Oracle® Solaris Studio 12.3 コードアナライザチュートリアル

2011 年 12 月

ORACLE®

概要

Oracle Solaris Studio コードアナライザは、Oracle Solaris 向けの C および C++ アプリケーションの開発者を支援するための統合ツールセットで、セキュリティーと品質の高い堅牢なソフトウェアを作成できるように設計されています。

コードアナライザには 3 種類の分析があります。

- コンパイルの一環としての静的コード検査
- 動的メモリーアクセス検査
- コードカバレージ分析

静的コード検査は、コード内の一般的なプログラミングエラーをコンパイル時に検出します。新しいコンパイラオプションは、Oracle Solaris Studio コンパイラの実績ある幅広い制御およびデータフロー分析フレームワークを活用して、アプリケーションのプログラミングおよびセキュリティー上の潜在的な欠陥を分析します。

コードアナライザは、メモリーエラー検出ツールである Discover で収集された動的メモリーデータを使用して、アプリケーションの実行時にメモリー関連のエラーを検出します。Studio のコードカバレージツールである Uncover で収集されたデータを使用して、コードカバレージを測定します。

個々の分析へのアクセスを提供するほかに、コードアナライザは静的コード検査と動的メモリーアクセス検査を統合して、コードで見つかるエラーの信頼度を高めます。静的コード検査を動的メモリーアクセス分析およびコードカバレージ分析とともに使用することで、単独で機能するほかのエラー検出ツールでは見つけることのできない多くの重要なエラーをアプリケーション内に見つけることができます。

サンプルアプリケーションの入手

このチュートリアルではサンプルプログラムを使用して、Oracle Solaris Studio コンパイラ、Discover メモリーエラー検出ツール、Uncover コードカバレージツール、およびコードアナライザ GUI を使用して一般的なプログラミングエラー、動的メモリーアクセスエラー、およびコードカバレージの問題を見つけて修正する方法を示します。

サンプル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。サンプルアプリケーションの zip ファイルをまだダウンロードしていない場合は、適当なディレクトリにダウンロードして展開します。

`sample` アプリケーションは、`SolarisStudioSampleApplications` ディレクトリの `CodeAnalyzer` サブディレクトリにあります。

`sample` ディレクトリには次のソースコードファイルがあります。

```
main.c
previse_1.c
previse_all.c
sample_1.c
```

```
sample_2.c
sample_3.c
```

データの収集と表示

コードアナライザのツールを使用して、1 種類、2 種類、または 3 種類すべてのデータを収集できます。

静的エラーデータの収集と表示

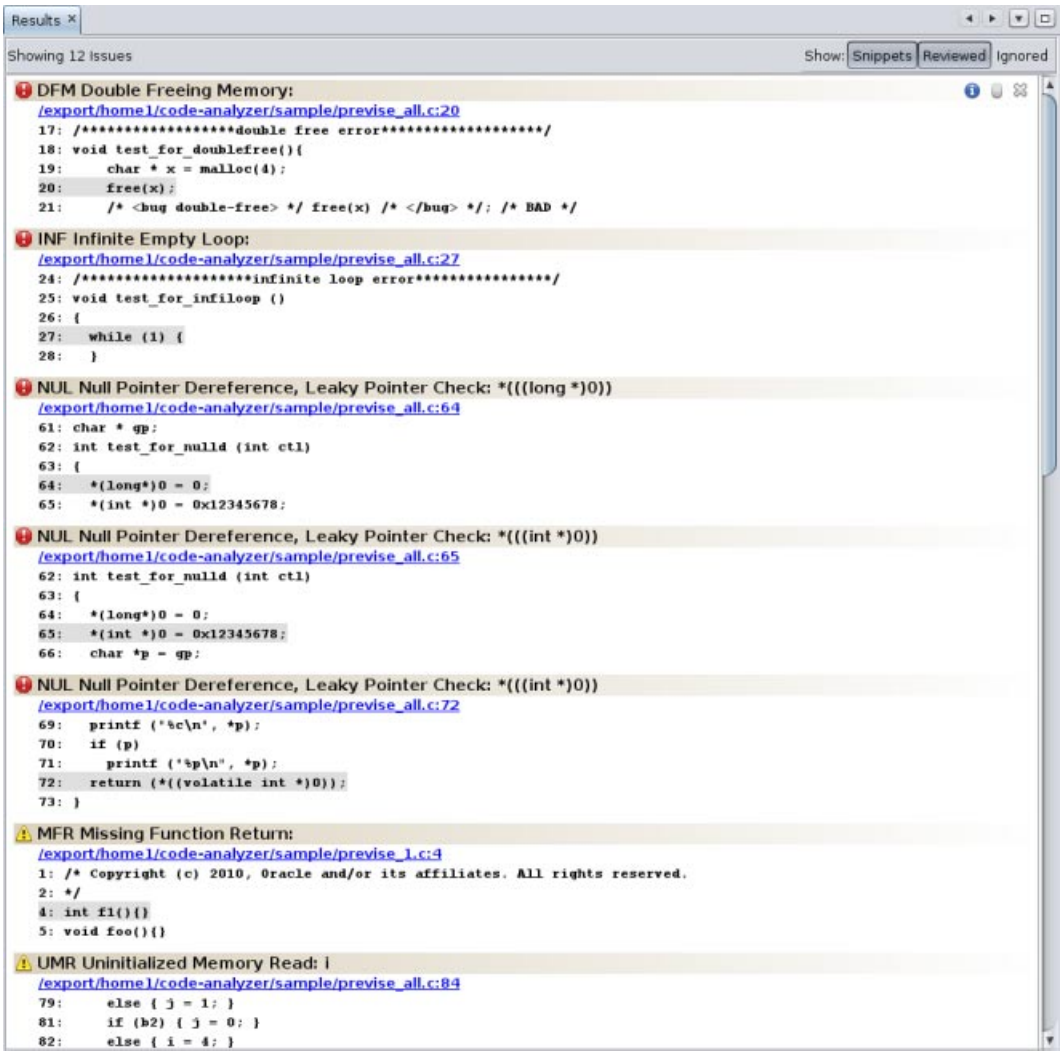
`-xanalyze=code` コンパイラオプションを使用してバイナリを構築すると、コンパイラは静的エラーを自動的に抽出し、データを *binary_name*`.analyze` ディレクトリの `static` サブディレクトリに書き込みます。コンパイラで検出される静的エラーの種類のリストについては、14 ページの「静的コードの問題」を参照してください。

1. `sample` ディレクトリで、次のように入力してアプリケーションを構築します。

   ```
   cc -xanalyze=code *.c
   ```

   静的エラーデータが、`sample` ディレクトリの `a.out.analyze` ディレクトリの `static` サブディレクトリに書き込まれます。
2. コードアナライザ GUI を開いて結果を表示します。

   ```
   code-analyzer a.out &
   ```
3. コードアナライザ GUI が開き、コンパイル時に見つかった静的コードの問題が「結果」タブに表示されます。「結果」タブの上部のテキストは、静的コードの問題が 12 件見つかったことを示しています。

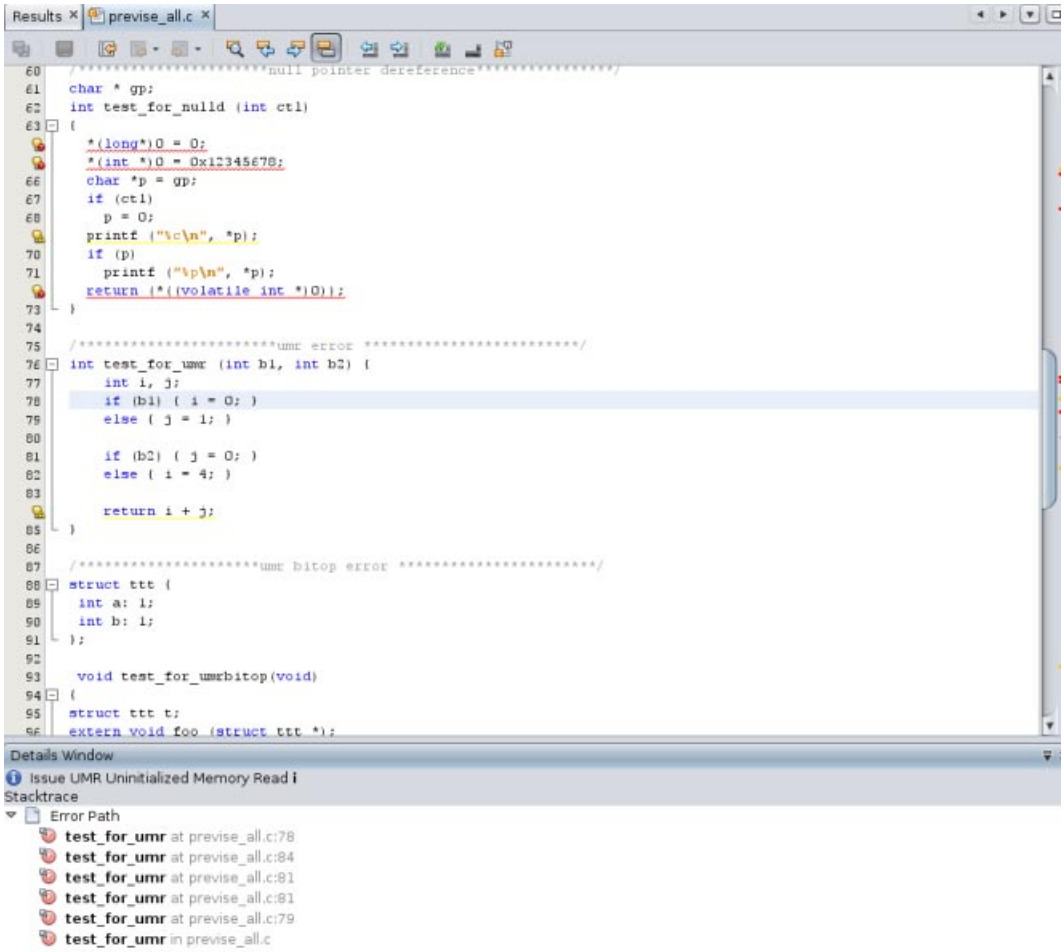


4. タブでは、各問題について、問題の種類、問題が見つかったソースファイルのパス名、およびそのファイルの該当するソース行を強調表示したコードスニペットが表示されます。
5. 最初の問題、「メモリーの二重解放」エラーの詳細を表示するには、エラーアイコンをクリックします  。問題のスタックトレースが開き、エラーパスが表示されます。

6. スタックトレースを開いたときに、問題の右上隅にあるアイコンが  から  に変わり、問題が確認済みであることを示します。

注 – 確認済みの問題を非表示にするには、「レビュー済み課題を非表示」ボタン Reviewed を「結果」タブの上部でクリックします。ボタンを再度クリックすると、問題の非表示が解除されます。

7. エラーアイコンをクリックして、スタックトレースを閉じます。
8. 次に、「非初期化メモリーからの読み取り」警告の 1 つを見てみましょう。警告アイコン ⚠ をクリックして、スタックトレースを開きます。この問題のエラーパスには、「メモリーの二重解放」問題のエラーパスよりもはるかに多くの関数呼び出しが含まれています。最初の関数呼び出しをダブルクリックします。ソースファイルが開き、その呼び出しが強調表示されます。エラーパスは、ソースコードの下の詳細ウィンドウに表示されます。



エラーパス内の残りの関数呼び出しをダブルクリックしていくと、エラーにつながったパスをコード内で追尾することができます。

9. UMR エラータイプの詳細を表示するには、「情報」ボタン ℹ を問題の説明の左でクリックします。コード例や考えられる原因などを含め、エラータイプの説明がオンラインヘルプブラウザに表示されます。
10. コードアナライザ GUI を閉じます。

動的メモリー使用データの収集と表示

静的データを収集したかどうかにかかわらず、アプリケーションをコンパイルし、計測機構を組み込み、実行して、動的メモリーアクセスデータを収集することができます。Discover で計測機構を組み込んでからアプリケーションを実行することによって検出される動的メモリーアクセスエラーのリストについては、14 ページの「動的メモリーアクセスの問題」を参照してください。

1. `sample` ディレクトリで、サンプルアプリケーションを `-g` オプションで構築します。このオプションではデバッグ情報が生成され、エラーおよび警告に関するソースコードおよび行番号情報をコードアナライザで表示できるようになります。

   ```
   cc -g *.c
   ```

2. すでに計測機構の付いたバイナリに計測機構を組み込むことはできないため、カバレージデータの収集時に使用するバイナリのコピーを保存します。

   ```
   cp a.out a.out.save
   ```

3. Discover でバイナリに計測機構を組み込みます。

   ```
   discover -a a.out
   ```

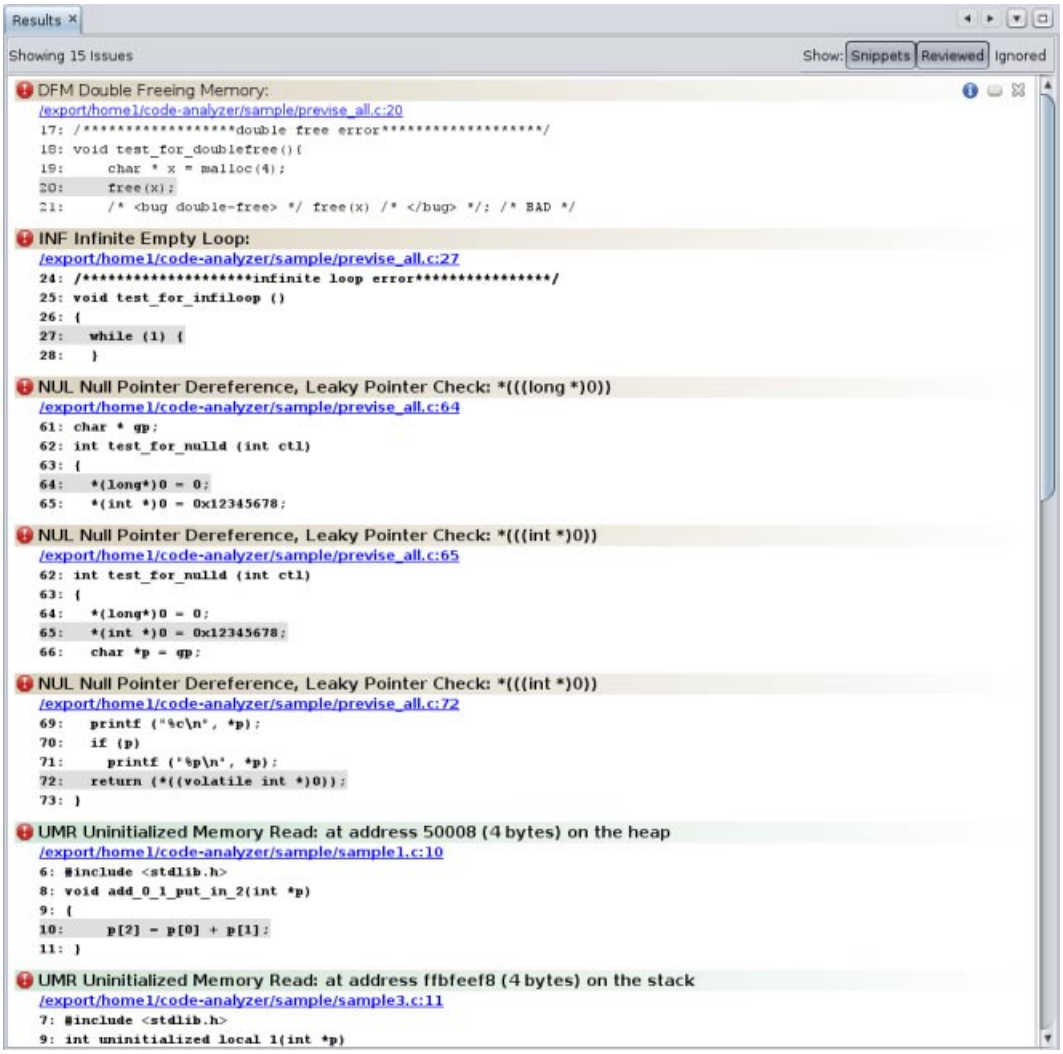
4. 計測機構付きバイナリを実行して動的メモリーアクセスデータを収集します。

   ```
   ./a.out
   ```

   動的メモリーアクセスエラーデータが、`sample` ディレクトリの `a.out.analyze` ディレクトリの `dynamic` サブディレクトリに書き込まれます。

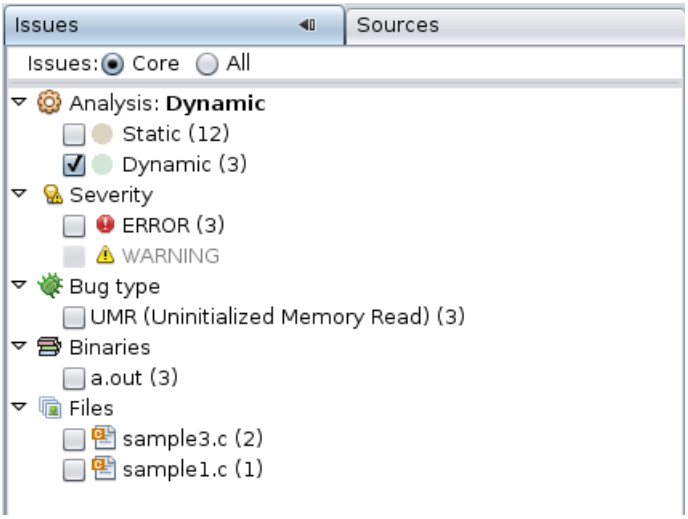
5. コードアナライザ GUI を開いて結果を表示します。

   ```
   code-analyzer a.out &
   ```



6. この時点で、「結果」タブには、静的な問題と動的メモリーの問題の両方が表示されます。問題の説明の背景色は、静的コードの問題 (褐色) か動的メモリーアクセスの問題 (淡い緑) かを示しています。

   結果をフィルタリングして動的メモリーの問題だけを表示するには、「問題」タブで「動的」チェックボックスを選択します。

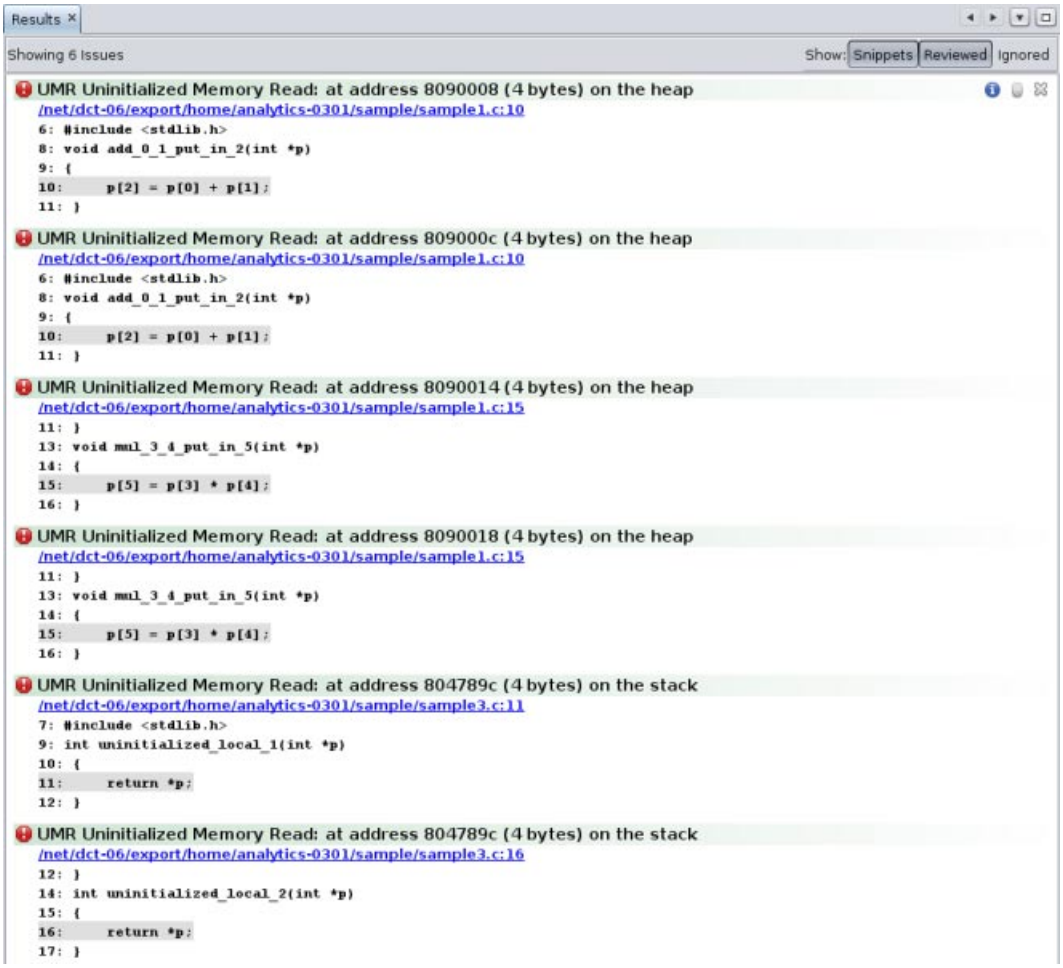


この時点で、「結果」タブには、中核となる 3 件の動的メモリーの問題だけが表示されます。

注 – 中核となる問題とは、それらを修正すればほかの問題も解消される可能性の高い問題のことです。通常、中核となる問題には、「すべて」のビューで一覧表示される問題のいくつかが関連しています。たとえば、それらの問題では割り当てポイントが共通であったり、問題が同じ関数の同じデータアドレスで発生したりするためです。

---

7. すべての動的メモリーの問題を表示するには、「問題」タブの上部にある「すべて」ラジオボタンを選択します。この時点で、「結果」タブには、6 件の動的メモリーの問題が表示されます。



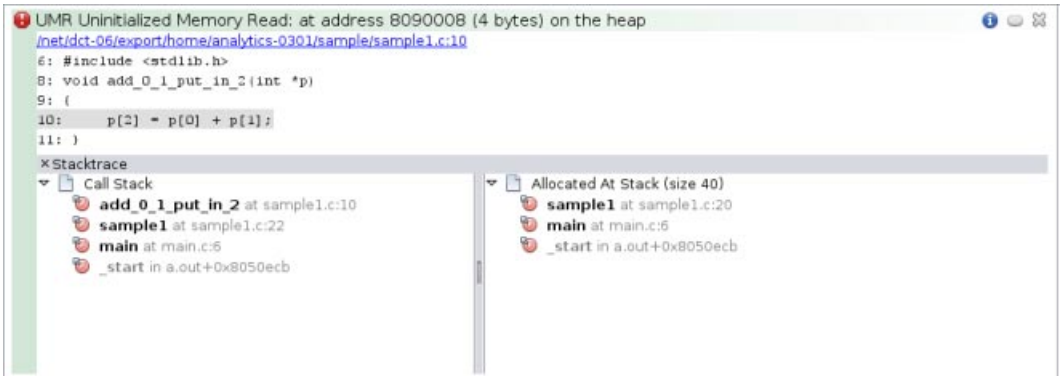
表示に追加された 3 件の問題を調べ、中核となる問題にどのように関連しているかを確認します。表示されている最初の問題の原因を修正すれば、2 番目と 3 番目の問題も解消されるように見えます。

動的メモリーアクセスの問題のうちでこれらの最初の問題を調査している間、ほかの 3 件を非表示にするには、「無視」ボタン を各問題についてクリックします。

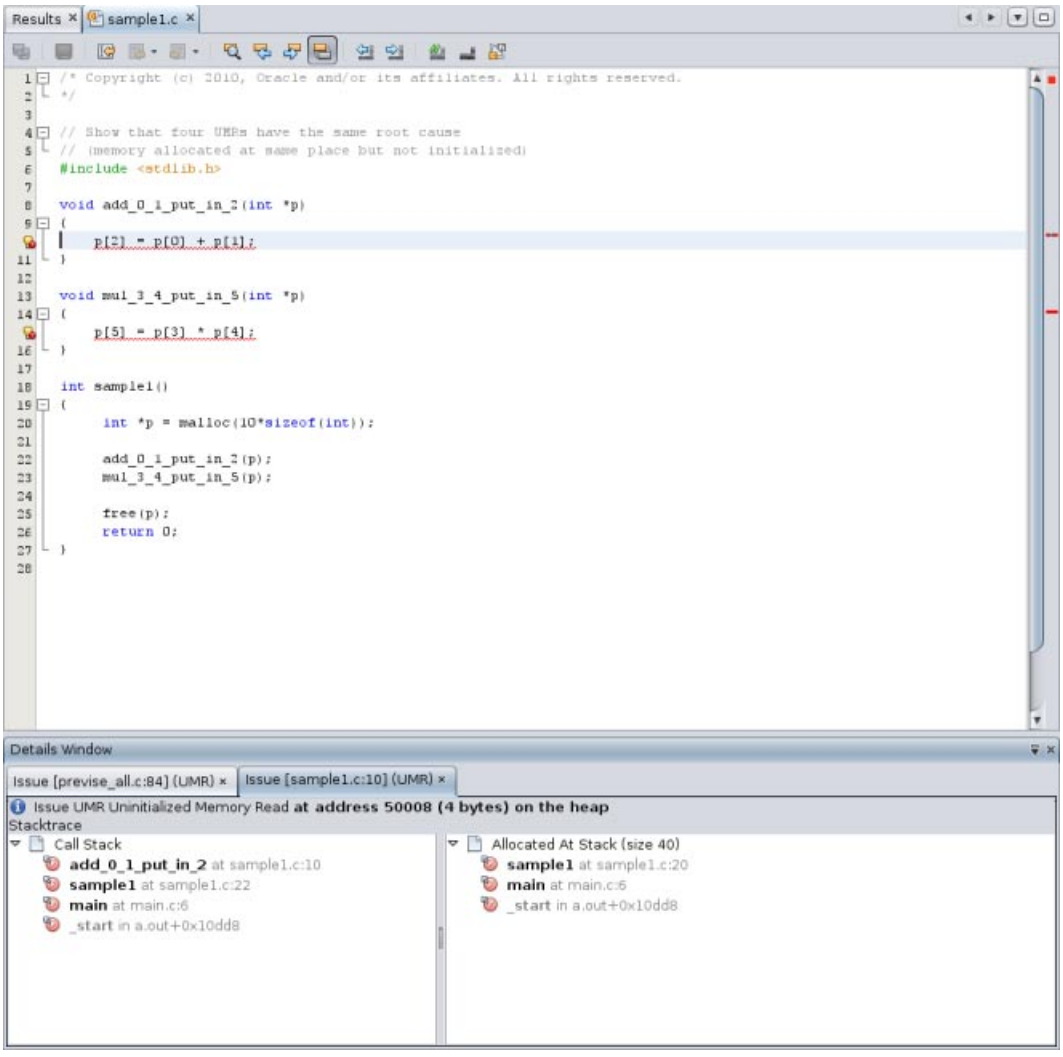
注 – あとで「無視された問題を表示」ボタンをクリックして、閉じた問題を再度表示することができます Ignored を「結果」タブの上部でクリックします。

---

8. 最初の問題を調査するために、エラーアイコン をクリックして、スタックトレースを表示します。この問題の場合、スタックトレースには呼び出しスタックと割り当てスタックが含まれます。



スタック内の関数呼び出しをダブルクリックして、ソースファイル内の関連する行を表示します。ソースファイルが開くと、ファイルの下の詳細ウィンドウにスタックトレースが表示されます。



9. コードアナライザ GUI を閉じます。

コードカバレージデータの収集と表示

静的データや動的メモリーアクセスデータを収集したかどうかにかかわらず、アプリケーションをコンパイルし、計測機構を組み込み、実行して、コードカバレージデータを収集することができます。

1. 動的メモリーエラーデータを収集する前に `-g` オプションでアプリケーションを構築し、計測機構を組み込む前にバイナリのコピーを保存しておいたため、保存しておいたバイナリをコピーし、カバレージデータ収集用の計測機構を組み込むことができます。

   ```
   cp a.out.save a.out
   ```

2. Uncover でバイナリに計測機構を組み込みます。

   ```
   uncover a.out
   ```

3. 計測機構付きバイナリを実行してコードカバレージデータを収集します。

   ```
   ./a.out
   ```

   コードカバレージデータが、sample ディレクトリの `a.out.uc` ディレクトリに書き込まれます。

4. `a.out.uc` ディレクトリに対して Uncover を実行します。

   ```
   uncover -a a.out.uc
   ```

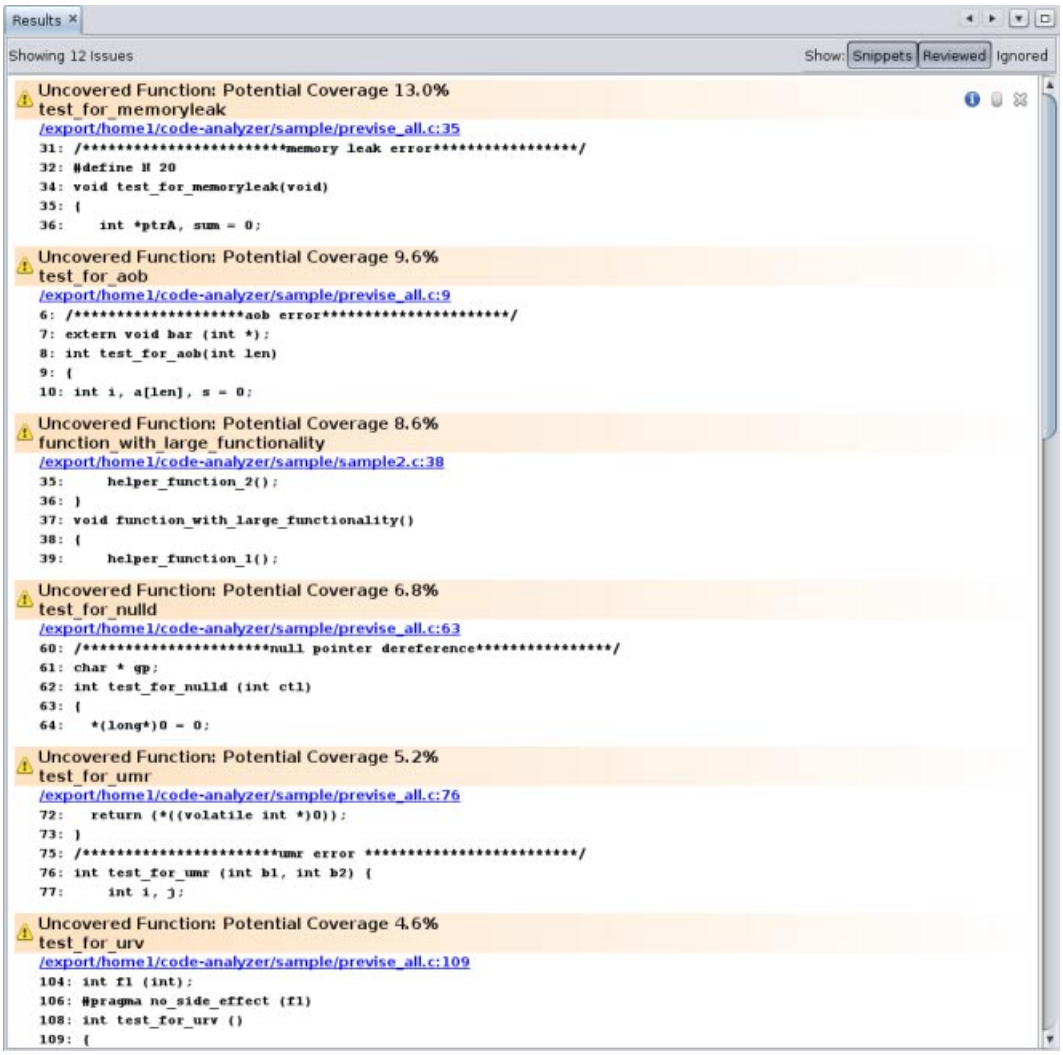
   コードカバレージデータが、sample ディレクトリの `a.out.analyze` ディレクトリの uncover サブディレクトリに書き込まれます。

5. コードアナライザ GUI を開いて結果を表示します。

   ```
   code-analyzer a.out &
   ```

6. この時点で、「結果」タブには、静的な問題、動的メモリーの問題、およびコードカバレージの問題が表示されます。結果をフィルタリングしてコードカバレージの問題だけを表示するには、「問題」タブで「カバレージ」チェックボックスを選択します。

   この時点で、「結果」タブには、12 件のコードカバレージの問題だけが表示されます。各問題の説明には、潜在的なカバレージの割合が含まれています。この割合は、該当する関数をカバーするテストを追加した場合にアプリケーションの合計カバレージが何パーセント増加するかを示しています。

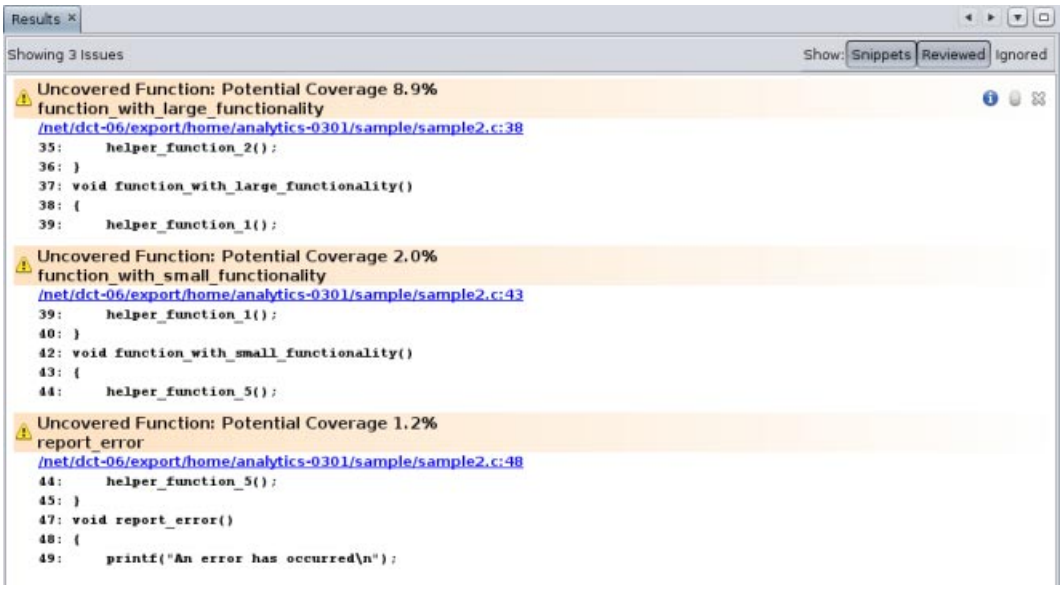


注 – 上下にスクロールすることなくすべての問題を表示するには、「スニペットを非表示」ボタン  を「結果」タブの上部でクリックして、コードスニペットを非表示にします。

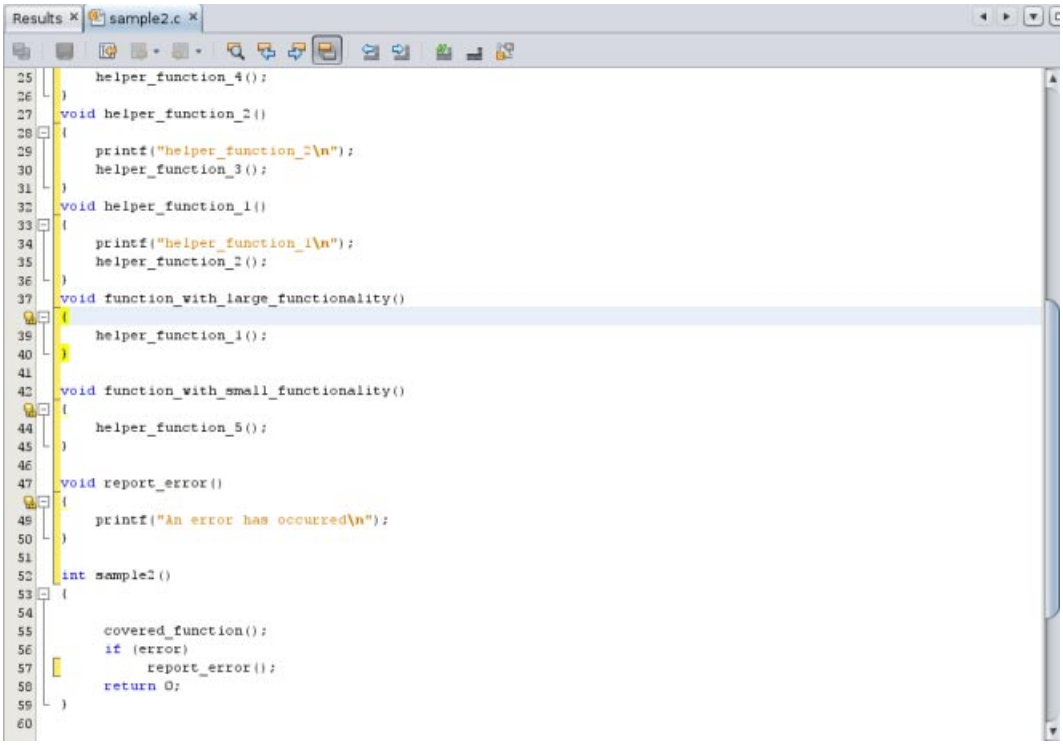
---

7. 「問題」タブでは、カバレージの問題の 9 件が `previse_all.c` ソースファイル、3 件が `sample2.c`、および 1 件が `previse_1.c` に含まれていることがわかります。結果をさらにフィルタリングして `sample2.c` ファイルの問題だけを表示するには、「問題」タブでそのファイルのチェックボックスを選択します。

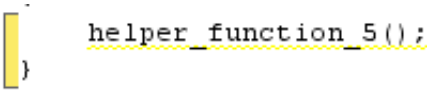
   この時点で、「結果」タブには、`sample2.c` で見つかった 3 件のコードカバレージの問題だけが表示されます。



8. いずれかの問題でソースファイルパスのリンクをクリックして、ソースファイルを開きます。左の余白に警告アイコンが現れるまで、ソースファイルを下へスクロールします。



カバーされていないコードは黄色の各括弧でマークされ、たとえば



のようになります。ファイルで見つかったカバレージの問題は、警告アイコンでマークされます 

コードアナライザで検出された問題をコードの改善に使用する

コードアナライザで検出された中核となる問題を修正することで、コードに見つかったほかの問題も解消でき、コードの品質と安定性を大きく改善できるはずです。

静的エラー検査を実行することにより、アプリケーション内の危険なコードを見つけることができます。ただし、静的エラー検査では偽陽性が発生することがあります。動的検査はコードの問題をより精密に描写して、このようなエラーの確認と解消に役立ちます。コードカバレージ検査は、動的テストスイートの改善に役立ちます。

コードアナライザではこれら 3 種類の検査の結果が統合され、もっとも精密なコード分析をすべて 1 つのツールで行うことができます。

コードアナライザツールで検出される潜在的なエラー

コンパイラ、Discover、および Uncover は、コード内の静的コードの問題、動的メモリーアクセスの問題、およびカバレージの問題を検出します。これらのツールで検出され、コードアナライザで分析されるエラーの種類について、以降の節で説明します。

静的コードの問題

静的コード検査では、次の種類のエラーが検出されます。

- ABR: 配列範囲外からの読み取り (beyond Array Bounds Read)
- ABW: 配列範囲外への書き込み (beyond Array Bounds Write)
- DFM: メモリーの二重解放 (Double Freeing Memory)
- ECV: 明示的型キャスト違反 (Explicit type Cast Violation)
- FMR: 解放されたメモリーからの読み取り (Freed Memory Read)
- FMW: 解放されたメモリーへの書き込み (Freed Memory Write)
- FOU: `PM_OUT` 定義前の使用
- INF: 空の無限ループ (INFinite empty loop)
- メモリーリーク
- MFR: 関数の復帰なし (Missing Function Return)
- MRC: malloc 戻り値の検査なし (Missing malloc Return value Check)
- NFR: 初期化されていない関数の復帰 (uNinitialized Function Return)
- NUL: NULL ポインタ間接参照、リークの可能性があるポインタの検査 (NULl pointer dereference, leaky pointer check)
- RFM: 解放済みメモリーを返す (Return Freed Memory)
- UMR: 初期化されていないメモリーの読み取り、初期化されていないメモリーの読み取りビット操作 (Uninitialized Memory Read, Uninitialized Memoey Read bit operation)
- URV: 使用されていない戻り値 (Unused Return Value)
- VES: スコープ外での局所変数の使用 (out-of-scope local Variable usage)

動的メモリーアクセスの問題

動的メモリーアクセス検査では、次の種類のエラーが検出されます。

- ABR: 配列範囲外からの読み取り (beyond Array Bounds Read)
- ABW: 配列範囲外への書き込み (beyond Array Bounds Write)
- BFM: 不正な空きメモリー (Bad Free Memory)
- BRP: 不正な realloc アドレスパラメータ (Bad Realloc address Parameter)

- CGB: 破損したガードブロック (Corrupted Guard Block)
- DFM: メモリーの二重解放 (Double Freeing Memory)
- FMR: 解放されたメモリーからの読み取り (Freed Memory Read)
- FMW: 解放されたメモリーへの書き込み (Freed Memory Write)
- IMR: 無効なメモリーからの読み取り (Invalid Memory Read)
- IMW: 無効なメモリーへの書き込み (Invalid Memory Write)
- メモリーリーク
- OLP: 送り側と受け側の重複 (OverLaPping source and destination)
- PIR: 部分的に初期化された領域からの読み取り (Partially Initialized Read)
- SBR: スタック境界を越える読み取り (beyond Stack Bounds Read)
- SBW: スタック境界を越える書き込み (beyond Stack Bounds Write)
- UAR: 非割り当てメモリーからの読み取り (UnAllocated memory Read)
- UAW: 非割り当てメモリーへの書き込み (UnAllocated memory Write)
- UMR: 非初期化メモリーからの読み取り (Unitialized Memory Read)

動的メモリーアクセス検査では、次の種類の警告が検出されます。

- AZS: 0 サイズの割り当て (Allocating Zero Size)
- メモリーリーク
- SMR: 投機的な非初期化メモリーからの読み取り (Speculative unitialized Memory Read)

コードカバレージの問題

コードカバレージ検査では、カバーされていない関数が特定されます。結果では、見つかったコードカバレージの問題に「カバーされていない関数」というラベルが付けられ、潜在的なカバレージの割合が示されます。この割合は、該当する関数をカバーするテストを追加した場合にアプリケーションの合計カバレージが何パーセント増加するかを示しています。

E26468

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

ORACLE