Oracle® Solaris Studio 12.3 代码分析器教程

2011 年 12 月

ORACLE®

简介

Oracle Solaris Studio 代码分析器是一套集成的工具，用于帮助 C 和 C++ 应用程序开发者针对 Oracle Solaris 开发出既安全又强健的高质量软件。

代码分析器包含以下三种分析类型：

- 编译过程中的静态代码检查
- 动态内存访问检查
- 代码覆盖分析

静态代码检查可在编译期间检测代码中出现的常见编程错误。新编译器选项利用 Oracle Solaris Studio 编译器既广泛又成熟的控制和数据流分析框架来分析应用程序是否存在潜在的编程和安全缺陷。

在运行您的应用程序时，代码分析器使用 Discover（Oracle Solaris Studio 中的内存错误搜索工具）所收集的动态内存数据来查找与内存相关的错误。此外，它使用 Uncover（Studio 中的代码覆盖工具）所收集的数据来测量代码覆盖。

除了允许您访问各种分析类型以外，代码分析器还集成了静态代码检查和动态内存访问检查，用于向在代码中发现的错误添加置信度级别。通过将静态代码检查与动态内存访问分析和代码覆盖分析结合使用，将可以在应用程序中找到其他错误检测工具通过自身无法找到的许多重要错误。

获取样例应用程序

本教程使用样例程序说明如何使用 Oracle Solaris Studio 编译器、Discover 内存错误搜索工具、Uncover 代码覆盖工具以及代码分析器 GUI 来查找和更正常见的编程错误、动态内存访问错误以及代码覆盖问题。

可从 Oracle Solaris Studio 12.3 Sample Applications（Oracle Solaris Studio 12.3 样例应用程序）Web 页面上的样例应用程序 zip 文件中获取该样例程序的源代码，网址为 `http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-samples-1408618.html`。如果您尚未下载样例应用程序 zip 文件，请下载该文件并将其解压缩到您选择的目录中。

`sample` 应用程序位于 `SolarisStudioSampleApplications` 目录的 `CodeAnalyzer` 子目录中。

`sample` 目录包含以下源代码文件：

```
main.c
previse_1.c
previse_all.c
sample_1.c
sample_2.c
sample_3.c
```

收集和显示数据

您可以使用代码分析器工具收集其中一种、两种或全部三种类型的数据。

收集和显示静态错误数据

使用 `-xanalyze=code` 编译器选项生成二进制文件时，编译器会自动提取静态错误并将数据放入源代码所在目录中的 *binary_name*`.analyze` 目录的 `static` 子目录中。有关编译器找到的静态错误的类型列表，请参见第 13 页中的 “静态代码问题”。

1. 在您的 `sample` 目录中，通过键入以下内容来生成应用程序：

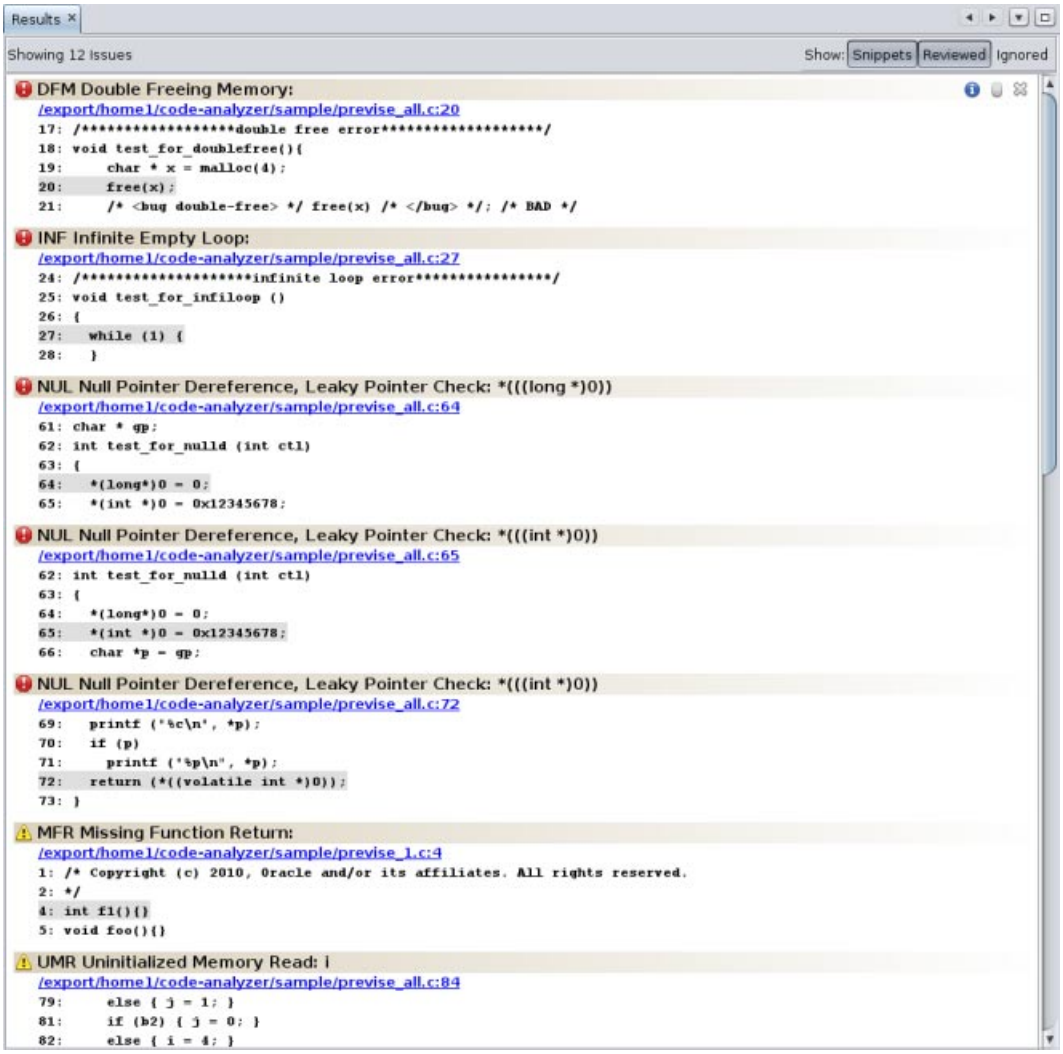
   ```
   cc -xanalyze=code *.c
   ```

   静态错误数据将被写入 `sample` 目录中 `a.out.analyze` 目录的 `static` 子目录中。

2. 打开代码分析器 GUI 以查看结果：

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

3. 此时将打开代码分析器 GUI，"Results"（结果）标签中会显示编译期间发现的静态代码问题。"Results"（结果）标签顶部的文本显示发现了十二个静态代码问题。



4. 对于每个问题，此标签显示问题类型、发现问题的源文件的路径名以及该文件中突出显示相关源代码行的代码片段。

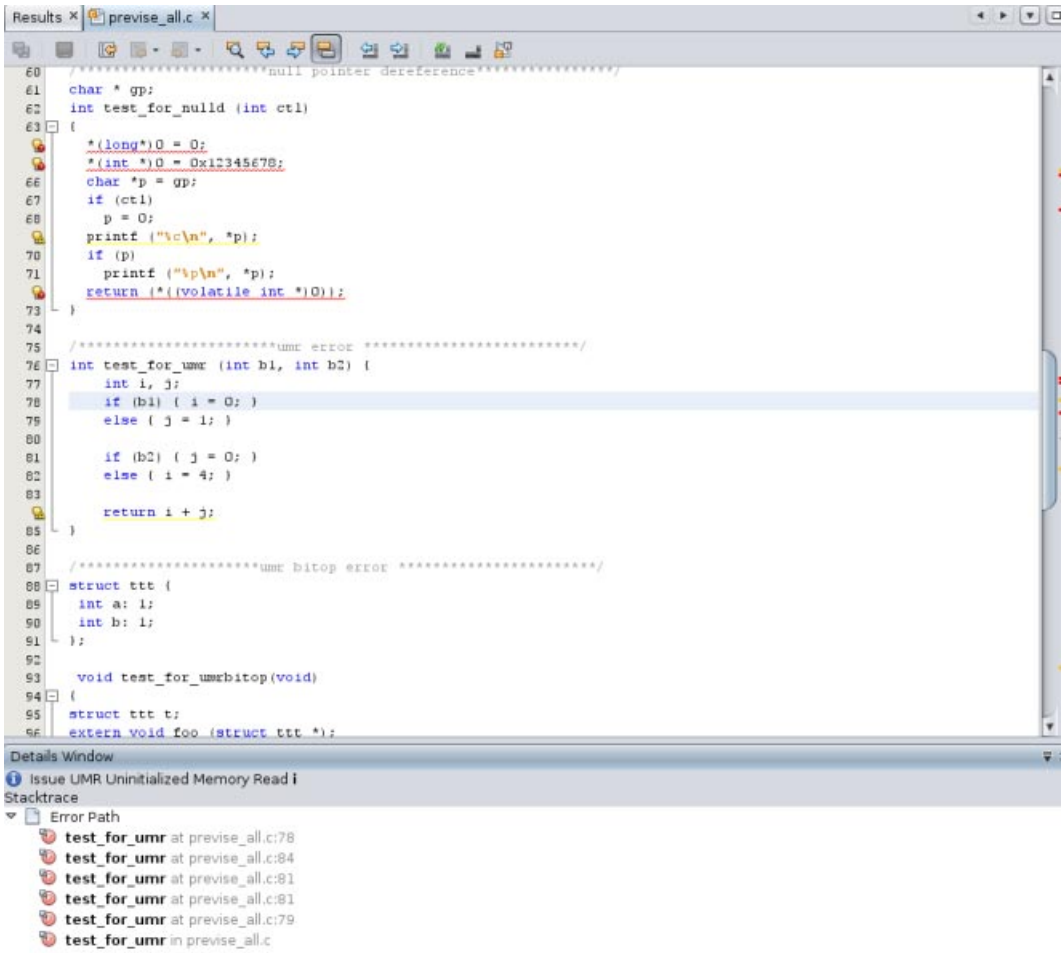
5. 要查看有关第一个问题（"Double Freeing Memory"（双重释放内存）错误）的更多信息，请单击错误图标 。此时将打开问题的堆栈跟踪，其中显示错误路径。

6. 请注意，打开堆栈跟踪后，问题右上角的图标从 更改为 ，表明您已查看此问题。

注 – 单击 "Results"（结果）标签顶部的 "Reviewed"（已查看）按钮 Reviewed 可隐藏已查看的问题。再次单击此按钮可取消隐藏问题。

---

7. 单击错误图标以关闭堆栈跟踪。
8. 现在，查看其中一个 "Unitialized Memory Read"（读取未初始化的内存）警告。单击警告图标 以打开堆栈跟踪。请注意，此问题的错误路径所包含的函数调用远远多于 "Double Freeing Memory"（双重释放内存）问题的错误路径所包含的函数调用。双击第一个函数调用。此时将打开源文件，并且突出显示该调用。错误路径显示在源代码下的 "Details Window"（详细信息窗口）中。



双击错误路径中的其余函数调用以沿着路径找到导致错误的代码。

9. 要查看有关 UMR 错误类型的更多信息，请单击问题描述左侧的 "Info"（信息）按钮 。联机帮助浏览器中显示了错误类型的描述，其中包括代码示例和可能的原因。
10. 关闭代码分析器 GUI。

收集和显示动态内存使用情况数据

无论是否收集了静态数据，您都可以编译、检测和运行应用程序来收集动态内存访问数据。有关通过使用 Discover 检测应用程序然后运行应用程序找到的动态内存访问错误的列表，请参见第 13 页中的 “动态内存访问问题”。

1. 在 `sample` 目录中，使用 `-g` 选项生成样例应用程序。此选项可生成调试信息，从而使代码分析器可以显示错误和警告的源代码和行号信息。

   ```
   cc -g *.c
   ```
2. 不能检测已检测过的二进制文件，因此请在收集覆盖数据时保存要使用的二进制文件的副本。

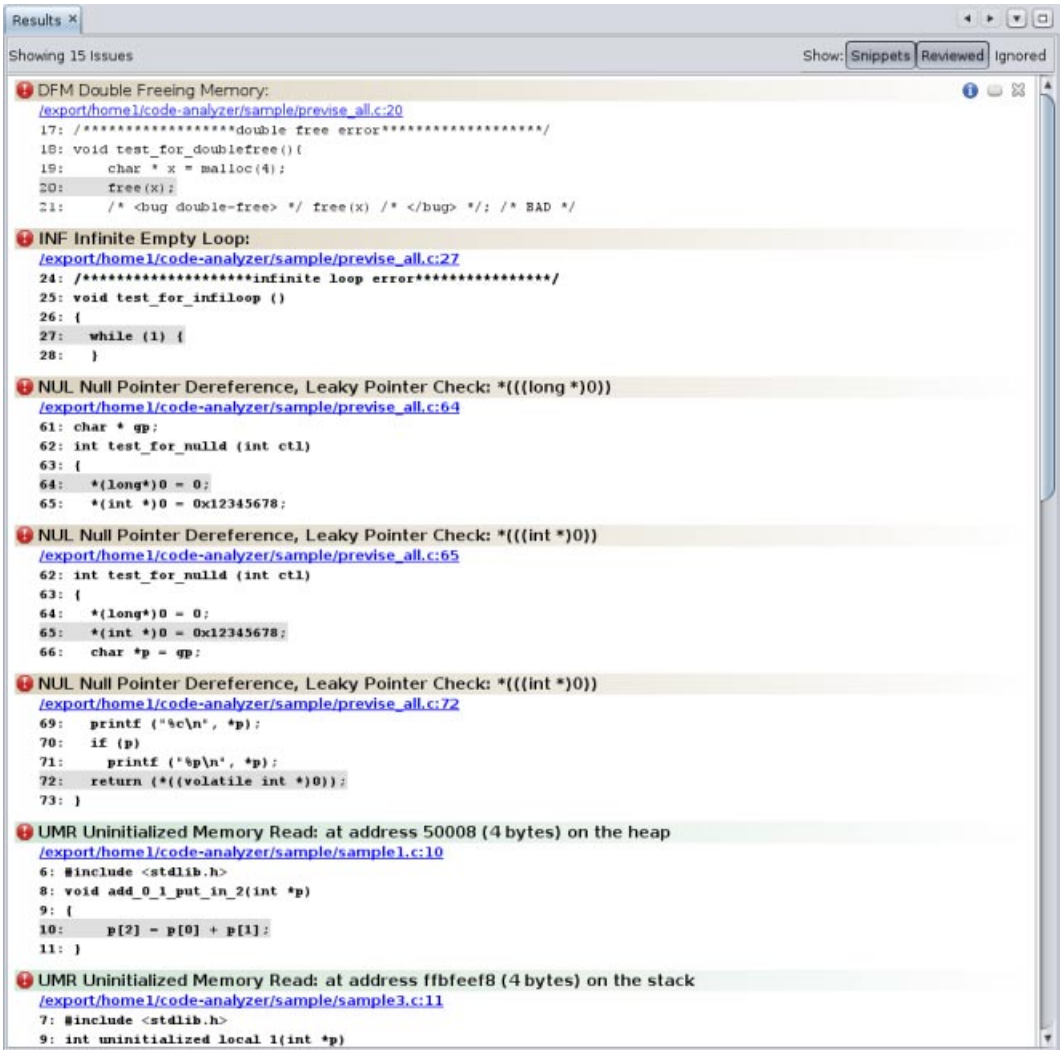
   ```
   cp a.out a.out.save
   ```
3. 使用 Discover 检测二进制文件：

   ```
   discover -a a.out
   ```
4. 运行检测过的二进制文件以收集动态内存访问数据。

   ```
   ./a.out
   ```

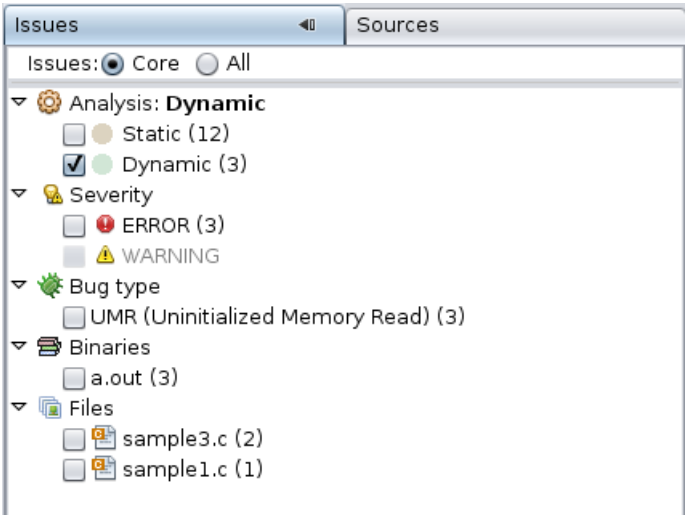
   动态内存访问错误数据将被写入 `sample` 目录中 `a.out.analyze` 目录的 `dynamic` 子目录中。
5. 打开代码分析器 GUI 以查看结果：

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



6. "Results"（结果）标签现在同时显示静态问题和动态内存问题。问题描述的背景颜色可以指明问题是静态代码问题（棕褐色）还是动态内存访问问题（淡绿色）。

   要过滤结果以便仅显示动态内存问题，请在 "Issues"（问题）标签中选中 "Dynamic"（动态）复选框。

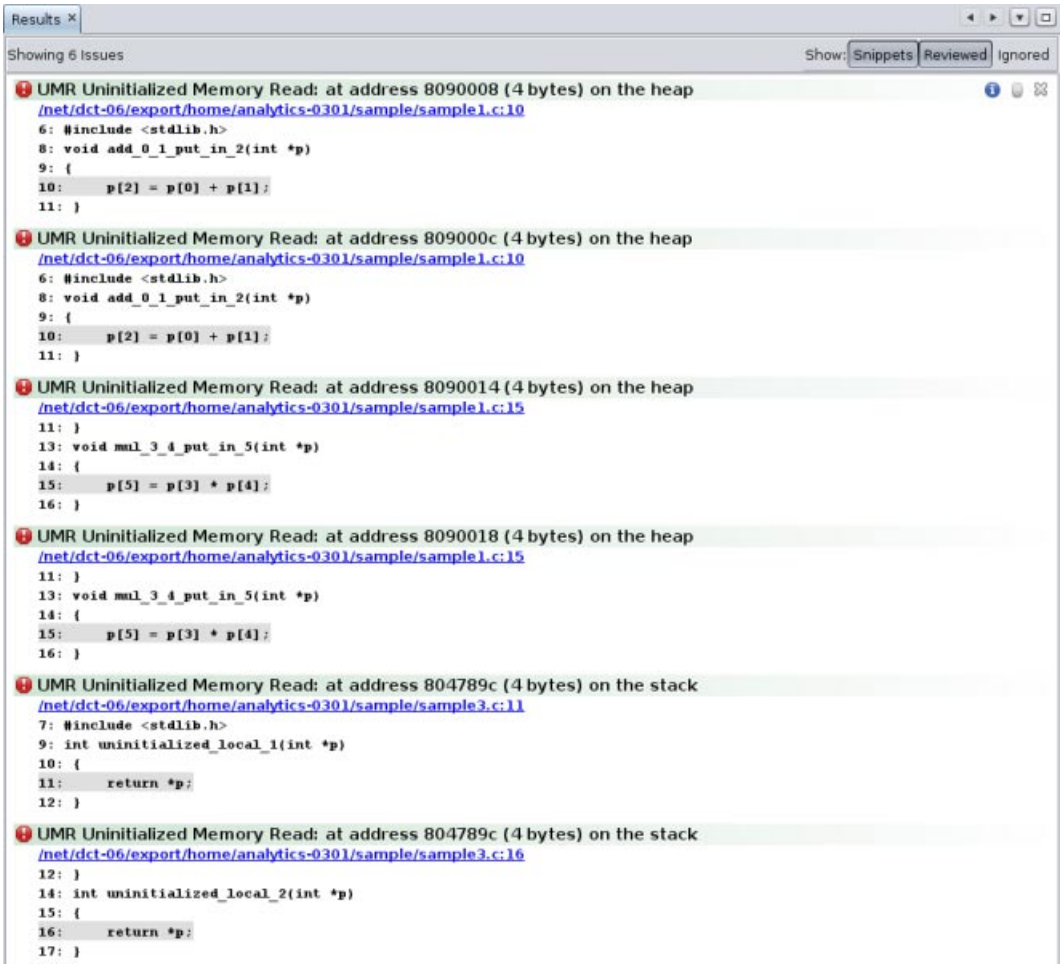


现在，"Results"（结果）标签仅显示三个核心动态内存问题。

注 – 修复核心问题后可能会消除其他问题。一个核心问题通常会伴有 "All"（所有）视图中列出的多个问题，例如，因为这些问题具有一个通用分配点或出现在同一函数中的同一数据地址上。

---

7. 要查看所有动态内存问题，请选择 "Issues"（问题）标签顶部的 "All"（所有）单选按钮。现在，"Results"（结果）标签显示六个动态内存问题。



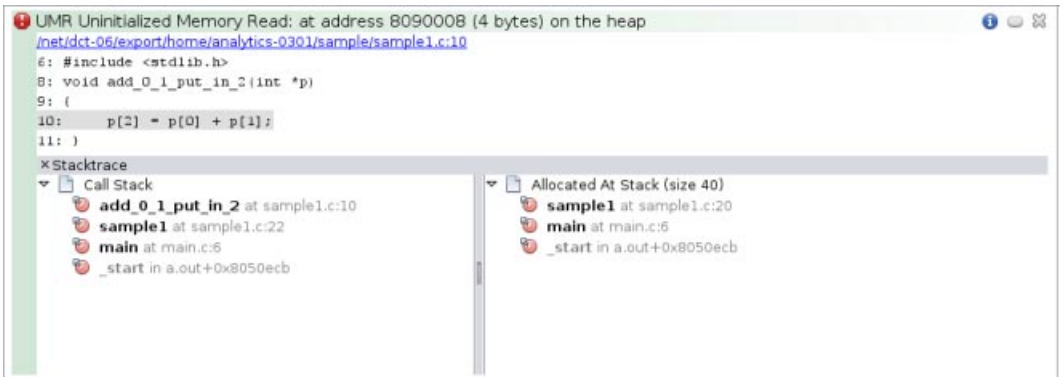
查看已添加到显示屏幕中的三个问题，并了解它们如何与核心问题相关联。看起来好像在基于相应成因修复显示屏幕中的第一个问题后可能还会消除第二个和第三个问题。

要在调查第一个问题的成因时隐藏其他三个动态内存访问问题，请单击每个问题对应的 "Ignore"（忽略）按钮 。

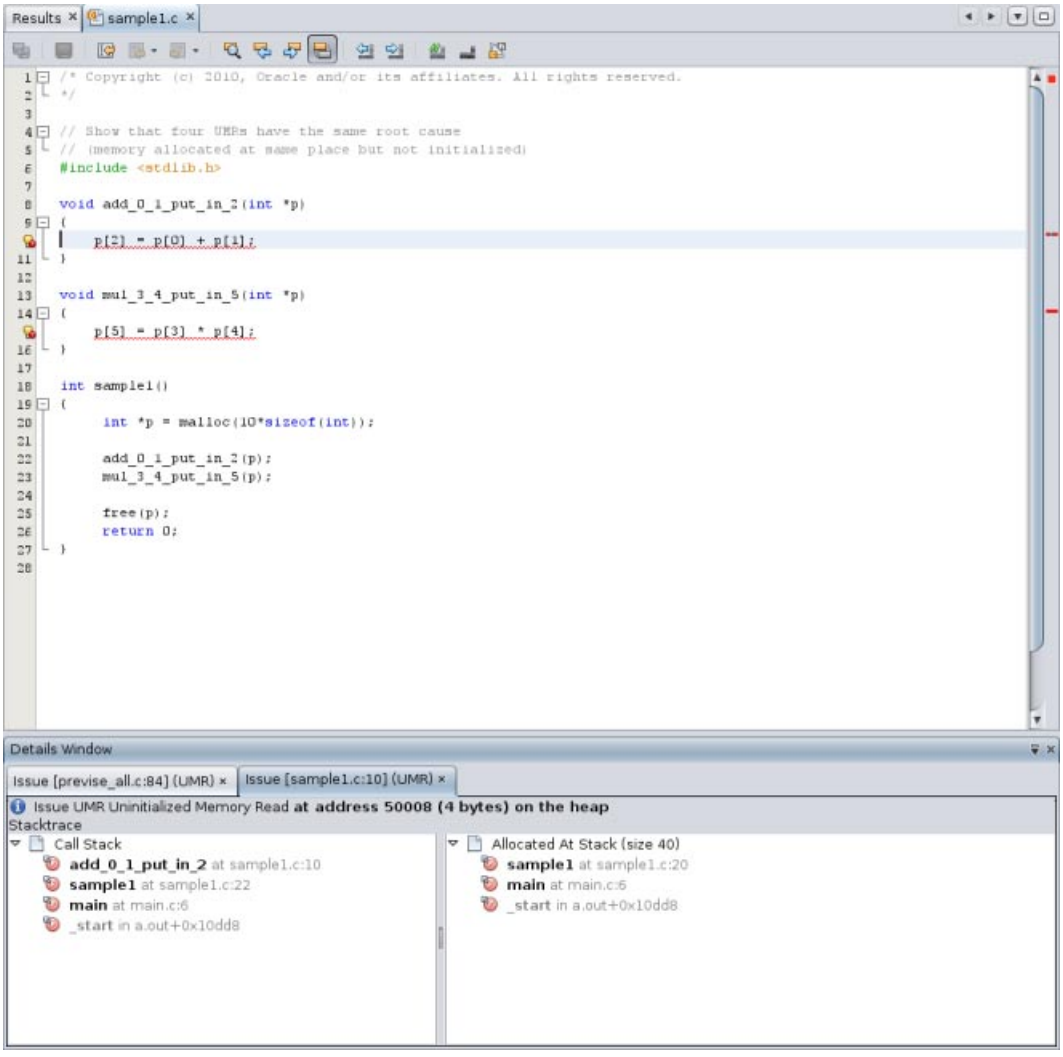
注 – 可以在以后通过单击 "Results"（结果）标签顶部的 "Ignored"（已忽略）按钮 Ignored 来重新显示已关闭的问题。

---

8. 通过单击错误图标 以显示堆栈跟踪来调查第一个问题。对于此问题，堆栈跟踪包括 "Call Stack"（调用堆栈）和 "Allocated At Stack"（堆栈上的已分配空间）。



双击堆栈中的函数调用以查看源文件中的关联行。打开源文件时，堆栈跟踪显示在该文件下的 "Details Window"（详细信息窗口）中。



9. 关闭代码分析器 GUI。

收集和显示代码覆盖数据

无论是否收集了静态数据或动态内存访问数据，您都可以编译、检测和运行应用程序来收集代码覆盖数据。

1. 由于在收集动态内存错误数据之前使用 `-g` 选项生成了应用程序，并在检测该应用程序之前保存了二进制文件的副本，因此，您可以复制保存的二进制文件进行检测，以便收集覆盖数据。

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

2. 使用 Uncover 检测二进制文件：

   ```
   uncover a.out
   ```

3. 运行检测过的二进制文件以收集代码覆盖数据。

   ```
   ./a.out
   ```

   代码覆盖数据将被写入 sample 目录的 `a.out.uc` 目录中。

4. 对 `a.out.uc` 目录运行 Uncover。

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

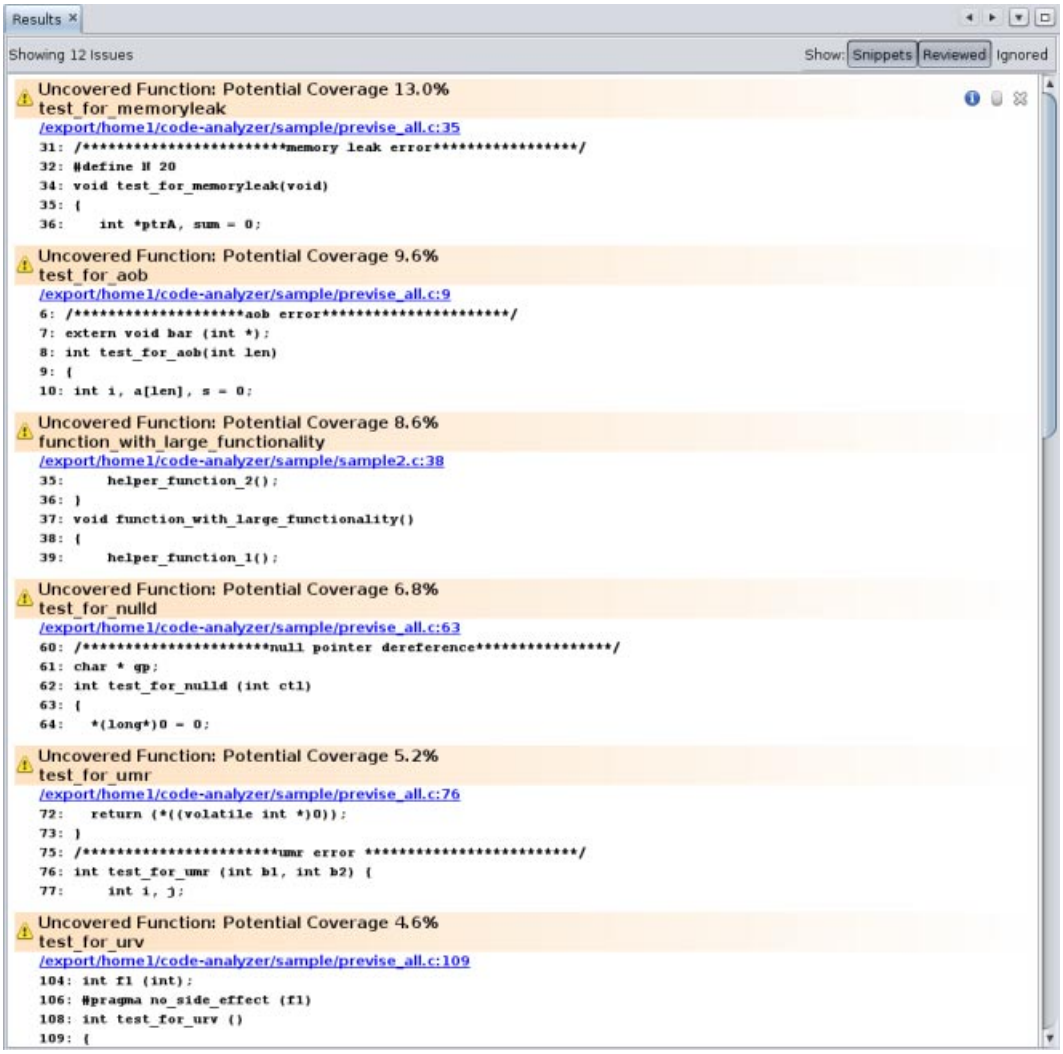
   代码覆盖数据将被写入 sample 目录中 `a.out.analyze` 目录的 uncover 子目录中。

5. 打开代码分析器 GUI 以查看结果：

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

6. "Results"（结果）标签现在同时显示静态问题、动态内存问题和代码覆盖问题。要过滤结果以便仅显示代码覆盖问题，请在 "Issues"（问题）标签中选中 "Coverage"（覆盖）复选框。

   现在，"Results"（结果）标签仅显示十二个代码覆盖问题。每个问题的描述中均包含一个潜在的覆盖百分比，此百分比是在增加覆盖相关函数的测试时将加入该应用程序总覆盖范围的覆盖百分比。

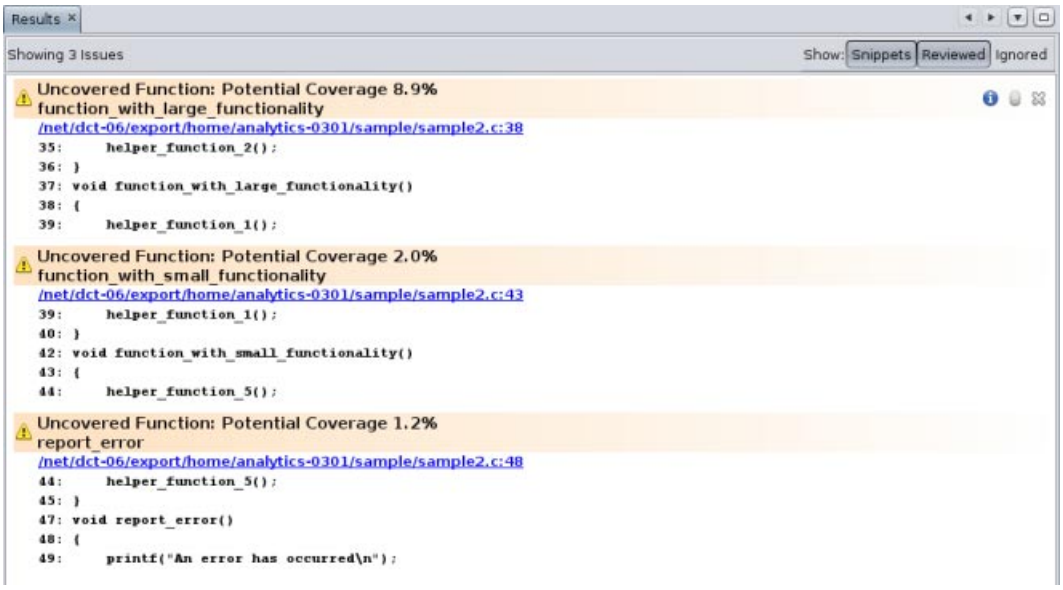


注 – 要在无需上下滚动的情况下查看所有问题，请单击 "Results"（结果）标签顶部的 "Snippets"（代码片段）按钮  以隐藏代码片段。

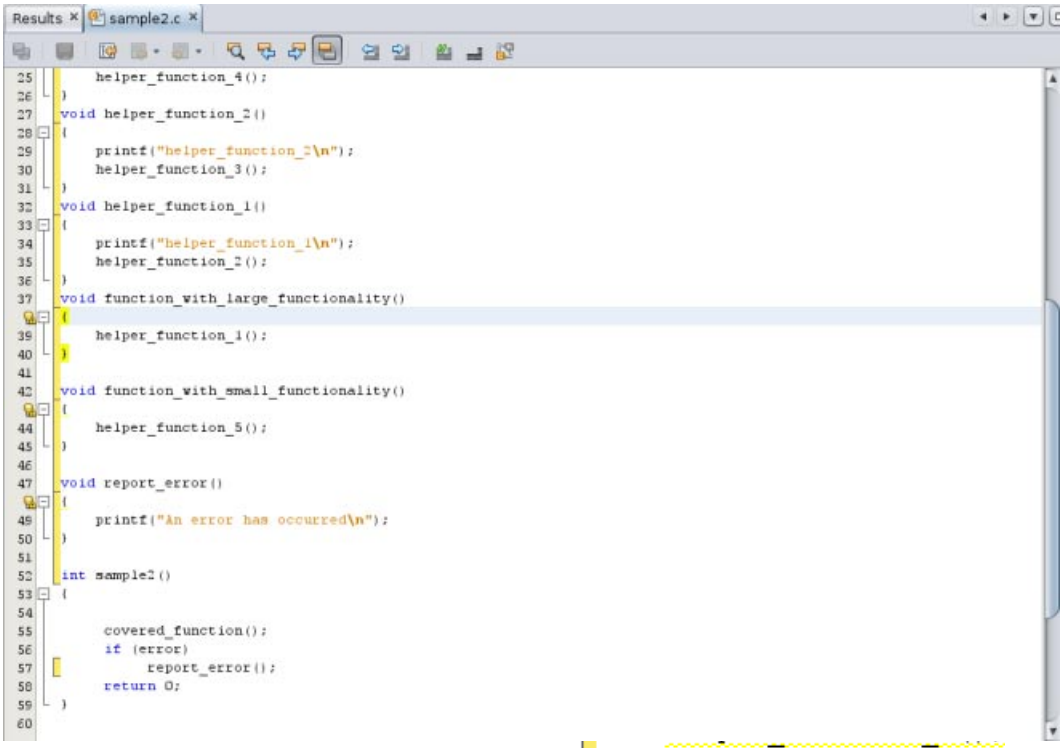
---

7. 在 "Issues"（问题）标签中，请注意，`previse_all.c` 源文件中有九个覆盖问题，`sample2.c` 源文件中有三个覆盖问题，而 `previse_1.c` 源文件中有一个覆盖问题。要进一步过滤结果以便仅显示 `sample2.c` 文件中的问题，请在 "Issues"（问题）标签中选中与该文件对应的复选框。

   "Results"（结果）标签现在仅显示在 `sample2.c` 中找到的三个代码覆盖问题。



8. 单击其中一个问题的源文件路径链接以打开源文件。在源文件中向下滚动，直到在左边界看到警告图标。



将使用黄色的方括号来标记未覆盖的代码，例如  。在文件中找到的覆盖问题标记有警告图标 

利用代码分析器找到的问题来改进代码

通过修复代码分析器找到的核心问题，应该能够消除在代码中找到的其他问题并大大提高代码的质量和稳定性。

通过执行静态错误检查，可以在应用程序中找到存在风险的代码。但静态错误检查可能会产生误报。动态检查可帮助检验并消除这些错误，使您能够更准确地了解代码中存在的问题。而代码覆盖检查可帮助改进您的动态测试套件。

代码分析器整合了这三种检查类型的结果，使您能够在一个工具中对所有代码进行最准确的分析。

代码分析器工具找到的潜在错误

编译器、Discover 和 Uncover 可在您的代码中查找静态代码问题、动态内存访问问题以及覆盖问题。以下各节列出了可由这些工具找到并由代码分析器分析的特定错误类型。

静态代码问题

静态代码检查可查找以下类型的错误：

- ABR：数组越界读
- ABW：数组越界写
- DFM：双重释放内存
- ECV：显式强制类型转换违规
- FMR：读取释放的内存
- FMW：写入释放的内存
- FOU：`PM_OUT` 在定义前使用
- INF：无限空循环
- 内存泄漏
- MF：返回缺少的函数
- MRC：缺少 malloc 返回值检查
- NFR：返回未初始化的函数
- NUL：NULL 指针解除引用，泄漏指针检查
- RFM：返回释放的内存
- UMR：未初始化的内存读取，未初始化的内存读取位操作
- URV：未使用的返回值
- VES：超出范围的局部变量使用

动态内存访问问题

动态内存访问检查可查找以下类型的错误：

- ABR：数组越界读
- ABW：数组越界写
- BFM：释放错误的内存块
- BRP：错误的重新分配地址参数
- CGB：损坏的保护块
- DFM：双重释放内存
- FMR：读取释放的内存
- FMW：写入释放的内存
- IMR：无效的内存读取
- IMW：无效的内存写入
- 内存泄漏
- OLP：重叠源和目标
- PIR：部分初始化的读取
- SBR：堆栈越界读
- SBW：堆栈越界写
- UAR：读取未分配的内存

- UAW：写入未分配的内存
- UMR：读取未初始化的内存

动态内存访问检查可查找以下类型的警告：

- AZS：分配零大小
- 内存泄漏
- SMR：推测性未初始化内存读取

代码覆盖问题

代码覆盖检查可确定哪些函数未被覆盖。在结果中，发现的代码覆盖问题会标记为 "Uncovered Function"（未覆盖的函数），并且带有潜在覆盖百分比，此百分比是指在添加覆盖相关函数的测试时将添加到应用程序总覆盖的覆盖百分比。

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

ORACLE