

Oracle® Developer Studio 12.5: コードアナ  
ライザユーザーズガイド

ORACLE®

Part No: E71973  
2016年6月



## Part No: E71973

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

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.

このソフトウェアまた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。このソフトウェアまたはハードウェアは、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。このソフトウェアまた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場合、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。このソフトウェアまた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起因して損害が発生しても、Oracle Corporation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はオラクル およびその関連会社の登録商標です。その他の社名、商品名等は各社の商標または登録商標である場合があります。

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。適用されるお客様とOracle Corporationとの間の契約に別段の定めがある場合を除いて、Oracle Corporation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。適用されるお客様とOracle Corporationとの間の契約に定めがある場合を除いて、Oracle Corporation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。

### ドキュメントのアクセシビリティについて

オラクルのアクセシビリティについての詳細情報は、Oracle Accessibility ProgramのWeb サイト(<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>)を参照してください。

### Oracle Supportへのアクセス

サポートをご契約のお客様には、My Oracle Supportを通して電子支援サービスを提供しています。詳細情報は(<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info>)か、聴覚に障害のあるお客様は (<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs>)を参照してください。



# 目次

---

このドキュメントの使用方法 .....	9
<b>1 コードアナライザの使用 .....</b>	<b>11</b>
コードアナライザで分析されるデータ .....	11
静的コード検査 .....	12
動的メモリアクセス検査 .....	12
コードカバレッジ検査 .....	12
コードアナライザを使用するための要件 .....	13
コードアナライザ GUI .....	13
コードアナライザのコマンド行インタフェース .....	13
リモートデスクトップ配布 .....	14
クイックスタート .....	15
▼ クイックスタート .....	15
<b>2 データの収集とコードアナライザの起動 .....</b>	<b>17</b>
静的エラーデータの収集 .....	17
動的メモリアクセスデータの収集 .....	18
▼ バイナリから動的メモリアクセスデータを収集する方法: .....	19
コードカバレッジデータの収集 .....	19
▼ バイナリからコードカバレッジデータを収集する方法 .....	19
コードアナライザ GUI の使用 .....	20
コードアナライザコマンド行ツール (codean) の使用 .....	21
codean のオプション .....	22
codean ワークフローの例 .....	25
問題のラベル付け .....	26
テストスイートでの codean の使用 .....	29
<b>A コードアナライザで分析されるエラー .....</b>	<b>35</b>
コードカバレッジの問題 .....	35
静的コードの問題 .....	35

配列境界を越える読み取り (ABR) .....	36
配列境界を越える書き込み (ABW) .....	36
メモリの二重解放 (DFM) .....	36
解放済みメモリの読み取り (FMR) .....	37
解放済みメモリの書き込み (FMW) .....	37
空の無限ループ (INF) .....	37
メモリリーク .....	37
関数の復帰なし (MFR) .....	38
malloc 戻り値の検査なし (MRC) .....	38
リークの可能性があるポインタの検査: NULL ポインタ間接参照 (NUL) .....	38
解放済みメモリを返す (RFM) .....	39
初期化されていないメモリの読み取り (UMR) .....	39
使用されていない戻り値 (URV) .....	40
スコープ外での局所変数の使用 (VES) .....	40
動的メモリアクセスエラー .....	40
配列境界を越える読み取り (ABR) .....	41
配列境界を越える書き込み (ABW) .....	41
不正な空きメモリ (BFM) .....	41
不正な Realloc アドレスパラメータ (BRP) .....	42
破損したガードブロック (CGB) .....	42
メモリの二重解放 (DFM) .....	42
解放済みメモリの読み取り (FMR) .....	43
解放済みメモリの書き込み (FMW) .....	43
解放済み Realloc パラメータ (FRP) .....	43
無効なメモリの読み取り (IMR) .....	43
無効なメモリの書き込み (IMW) .....	44
メモリリーク .....	44
送り側と受け側の重複 (OLP) .....	44
部分的に初期化された読み取り (PIR) .....	45
スタック境界を越える読み取り (SBR) .....	45
スタック境界を越える書き込み (SBW) .....	45
割り当てられていないメモリの読み取り (UAR) .....	46
割り当てられていないメモリの書き込み (UAW) .....	46
初期化されていないメモリの読み取り (UMR) .....	46
動的メモリアクセスの警告 .....	47
サイズ 0 の割り当て (AZS) .....	47
メモリリーク (MLK) .....	47
投機的なメモリの読み取り (SMR) .....	47

索引 ..... 49



## このドキュメントの使用方法

---

- **概要** - コードアナライザツールを使用してデータを分析および表示する方法を説明します。
- **対象読者** - アプリケーション開発者、システム開発者、アーキテクト、サポートエンジニア
- **必要な知識** - プログラミングの経験、ソフトウェア開発テスト、ソフトウェア製品の構築およびコンパイルの経験

## 製品ドキュメントライブラリ

この製品および関連製品のドキュメントとリソースは [http://docs.oracle.com/cd/E37069\\_01](http://docs.oracle.com/cd/E37069_01) で入手可能です。

## フィードバック

このドキュメントに関するフィードバックを <http://www.oracle.com/goto/docfeedback> からお聞かせください。



# ◆◆◆ 第 1 章

## コードアナライザの使用

---

Oracle Developer 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がセキュアで堅牢かつ高品質なソフトウェアを作成できるよう支援する統合ツールセットです。

この章には、次の情報が含まれます。

- 11 ページの「コードアナライザで分析されるデータ」
- 13 ページの「コードアナライザを使用するための要件」
- 13 ページの「コードアナライザ GUI」
- 13 ページの「コードアナライザのコマンド行インタフェース」
- 14 ページの「リモートデスクトップ配布」
- 15 ページの「クイックスタート」

### コードアナライザで分析されるデータ

コードアナライザは 3 種類のデータを分析します。

- コンパイル時に検出される静的コードエラー
- メモリエラー検出ツールである discover ユーティリティで検出される動的メモリアクセスエラーおよび警告
- コードカバレッジツールである uncover ユーティリティで測定されるコードカバレッジデータ

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アクセス分析およびコードカバレッジ分析を統合して、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。

また、コードアナライザはコード内の中核となる問題、つまり、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を特定します。通常、中核となる問題にはほかのいくつかの問題が関連しています。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。

## 静的コード検査

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。C および C++ コンパイラの `-xprewise=yes` オプションは、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上の潜在的な欠陥を分析します。

---

**注記** - オプションで、`-xanalyze=code` オプションを使用して静的コードエラーを収集できますが、このオプションは EOL です。`-xprewise=yes` オプションを使用することを推奨します。

---

静的エラーデータの収集については、[17 ページの「静的エラーデータの収集」](#)を参照してください。

コードアナライザで分析される静的コードエラーのリストについては、[35 ページの「静的コードの問題」](#)を参照してください。

## 動的メモリアクセス検査

多くの場合、コード内のメモリアクセスエラーは見つけることが困難です。プログラムを実行する前に `discover` で計測機構を組み込むと、プログラムの実行中に `discover` はメモリアクセスエラーを動的に検出して報告します。たとえば、プログラムが配列を割り当て、それを初期化せずに、配列内のある場所から読み取ろうとする場合、プログラムは動作が不安定になることがあります。プログラムに `Discover` で計測機構を組み込んでから実行すると、`discover` はこのエラーを検出します。

動的メモリアクセスエラーデータの収集については、[18 ページの「動的メモリアクセスデータの収集」](#)を参照してください。

コードアナライザで分析される動的メモリアクセスの問題のリストについては、[40 ページの「動的メモリアクセスエラー」](#)を参照してください。

## コードカバレッジ検査

コードカバレッジは、テストで実行されるコード領域、および実行されないコード領域に関する情報を提供するので、より多くのコードをテストできるようにテストスイートを改善できます。コードアナライザは、`uncover` によって収集されたデータを使用して、プログラム内のどの関数がカバーされていないか、また、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ッジが何パーセント増加するかを調べます。

コードカバレッジデータの収集については、[19 ページの「コードカバレッジデータの収集」](#)を参照してください。

## コードアナライザを使用するための要件

コードアナライザは、Oracle Developer Studio 12.3、12.4、または 12.5 の C または C++ コンパイラでコンパイルされたバイナリから収集される静的エラーデータ、動的メモリアクセスエラーデータ、およびコードカバレッジデータを処理します。

コードアナライザは、Oracle Solaris 10 10/08 以上のオペレーティングシステム、Oracle Solaris 11、Oracle Enterprise Linux 5.x、または Oracle Enterprise Linux 6.x 以上を実行する SPARC ベースまたは x86 ベースのシステムで実行されます。

## コードアナライザ GUI

コンパイラ、Discover、または Uncover でデータを収集したあと、`code-analyzer` コマンドを発行してコードアナライザ GUI を起動し、問題の表示と分析を行うことができます。

コードアナライザは各問題について、問題の説明、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を表示します。

コードアナライザでは次の操作を実行できます。

- 問題の詳細を表示する。静的な問題の場合、詳細にはエラーパスが含まれます。動的メモリアクセスの問題の場合、詳細には呼び出しスタックが含まれ、データが使用可能であれば割り当てスタックと解放スタックも含まれます。
- 問題が見つかったソースファイルを開く。
- エラーパスまたはスタック内の関数呼び出しから関連するソースコード行に移動する。
- プログラムでの関数の使用箇所をすべて見つける。
- 関数宣言に移動する。
- オーバーライドされるまたはオーバーライドする関数の宣言に移動する。
- 関数のコールグラフを表示する。
- 問題の種類ごとに、コード例や考えられる原因などの詳細を表示する。
- 表示する問題を分析の種類、問題の種類、およびソースファイルによってフィルタリングする。
- すでに確認した問題を非表示にし、関心のない問題を閉じる。

GUI の使用に関する詳細は、GUI のオンラインヘルプと『[Oracle Developer Studio 12.5: コードアナライザチュートリアル](#)』を参照してください。

## コードアナライザのコマンド行インタフェース

コードアナライザのコマンド行インタフェースバージョンである `codean` は、静的コード検査、Discover、および Uncover を使用して、Analytics ファイルを入力として読み取り、テキストま

たは HTML 形式で出力を生成します。また、あとで履歴データと新しいデータを比較できるように、履歴アーカイブにデータを保管するメカニズムを提供します。codean では、次の操作を実行できます。

- API 形式でのレポートの読み取りと、テキストおよび HTML 形式への情報の変換。codean はテキスト出力を `.type.html` ファイル (`type` は `static`、`dynamic`、または `coverage`) に保存します。
- `.analyze/type/latest` レポートでは、各問題のチェックサムが計算され、元の問題情報が `.analyze/history//type` ファイル (`type` は `static`、`dynamic`、または `coverage`) に保管されます。
- 最新レポートに新しい問題または修正された問題だけを表示し、以前に保存されているレポートと比較する。
- 収集するデータの種類 (`dynamic`、`static`、`coverage`、または `all`) を指定する。
- フルパス名を表示する。
- 特定のソースファイルの問題を表示する。
- 特定の行数のソースコードを表示する。
- 最新のレポートを保存します。
- 同じタグ名を使用して最後に保存されたレポートを上書きする。
- 新しい問題または修正された問題だけをレポートに表示する。
- レポートを保存するディレクトリを指定する。
- 表示するエラーと警告の種類をフィルタリングする。

詳細については、codean(1) のマニュアルページを参照してください。

## リモートデスクトップ配布

ほとんどすべてのオペレーティングシステム上で動作し、リモートサーバー上の Oracle Developer Studio のコンパイラやツールを使用するコードアナライザのリモートデスクトップ配布を作成できます。インストール時にリモートデスクトップ配布を生成して、「デフォルトのユーザーディレクトリからユーザー設定をエクスポートする」オプションをチェックすると、コードアナライザは、配布の生成元となったサーバーをリモートホストとして認識し、Oracle Developer Studio インストール内のツールコレクションにアクセスします。デフォルトでは、このオプションは選択されません。

リモートオペレーティングシステムでコードアナライザを起動するには、該当する実行可能ファイルを実行します。

```
./codeanalyzer/bin/codeanalyzer.exe
```

リモートデスクトップ配布のインストール方法については、『[Oracle Developer Studio 12.5: インストールガイド](#)』を参照してください。

リモートデスクトップ配布については、コードアナライザ GUI オンラインヘルプを参照してください。

## クイックスタート

次に、コードに関する情報を収集するために必要な手順の例と、コードアナライザで結果を表示する方法を、サンプル C プログラムを使用して示します。

### ▼ クイックスタート

1. プログラムをコンパイルして静的データを収集します。

```
% cc -xprevisive=yes *.c
```

---

注記 - 以前は `-xanalyze=code` オプションを使用してコンパイルできました。このオプションは Oracle Developer Studio 12.5 でも有効ですが、EOL です。

---

2. デバッグ情報付きでプログラムを再コンパイルします。

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

3. プログラムに `discover` で計測機構を組み込んでからプログラムを実行し、動的メモリアクセスデータを収集します。

```
% cp a.out a.out.save  
% discover -a a.out  
% a.out
```

4. プログラムに `uncover` で計測機構を組み込んでから、コードカバレッジデータを収集します。

```
% a.out  
% cp a.out.save a.out  
% a.out  
% uncover a.out
```

5. 情報の収集後に、収集データを表示するためにコードアナライザを GUI または `codean` コマンド行ツールで使用するよう選択できます。

- GUI でコードアナライザにアクセスするには、次のコマンドを使用します。

```
% code-analyzer a.out
```

- コマンド行ツールでコードアナライザにアクセスするには、次のコマンドを使用します。

```
% codean a.out
```



## ◆◆◆ 第 2 章

# データの収集とコードアナライザの起動

---

コードアナライザでの分析のために収集されるデータは、ソースコードファイルが入っているディレクトリの `binary-name.analyze` ディレクトリに保存されます。`binary-name.analyze` ディレクトリは、コンパイラ、`discover`、または `uncover` により作成されます。

この章には、次のトピックに関する情報が含まれています。

- 17 ページの「静的エラーデータの収集」
- 18 ページの「動的メモリアクセスデータの収集」
- 19 ページの「コードカバレッジデータの収集」
- 20 ページの「コードアナライザ GUI の使用」

## 静的エラーデータの収集

C または C++ プログラムの静的エラーデータを収集するには、Oracle Developer Studio 12.3、Oracle Developer Studio 12.4、または Oracle Developer Studio 12.5 の C または C++ コンパイラで `-xprevisive=yes` オプションを指定して、プログラムをコンパイルします。以前は `-xanalyze=code` オプションを使用していましたが、このオプションは EOL であるため、代わりに `-xprevisive=yes` オプションを使用することをお勧めします。`-xprevisive=yes` オプションは、以前のリリースの Oracle Developer Studio のコンパイラでは使用できません。このオプションを使用すると、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。

プログラムを `-xprevisive=yes` オプションでコンパイルしたあと、別の手順でリンクする場合は、リンク手順でも `-xanalyze=code` オプションを指定する必要があります。

Linux では、静的エラーデータを収集するには `-xannotate` オプションと `-xprevisive=yes` を指定する必要があります。例:

```
% cc -xprevisive=yes -xannotate -g t.c
```

コンパイラはコード内の静的エラーをすべて検出できるわけではないことに注意してください。

- 実行時にのみ使用可能になるデータに依存するエラーもあります。たとえば、次のコードの場合、ファイルから読み取られる `ix` の値が `[0,9]` の範囲外にあることを検出できないため、コンパイラは ABW (配列範囲外への書き込み) エラーを検出しません。

```
void f(int fd, int array[10])
{
    int ix;
    read(fd, &ix, sizeof(ix));
    array[ix] = 0;
}
```

- 一部のエラーはあいまいであり、実際のエラーではないことがあります。コンパイラはこのようなエラーを報告しません。
- 一部の複雑なエラーは、このリリースのコンパイラでは検出されません。

静的エラーデータを収集したあと、コードアナライザ GUI またはコマンド行ツール (codean) を起動してデータの分析と表示を行ったり、動的メモリアクセスまたはコードカバレッジのデータを収集できるようにするためにプログラムを再度コンパイルできます。

## 動的メモリアクセスデータの収集

C または C++ プログラムの動的メモリアクセスデータの収集は、discover でバイナリに計測機構を組み込む手順と、計測機構付きバイナリを実行する手順の 2 つから成ります。

コードアナライザ用のデータを収集するために discover でプログラムに計測機構を組み込むには、プログラムを Oracle Developer Studio バージョン 12.3、12.4、12.5 の C または C++ コンパイラでコンパイルしておく必要があります。-g オプションでコンパイルするとデバッグ情報が生成され、動的メモリアクセス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。

プログラムが最適化なしでコンパイルされている場合に、discover はソースコードレベルでもっとも完全なメモリアクセスエラー検出を提供します。最適化を使用してコンパイルした場合、一部のメモリアクセスエラーは検出されません。

Discover で計測機構を組み込むことのできる、またはできないバイナリの種類については、『Oracle Developer Studio 12.5: Discover および Uncover ユーザーズガイド』の「サポートされるバイナリ」および『Oracle Developer Studio 12.5: Discover および Uncover ユーザーズガイド』の「プリロードまたは監査を使用するバイナリは互換性がない」を参照してください。

---

**注記** - discover と uncover の両方で使用するためのプログラムを 1 回で構築できます。ただし、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、uncover を使用してカバレッジデータも収集する予定であれば、discover でバイナリに計測機構を組み込む前に uncover 用のコピーを保存してください。例:

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

---

## ▼ バイナリから動的メモリアクセスデータを収集する方法:

1. Discover で `-a` オプションを使用してバイナリに計測機構を組み込みます。

```
% discover -a binary_name
```

---

注記 - Oracle Developer Studio バージョン 12.3、12.4、または 12.5 のバージョンの `discover` を使用する必要があります。以前のバージョンの `discover` では `-a` オプションは使用できません。

---

2. 計測機構付きバイナリを実行します。

動的メモリアクセスデータが、`binary_name.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。

---

注記 - `discover` でバイナリに計測機構を組み込むときに指定できるその他の計測オプションについては、『[Oracle Developer Studio 12.5: Discover および Uncover ユーザーズガイド](#)』の「計測オプション」または `discover` のマニュアルページを参照してください。

---

3. (オプション) コードアナライザの GUI またはコマンド行ツール (`codean`) を起動し、以前に収集した静的コードデータを使用してデータを分析および表示します。または、計測機構の付いていないバイナリのコピーを使用して、コードカバレッジデータを収集できます。

## コードカバレッジデータの収集

C または C++ プログラムのコードカバレッジデータの収集は、3 段階のプロセスです。

1. `uncover` でバイナリに計測機構を組み込みます。
2. 計測済みバイナリの実行
3. `uncover` を再度実行してコードアナライザが使用するカバレッジレポートを生成します。

バイナリに計測機構を組み込んだあと、その計測機構付きバイナリを複数回実行してすべての実行に関するデータを蓄積してから、カバレッジレポートを生成できます。

## ▼ バイナリからコードカバレッジデータを収集する方法

始める前に コードアナライザで使用するデータを収集するために `uncover` でプログラムに計測機構を組み込むには、プログラムを Oracle Developer Studio バージョン 12.3、12.4、または 12.5 の C または C++ コンパイラでコンパイルしておく必要があります。`-g` オプションでコンパイルするとデバッグ情報が生成され、ソースコードレベルのカバレッジ情報をコードアナライザで使用できるようになります。

---

**注記** - `discover` で計測機構を組み込むためにプログラムをコンパイルしたときに、バイナリのコピーを保存した場合は、そのコピーの名前を元のバイナリ名に変更し、それを使用して `uncover` で計測機構を組み込むことができます。例:

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

---

1. **Uncover でバイナリに計測機構を組み込みます。**

```
% uncover binary-name
```

2. **計測機構付きバイナリを 1 回以上実行します。**

コードカバレッジデータが、`binary-name.uc` ディレクトリに書き込まれます。

3. **Uncover で `-a` オプションを使用して、蓄積されたデータからコードカバレッジレポートを生成します。**

```
% uncover -a binary-name.uc
```

カバレッジレポートが、`binary-name.analyze` ディレクトリの `coverage` サブディレクトリに書き込まれます。

---

**注記** - Oracle Developer Studio バージョン 12.3、12.4、または 12.5 のバージョンの `uncover` を使用する必要があります。以前のバージョンの `uncover` では `-a` オプションは使用できません。

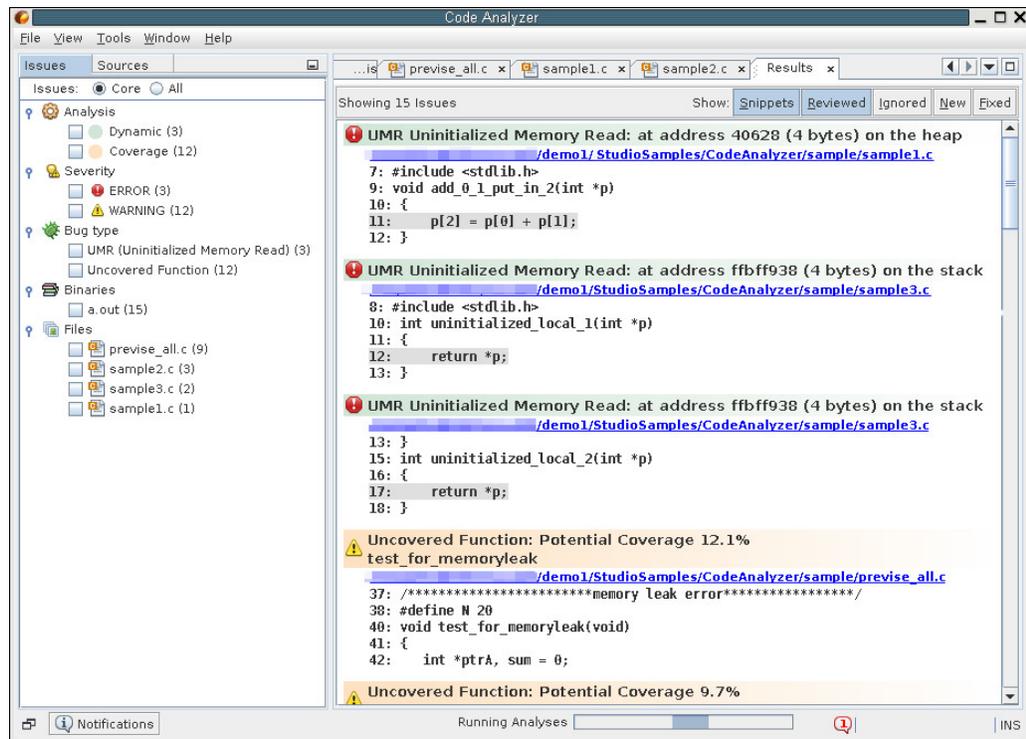
---

## コードアナライザ GUI の使用

コードアナライザ GUI を使用して、3 種類までのデータを分析できます。GUI を起動するには、`code-analyzer` コマンドと、収集したエラーデータを分析するバイナリへのパスを入力します。

```
% code-analyzer binary-name
```

次の図に示すように、コードアナライザ GUI は、`binary-name.analyze` ディレクトリのデータを開き、表示します。



コードアナライザ GUI の実行中に、「開く」->「ファイル」を選択して別のバイナリを指定すると、そのバイナリについて収集されたデータの表示に切り替えることができます。

GUI のオンラインヘルプには、表示する結果のフィルタリング、問題の表示/非表示、および特定の問題に関する詳細の表示を行う、すべての機能の使用方法が説明されています。『Oracle Developer Studio 12.5: コードアナライザチュートリアル』では、サンプルプログラムを使用して、データの収集と分析のシナリオ全体を説明します。

## コードアナライザコマンド行ツール (codean) の使用

コードアナライザコマンド行ツール codean を使用して、3 種類までのデータを分析できます。codean を起動するには、codean コマンド、オプション、および実行可能ファイルまたはディレクトリのパスを入力します。

```
codean options executable-path|directory
```

codean ツールは、画面にテキスト出力を表示します。実行可能ファイルと同じ場所にある .type.html ファイルで結果を確認することもできます。このセクションでは、コマンドオプションについて説明します。

## codean のオプション

以降のセクションでは、codean に使用できる各種オプションについて説明します。

### データの種類のオプション

次のオプションは、収集するデータの種類を指定します。

- s                   静的データを処理および表示します。
- d                   動的データを処理および表示します。
- c                   カバレッジデータを処理および表示します。

複数のオプションを指定するか、オプションを指定しないでおくことができます。何も選択しないと、デフォルトで、`.analyze/type/latest` ファイル (`type` は `static`、`dynamic`、`coverage` のいずれか) が存在しているかどうかに応じて、すべてのオプションが処理されます。

### 表示オプション

次のオプションは、結果のテキスト出力の内容を決定します。

- `--fullpath`       ファイルのフルパス名を表示します。
- `-f source-file`   指定したソースファイルに関する問題だけを表示します。
- `-n number`       指定された行数のソースコードを表示します。
- `-v`               バージョンを出力し、それ以上の処理を行わずに終了します。

### フィルタリングオプション

次のオプションは、エラーや警告の種類、ハッシュ文字列、またはラベル名でフィルタリングすることで、結果に報告される問題を決定します。

エラーまたは警告の種類は、次のいずれかです。

- 3 文字のエラーコードまたは 3 文字の警告コード。考えられるエラーと警告のリストについては、[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。
- MLK または mlk (メモリーリーク)。

- ALL または all (すべての警告またはエラー)。

エラーまたは警告が指定されていない場合、デフォルトは all です。

フィルタリングオプションは次のとおりです。

<code>--showerrors</code> <i>error-type</i>	指定された種類のエラーだけを表示します。
<code>--showwarnings</code> <i>warning-type</i>	指定された種類の警告だけを表示します。
<code>--hideerrors</code> <i>error-type</i>	指定された種類のエラーを表示しません。
<code>--hidewarnings</code> <i>warning-type</i>	指定された種類の警告を表示しません。

## ラベル付けオプション

ハッシュ文字列を使用して問題にラベルを付けることができます。codean には、`false_positive`、`verified`、`wont_be_fixed` という 3 つのラベルが用意されています。これらのラベル付けオプションの使用の詳細は、[26 ページの「問題のラベル付け」](#)を参照してください。

<code>--showhash</code> <i>hash</i>	各問題に関連付けられたハッシュ文字列を表示します。
<code>--showlabel</code> [ <i>verified</i>   <i>false_positive</i>   <i>wont_be_fixed</i> ]	指定されたラベルの問題だけを表示します。
<code>--hidelabel</code> [ <i>verified</i>   <i>false_positive</i>   <i>wont_be_fixed</i> ]	指定されたラベルの問題を非表示にします。
<code>--findhash</code> <i>hash</i> [: <i>hash2</i> ...]	指定されたハッシュ文字列 <i>hash</i> に関連付けられた問題だけを表示します。複数のハッシュ文字列を一覧表示できます。

## 結果保存オプション

最新の結果をファイルに保存し、このファイルに特定のタグ名を付けて特定のディレクトリに配置できます。

<code>--save</code>	最新のレポートを保存します。
<code>--tag tag-name</code>	<code>--save</code> と組み合わせて使用する場合は、保存されるコピーの名前としてタグ名 <code>tag-name</code> が使用されます。保存されているコピーに同じタグ名が使用されている場合、codean は警告メッセージを発行し、ファイルを上書きせずに終了します。タグ名が指定されていない場合、codean は実行可能ファイルの最新レポートの最終変更時間をチェックし、そのタイムスタンプをタグ名として使用します。
<code>-t</code>	同じタグ名で保存されているレポートを上書きします。
<code>-D directory</code>	レポートをディレクトリ <code>directory</code> に保存します。

## 結果比較オプション

次のオプションを使用して、以前に生成されたレポートと結果を比較できます。

<code>--whatisnew</code>	新しい問題だけを表示します。このオプションは <code>--whatisfixed</code> と一緒に使用できません。
<code>--whatisfixed</code>	修正された問題だけを表示します。このオプションは <code>--whatisnew</code> と一緒に使用できません。
<code>--tag tag-name</code>	<code>--whatisnew</code> または <code>--whatisfixed</code> と組み合わせて使用すると、新しく生成されたレポートとの比較に、タグ名 <code>tag-name</code> が付いているレポートの履歴コピーが使用されます。タグ名が指定されていない場合、最新レポートが最終保存コピーと比較されます。
<code>--ref file directory</code>	<code>--whatisnew</code> または <code>--whatisfixed</code> と組み合わせ、後にパス名を指定する必要があります。このオプションは、新しいレポートと比較するファイルまたはディレクトリを指定します。

## テストスイートのオプション

codean の `--union` オプションを使用すると、複数のテストレポートを処理して、テストスイートでの `discover` の実行結果の表示、保存、新しい問題の表示、または修正された問題の表示を行うことができます。詳細は、[29 ページの「テストスイートでの codean の使用」](#)を参照してください。

<code>--union</code>	複数の動的レポートを表示します。 <code>--save</code> 、 <code>--whatisnew</code> 、および <code>--whatisfixed</code> とともに指定すると、複数の動的レポートの保存、新しい問
----------------------	---

題の表示、および修正された問題の表示を行います。複数のレポートに表示される問題の詳細は、一度だけ表示されます。

## codean ワークフローの例

このセクションでは、バグ修正の効果をモニターする例を説明します。

### 例 1 ワークフローの例

1. 修正の前にターゲットソースをコンパイルします。

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

2. Discover を使用してバイナリに計測機構を組み込み、Analytics 出力が生成されるようにします。

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

3. 計測されたバイナリを実行します。
4. codean を使用して Analytics 出力を格納します。履歴アーカイブが `a.out.analyze/history/before_bugfix` に作成され、`dynamic` という名前の履歴ファイルがこのディレクトリ内に作成されます。

```
% codean --save --tag before_bugfix -d a.out
```

5. バグを修正します。
6. ターゲットソースをもう一度コンパイルします。

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

7. discover を使用して再度バイナリに計測機構を組み込みます。

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

8. 計測されたバイナリを実行します。

```
% a.out
```

9. 比較結果を表示し、バグが原因で発生していた無効なメモリアクセスが修正されたことを確認します。

```
% codean --whatisfixed --tag before_bugfix -d a.out
```

これにより、修正された動的な問題だけが含まれる新しい Analytics 出力ファイルが `a.out.analyze/dynamic/fixed_before_bugfix` に作成されます。修正された問題を表示するには、codean またはコードアナライザ GUI を使用できます。

10. (オプション) codean を実行して、新しいバグが発生していないことを確認します。

```
% codean --whatisnew --tag before_bugfix -d a.out
```

このコマンドにより、新しい動的な問題だけが含まれる新しい Analytics ファイルが `a.out.analyze/dynamic/new_before_bugfix` に作成されます。

## 問題のラベル付け

次のセクションでは、ラベルを使用して問題をソートおよび表示する一般的なワークフローについて説明します。

### ▼ 問題にラベル付けする方法

1. `--showhash` オプションを使用して、問題に関連付けられたハッシュを表示します。
2. ラベル付けする問題とそのハッシュ文字列を特定します。
3. バイナリの `labels` サブディレクトリを作成します。  
たとえば、バイナリが `a.out` の場合は、`a.out.analyze/labels` を作成します。
4. ラベル付けする問題の (ハッシュとラベルの) ペアを次の 3 つのファイルに入れます。
  - `a.out.analyze/labels/static_report_labels`
  - `a.out.analyze/labels/dynamic_report_labels`
  - `a.out.analyze/labels/coverage_report_labels`

各ディレクトリにはそれぞれ、静的レポート、動的レポート、およびカバレッジレポート内の問題が入ります。ラベルファイルの形式は `hash-name:label-name:comment` です。  
次に、ラベルファイルの例を示します。

```
$ cat codean.analyze/labels/dynamic_report_labels
54f3a6f0160dceb58156be03d07090a2:false_positive:bug 12345678 has been filed
3b7ee9d573847e2dbf80652b7a89026e:false_positive
6c575302146d147f5f1d2d2e6e1710a5:false_positive
```

`codean` を使用して `a.out` というレポートを処理しているときに、一致するラベルが問題に含まれている場合は、ラベル名がデフォルトでその問題のあとに追加情報として表示されます。

### ▼ ラベルで問題を表示または非表示にする方法

1. 問題を表示するには、`--showlabel` オプションを使用します。  
たとえば、誤検出のみを表示する場合:  

```
% codean --showlabel false_positive a.out
```
2. 問題を非表示にするには、`--hidelabel` オプションを使用します。

たとえば、`wont_be_fixed` ラベルの付いた問題を非表示にする場合:

```
% codean --hidelabel wont_be_fixed a.out
```

## ▼ 特定のハッシュを検索する方法

1. ラベルファイル内の特定のハッシュが期限切れかどうかを調べるには、`--findhash hash-string` を使用して、そのハッシュと一致する問題のみを表示するよう `codean` に指示します。
2. 複数のハッシュを検索するには、複数の `hash-string` をコロン (:) で区切って指定します。例:

```
% codean --findhash 54f3a6f0160dceb58156be03d07090a2:3b7ee9d573847e2dbf80652b7a89026e a.out
```

### 例 2 さまざまなラベルの使用

次に、さまざまなラベルの使用例を示します。

```
$ cat t.c
#include <stdlib.h>

int main()
{
int *p = (int *)malloc(sizeof(int));
int i = *p;
free(p);
return i;
}

$ cc -g t.c
$ discover -a -o a.out.disc a.out
$ ./a.out.disc

$ codean -d --showhash a.out
DYNAMIC report of a.out:
ERROR 1 (UMR): accessing uninitialized data in "*p" at address 0x1001208e0 (4 bytes) on the
heap:
hash: 79b6e1b242a057deec8762328b6860e6
main() + 0xac <t.c : 6>
3: int main()
4: {
5: int *p = (int *)malloc(sizeof(int));
6:=> int i = *p;
7: free(p);
_start() + 0x108
was allocated at (4 bytes):
main() + 0x20 <t.c : 5>
1: #include <stdlib.h>
3: int main()
4: {
5:=> int *p = (int *)malloc(sizeof(int));
6: int i = *p;
_start() + 0x108
```

```
DISCOVER SUMMARY for a.out: 1 non-leak issues, 0 leak issues
unique errors : 1 (1 total)
unique warnings : 0 (0 total)
unique leaks : 0 (0 blocks, 0 bytes)
unique possible leaks : 0 (0 blocks, 0 bytes)

$ cat a.out.analyze/labels/dynamic_report_labels
79b6e1b242a057deec8762328b6860e6:verified:I have verified that this is a bug.

$ codean -d a.out
DYNAMIC report of a.out:
ERROR 1 (UMR): accessing uninitialized data in "*" at address 0x1001208e0 (4 bytes) on the
  heap:
label: verified "I have verified that this is a bug."
main() + 0xac <t.c : 6>
3: int main()
4: {
5: int *p = (int *)malloc(sizeof(int));
6: => int i = *p;
7: free(p);
_start() + 0x108
was allocated at (4 bytes):
main() + 0x20 <t.c : 5>
1: #include <stdlib.h>
3: int main()
4: {
5: => int *p = (int *)malloc(sizeof(int));
6: int i = *p;
_start() + 0x108
DISCOVER SUMMARY for a.out: 1 non-leak issues, 0 leak issues
unique errors : 1 (1 total)
unique warnings : 0 (0 total)
unique leaks : 0 (0 blocks, 0 bytes)
unique possible leaks : 0 (0 blocks, 0 bytes)

$ codean -d --showlabel verified a.out
DYNAMIC report of a.out:
ERROR 1 (UMR): accessing uninitialized data in "*" at address 0x1001208e0 (4 bytes) on the
  heap:
label: verified "I have verified that this is a bug."
main() + 0xac <t.c : 6>
3: int main()
4: {
5: int *p = (int *)malloc(sizeof(int));
6: => int i = *p;
7: free(p);
_start() + 0x108
was allocated at (4 bytes):
main() + 0x20 <t.c : 5>
1: #include <stdlib.h>
3: int main()
4: {
5: => int *p = (int *)malloc(sizeof(int));
6: int i = *p;
```

```
_start() + 0x108
DISCOVER SUMMARY for a.out: 1 non-leak issues, 0 leak issues
unique errors : 1 (1 total)
unique warnings : 0 (0 total)
unique leaks : 0 (0 blocks, 0 bytes)
unique possible leaks : 0 (0 blocks, 0 bytes)

$ codean -d --hidelabel verified a.out
DYNAMIC report of a.out:
DISCOVER SUMMARY for a.out: 0 issues found (1 issues suppressed)
```

## テストスイートでの codean の使用

--union を使用すると、codean で複数の動的レポートを処理できます。これを使用すると、テストスイートでの discover の実行結果の表示、保存、新しい問題の表示、または修正された問題の表示を行うことができます。次の例は、discover および codean --union の使用方法を示しています。

### ▼ discover を使用したテストスイートでのバイナリの準備

1. discover を使用して a.out に計測機構を組み込みます。

2. Analytics 出力を選択します: discover -a a.out。

これにより、a.out.analyze/dynamic/ の下にある既存の動的レポートがすべて削除されます。

3. テストスイートに対して計測機構付きバイナリを実行します。

デフォルトでは、各実行の結果が別のファイルに保存され、latest レポートが最新のレポートへのシンボリックリンクになります。たとえば、a.out を 5 回実行したあとの a.out.analyze/dynamic/ ディレクトリは次のようになります。

```
$ ls -l a.out.analyze/dynamic/
total 11
lrwxrwxrwx 1 demoUser demo 18 May 1 15:14 latest -> ./latest.AAAvCaWri
-rwxrwxrwx 1 demoUser demo 588 Apr 30 10:05 latest.AAACRaOId
-rwxrwxrwx 1 demoUser demo 587 Apr 15 15:03 latest.AAAQcayId
-rwxrwxrwx 1 demoUser demo 587 Apr 30 10:05 latest.AAAe5aWId
-rwxrwxrwx 1 demoUser demo 587 Apr 15 15:03 latest.AAAlCaGIId
-rwxrwxrwx 1 demoUser demo 587 May 1 15:14 latest.AAAvCaWri
```

discover によるレポートの準備が完了したら、codean を使用して a.out.analyze/dynamic/ の下にあるすべての動的レポートを処理できます。

## テストスイートでのレポートの表示

個々のレポートを表示するには:

```
% codean --union -d a.out
```

codean コマンドは、a.out.analyze/dynamic の下にある個々の動的レポートをすべてアルファベット順に処理します。これによってテキスト出力のみが生成されます (つまり、標準の codean の実行とは異なり、Analytics 出力や HTML 出力が合わせて生成されることはありません)。同じ問題については、どのレポートが存在していても、最初に出現したときのみ問題の詳細が表示されます。それ以降に出現した場合、codean は次のような限られた情報のみを表示します。

```
LEAK 1: repeated, 1 blocks, 4 bytes
ERROR 1: repeated 1 time
```

discover 情報は詳細レポートの最後に一度だけ表示され、テストスイート全体での問題の総数がまとめられています。codean の完全な出力の例は次のとおりです。

```
$ codean --union -d a.out
```

```
Displaying dynamic report of a.out.analyze/dynamic/latest.AAACRa0Id:
```

```
ERROR 1 (UMR): accessing uninitialized data in "*" at address 0x8090010 (4 bytes) on the heap
  at:
main() + 0xe1 <hello.c : 10>
5: {
6: int *i = malloc(sizeof(int));
8: int j = 0;
10: => j = *i;
12: return 0;
_start() + 0x71
was allocated at (4 bytes):
main() + 0x5e <hello.c : 6>
2: #include <stdio.h>
4: int main()
5: {
6: => int *i = malloc(sizeof(int));
8: int j = 0;
_start() + 0x71
```

```
Displaying dynamic report of a.out.analyze/dynamic/latest.AAAQcayId:
```

```
LEAK 1: 1 allocation with total size of 4 bytes
main() + 0x5e <hello.c : 6>
2: #include <stdio.h>
4: int main()
5: {
6: => int *i = malloc(sizeof(int));
8: int j = 0;
_start() + 0x71
ERROR 1: repeated 1 time
```

```
Displaying dynamic report of a.out.analyze/dynamic/latest.AAAe5aWId:
```

```
LEAK 1: repeated, 1 blocks, 4 bytes
ERROR 1: repeated 1 time
```

```
Displaying dynamic report of a.out.analyze/dynamic/latest.AAA1CaGId:
```

```
LEAK 1: repeated, 1 blocks, 4 bytes
ERROR 1: repeated 1 time
```

```
Displaying dynamic report of a.out.analyze/dynamic/latest.AAAvCaWri:
```

```
LEAK 1: repeated, 1 blocks, 4 bytes
ERROR 1: repeated 1 time
```

```
DISCOVER SUMMARY for a.out: 1 non-leak issues, 1 leak issues
unique errors : 1 (5 total)
unique warnings : 0 (0 total)
unique leaks : 1 (4 blocks, 16 bytes)
unique possible leaks : 0 (0 blocks, 0 bytes)
```

明らかなリークと可能性のあるリークの扱い方が異なることに注意してください。標準の codean を実行している場合、リークが明らかなリークまたは可能性のあるリークのどちらであるかは、Analytics レポートの信頼値に完全に依存しています。しかし、「テストスイート」の codean を実行している場合、動的レポートのどれかで明らかなリークと判断されたリークは、レポートの信頼値に関係なく、残りのレポートでも明らかなリークと見なされます。次の「テストスイート」の codean と標準の codean のレポートで、LEAK 1 がどのように表示されるかを確認してください。

```
$ codean --union -d a.out
```

```
Displaying dynamic report of a.out.analyze/dynamic/latest.AAACRaOId:
```

```
...
```

```
Displaying dynamic report of a.out.analyze/dynamic/latest.AAAQcayId:
```

```
LEAK 1: 1 allocation with total size of 4 bytes
main() + 0x5e <hello.c : 6>
2: #include <stdio.h>
4: int main()
5: {
6: => int *i = malloc(sizeof(int));
8: int j = 0;
_start() + 0x71
...
```

```
Displaying dynamic report of a.out.analyze/dynamic/latest.AAAe5aWId:
```

```
LEAK 1: repeated, 1 blocks, 4 bytes
...
```

```
Displaying dynamic report of a.out.analyze/dynamic/latest.AAA1CaGId:
```

```
LEAK 1: repeated, 1 blocks, 4 bytes
...

Displaying dynamic report of a.out.analyze/dynamic/latest.AAAvCaWri:

LEAK 1: repeated, 1 blocks, 4 bytes
...

DISCOVER SUMMARY for a.out: 1 non-leak issues, 1 leak issues
unique errors : 1 (5 total)
unique warnings : 0 (0 total)
unique leaks : 1 (4 blocks, 16 bytes)
unique possible leaks : 0 (0 blocks, 0 bytes)

tests$ codean -d a.out
DYNAMIC report of a.out:
...
LEAK (Possible leak) 1: 1 allocation with total size of 4 bytes
main() + 0x5e <hello.c : 6>
2: #include <stdio.h>
4: int main()
5: {
6: => int *i = malloc(sizeof(int));
8: int j = 0;
_start() + 0x71
DISCOVER SUMMARY for a.out: 1 non-leak issues, 1 leak issues
unique errors : 1 (1 total)
unique warnings : 0 (0 total)
unique leaks : 0 (0 blocks, 0 bytes)
unique possible leaks : 1 (1 blocks, 4 bytes)
```

## テストスイートでのレポートの保存

テストスイートでレポートを保存するには:

```
% codean --save --union -d --tag run1 a.out
```

a.out.analyze/dynamic/ の下にある各動的レポートは、別のファイルに保存されます。

```
$ ls -l a.out.analyze/history/run1/
total 15
lrwxrwxrwx 1 demoUser demo 26 Sep 30 11:09 dynamic -> ./dynamic.latest.AAACRaOId
-rw-r--r-- 1 demoUser demo 674 Sep 30 11:09 dynamic.latest.AAACRaOId
-rw-r--r-- 1 demoUser demo 847 Sep 30 11:09 dynamic.latest.AAAQcayId
-rw-r--r-- 1 demoUser demo 847 Sep 30 11:09 dynamic.latest.AAAe5aWId
-rw-r--r-- 1 demoUser demo 847 Sep 30 11:09 dynamic.latest.AAALCaGIId
-rw-r--r-- 1 demoUser demo 847 Sep 30 11:09 dynamic.latest.AAAvCaWri
```

## テストスイートでのレポートの比較

テストスイートでレポートを比較するには:

```
% codean --whatisnew --union -d --tag run1 a.out
% codean --whatisfixed --union -d --tag run1 a.out
```

codean コマンドは、現在 a.out.analyze/dynamic/ の下にあるすべての動的レポートの新しい問題と修正された問題を 1 セットで表示し、a.out.analyze/history/run1/ の下にある保存されたすべての動的なレポートを 1 セットで表示します。次に出力例を示します。

---

**注記** - 可能性のあるリークと明らかでないリークの扱いは、[30 ページの「テストスイートでのレポートの表示」](#)での説明と同じです。

---

```
$ codean --whatisnew --union -d --tag run1 a.out
DYNAMIC report of a.out showing new issues:
New issues in a.out.analyze/dynamic/latest.AAARTa0xS:
ERROR 1 (ABR): reading memory beyond array bounds at address 0xffeffdef8 (4 bytes) on the stack
at:
main() + 0x68 <hello.c : 11>
6: // int *i = malloc(sizeof(int));
7: int i[30];
9: int j = 0;
11: => j = i[35];
13: return 0;
_start() + 0x71
New issues in a.out.analyze/dynamic/latest.AAATDaGxS:
ERROR 1 is a new, but repeated error. It was first seen as ERROR 1 in latest.AAARTa0xS.
New issues in a.out.analyze/dynamic/latest.AAARca4wS:
ERROR 1 is a new, but repeated error. It was first seen as ERROR 1 in latest.AAARTa0xS.

DISCOVER SUMMARY for a.out: 1 new non-leak issues, 0 new leak issues
new unique errors : 1 (3 total)
new unique warnings : 0 (0 total)
new unique leaks : 0 (0 blocks, 0 bytes)
new unique possible leaks : 0 (0 blocks, 0 bytes)

tests$ codean --whatisfixed --union -d --tag run1 a.out
DYNAMIC report of a.out showing fixed issues:
Fixed issues in a.out.analyze/history/run1/dynamic.latest.AAACRa0Id:
ERROR 1 (UMR): accessing uninitialized data in "*i" at address 0x8090010 (4 bytes) on the heap
at:
(Warning: Source files have changed. Source code shown below may not be accurate.)
main() + 0xe1 <hello.c : 10>
6: // int *i = malloc(sizeof(int));
7: int i[30];
9: int j = 0;
11: j = i[35];
_start() + 0x71
was allocated at (4 bytes):
main() + 0x5e <hello.c : 6>
2: #include <stdio.h>
4: int main()
5: {
6: => // int *i = malloc(sizeof(int));
7: int i[30];
_start() + 0x71
```

```
Fixed issues in a.out.analyze/history/run1/dynamic.latest.AAAQcayId:
ERROR 1 is a fixed, but repeated error. It was first seen as ERROR 1 in dynamic.latest.
AAACRa0Id.
LEAK 1: 1 allocation with total size of 4 bytes
(Warning: Source files have changed. Source code shown below may not be accurate.)
main() + 0x5e <hello.c : 6>
2: #include <stdio.h>
4: int main()
5: {
6: => // int *i = malloc(sizeof(int));
7: int i[30];
_start() + 0x71
Fixed issues in a.out.analyze/history/run1/dynamic.latest.AAAe5aWId:
ERROR 1 is a fixed, but repeated error. It was first seen as ERROR 1 in dynamic.latest.
AAACRa0Id.
LEAK 1 is a fixed, but repeated leak. It was first seen as LEAK 1 in dynamic.latest.AAAQcayId.
Fixed issues in a.out.analyze/history/run1/dynamic.latest.AAA1CaGId:
ERROR 1 is a fixed, but repeated error. It was first seen as ERROR 1 in dynamic.latest.
AAACRa0Id.
LEAK 1 is a fixed, but repeated leak. It was first seen as LEAK 1 in dynamic.latest.AAAQcayId.
Fixed issues in a.out.analyze/history/run1/dynamic.latest.AAAvCaWri:
ERROR 1 is a fixed, but repeated error. It was first seen as ERROR 1 in dynamic.latest.
AAACRa0Id.
LEAK 1 is a fixed, but repeated leak. It was first seen as LEAK 1 in dynamic.latest.AAAQcayId.

DISCOVER SUMMARY for a.out: 1 fixed non-leak issues, 1 fixed leak issues
fixed unique errors : 1 (5 total)
fixed unique warnings : 0 (0 total)
fixed unique leaks : 1 (4 blocks, 16 bytes)
fixed unique possible leaks : 0 (0 blocks, 0 bytes)
```

# ◆◆◆ 付録 A

## コードアナライザで分析されるエラー

---

コンパイラ、discover、および uncover は、コード内の静的コードの問題、動的メモリアクセスの問題、およびカバレッジの問題を検出します。この付録では、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特有のエラーの種類について説明します。

- 35 ページの「コードカバレッジの問題」
- 35 ページの「静的コードの問題」
- 40 ページの「動的メモリアクセスエラー」
- 47 ページの「動的メモリアクセスの警告」

### コードカバレッジの問題

コードカバレッジ検査では、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ッジの問題に「カバーされていない関数」というラベルが付けられ、潜在的なカバレッ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ッジが何パーセント増加するかを示しています。

**考えられる原因:** 関数を実行するテストが行われなかったか、デッドコードまたは古いコードを削除していません。

### 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)

このセクションでは、エラーの考えられる原因と、エラーが発生する可能性があるコードの例を説明します。

## 配列境界を越える読み取り (ABR)

考えられる原因: 配列境界を越えてメモリーを読み取ろうとしました。

例:

```
int a[5];
...
printf("a[5] = %d\n",a[5]); // Reading memory beyond array bounds
```

## 配列境界を越える書き込み (ABW)

考えられる原因: 配列境界を越えてメモリーに書き込もうとしました。

例:

```
int a [5];
...
a[5] = 5; // Writing to memory beyond array bounds
```

## メモリーの二重解放 (DFM)

考えられる原因: 同じポインタを使用して free()() を複数回呼び出しました。C++ では、同じポインタに対して delete 演算子を 2 回以上使用しています。

例:

```
int *p = (int*) malloc(sizeof(int));
```

```
free(p);
. . . // p was not signed a new value between the free statements
free(p); // Double freeing memory
```

## 解放済みメモリの読み取り (FMR)

例:

```
int *p = (int*) malloc(sizeof(int));
free(p);
. . . // Nothing assigned to p in between
int j = *p; // Reading from freed memory
```

## 解放済みメモリの書き込み (FMW)

例:

```
int *p = (int*) malloc(sizeof(int));
free(p);
. . . // Nothing assigned to p in between
*p = 1; // Writing to freed memory
```

## 空の無限ループ (INF)

例:

```
int x=0;
int i=0;
while (i<200) {
    x++; } // Infinite loop
```

## メモリーリーク

考えられる原因: メモリーが割り当てられるが、関数の終了またはエスケープの前に解放されていません。

例:

```
int foo()
{
    int *p = (int*) malloc(sizeof(int));
    if (x) {
        p = (int *) malloc(5*sizeof(int)); // will cause a leak of the 1st malloc
    }
} // The 2nd malloc leaked here
```

## 関数の復帰なし (MFR)

考えられる原因: 終了するパスの一部に戻り値がありません。

例:

```
#include <stdio.h>
int foo (int a, int b)
{
    if (a)
    {
        return b;
    }
    // If foo returns here, the return is uninitialized
int main ( )
{
    printf("%d\n", foo(0,30));
}
```

## malloc 戻り値の検査なし (MRC)

考えられる原因: Cの malloc または C++ の new 演算子からの戻り値に null 検査なしでアクセスします。

例:

```
#include <stdlib.h>
int main()
{
    int *p3 = (int*) malloc(sizeof(int)); // Missing null-pointer check after malloc.
    *p3 = 0;
}
```

## リークの可能性のあるポインタの検査: NULL ポインタ間接参照 (NUL)

考えられる原因: null に等しくなる可能性のあるポインタにアクセスしているか、null になることのないポインタに冗長な null 検査を行なっています。

例:

```
#include <stdio.h>
#include <stdlib.h>
int gp, ctl;
int main()
{
    int *p = gp;
```

```
if (ctl)
    p = 0;
printf ("%c\n", *p); // May be null pointer dereference
if (!p)
    *p = 0; // Surely null pointer dereference

int *p2 = gp;
*p2 = 0; // Access before checking against NULL.
assert (p2!=0);

int *p3 = gp;
if (p3) {
    printf ("p3 is not zero.\n");
}
*p3 = 0; // Access is not protected by previous check against NULL.
}
```

## 解放済みメモリーを返す (RFM)

例:

```
#include <stdlib.h>
int *foo ()
{
    int *p = (int*) malloc(sizeof(int));
    free(p);
    return p; // Return freed memory is dangerous
}
int main()
{
    int *p = foo();
    *p = 0;
}
```

## 初期化されていないメモリーの読み取り (UMR)

考えられる原因: 初期化されていないローカルデータまたはヒープデータの読み取り。

例:

```
#include <stdio.h>
#include <stdlib.h>
struct ttt {
    int a;
    int b;
};

int main()
{
    int *p = (int*) malloc(sizeof(int));
```

```
printf("*p = %d\n",*p); // Accessing uninitialized data

struct ttt t;
extern void foo (struct ttt *);

t.a = 1;
foo (&t); // Access uninitialized bitfield data "t.b"
}
```

## 使用されていない戻り値 (URV)

考えられる原因: 初期化されていないローカルデータまたはヒープデータの読み取り。

例:

```
int foo();
int main()
{
    foo(); // Return value is not used.
}
```

## スコープ外での局所変数の使用 (VES)

考えられる原因: 初期化されていないローカルデータまたはヒープデータの読み取り。

例:

```
int main()
{
    int *p = (int *)0;
    void bar (int *);
    {
        int a[10];
        p = a;
    } // local variable 'a' leaked out
    bar(p);
}
```

## 動的メモリアクセスエラー

動的メモリアクセス検査では、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)

このセクションでは、エラーの考えられる原因と、エラーが発生する可能性があるコードの例を説明します。

## 配列境界を越える読み取り (ABR)

考えられる原因: 配列境界を越えてメモリーを読み取ろうとしました。

例:

```
int a[5];
...
printf("a[5] = %d\n",a[5]); // Reading memory beyond array bounds
```

## 配列境界を越える書き込み (ABW)

考えられる原因: 配列境界を越えてメモリーに書き込もうとしました。

例:

```
int a [5];
...
a[5] = 5; // Writing to memory beyond array bounds
```

## 不正な空きメモリー (BFM)

考えられる原因: free()() または realloc()() にヒープデータ以外のポインタを渡しました。

例:

```
#include <stdlib.h>
int main()
{
    int *p = (int*) malloc(sizeof(int));
    free(p+1); // Freeing wrong memory block
}
```

## 不正な Realloc アドレスパラメータ (BRP)

例:

```
#include <stdlib.h>
int main()
{
    int *p = (int*) realloc(0,sizeof(int));
    int *q = (int*) realloc(p+20,sizeof(int[2])); // Bad address parameter for realloc
}
```

## 破損したガードブロック (CGB)

考えられる原因: 動的に割り当てられた配列の末尾を越えて「レッドゾーン」に書き込んでいます。

例:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int *) malloc(sizeof(int)*4);
    *(p+5) = 10; // Corrupted array guard block detected (only when the code is not annotated)
    free(p);

    return 0;
}
```

## メモリの二重解放 (DFM)

考えられる原因: 同じポインタを使用して free()() を複数回呼び出しました。C++ では、同じポインタに対して delete 演算子を 2 回以上使用しています。

例:

```
int *p = (int*) malloc(sizeof(int));
free(p);
. . . // p was not assigned a new value between the free statements
free(p); // Double freeing memory
```

## 解放済みメモリの読み取り (FMR)

例:

```
int *p = (int*) malloc(sizeof(int));
free(p);
. . . // Nothing assigned to p in between
printf("p = 0x%h\n",p); // Reading from freed memory
```

## 解放済みメモリの書き込み (FMW)

例:

```
int *p = (int*) malloc(sizeof(int));
free(p);
. . . // Nothing assigned to p in between
*p = 1; // Writing to freed memory
```

## 解放済み Realloc パラメータ (FRP)

例:

```
#include <stdlib.h>

int main() {
    int *p = (int *) malloc(sizeof(int));
    free(0);
    int *q = (int*) realloc(p,sizeof(it[2])); //Freed pointer passed to realloc
}
```

## 無効なメモリの読み取り (IMR)

**考えられる原因:** ハーフワード境界、ワード境界、またはダブルワード境界に整列していないアドレスから、それぞれ 2、4、または 8 バイトを読み取っています。

例:

```
#include <stdlib.h>
int main()
```

```
{
  int *p = 0;
  int i = *p;  // Read from invalid memory address
}
```

## 無効なメモリーの書き込み (IMW)

**考えられる原因:** ハーフワード境界、ワード境界、またはダブルワード境界に整列していないアドレスに、それぞれ 2、4、または 8 バイトを書き込んでいます。テキストアドレスに書き込んでいるか、読み取り専用データセクション (.rodata) に書き込んでいるか、mmap によって読み取り専用になっているページに書き込んでいます。

例:

```
int main()
{
  int *p = 0;
  *p = 1;  // Write to invalid memory address
}
```

## メモリーリーク

**考えられる原因:** メモリーが割り当てられるが、関数の終了またはエスケープの前に解放されていません。

例:

```
int foo()
{
  int *p = (int*) malloc(sizeof(int));
  if (x) {
    p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
  }
}  // The 2nd malloc leaked here
```

## 送り側と受け側の重複 (OLP)

**考えられる原因:** 正しくないソース、宛先、または長さが指定されました。ソースと宛先が重複している場合、プログラムの動作は不定になります。

例:

```
#include <stdlib.h>
#include <string.h>
int main() {
```

```
char *s=(char *) malloc(15);
memset(s, 'x', 15);
memcpy(s, s+5, 10);
return 0;
}
```

## 部分的に初期化された読み取り (PIR)

例:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *p = (int*) malloc(sizeof(int));
    *((char*)p) = 'c';
    printf("**(p = %d\n",*(p+1)); // Accessing partially initialized data
}
```

## スタック境界を越える読み取り (SBR)

考えられる原因: ローカル配列の末尾よりあと、または先頭より前を読み取っています。

例:

```
#include <stdio.h>

int main() {
    int a[2] = {0, 1};
    printf("a[-10]=%d\n",a[-10]); // Read is beyond stack frame bounds

    return 0;
}
```

## スタック境界を越える書き込み (SBW)

考えられる原因: ローカル配列の末尾よりあと、または先頭より前に書き込んでいます。

例:

```
#include <stdio.h>

int main() {
    int a[2] = {0, 1};
    a[-10] = 2; // Write is beyond stack frame bounds
}
```

```
    return 0;  
}
```

## 割り当てられていないメモリの読み取り (UAR)

**考えられる原因:** ストレイポインタ (不正な値を持つポインタ)、ヒープブロック境界のオーバーフロー、すでに解放されたヒープブロックへのアクセス。

例:

```
#include <stdio.h>  
#include <stdlib.h>  
int main()  
{  
    int *p = (int*) malloc(sizeof(int));  
    printf("(p+1) = %d\n",*(p+1)); // Reading from unallocated memory  
}
```

## 割り当てられていないメモリの書き込み (UAW)

**考えられる原因:** ストレイポインタ (不正な値を持つポインタ)、ヒープブロック境界のオーバーフロー、すでに解放されたヒープブロックへのアクセス。

例:

```
#include <stdio.h>  
#include <stdlib.h>  
int main()  
{  
    int *p = (int*) malloc(sizeof(int));  
    *(p+1) = 1; // Writing to unallocated memory  
}
```

## 初期化されていないメモリの読み取り (UMR)

**考えられる原因:** 初期化されていないローカルデータまたはヒープデータの読み取り。

例:

```
#include <stdio.h>  
#include <stdlib.h>  
int main()  
{  
    int *p = (int*) malloc(sizeof(int));  
    printf("**p = %d\n",*p); // Accessing uninitialized data  
}
```

## 動的メモリアクセスの警告

動的メモリアクセス検査では、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 uninitialized memory read)

このセクションでは、警告の考えられる原因と、警告が発生する可能性があるコードの例を説明します。

### サイズ 0 の割り当て (AZS)

例:

```
#include <stdlib>
int main()
{
    int *p = malloc(); // Allocating zero size memory block
}
```

### メモリーリーク (MLK)

考えられる原因: メモリーが割り当てられるが、関数の終了またはエスケープの前に解放されていません。

例:

```
int foo()
{
    int *p = (int*) malloc(sizeof(int));
    if (x) {
        p = (int *) malloc(5*sizeof(int)); // will cause a leak of the 1st malloc
    }
} // The 2nd malloc leaked here
```

### 投機的なメモリーの読み取り (SMR)

例:

```
int i;
if (foo(&i) != 0) /* foo returns nonzero if it has initialized i */
    printf("5d\n", i);
```

コンパイラは、前述のソースに対して次の同等のコードを生成する可能性があります。

```
int i;
int t1, t2;
t1 = foo(&i);
t2 = i; /* value in i is loaded. So even if t1 is 0, we have uninitialized read due to
speculative load */
if (t1 != 0)
    printf("%d\n", t2);
```

# 索引

---

## あ

- オプション, 22
  - 結果比較, 24
  - 結果保存, 23
  - データの種類, 22
  - テストスイート, 24
  - 表示, 22
  - フィルタリング, 22
  - ラベル付け, 23

## か

- コードアナライザ
  - 使用するための要件, 13
- コードアナライザのコマンド行インタフェース
  - 機能, 13
- コードアナライザ GUI
  - 起動, 20
  - 機能, 13
  - クイックスタート, 15
- コードカバレッジ検査, 12
- コードカバレッジの問題, 35

## さ

- 最適化、メモリーエラーに対する影響, 18
- 静的コード検査, 12
- 静的コードの問題, 35

## た

- 中核となる問題, 11
- データの収集
  - `binary-name.analyze` ディレクトリ, 17

- コードカバレッジ, 19
- 静的エラー, 17
  - 制限, 17
- 動的メモリーアクセスエラー, 18
- テストスイート
  - オプション, 24
- 動的メモリーアクセス検査, 12
- 動的メモリーアクセスの問題
  - エラー, 40
  - 警告, 47

## は

- プログラムへの計測機構の組み込み
  - Discover による, 18
  - discover による, 19
  - Uncover による, 19, 20

## や

- 要件
  - Discover によるプログラムへの計測機構の組み込み, 18
  - uncover によるプログラムへの計測機構の組み込み, 19
  - コードアナライザの使用, 13

## B

- `binary_name.analyze` ディレクトリ
  - dynamic サブディレクトリ, 19
- `binary-name.analyze` ディレクトリ, 17, 20
  - coverage サブディレクトリ, 20
  - static サブディレクトリ, 17

## C

- codean, 21
  - オプション, 22, 22, 22, 22, 23, 23, 24, 24
  - 機能, 13
  - テストスイート, 29
  - 問題のラベル付け, 26
  - ワークフローの例, 25
- codean コマンド, 13
- codean コマンド行ツール, 21

## G

- g コンパイラオプション, 18, 19

## X

- xanalyze=code コンパイラオプション, 12, 17, 17
  - Linux, 17
- xprewise=yes コンパイラオプション, 12, 17, 17
  - Linux, 17