Oracle® Solaris Studio 12.4: コードアナライザチュートリアル

2014 年 10 月

Part No: E57232

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。

コードアナライザの概要

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、セキュア、堅牢、高品質なソフトウェアを作成できるように設計されています。Oracle Solaris Studio コンパイラ、`discover` (メモリーエラー検出ツール)、および `uncover` (コードカバレージツール) と連携します。

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、`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。`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分析およびコードカバレージ分析を統合して、単独で機能するほかのエラー検出ツールでは検出できない多くの重要なエラーをアプリケーションで検出できます。

サンプルアプリケーションの入手

サンプルプログラムのソースコードは、Oracle Solaris Studio 12.4 のダウンロード 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ら、目的のディレクトリに解凍できます。

`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
sample1.c
sample2.c
sample3.c
sample4.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、3 種類まで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

-xprevise 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、データをソースコードと同じディレクトリの *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、17 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。

注記 - -xprevise オプションは、非推奨となった -xanalyze=code と同じです。

- Oracle Solaris の場合:

```
$ cc -xprevise main.c previse*.c sample1.c sample2.c sample3.c
```

- Oracle Linux の場合:

```
$ cc -xannotate -xprevise main.c previse*.c sample1.c sample2.c sample3.c
```

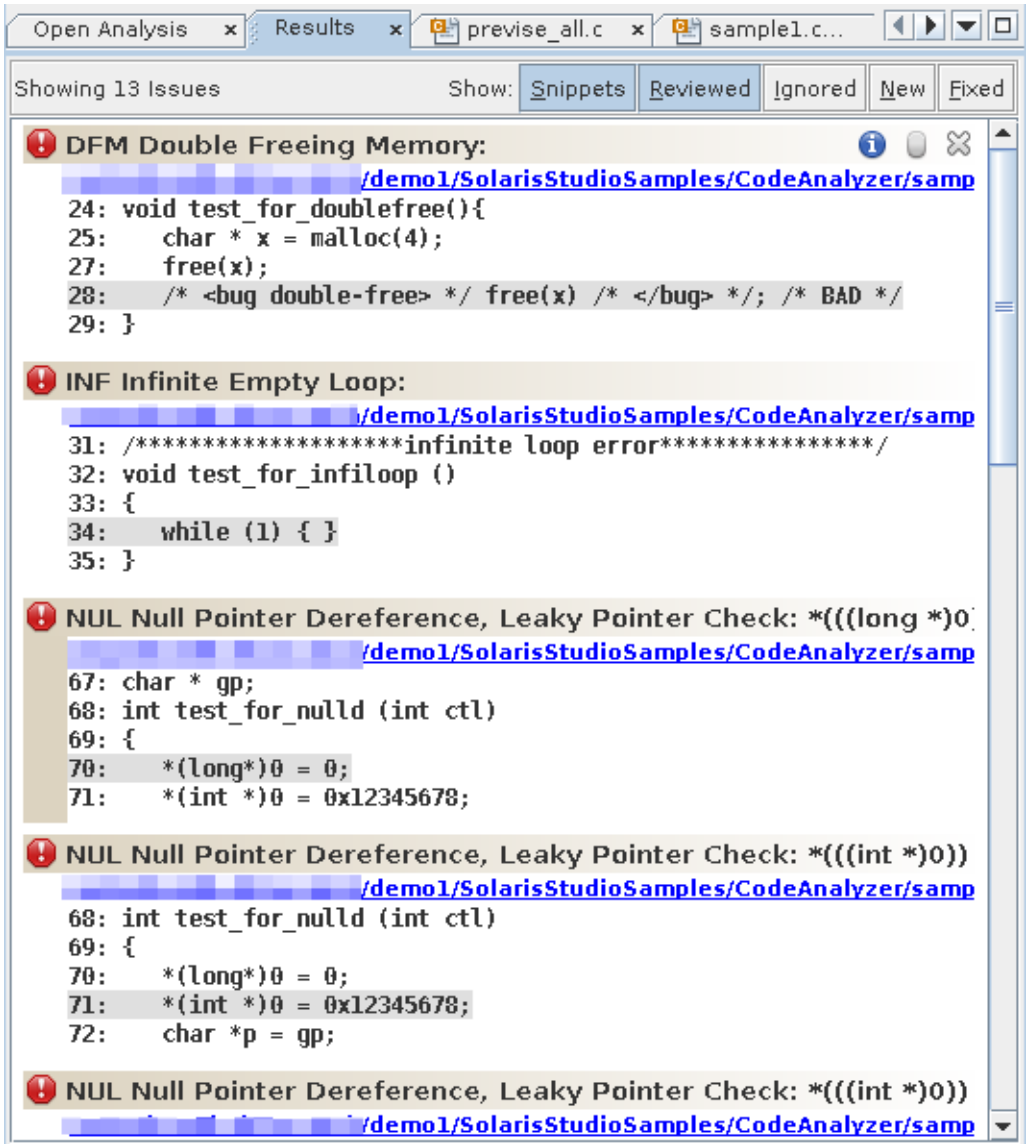
注記 - チュートリアルのこの部分では sample4.c をコンパイルしません。

   静的エラーデータが sample/a.out.analyze/static ディレクトリに書き込まれます。

2. コードアナライザ GUI を起動して結果を表示します。

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

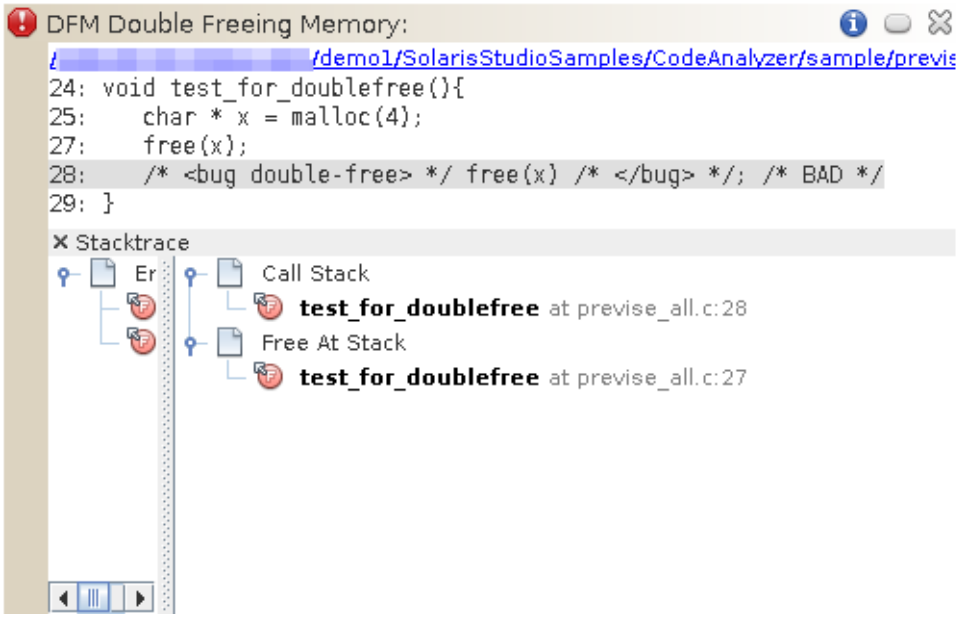
3. コードアナライザ GUI が開き、コンパイル時に見つかった静的コードの問題が「結果」タブに表示されます。「結果」タブの左上にあるテキストは、13 件の静的コードの問題が検出されたことを示しています。



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。

4. 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。



スタックトレースを開くと、問題の右上隅に表示されているアイコンが から、問題を確認したことを示す に変わることがわかります。

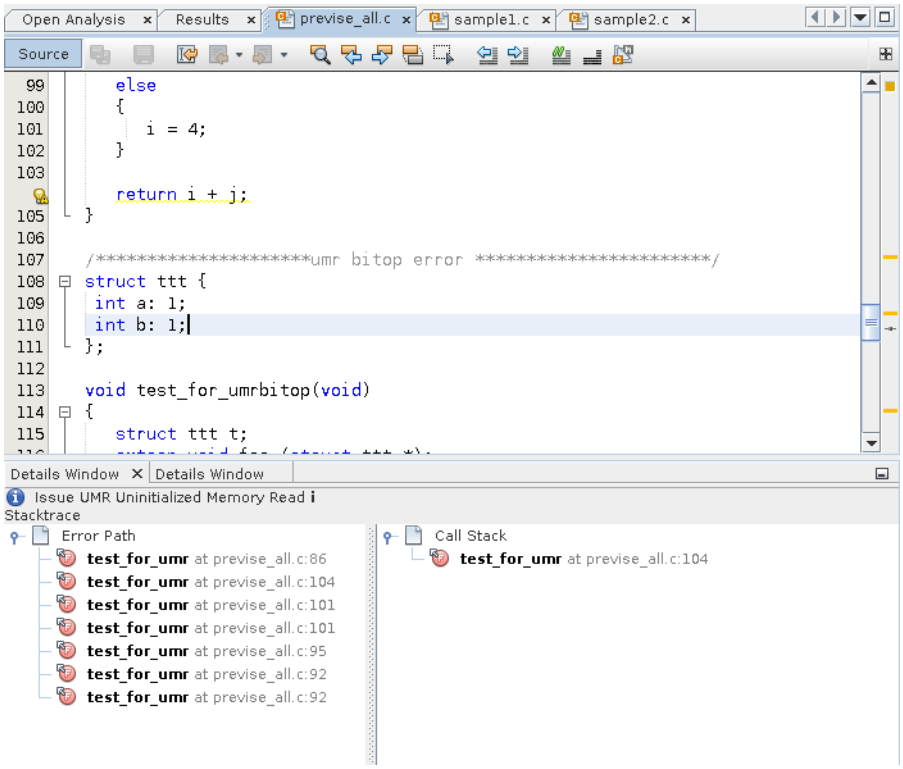
注記 - 確認済みの問題を非表示にするには、「結果」タブの上部にある「確認済み」ボタン Reviewed をクリックします。ボタンを再度クリックすると、問題の非表示が解除されます。

---

5. 同じ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。
6. 「初期化されていないメモリーの読み取り」警告の 1 つの警告アイコン 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。
7. 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。



8. エラーパス内のその他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できます。
9. 問題の説明の左側にある情報ボタン をクリックして、UMR エラータイプに関する詳細情報を確認します。

   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. 右上隅の X をクリックしてコードアナライザ 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、17 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。

   - Oracle Solaris の場合:

```
$ cc -g main.c previse*.c sample1.c sample2.c sample3.c
```

- Oracle Linux の場合:

```
$ cc -xannotate -g main.c previse*.c sample1.c sample2.c sample3.c
```

注記 - チュートリアルのこの部分では `sample4.c` をコンパイルしません。

---

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
```

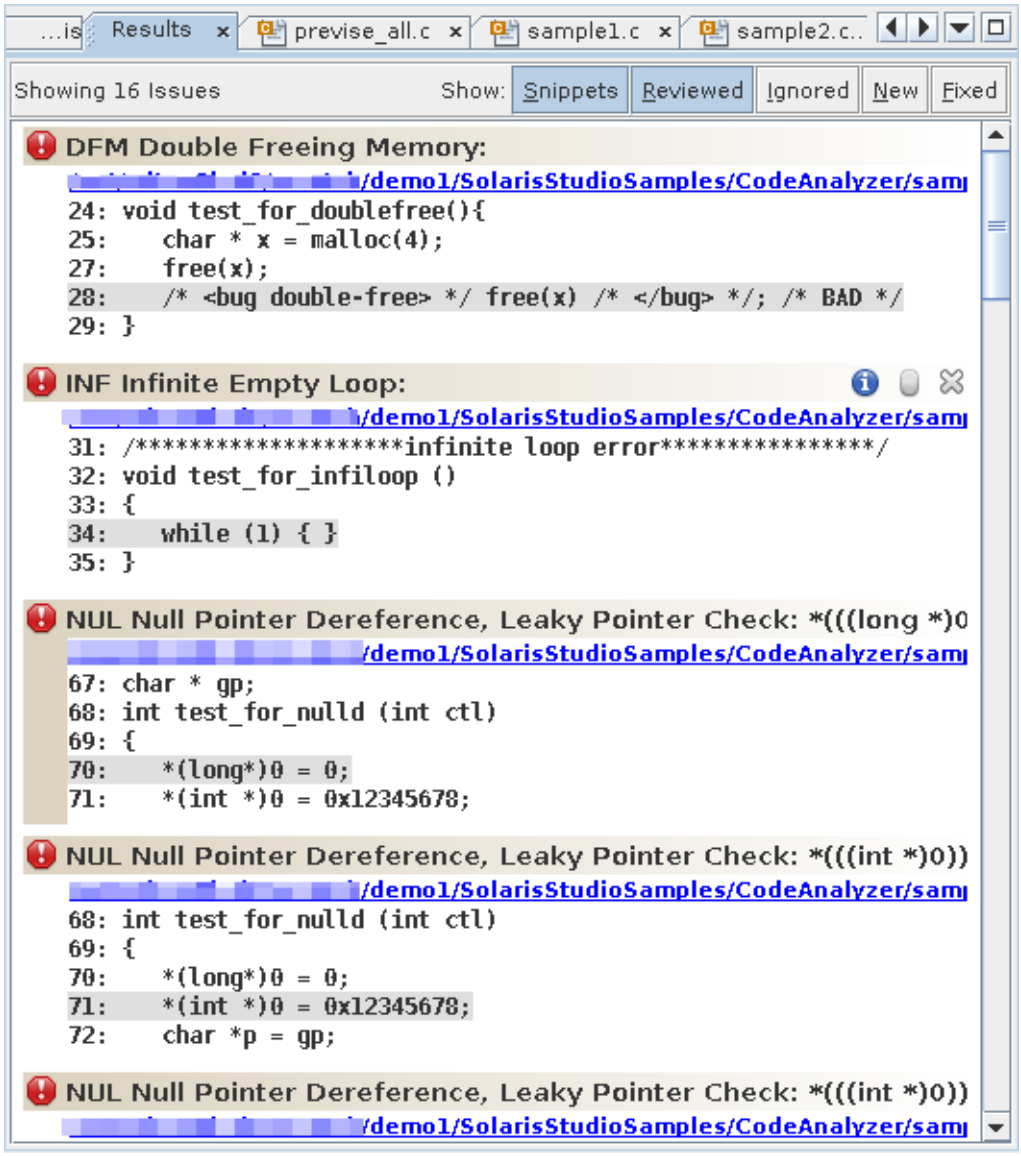
4. 計測機構付きバイナリを実行して動的メモリーアクセスデータを収集します。

```
$ ./a.out
```

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

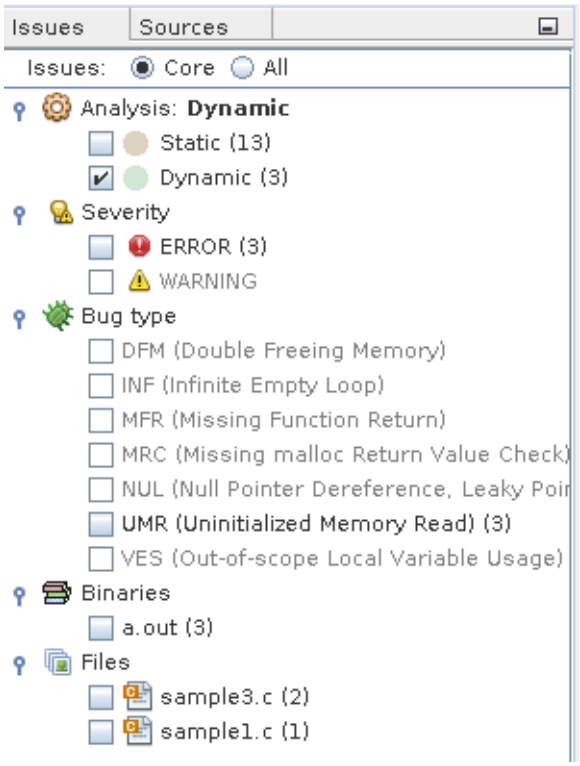
5. コードアナライザ GUI を起動して結果を表示します。

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



「結果」タブに、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) または動的メモリーアクセスの問題 (淡い緑) のいずれであるかを示しています。

6. 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」オプションを選択します。

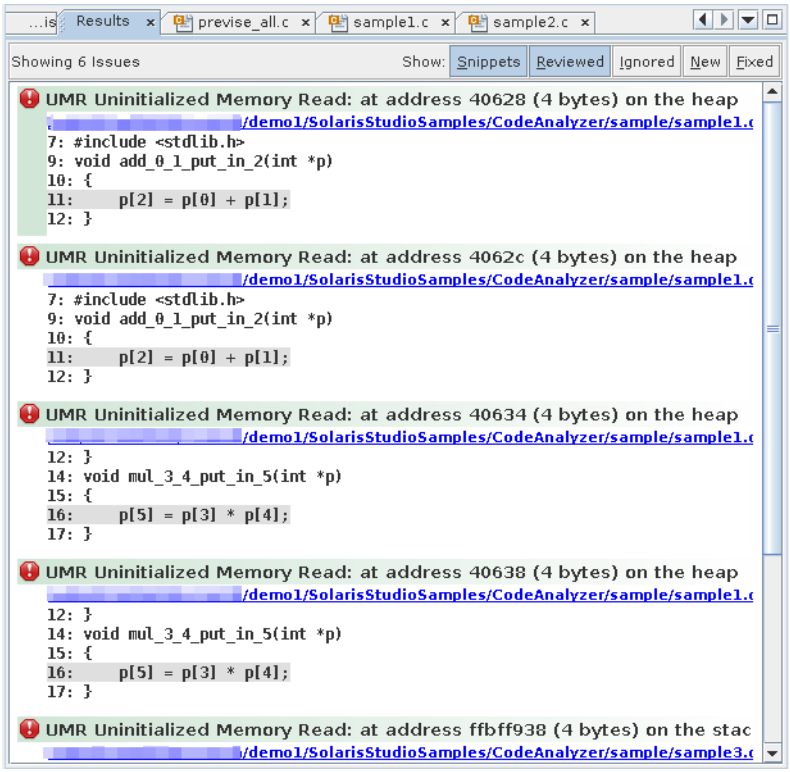


「結果」タブに、中核となる 3 件の動的メモリーの問題だけが表示されます。

注記 - 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おり、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。「結果」タブに、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可能性があります。

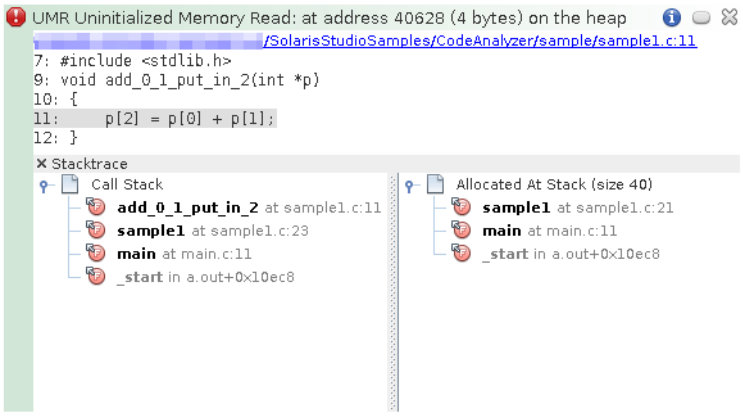
最初の問題を調査している間、ほかの動的メモリーアクセスの問題を非表示にするには、各問題の「無視」ボタン をクリックします。

注記 - 「結果」タブの上部にある「無視」ボタンをクリックすると、終了した問題をあとでふたたび表示できるようになります。

---

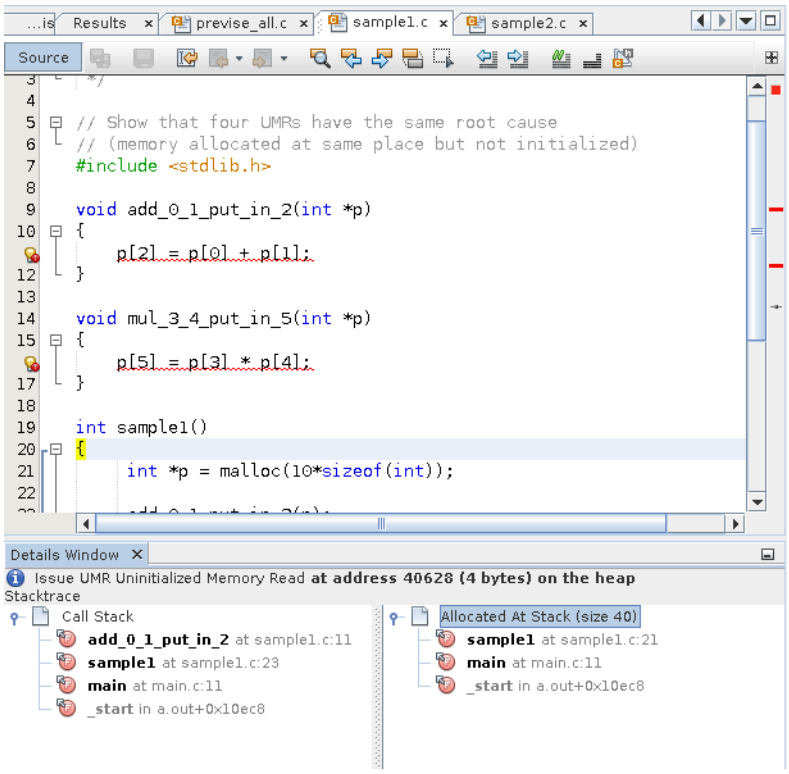
8. エラーアイコンをクリックしてスタックトレースを表示し、最初の問題を調査します。

   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。



9. 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。



10. 右上隅の X をクリックしてコードアナライザ GUI を閉じます。

コードカバレージデータの収集と表示

静的データまたは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できます。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た際には、計測機構を組み込む前にバイナリのコピーを保存していることになります。

1. カバレージデータ収集のために計測機能を組み込むよう、保存バイナリをコピーします。

   ```
   $ 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 ディレクトリ内の 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/a.out.analyze/uncover ディレクトリに書き込まれます。

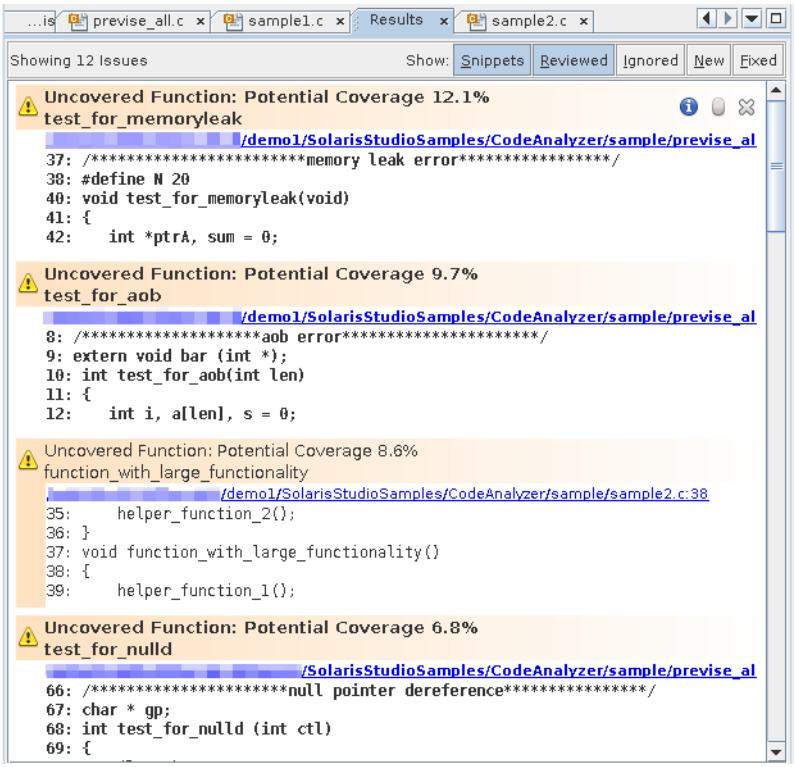
5. コードアナライザ GUI を起動して結果を表示します。

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

   「結果」タブに、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。

6. 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」オプションを選択します。

   「結果」タブに、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。

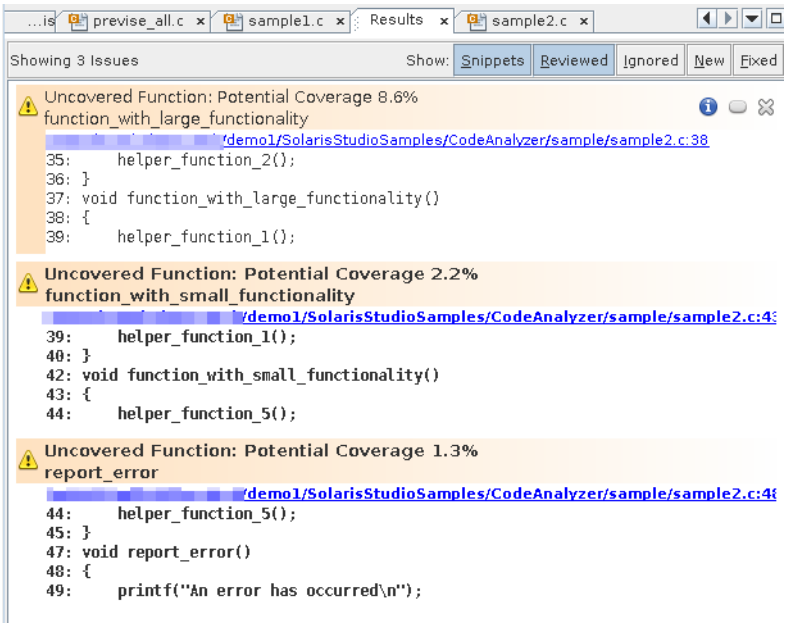


ヒント - 上下にスクロールすることなくすべての問題を表示するには、「結果」タブの上部にある「スニペット」ボタン  をクリックしてコードスニペットを非表示にします。

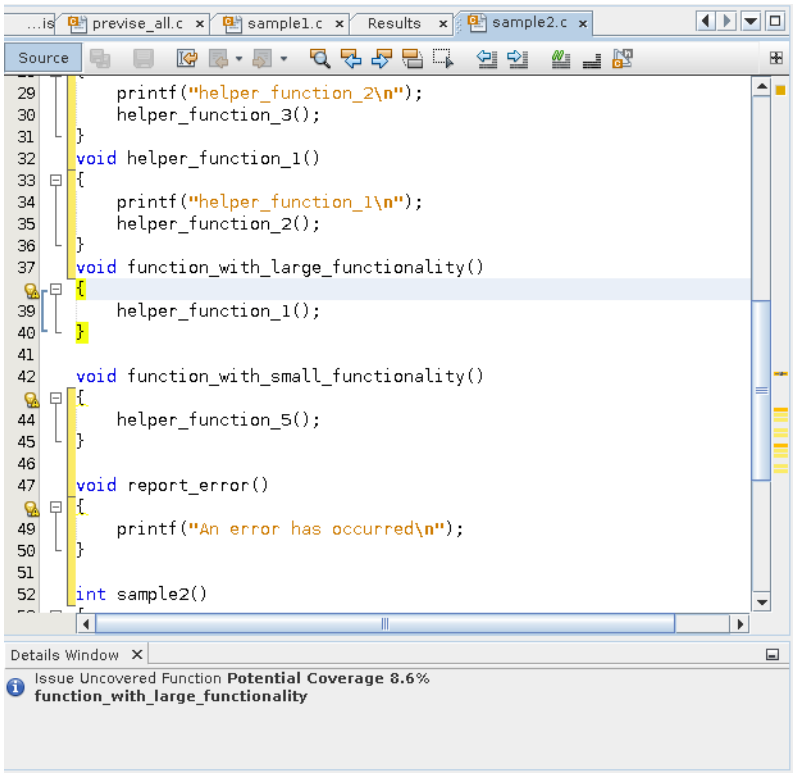
「問題」タブでは、カバレージの問題の 9 件が `previse_all.c` ソースファイル、3 件が `sample2.c`、および 1 件が `previse_1.c` に含まれています。

7. `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オプションを選択します。

   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。

codean コマンド行ツールの使用

コードアナライザのすべての機能を、codean コマンドでも同様に使用できます。このセクションは、codean コマンドを使用してコードで新しい静的コードの問題を検出する方法を、SolarisStudioSampleApplications の同じ sample プログラムを使って説明する短いチュートリアルです。

1. このチュートリアルの以前のセクションでは sample4.c をコンパイルしていませんでした。cat コマンドを使用してこのファイルをプレビューします。

```
$ cat sample_4.c
int another_new_umr()
{
  int i;
  if (i)
    return 0;
  else
    return 1;
}
```

   int i が初期化されていないことを確認します。

2. ソースをコンパイルして静的レポートを生成します。

   Oracle Solaris の場合:

```
$ cc -g -xprevise main.c previse_1.c previse_all.c sample1.c sample2.c sample3.c
```

   Oracle Linux の場合:

```
$ cc -xannotate -g -xprevise main.c previse_1.c previse_all.c sample1.c sample2.c sample3.c
```

3. codean --save オプションを使用して静的レポートを a.out に保存します。

```
$ codean --save -s a.out
```

4. sample4.c を含めて sample アプリケーションを再コンパイルします。

   Oracle Solaris の場合:

```
$ cc -g -xprevise *.c
```

   Oracle Linux の場合:

```
$ cc -g -xannotate -xprevise *.c
```

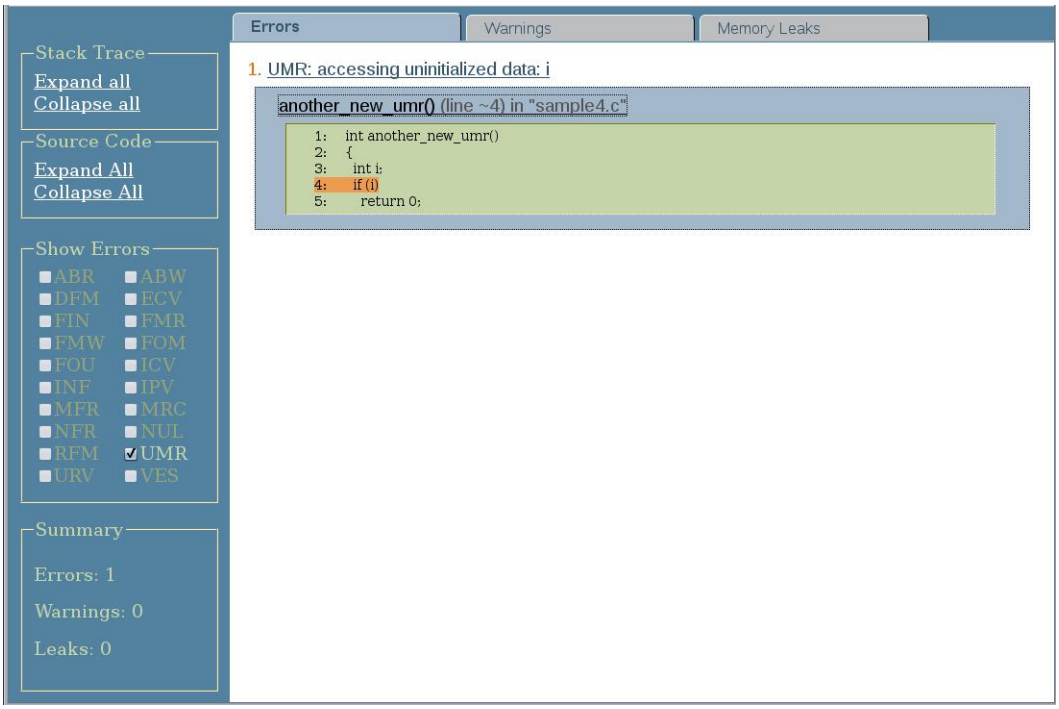
   この新しい関数は main から呼び出されることはありませんが、新しい UMR エラーを発生させます。

5. --whatisnew オプションを使用して、新たに追加された静的な問題に関するレポートを取得します。

```
$ codean --whatisnew -s a.out
```

```
STATIC report of a.out showing new issues:
Compare the latest results against a.out.analyze/history/2014.8.4.14.49.56...
ERROR 1 (UMR): accessing uninitialized data: i at:
        another_new_umr()  <sample_4.c : 4>
                1:      int another_new_umr()
                2:      {
                3:        int i;
                4:=>      if (i)
                5:          return 0;
PREVISE SUMMARY for a.out: 1 new error(s), 0 new warning(s), 0 new leak(s) in total
```

次の図に、codean によって生成される静的コードの問題に関する HTML レポートを示します。



codean の詳細については、『Oracle Solaris Studio 12.4: コードアナライザユーザーズガイド 』の「コードアナライザのコマンド行インタフェース」および codean(1) のマニュアルページを参照してください。

コードアナライザで検出された問題を使用したコードの改善

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多くの問題も解消でき、コードの品質と安定性を大きく改善できます。

静的エラー検査ではアプリケーション内の危険なコードを検出できますが、誤検出も発生する可能性があります。動的検査はコードの問題をより精密に描写するため、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も役立ちます。

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。このセクションでは、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説明します。

これらのエラーと警告の詳細については、『Oracle Solaris Studio 12.4: コードアナライザユーザーズガイド 』の付録 A「コードアナライザで分析されるエラー」を参照してください。

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: 配列境界を越える読み取り (beyond array bounds read)
- ABW: 配列境界を越える書き込み (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: 解放済みメモリーの読み取り (freed memory read)
- FMW: 解放済みメモリーの書き込み (freed memory write)
- INF: 空の無限ループ (infinite empty loop)
- MLK: メモリーリーク (memory leak)
- 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
- 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 memory 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: 配列境界を越える読み取り (beyond array bounds read)
- ABW: 配列境界を越える書き込み (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: 解放済みメモリーの読み取り (freed memory read)
- FMW: 解放済みメモリーの書き込み (freed memory write)
- FRP: 解放済み Realloc パラメータ (freed realloc parameter)
- IMR: 無効なメモリーの読み取り (invalid memory read)
- IMW: 無効なメモリーの書き込み (invalid memory write)
- MLK: メモリーリーク (memory leak)
- 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: 割り当てられていないメモリーの読み取り (unallocated memory read)
- UAW: 割り当てられていないメモリーの書き込み (unallocated memory write)
- UMR: 初期化されていないメモリーの読み取り (uninitialized 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)
- MLK: メモリーリーク (memory leak)
- SMR: 投機的な非初期化メモリーからの読み取り (speculative uninitialized 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。

ORACLE