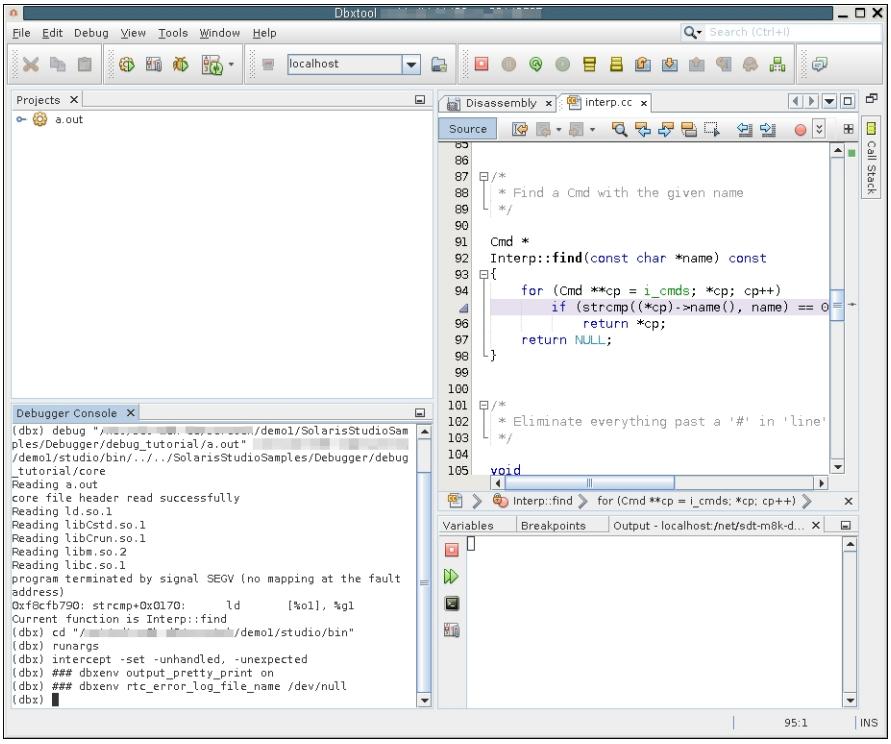
```
$ a.out
> display var
will display 'var'
>
Segmentation Fault (core dumped)
$
```

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
```

`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` に、次の例のような出力が表示されます。

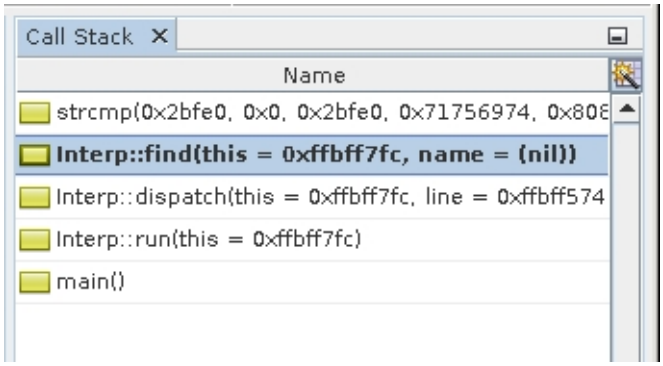


次の点に注意してください。

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

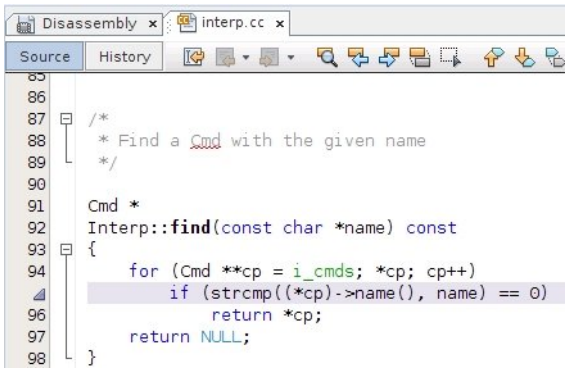
  ```
  program terminated by signal SEGV (no mapping at fault address)
  0xf8cfb790: strcmp+0x0170:      ld       [%ol], %gl
  Current function is Interp::find
  ```

- `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。この例では、strcmp() に渡された 2 番目のパラメータは 0x0 であり、name の値は NULL です。

- 「エディタ」ウィンドウで、95 行目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、実際にエラーの発生した場所は緑色のストライプと矢印で示されます。

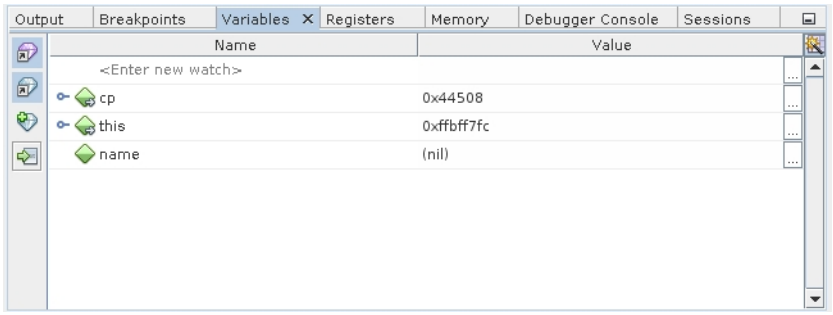


パラメータの値が表示されない場合は、dbxenv 変数 stack_verbose が .dbxrc ファイルで on に設定されていることを確認してください。ウィンドウ内を右クリックして「詳細」オプションを選択し、「呼び出しスタック」ウィンドウで詳細モードを設定することもできます。dbxenv 変数と .dbxrc の詳細は、『Oracle Solaris Studio 12.4: dbx コマンドによるデバッグ 』の第 3 章「dbx のカスタマイズ」を参照してください

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。strcmp() に渡された値を確認するには:

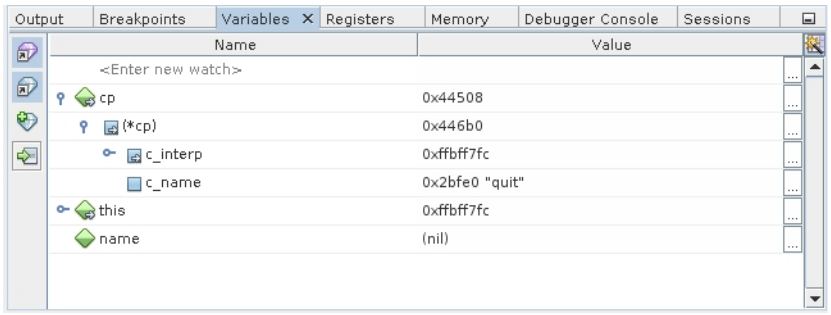
- 「変数」ウィンドウのパラメータの値を確認します。

  1. 「変数」タブをクリックします。



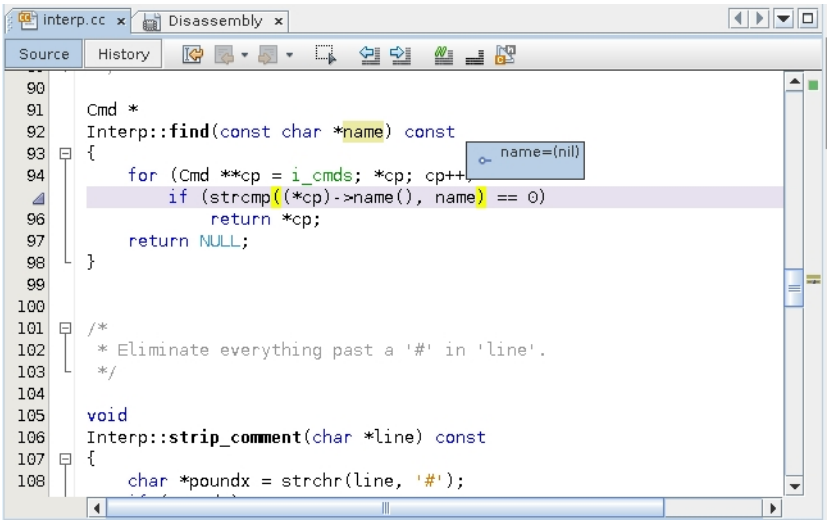
name の値が NULL であることに注意してください。その値が SEGV の原因である可能性が高いですが、もう一方のパラメータ (*cp)->name() の値を確認します。

2. 「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 name は "quit" になっており、問題ありません。



*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」ウィンドウ内をクリックし、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。

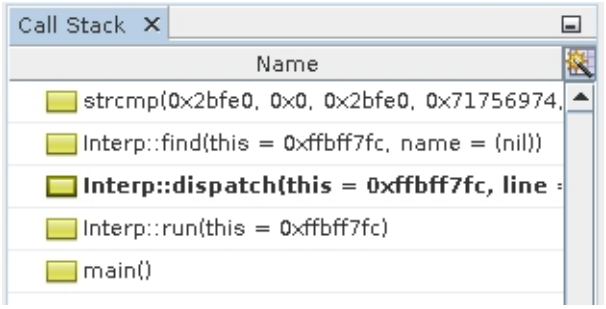


バルーン評価を使用すると、`(*cp)->name()` のような式の上にカーソルを置くこともできます。ただし、関数呼び出しを含む式のバルーン評価は次の理由で無効になります。

- デバッグしているのがコアファイルである。
- 「エディタ」ウィンドウにカーソルを偶然置いた結果、関数呼び出しに副作用が発生する場合がある。

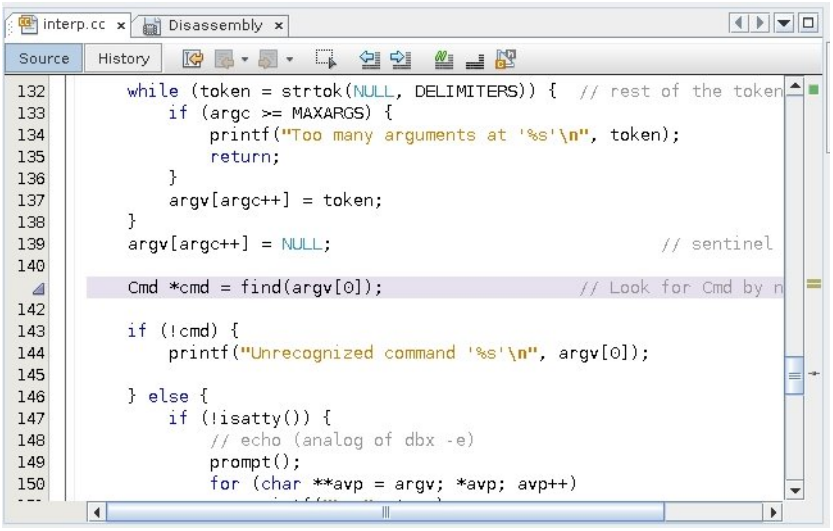
`name` の値を NULL にすることはできないため、この不正な値を `Interp::find()` に渡したコードを見つける必要があります。見つけるには:

1. 「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、ツールバーの「呼び出し元を現在に設定」ボタン (Alt+Page Down)  をクリックします。



2. 「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あり、`argv[0]` の値が NULL であること以外に手掛かりがありません。

この問題のデバッグは、ブレークポイントとステップ動作を使用して動的に行う方が簡単である可能性があります。

ブレークポイントとステップ動作の使用法

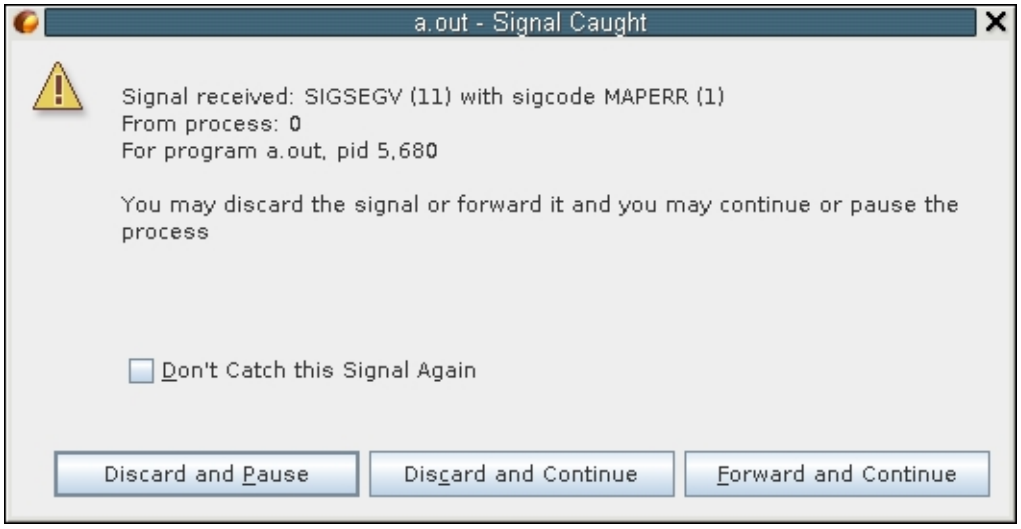
ブレークポイントにより、バグの出現箇所の前でプログラムを停止したり、何が間違っているかを検出するためにコードをステップスルーしたりできます。

まだ行なっていない場合は、「出力」ウィンドウの連結を解除します。

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

1. ツールバーの「再起動」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. 「デバッガ・コンソール」ウィンドウで Return キーを押します。

   警告ボックスに、SEGV に関する情報が表示されます。



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」ウィンドウで、`Interp::find()` 内の `strcmp()` 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、以前に `Interp::dispatch()` に表示された不明なコードに移動します。

5. 次のセクションでは、`find()` の呼び出し箇所の少し前にブレークポイントを設定し、コードをステップスルーして不具合の理由を特定できるようにします。

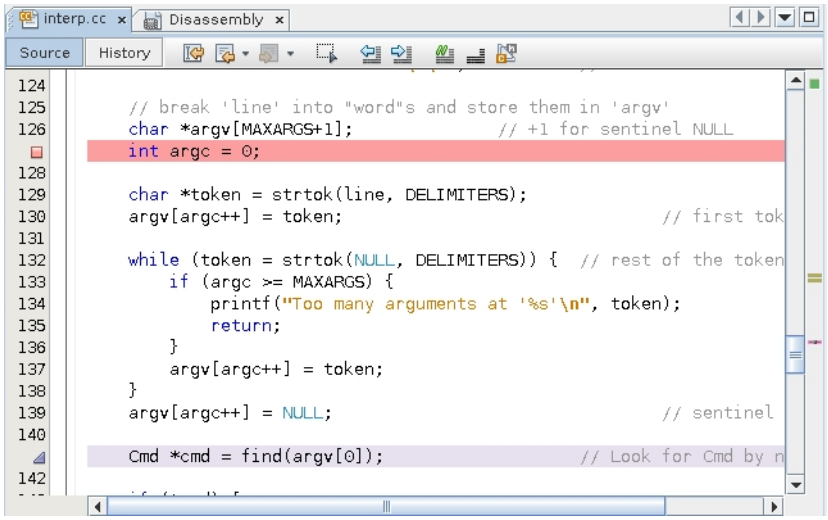
ブレークポイントを設定する

ブレークポイントは、行ブレークポイントや関数ブレークポイントなど、いくつかの方法で設定できます。次のリストで、ブレークポイントを作成する複数の方法について説明します。

注記 - 行番号が表示されていない場合は、左マージン内を右クリックし、「行番号を表示」オプションを選択して、エディタの行番号を有効にします。

■ **行ブレークポイントの設定**

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。

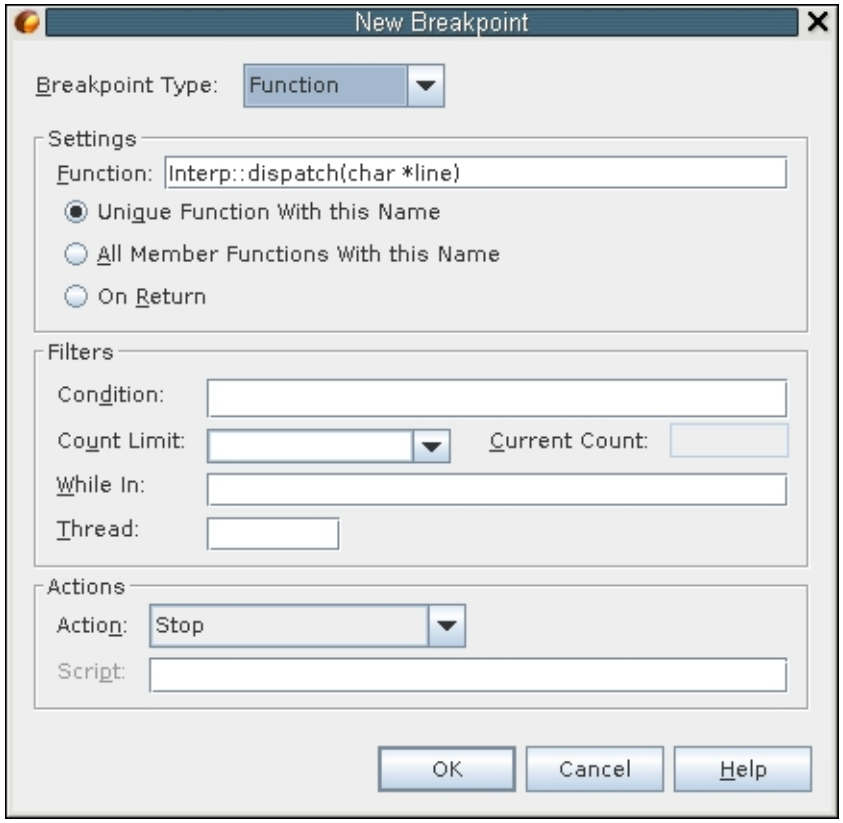


■ **関数ブレークポイントの設定**

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、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。

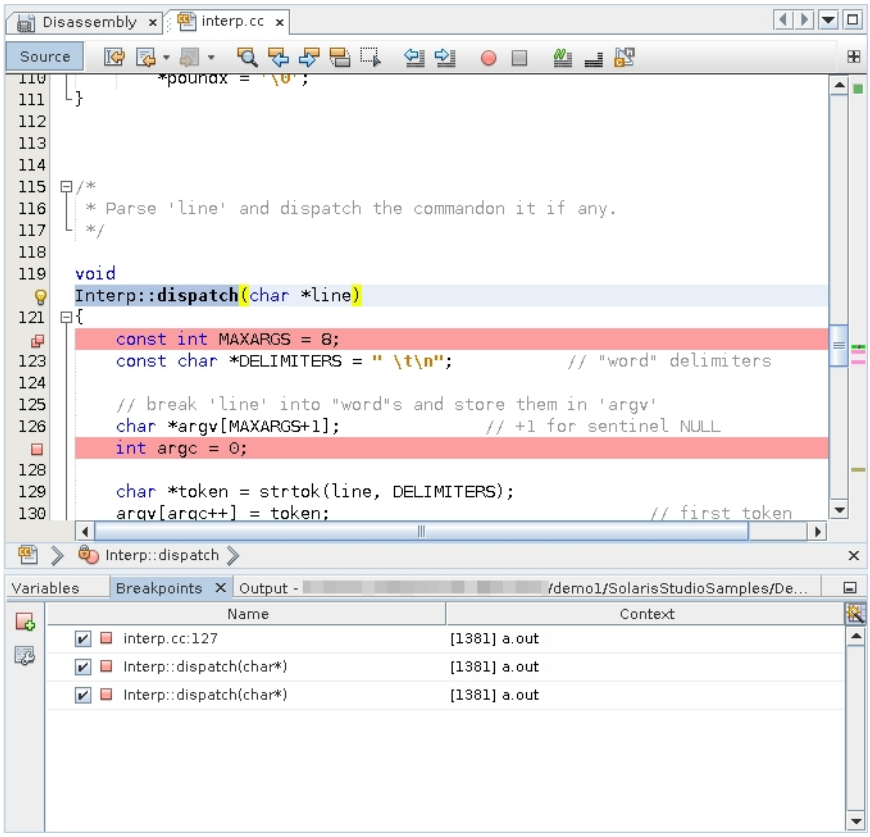
■ **コマンド行でのブレークポイントの設定**

関数ブレークポイントを設定するもっとも簡単な方法は、dbx コマンド行を使用することです。「デバッガ・コンソール」ウィンドウに `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かったことに注目してください。関数名を指定するだけで十分です。

「ブレークポイント」ウィンドウとエディタは、通常次のように表示されます。



エディタ内が乱雑になるのを避けるため、「ブレークポイント」ウィンドウを使用します。

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。

ブレークポイントの詳細は、『Oracle Solaris Studio 12.4: dbx コマンドによるデバッグ 』の第 6 章「ブレークポイントとトレースの設定」を参照してください。

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。

- 「デバッガ・コンソール」ウィンドウに `si dispatch` と入力すると、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必要がなくなります。
- エディタ内のテキストを選択すれば関数ブレークポイントを作成できるようになるため、ファイルを開かなくても、関数の呼び出し側で関数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` の構成ファイル `~/.dbxrc` に置きます。次に、一般的な例を示します。

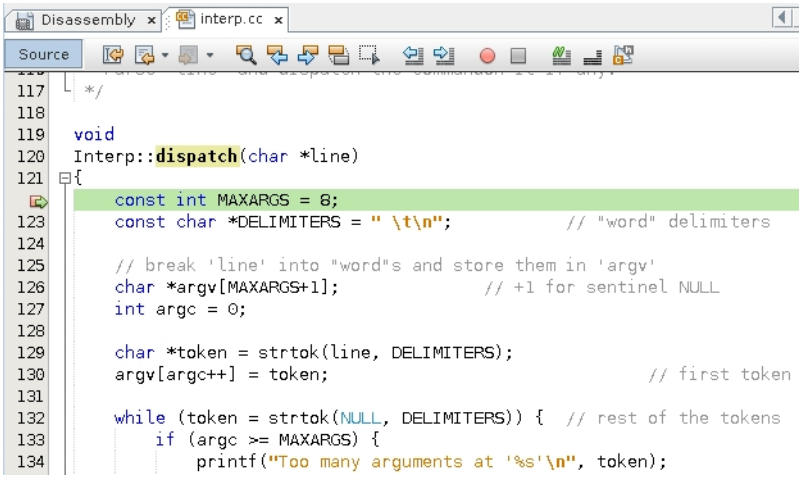
```
alias si stop in
alias sa stop at
alias s step
alias n next
alias r run
```

`.dbxrc` ファイルと dbxenv 変数のカスタマイズの詳細は、『Oracle Solaris Studio 12.4: dbx コマンドによるデバッグ 』の「dbxenv 変数の設定」を参照してください。

- 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説明的なものではありませんが、「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、「ソースへ移動」を選択するか、そのブレークポイントをダブルクリックすると、127 行目の内容を特定できます。
- 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永続させるため、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、行ブレークポイントが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

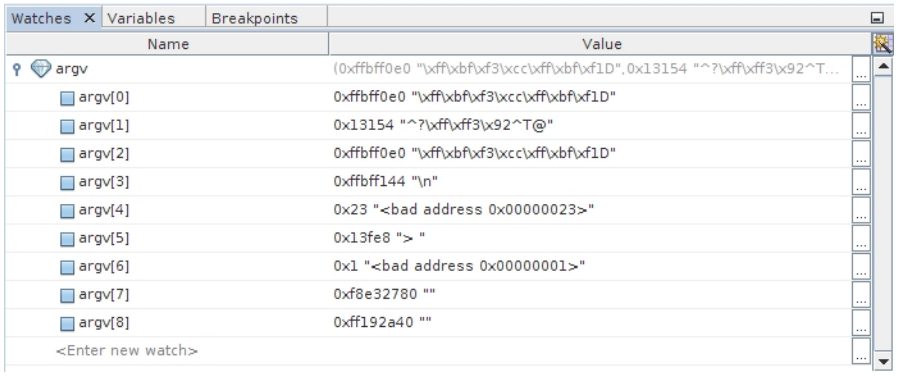
これで `Interp::dispatch()` にブレークポイントが 1 つできたので、「デバッガ・コンソール」ウィンドウで「再起動」  を再度クリックして Return キーを押すと、実行可能コードを含む `dispatch()` 関数の最初の行でプログラムが停止します。



`find()` に渡される `argv[0]` の問題が特定されたので、`argv` に対してウォッチポイントを使用します。

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。選択したテキストを含む「新規ウォッチ」ダイアログボックスが表示されます。

3. 「OK」をクリックします。
4. 「ウォッチ」ウィンドウを開くには、「ウィンドウ」->「ウォッチ」(Alt+Shift+2) を選択します。
5. 「ウォッチ」ウィンドウで、argv を展開します。



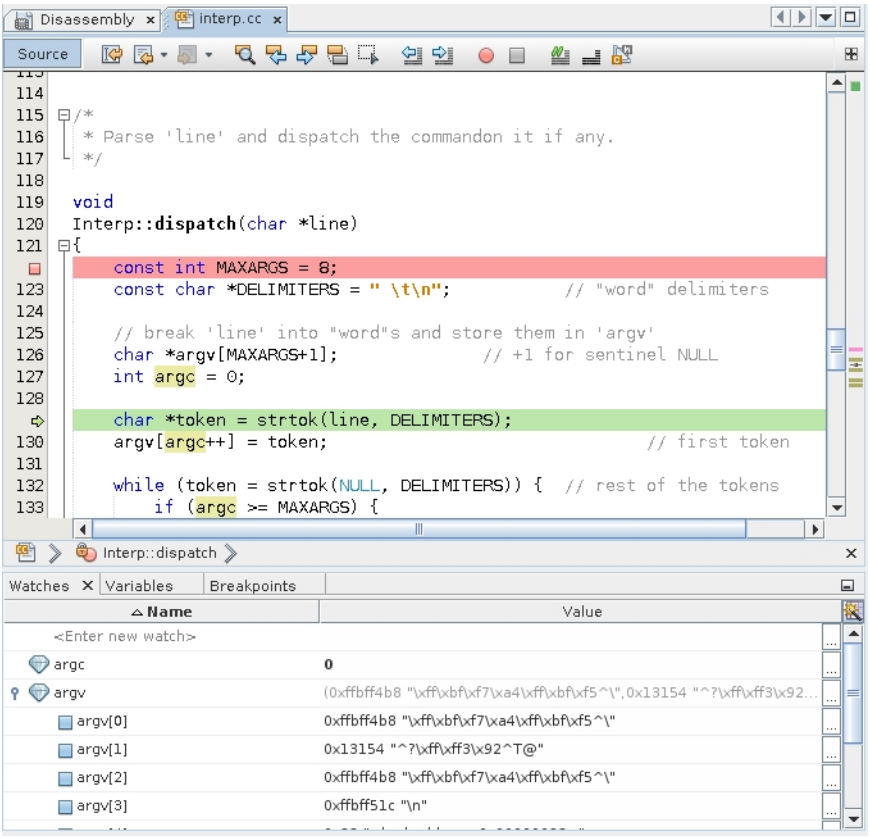
argv は初期化されておらず、argv はローカル変数であるため、前の呼び出しでスタックに残されたランダムな値を「継承」している可能性があることに注意してください。これは問題の原因でしょうか。

6. 緑色の PC の矢印が `int argc = 0;` を指すまで、「ステップ・オーバー」(F8)  を 2 回クリックします。
7. argc は argv のインデックスになるため、argc のウォッチポイントも作成します。argc も現時点では初期化されておらず、意図しない値が含まれている可能性があることに注意してください。

   argc のウォッチポイントは、argv のウォッチポイントのあとで作成したため、「ウォッチ」ウィンドウでは 2 番目に表示されます。
8. ウォッチポイント名をアルファベット順に並べるには、「名前」列見出しをクリックして列をソートします。次の図のソートの三角印を確認してください。

9. 「ステップ・オーバー」(F8)  をクリックします。

   argc は、初期化された値である 0 を示し、値がちょうど変更されたことを示す太字で表示されます。



   アプリケーションが strtok() を呼び出します。

10. 「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、token が NULL であることを監視します。

    strtok() 関数は、たとえば文字列をいずれかの DELIMITERS で区切られたトークンに分割するのに役立ちます。詳細は、strtok(3) のマニュアルページを参照してください。

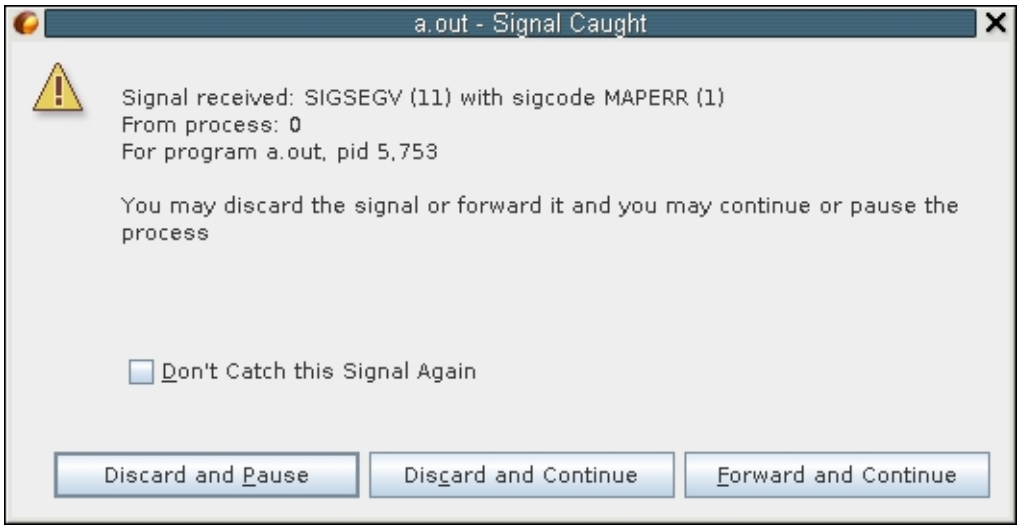
11. 「ステップ・オーバー」を再度クリックして、トークンを `argv` に割り当てます。次に、ループ内で `strtok()` の呼び出しが行われます。

    ステップオーバーすると、ループには入らずに (これ以上トークンがないため)、代わりに NULL が割り当てられます。

12. サンプルプログラムがどこでクラッシュしたかを特定するため、その割り当てもステップオーバーすると呼び出しのしきい値に達します。

13. プログラムがこの時点でクラッシュすることをダブルチェックするため、`find()` の呼び出しをステップオーバーします。

    「シグナルがキャッチされました」警告ボックスが再度表示されます。



14. 以前のように「破棄して一時停止 」をクリックします。

    `Interp::dispatch()` で停止したあとの `find()` の最初の呼び出しが、本当に不具合のある場所です。

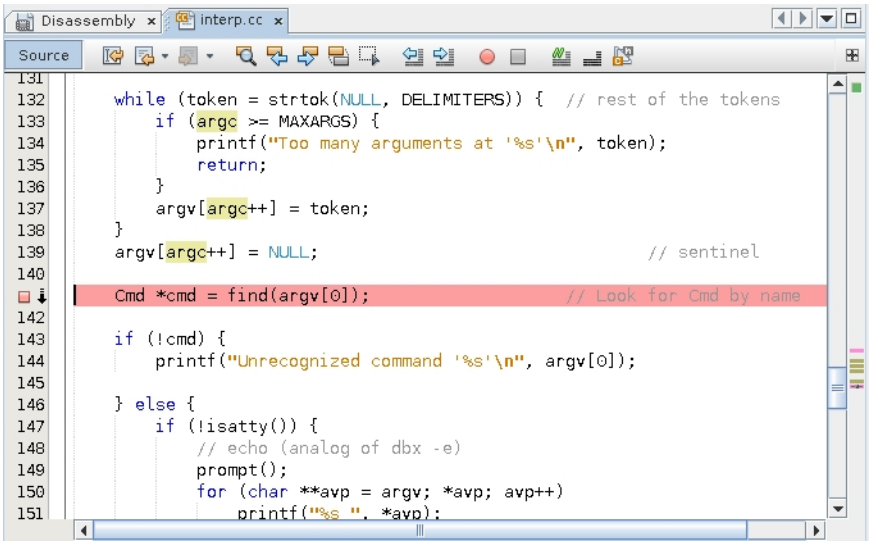
    最初に `find()` を呼び出した場所にすばやく戻ることができます。

    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` の表示は次の図のようになります。



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。

15. 「デバッガ・コンソール」ウィンドウで「再起動」  をクリックし、Return キーを押します。

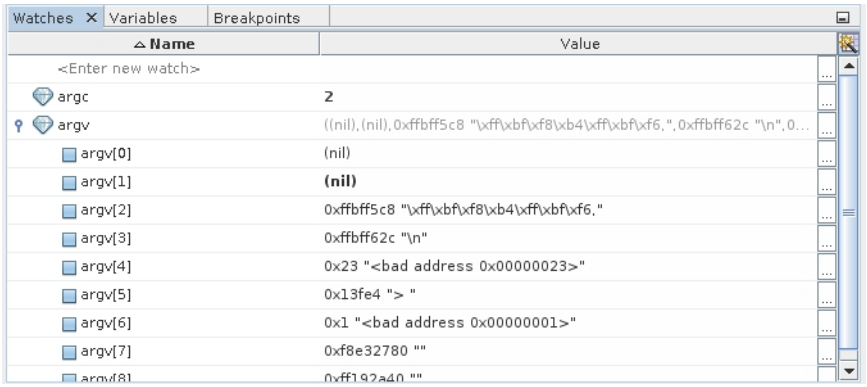
プログラムが `find()` の呼び出しの前に戻ります。「再起動」ボタンは再起動を実行します。デバッグ中は、初期の起動より再起動の方が頻繁に行われます。

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あとで、`dbxtool` を終了して、再起動する必要はありません。「再起動」ボタンをクリックすると、`dbx` はプログラム (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、デバッグ中の問題に対してすぐに使用できるように、`dbxtool` をデスクトップ上に、通常は最小化して保持することを検討してください。

---

16. バグはどこにあるでしょうか。ふたたびウォッチポイントをみてみましょう。



行が空であり、トークンを持っていなかったため、`strtok()` の最初の呼び出しが NULL を返したため、`argv[0]` が NULL であることに注意してください。

必要に応じて、このチュートリアルの残りに進む前にこのバグを修正してください。

デバッガでプログラムを実行する場合は、45 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」の説明に従って、デバッガでコードにパッチを適用できます。

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

この例は、誤動作するプログラムを不具合が発生する前の時点で停止し、コードの意図と実際のコードの動作を比較しながらコードをステップスルーする、もっとも一般的なデバッグパターンを示しています。

次のセクションでは、この例で使用したステップ動作とウォッチポイントの一部を回避するためにブレークポイントを使用する高度な技術について説明します。

高度なブレークポイント技術の使用法

このセクションでは、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

このセクション、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。

注記 - このセクションに示した正確な出力を得るには、プログラム例にまだバグが含まれている必要があります。バグを修正した場合は、http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-samples-1408618.html から `SolarisStudioSampleApplications` ディレクトリを再ダウンロードしてください。

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、これを使用してプログラム例でバグを発生させます。`in` には次のコードが含まれています。

```
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
```

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'

      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
```

この出力を見て量が多い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ステップスルーまたは追跡することが困難な場合に使用する技術を解説することを主眼としています。

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() 関数では、特定の状況での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。

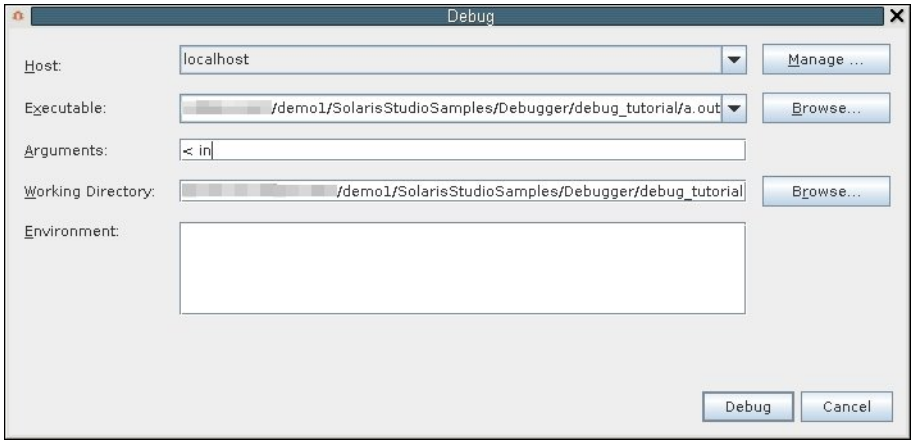
ステップ 1: 再現性

最初のステップでは、デバッグターゲットを設定し、「再起動」  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ターゲットを構成します。
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」を選択します。
3. 「実行可能ファイルをデバッグ」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または入力します。
4. 「引数 」フィールドで、次のものを入力します。

   ```
   < in
   ```

   実行可能ファイルのパスのディレクトリ部分が「作業ディレクトリ」フィールドに表示されます。
5. 「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。

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」を選択してから、目的の実行可能ファイルを選択すると、同じデバッグ構成を使用できます。

これらのプロパティーの多くを dbx コマンド行で設定できます。これらはデバッグターゲット構成に格納されます。

次の技術は、簡単な再現方法を維持するのに役立ちます。ブレークポイントを追加したときは、途中のさまざまなブレークポイントで「継続」をクリックしなくても、「再起動」をクリックすると目的の場所にすばやく移動できます。

ステップ 2: 最初のブレークポイント

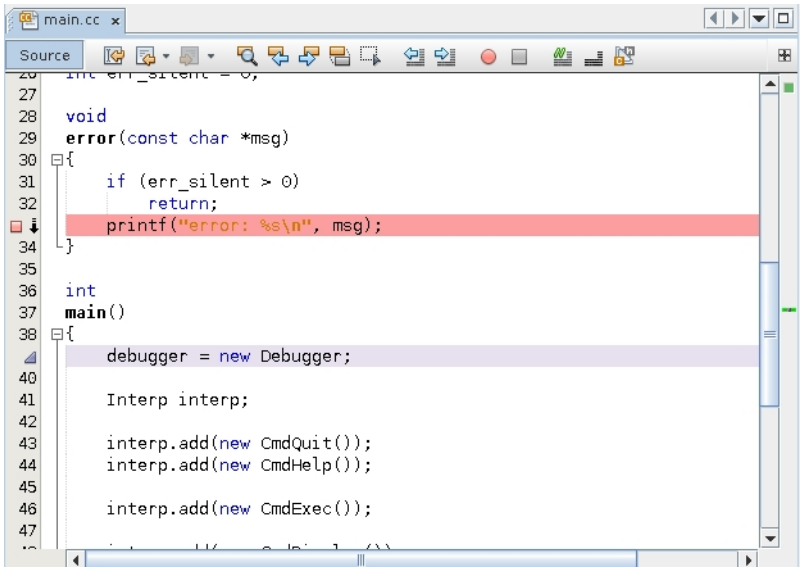
エラーメッセージが出力される場合は、error() 関数の内部に最初のブレークポイントを置きます。このブレークポイントは、33 行目の行ブレークポイントとなります。

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、次のように入力すると「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。

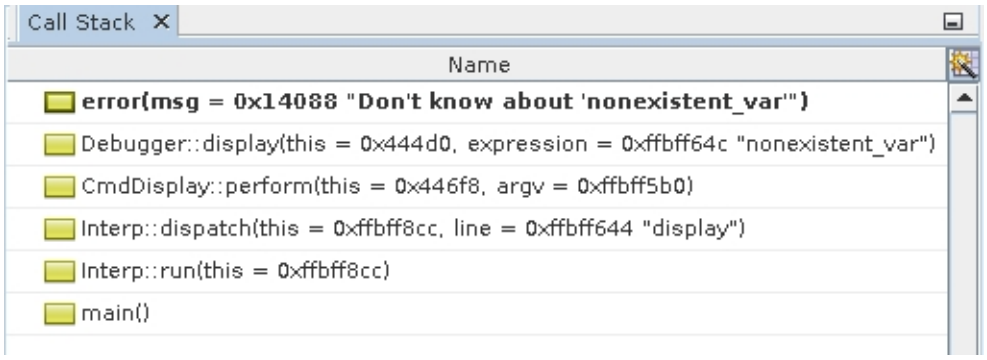
1. 33 番目の上の「エディタ 」ウィンドウの左マージンをクリックして、行ブレークポイントを作成します。



2. 「再起動」 をクリックしてプログラムを実行し、ブレークポイントにヒットすると、in ファイル内のシミュレートされたコマンドによって生成されたエラーメッセージがスタックトレースに表示されます。

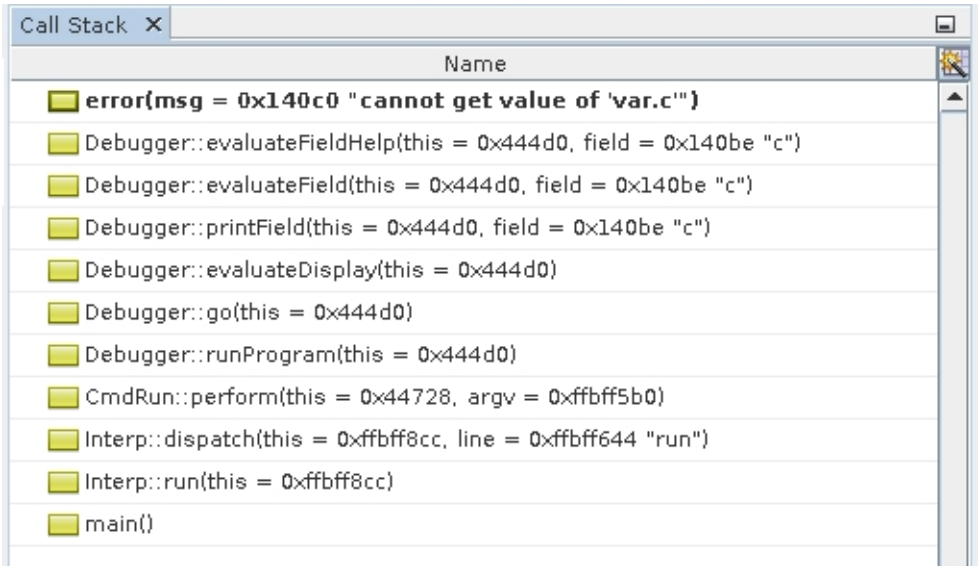
```
> display var # should yield an error
```

`error()` 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すると、再度ブレークポイントにヒットします。

予期しないエラーメッセージが表示されます。



ステップ 3: ブレークポイントカウント

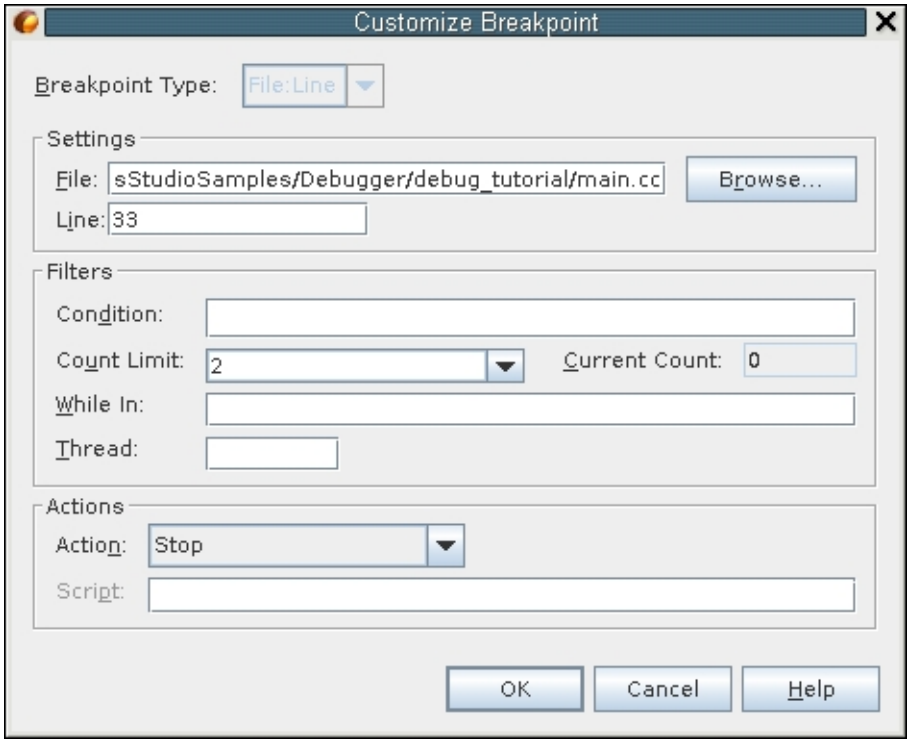
コマンドによるブレークポイントの最初のヒットの後で、「継続 」をクリックする必要なく、実行ごとに繰り返してこの場所に到着する方がよいでしょう。

```
> 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、使用している入力シーケンスはこのバグを再現する鍵となる可能性があるため、入力を変更する必要はありません。

再度このブレークポイントに到達するには、カウントを 2 に設定します。

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。



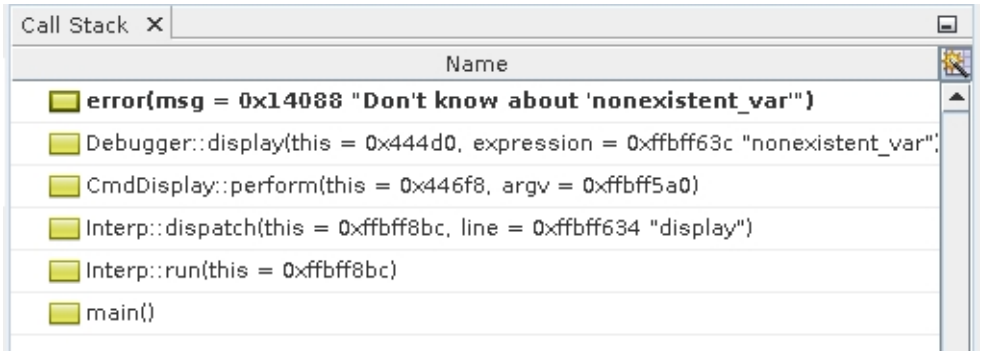
これで、目的の場所に繰り返し到達できます。

この場合は、カウントとして 2 を選択したので特に問題はありませんでした。しかし、目的の場所が何度も呼び出される場合もあります。適切なカウント値を簡単に選択するには、37 ページの「ステップ 7: カウント値の確認」を参照してください。ここでは、`error()` での停止を呼び出しでのみ行う別の方法について説明します。

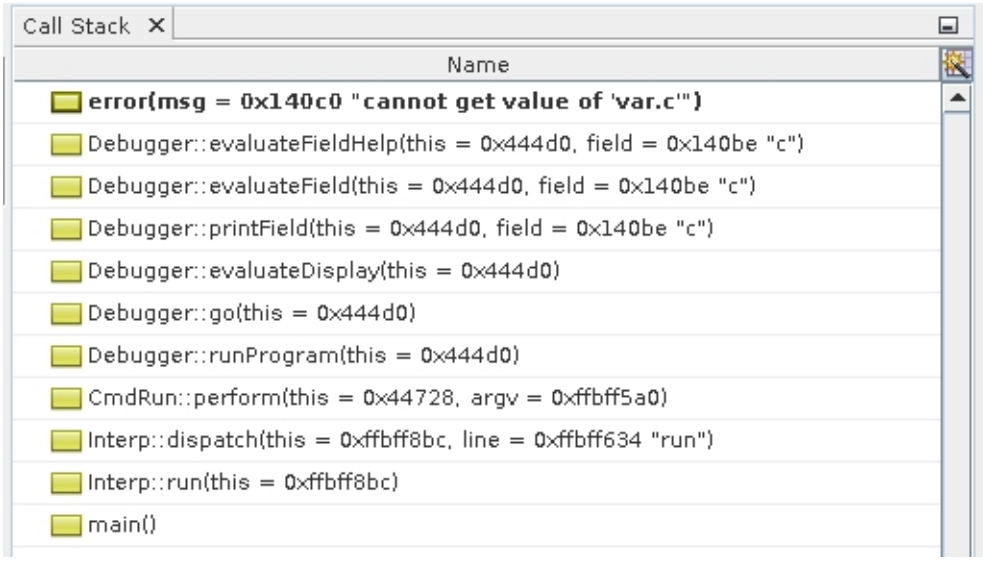
ステップ 4:境界ブレークポイント

1. `error()` 内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」ダイアログボックスを開き、「数の制限」のドロップダウンリストから「常に停止」を選択して、ブレークポイントのカウントを無効にします。
2. プログラムを再実行します。

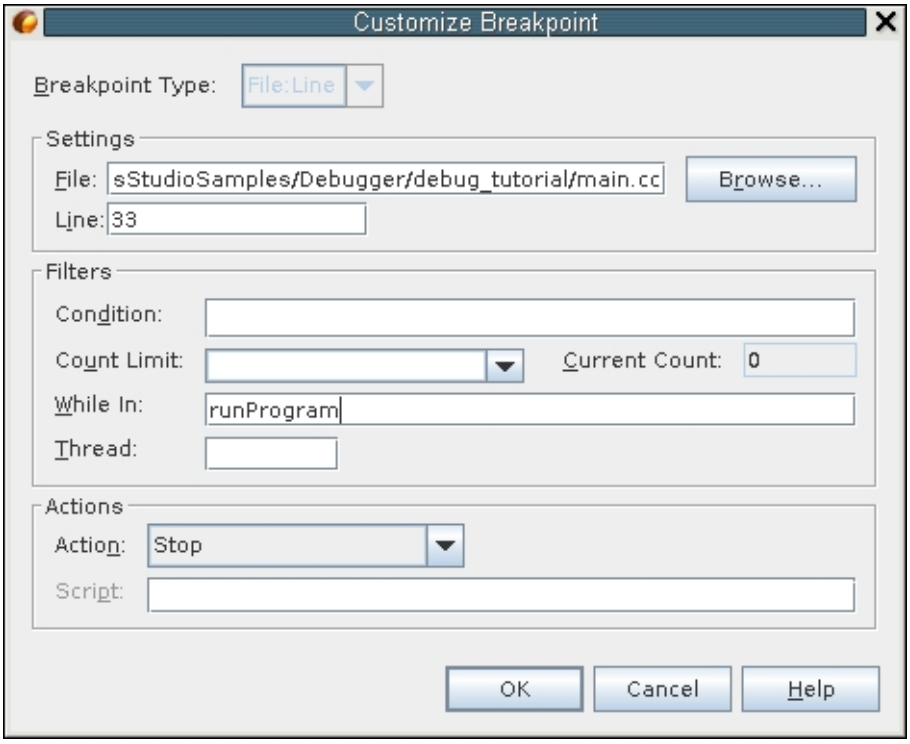
   `error()` で 2 回停止するときに、スタックトレースを確認します。1 回目の `error()` での停止は、次のような画面になります。



2 回目の `error()` での停止は、次のような画面になります。



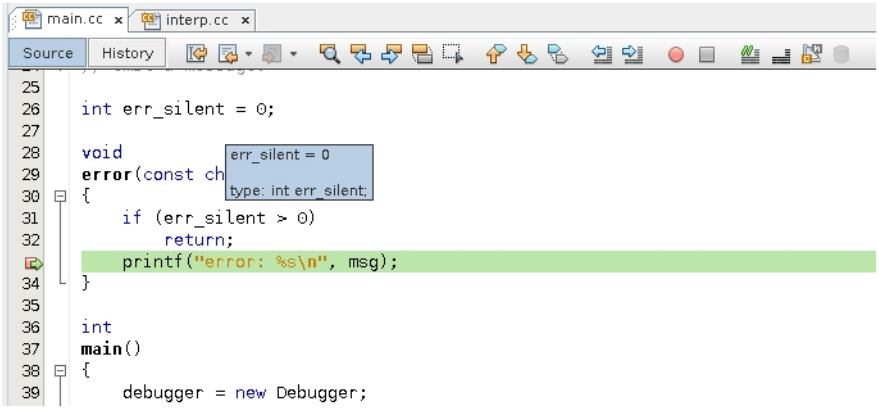
`runProgram` (フレーム [7]) から呼ばれるときにこのブレークポイントで停止するように調整するため、「ブレークポイントをカスタマイズ」ダイアログボックスを再度開き、「While In」フィールドを runProgram に設定します。



ステップ 5: 原因の調査

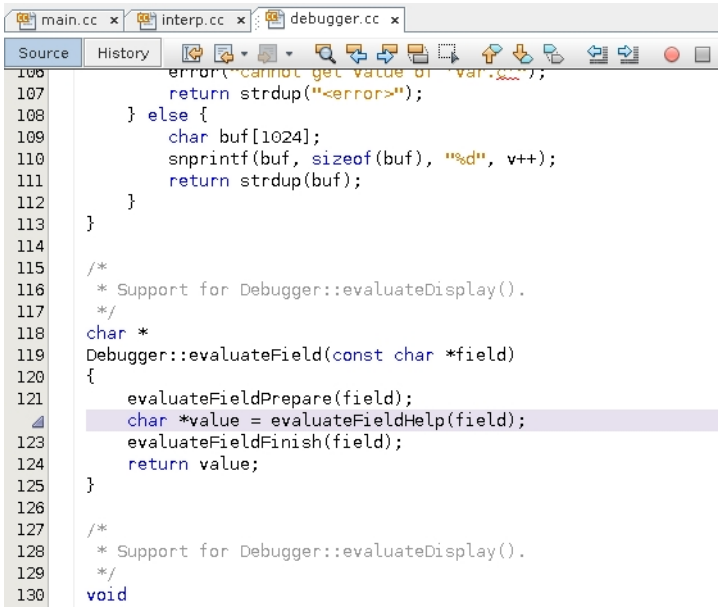
`err_silent` が 0 より大きくないため、意図しないエラーメッセージが出力されます。バルーン評価を使用して `err_silent` の値を調べます。

1. 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。

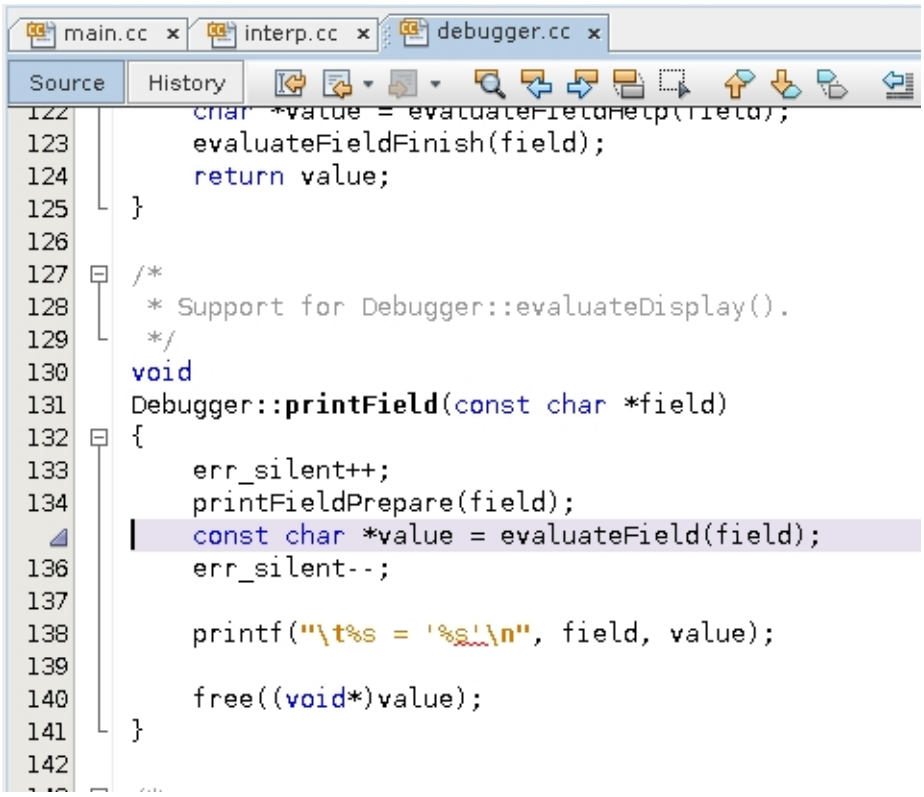


スタックを追跡して、`err_silent` が設定された場所を確認します。

2. 「呼び出し元を現在に設定」 を 2 回クリックすると、evaluateField() に到達し、この関数はすでに evaluateFieldPrepare() を呼び出し、err_silent を操作している可能性がある複雑な関数をシミュレートします。



3. 「呼び出し元を現在に設定」を再度クリックすると、printField() に到達し、err_silent が増分されます。printField() はすでに printFieldPrepare() を呼び出し、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。



一部のコードが `err_silent++` と `err_silent--` に囲まれていることを注目してください。

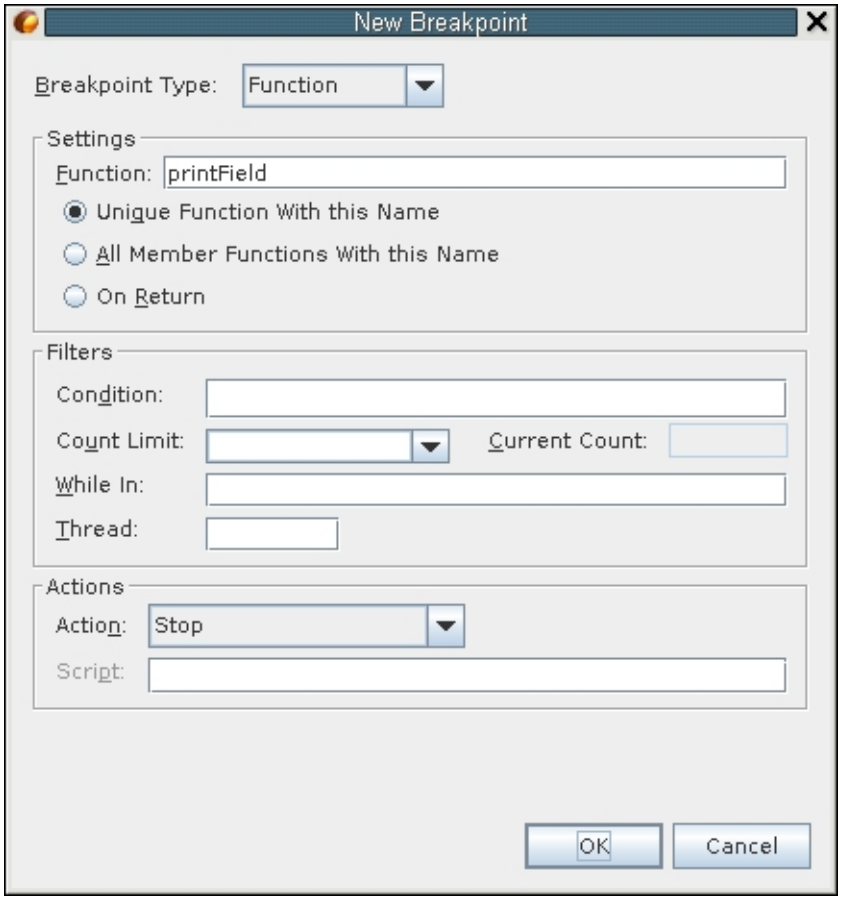
`err_silent` は、`printFieldPrepare()` または `evaluateFieldPrepare()` のいずれかで不正になったか、`printField()` に制御が到達したときにすでに不正だった可能性があります。

ステップ 6: より多くのブレークポイントカウント

`printField()` の呼び出し前と呼び出し後のどちらで `err_silent` が不正になったかを特定するため、`printField()` にブレークポイントを置きます。

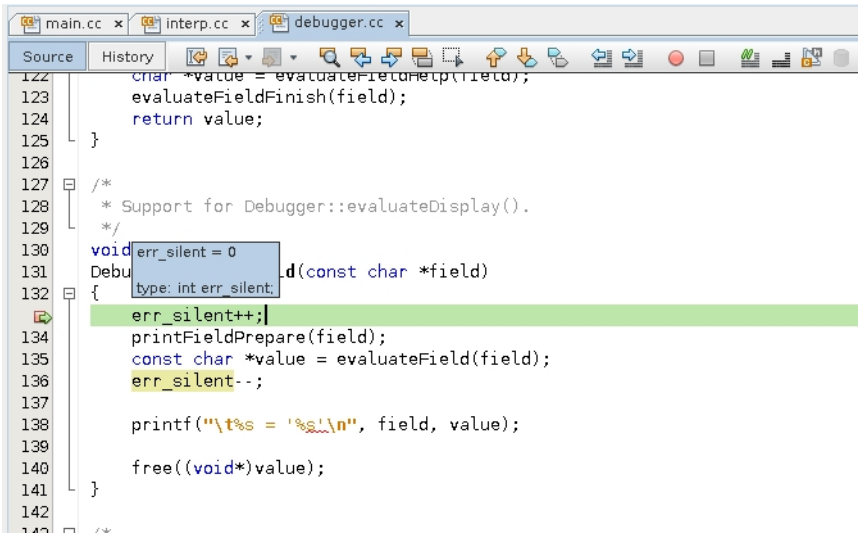
1. `printField()` を選択し、「新規ブレークポイント」を右クリックして、選択します。

   新しいブレークポイントのタイプが事前に選択され、「関数」フィールドに `printfield` が事前に入力されます。
2. 「OK」をクリックします。



3. 「再起動」  をクリックします。

   ブレークポイントに最初にヒットするのは、最初の実行中に最初に停止したときの最初のフィールド `var.a` 上です。`err_silent` は 0 で、これは問題ありません。



4. 「継続 」をクリックします。

   `err_silent` は引き続き問題ありません。

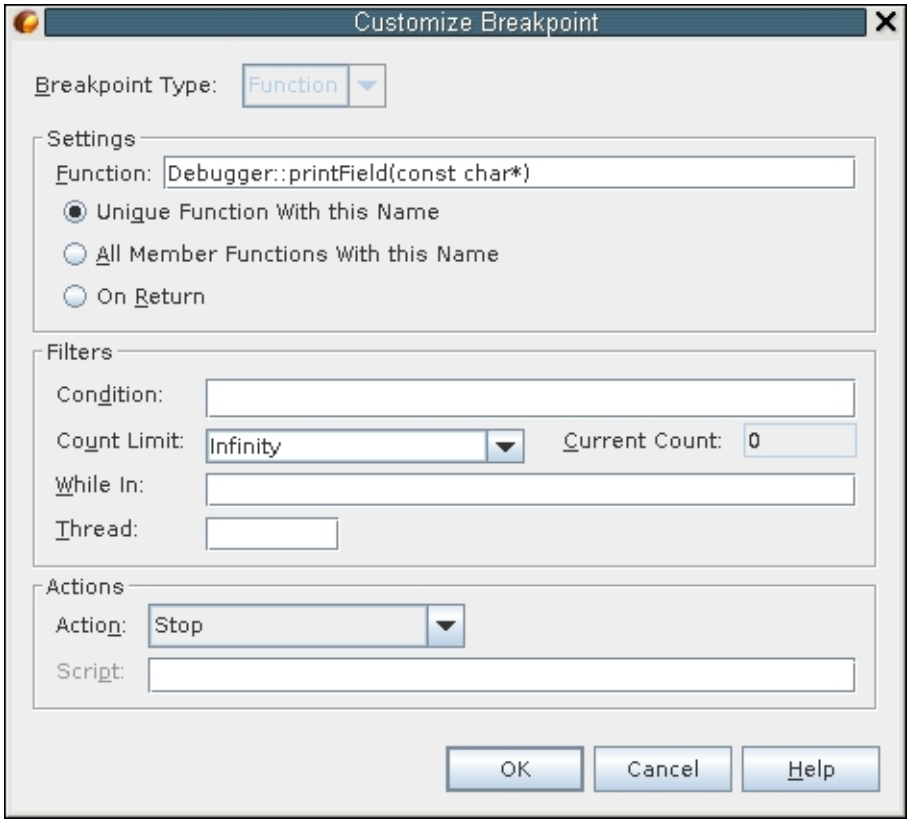
5. ふたたび「継続 」をクリックします。

   `err_silent` は引き続き問題ありません。

意図しないエラーメッセージが発生する `printField()` の呼び出しに到達するまで、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、表示される実行、停止、およびフィールドをカウントすることもできますが、実際のプロセスはもっと複雑である可能性があります。カウントを半自動的に確認する方法があります。

ステップ 7: カウント値の確認

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。



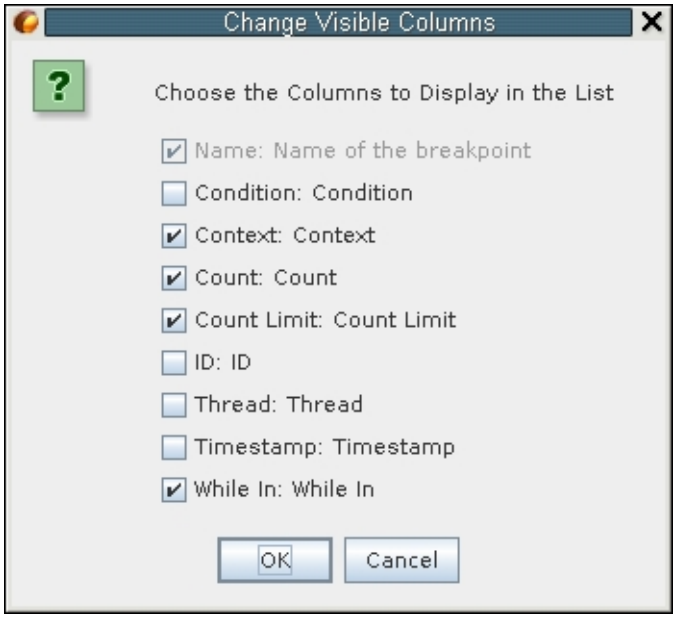
この設定は、このブレークポイントで停止しないことを意味します。ただし、依然として数えています。

2. カウントなどの追加のプロパティーが表示されるように「ブレークポイント」ウィンドウを設定します。

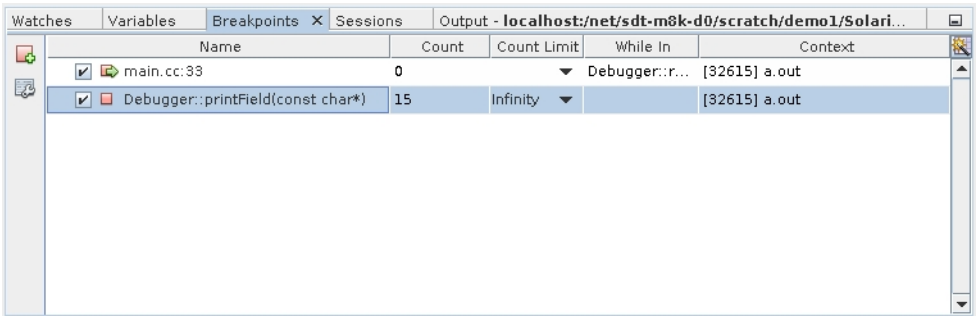
   a. 「ブレークポイント」ウィンドウの右上隅にある「表示可能項目の変更」ボタン  をクリックします。

   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. プログラムを再度実行します。`error()` 内の、`runProgram()` を境界とするブレークポイントにヒットします。
4. `printField()` 上のブレークポイントのカウントを確認します。



カウントは 15 です。

5. 再度「ブレークポイントをカスタマイズ」ウィンドウで、「数の制限」列のドロップダウンリストをクリックし、「現在のカウントの値を使用」を選択して現在のカウントを「数の制限」に転送し、「OK」をクリックします。

ここでプログラムを実行すると、意図しないエラーメッセージの前に最後に呼び出された `printField()` で停止します。

ステップ 8: 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。もっとも可能性の高い原因は、printField() に到達する前に、ある err_silent-- の実行回数が多すぎたか、ある err_silent++ の実行回数が少なすぎたことです。

このような小さいプログラムでは、コードを入念に検査することで、この err_silent のミスマッチのペアを特定できます。ただし大規模なプログラムには、膨大な数の次のペアが含まれてい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。

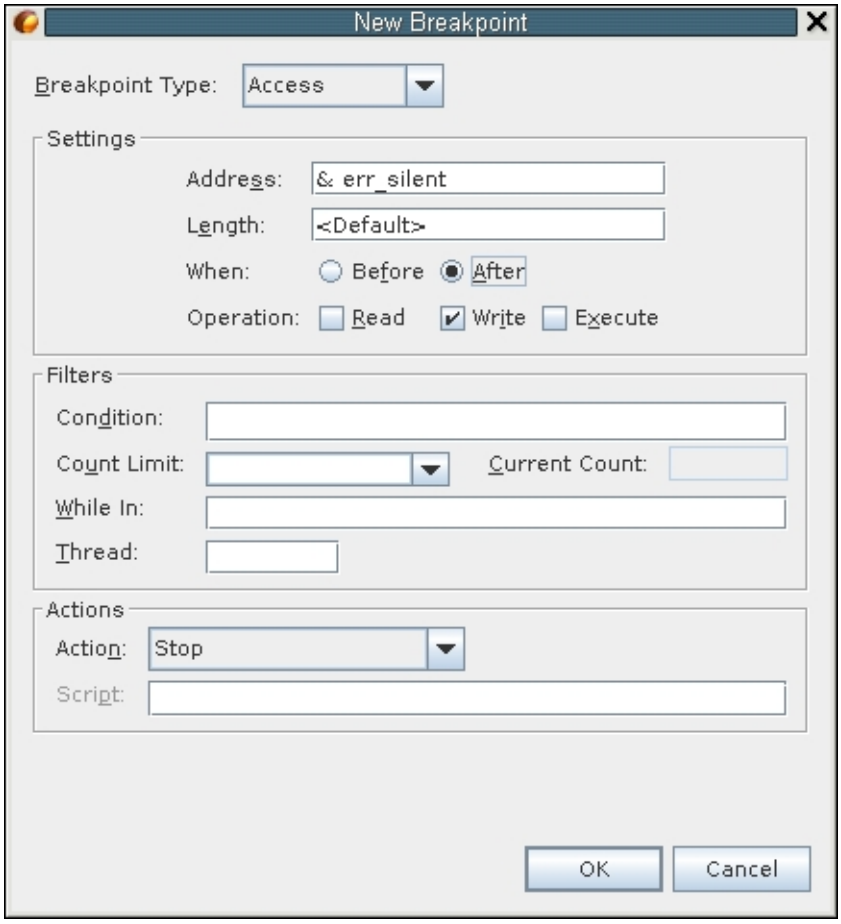
エラーの原因が、err_silent++; と err_silent--; のセットのミスマッチではなく、err_silent の内容を上書きする不正なポインタである可能性もあります。ウォッチポイントはそのような問題をとらえるのにより効果的でしょう。

ステップ 9: 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」セクションが変更され、「アドレス」フィールドが & err_silent になることに注意してください。
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. プログラムを実行します。

   `init()` で停止します。`err_silent` が 1 に増分され、そのあと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 -1 にはならず、問題はありません。

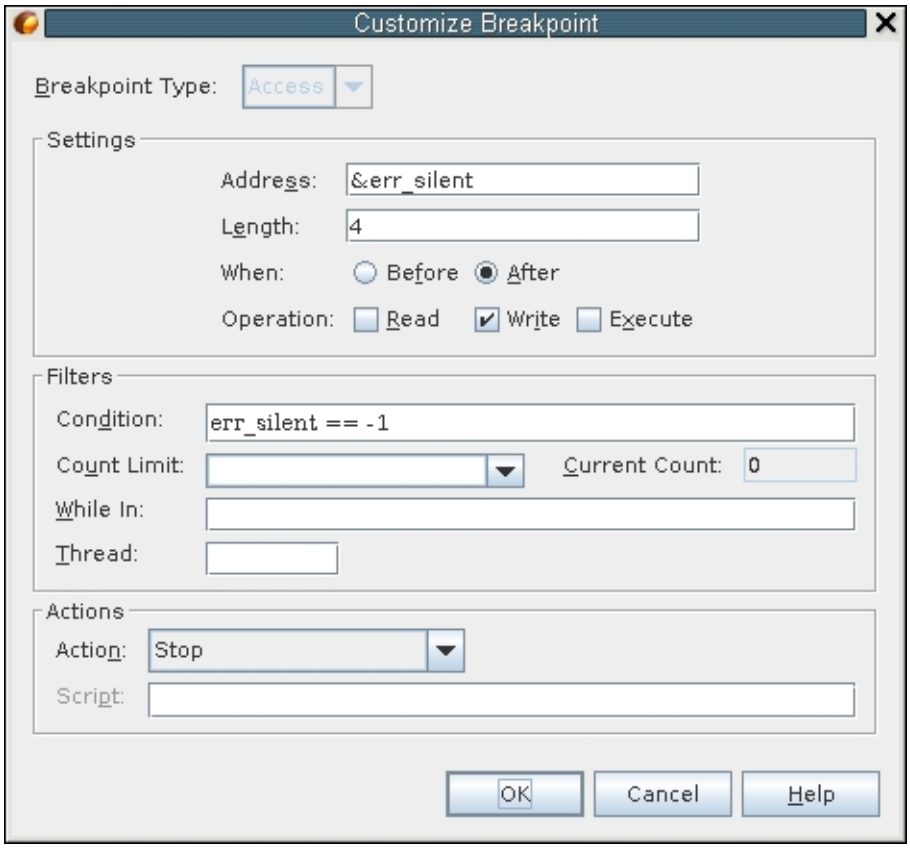
`err_silent` が -1 に設定されるまで何度も「継続」をクリックする代わりに、ブレークポイント条件を設定できます。

ステップ 10: ブレークポイント条件

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。

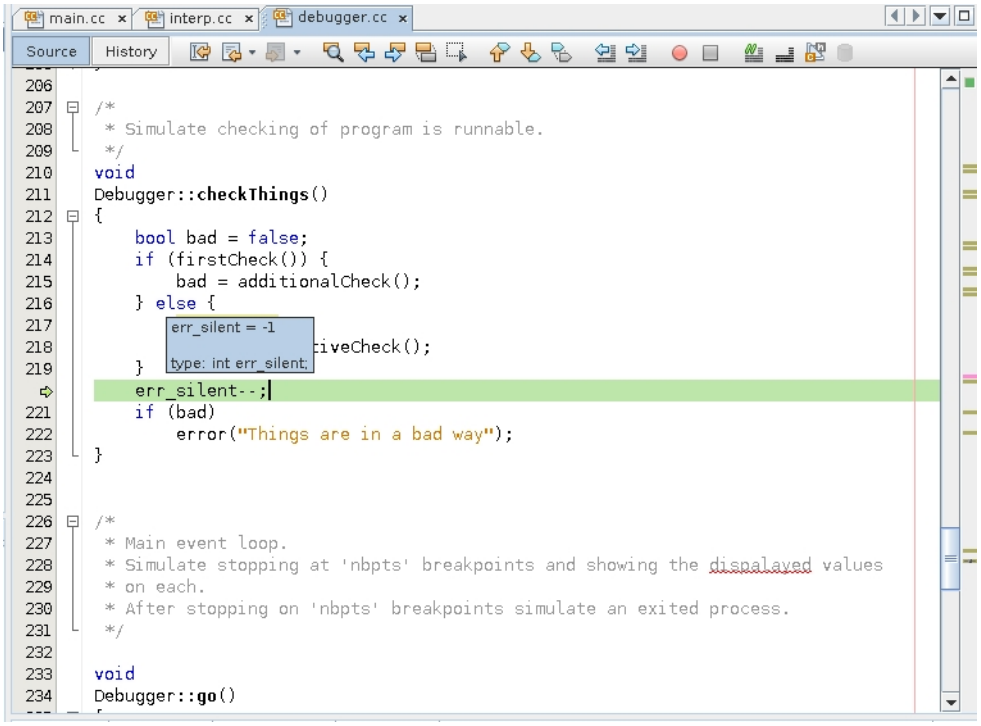
1. 「ブレークポイント」ウィンドウで、「後」の「書き込み」ブレークポイントを右クリックして、「カスタマイズ」を選択します。
2. 「時間」フィールドで「後」が選択されていることを確認します。

   「後」を選択すると、`err_silent` の変更後の値を確認できます。
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。



5. プログラムを再度実行します。

   `checkThings()` で停止します。ここではじめて `err_silent` が -1 に設定されます。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 11: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する方法の 1 つは、`checkThings()` に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。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して `checkThings()` の正しい呼び出しに到達できますが、最近実行された処理をよりすばやく再現するには、スタックをポップします。

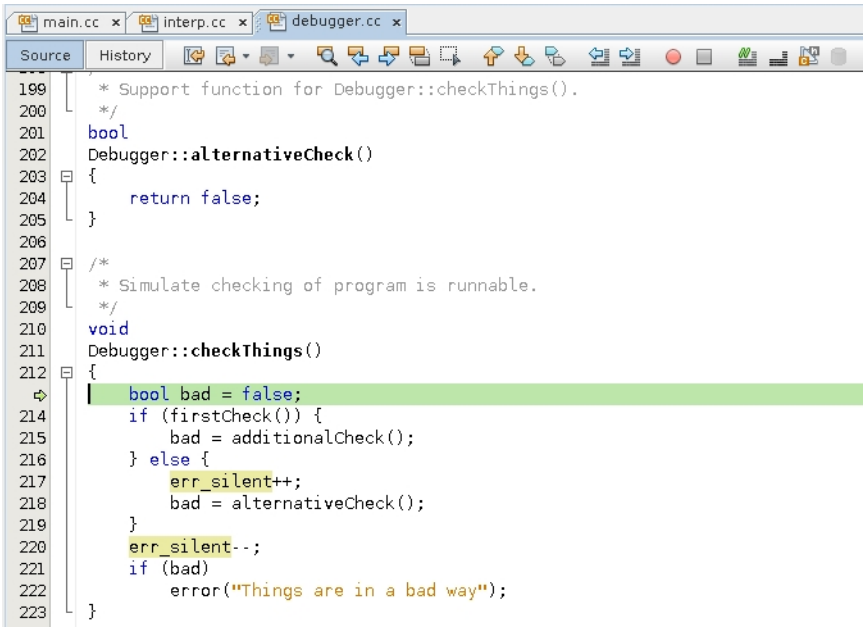
1. 「デバッグ」->「スタック」->「最上位の呼び出しをポップ」を選択します。

   「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、データデバッグからコントロールフローデバッグに切り替えているため、`err_silent` の値はすでに間違っています。

   プロセスの状態は、`checkThings()` の呼び出しを含む行の最初に戻ります。

2. 「ステップイン」  をクリックすると、`checkThings()` が再度呼び出されるたびに監視できます。

   `checkThings()` をステップスルーするにつれて、プロセスが `if` ブロックを実行することを確認でき、ここで `err_silent` は増分されず、-1 に減らされます。

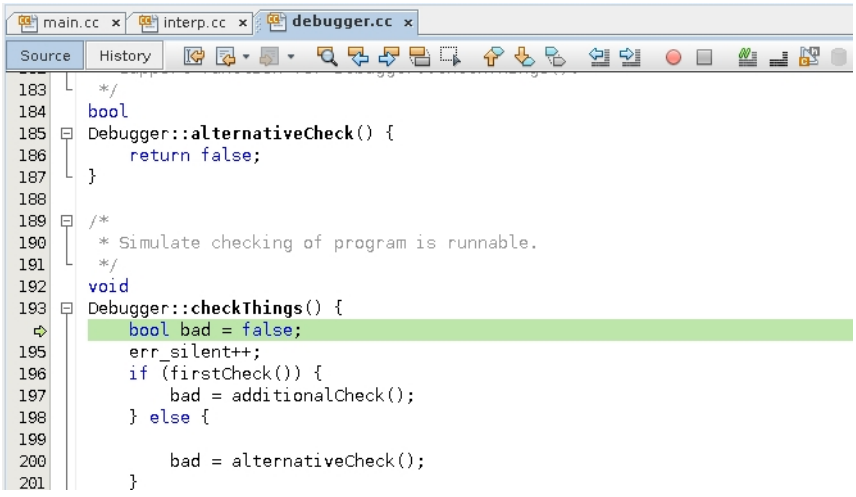


プログラミングエラーが見つかったようですが、それをトリプルチェックすることをお勧めします。

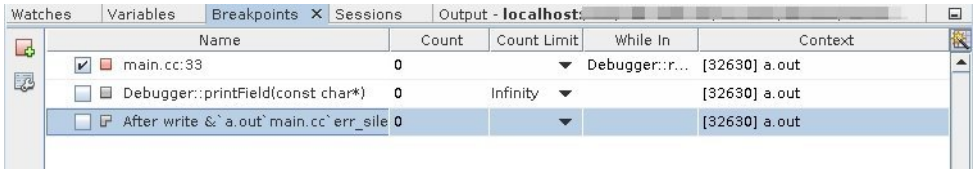
ステップ 12: 診断をさらに確認するための修正の使用法

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。

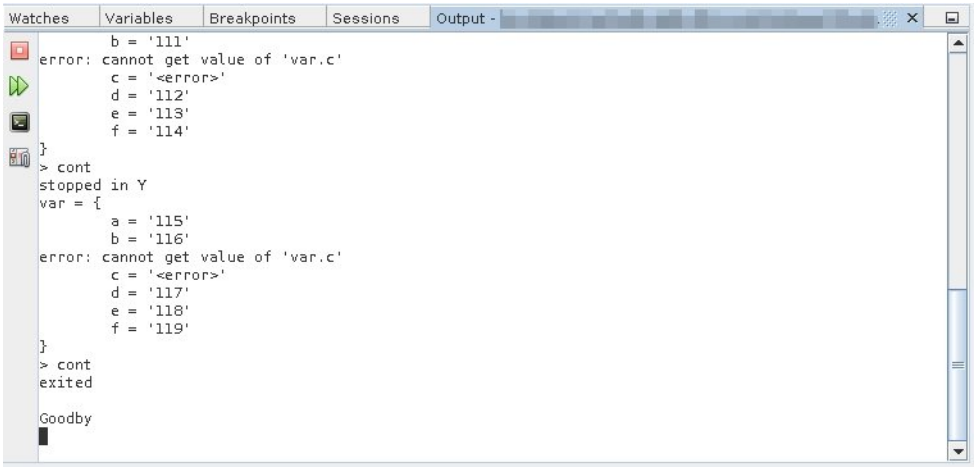


2. 「デバッグ」>「コード変更の適用」を選択するか、「コード変更の適用」ボタン  を押します。
3. `printField` ブレークポイントおよびウォッチポイントを無効にしますが、`error()` のブレークポイントは有効なままにします。



4. プログラムを再度実行します。

`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を確認してください。



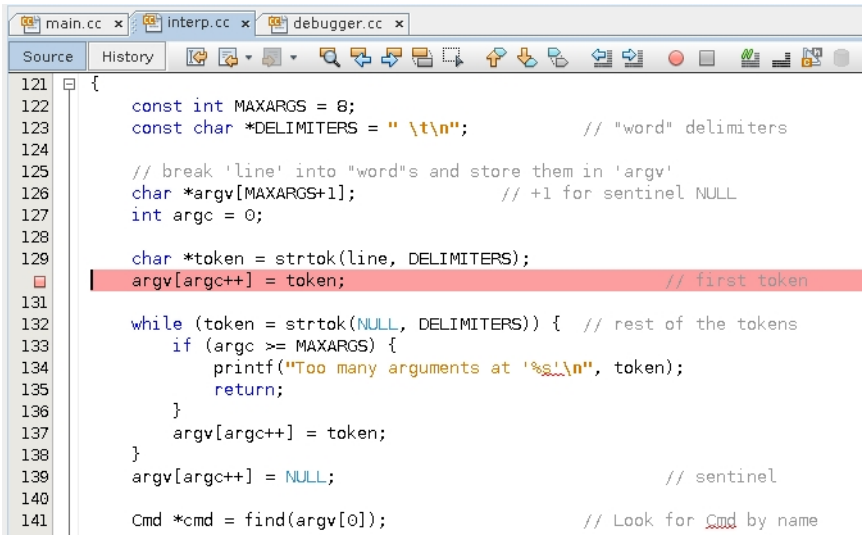
ディスカッション

この例は、15 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ます。つまり、誤動作するプログラムを不具合が発生する前の時点で停止し、コードの意図と実際のコードの動作を比較しながら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

15 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。エラーの回避策を使用できます。

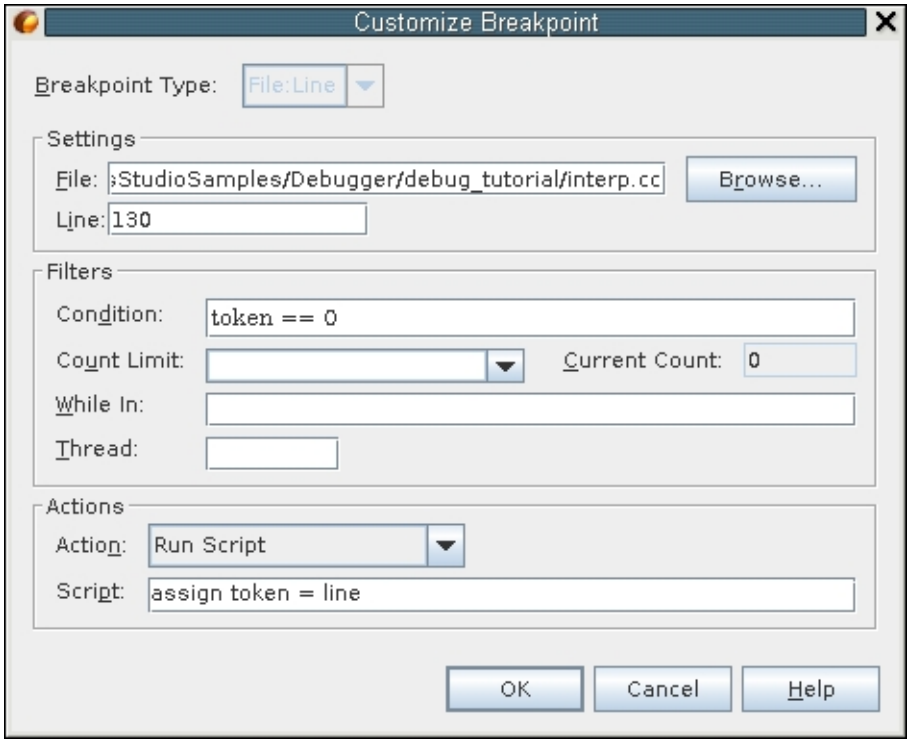
1. 前に作成したブレークポイントをすべて削除します。これをすばやく行うには、「ブレークポイント」ウィンドウを右クリックして「すべて削除」を選択します。
2. 「実行可能ファイルをデバッグします」ダイアログボックスの `<in` 引数を削除します。
3. `interp.cc` の 130 行目の行ブレークポイントを切り替えます。



4. 「ブレークポイント」ウィンドウで、先ほど作成した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。
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。

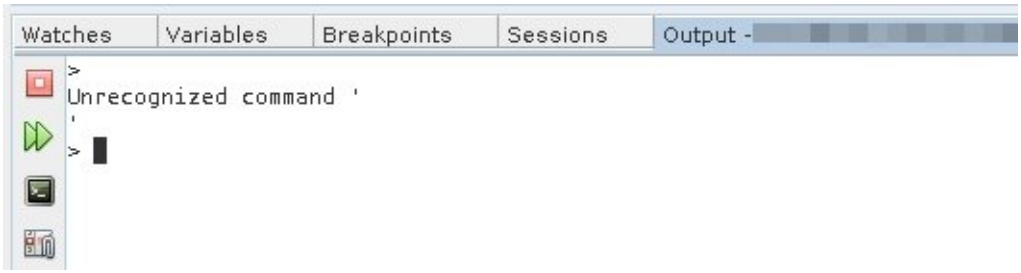
注記 - dbx ではデバッグプロセスの `dummy` 文字列を割り当てることができないため、`assign token = "dummy"` にはできません。その一方で `line` は `""` に等しいことが知られています。

ダイアログボックスは次のような画面になります。



8. 「OK」をクリックします。

ここで、プログラムを実行して空の行を入力すると、クラッシュせずに、次の画面に示すような警告が表示されます。



`dbxtool` から `dbx` に送信されたコマンドを見れば、この回避策がより明確になる可能性があります。

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

結論

Oracle Solaris Studio の `dbxtool` では、便利な GUI 形式を使用しながら、プログラムがクラッシュする原因となる問題領域を特定できます。`dbxtool` では、ブレークポイントを作成してコードをステップスルーすることにより、コードを簡単にデバッグできます。`dbxtool` では、高度なブレークポイント技術とウォッチポイン

ト、ブレークポイント条件、スタックのポップなどの機能を組み合わせることで、コード内のバグを特定し、それらの問題を修正することもできます。

ORACLE