# Oracle® Solaris Studio 12.3 Code Analyzer Tutorial

December 2011

- "Introduction" on page 2
- "Getting the Sample Application" on page 2
- "Collecting and Displaying Data" on page 3
- "Using the Issues Found by the Code Analyzer to Improve Your Code" on page 13
- "Potential Errors Found by Code Analyzer Tools" on page 13


## Introduction

The Oracle Solaris Studio Code Analyzer is an integrated set of tools designed to help developers of C and C++ applications for Oracle Solaris produce secure, robust, and quality software.

The Code Analyzer includes three types of analysis:

- Static code checking as part of compilation
- Dynamic memory access checking
- Code coverage analysis

Static code checking detects common programming errors in your code during compilation. A new compiler option leverages the Oracle Solaris Studio compilers' extensive and well proven control and data flow analysis frameworks to analyze an application for potential programming and security flaws.

The Code Analyzer uses dynamic memory data collected by Discover, the memory error discovery tool in Oracle Solaris Studio, to find memory-related errors when you run your application. It uses data collected by Uncover, the code coverage tool in Studio, to measure code coverage.

In addition to giving you access to each individual type of analysis, the Code Analyzer integrates static code checking and dynamic memory access checking to add confidence levels to errors found in code. By using static code checking together with dynamic memory access analysis and code coverage analysis, you will be able to find many important errors in your applications that cannot be found by other error detection tools working by themselves.

## Getting the Sample Application

This tutorial uses a sample program to demonstrate how to use the Oracle Solaris Studio compilers, the Discover memory error discovery tool, the Uncover code coverage tool, and the Code Analyzer GUI to find and correct common programming errors, dynamic memory access errors, and code coverage issues.

The source code for the sample program is available in the sample applications zip file on the Oracle Solaris Studio 12.3 Sample Applications web page at `http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-samples-1408618.html`. If you have not already done so, download the sample applications zip file and unpack it in a directory of your choice.

The `sample` application is located in the `CodeAnalyzer` subdirectory of the `SolarisStudioSampleApplications` directory.

The `sample` directory contains the following source code files:

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

# Collecting and Displaying Data

You can use the Code Analyzer tools to collect one, two, or all three types of data.

## Collecting and Displaying Static Error Data

When you build a binary using the `-xanalyze=code` compiler option, the compiler automatically extracts static errors and puts the data in a `static` subdirectory in a *binary_name*`.analyze` directory in the same directory as the source code. For a list of the types of static errors found by the compiler, see "Static Code Issues" on page 13.

1. In your `sample` directory, build the application by typing:

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

   The static error data is written to the `static` subdirectory in an `a.out.analyze` directory in your `sample` directory.

2. Open the Code Analyzer GUI to view the results:

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

3. The Code Analyzer GUI opens and the Results tab displays the static code issues found during compilation. The text at the top of the Results tab tells you that twelve static code issues were found.

```
Results ×
Showing 12 Issues                                   Show: Snippets Reviewed Ignored
DFM Double Freeing Memory:
/export/home1/code-analyzer/sample/previse_all.c:20
17: /******************double free error*******************/
18: void test_for_doublefree(){
19:     char * x = malloc(4);
20:     free(x);
21:     /* <bug double-free> */ free(x) /* </bug> */; /* BAD */
INF Infinite Empty Loop:
/export/home1/code-analyzer/sample/previse_all.c:27
24: /********************infinite loop error****************/
25: void test_for_infiloop ()
26: {
27:   while (1) {
28:   }
NUL Null Pointer Dereference, Leaky Pointer Check: *(((long *)0))
/export/home1/code-analyzer/sample/previse_all.c:64
61: char * gp;
62: int test_for_nulld (int ctl)
63: {
64:   *(long*)0 = 0;
65:   *(int *)0 = 0x12345678;
NUL Null Pointer Dereference, Leaky Pointer Check: *(((int *)0))
/export/home1/code-analyzer/sample/previse_all.c:65
62: int test_for_nulld (int ctl)
63: {
64:   *(long*)0 = 0;
65:   *(int *)0 = 0x12345678;
66:   char *p = gp;
NUL Null Pointer Dereference, Leaky Pointer Check: *(((int *)0))
/export/home1/code-analyzer/sample/previse_all.c:72
69:   printf ("%c\n", *p);
70:   if (p)
71:     printf ("%p\n", *p);
72:   return (*((volatile int *)0));
73: }
MFR Missing Function Return:
/export/home1/code-analyzer/sample/previse_1.c:4
1: /* Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
2: */
4: int f1(){}
5: void foo(){}
UMR Uninitialized Memory Read: i
/export/home1/code-analyzer/sample/previse_all.c:84
79:     else { j = 1; }
81:     if (b2) { j = 0; }
82:     else { i = 4; }
```

4. For each issue, the tab displays the issue type, the path name of the source file in which the issue was found, and a code snippet from that file with the relevant source line highlighted.
5. To see more information about the first issue, a Double Freeing Memory error, click the error icon . The stack trace for the issue opens displaying the error path.

6. Notice that when you opened the stack trace, the icon in the upper right corner of the issue changed from

to to indicate that you have reviewed the issue.

---

**Note –** You can hide the issues you have reviewed by clicking the Hide Reviewed Issues button

at the top of the Results tab. Clicking the button again unhides the issues.

---

7. Click the error icon to close the stack trace.
8. Now look at one of the Unitialized Memory Read warnings. Click the warning icon to open the stack trace. Notice that the error path for this issue contains many more function calls than the one for the Double Freeing Memory issue. Double click on the first function call. The source file opens with that call highlighted. The error path is displayed in a Details Window below the source code.

```
Results × | previse_all.c ×

60   /***********************null pointer dereference****************/
61   char * gp;
62   int test_for_nulld (int ctl)
63   {
       *(long*)0 = 0;
       *(int *)0 = 0x12345678;
66     char *p = gp;
67     if (ctl)
68       p = 0;
       printf ("%c\n", *p);
70     if (p)
71       printf ("%p\n", *p);
       return (*((volatile int *)0));
73   }
74
75   /***********************umr error ***********************/
76   int test_for_umr (int b1, int b2) {
77       int i, j;
78       if (b1) { i = 0; }
79       else { j = 1; }
80
81       if (b2) { j = 0; }
82       else { i = 4; }
83
         return i + j;
85   }
86
87   /*********************umr bitop error ***********************/
88   struct ttt {
89     int a: 1;
90     int b: 1;
91   };
92
93    void test_for_umrbitop(void)
94   {
95   struct ttt t;
96   extern void foo (struct ttt *);

Details Window
Issue UMR Uninitialized Memory Read
Stacktrace
Error Path
  test_for_umr at previse_all.c:78
  test_for_umr at previse_all.c:84
  test_for_umr at previse_all.c:81
  test_for_umr at previse_all.c:81
  test_for_umr at previse_all.c:79
  test_for_umr in previse_all.c
```

Double click the rest of the function calls in the error path to follow the path through the code that leads to the error.

9. To see more information about the UMR error type, click the Info button to the left of the issue description. A description of the error type, including a code example and possible causes, is displayed in the online help browser.
10. Close the Code Analyzer GUI.

## Collecting and Displaying Dynamic Memory Usage Data

Whether or not you have collected static data, you can compile, instrument, and run your application to collect dynamic memory access data. For a list of the dynamic memory access errors found by instrumenting your application with Discover and then running it, see "Dynamic Memory Access Issues" on page 13.

1. In your `sample` directory, build the sample application with the `-g` option. This option generates debug information that allows the Code Analyzer to display source code and line number information for errors and warnings.

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

2. You cannot instrument a binary that is already instrumented, so save a copy of the binary to use when you collect coverage data.

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

3. Instrument the binary with Discover:

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

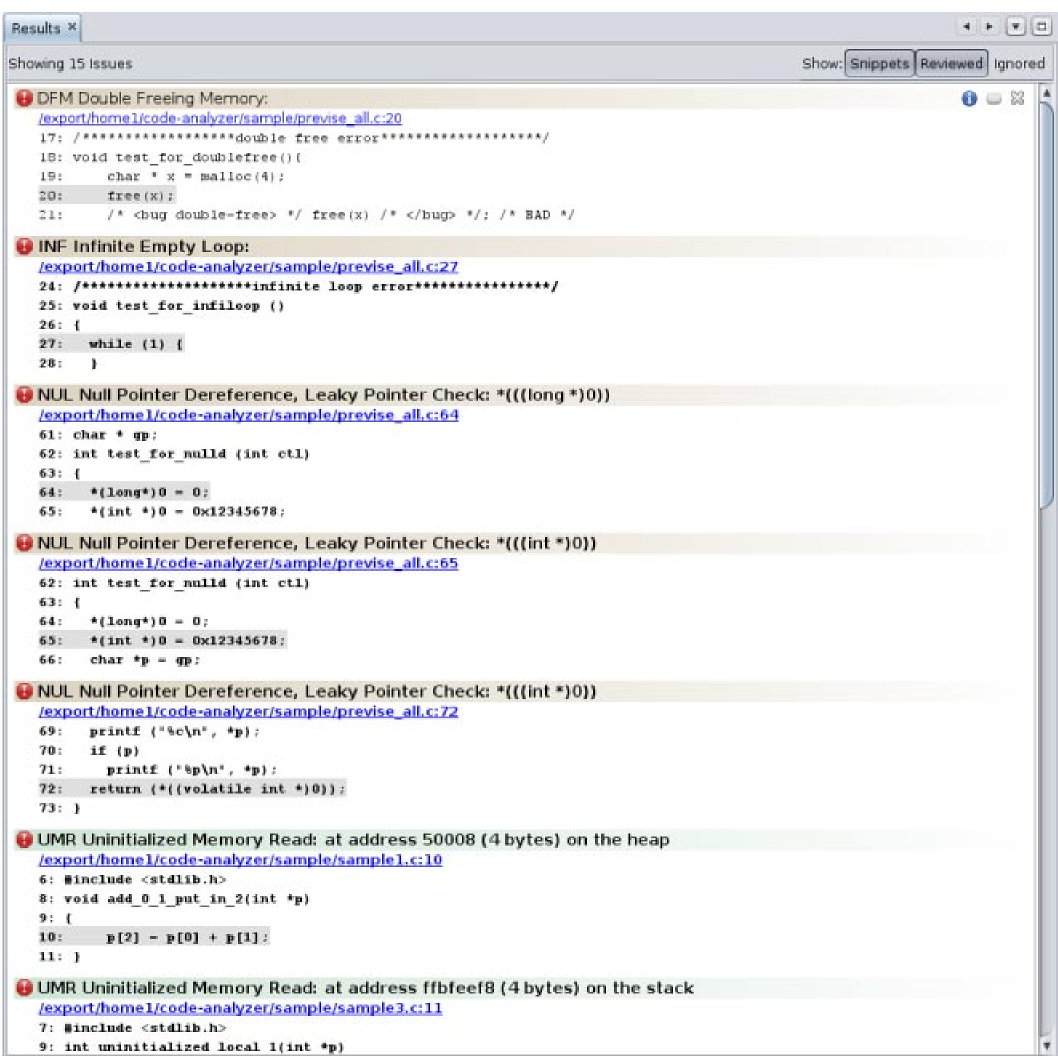
4. Run the instrumented binary to collect the dynamic memory access data.

   ```
   ./a.out
   ```

The dynamic memory access error data is written to the `dynamic` subdirectory in the `a.out.analyze` directory in your `sample` directory.

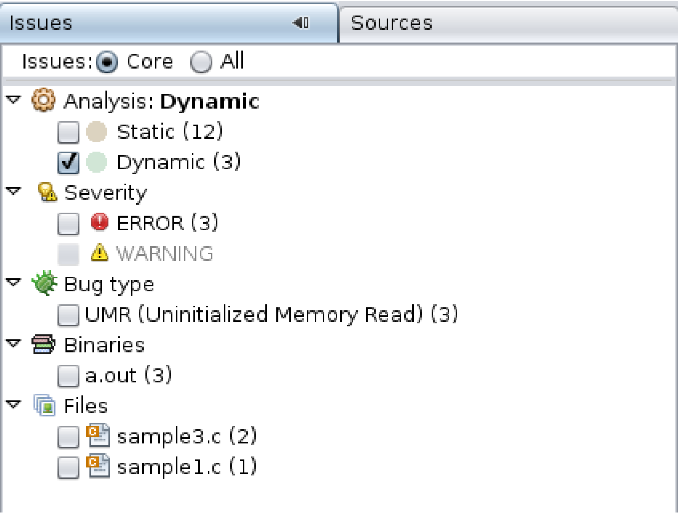
5. Open the Code Analyzer GUI to view the results:

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




6. The Results tab now shows both static issues and dynamic memory issues. The background color behind an issue description indicates a static code issue (tan) or a dynamic memory access issue (pale green).

   To filter the results and show just the dynamic memory issues, select the Dynamic checkbox on the Issues tab.



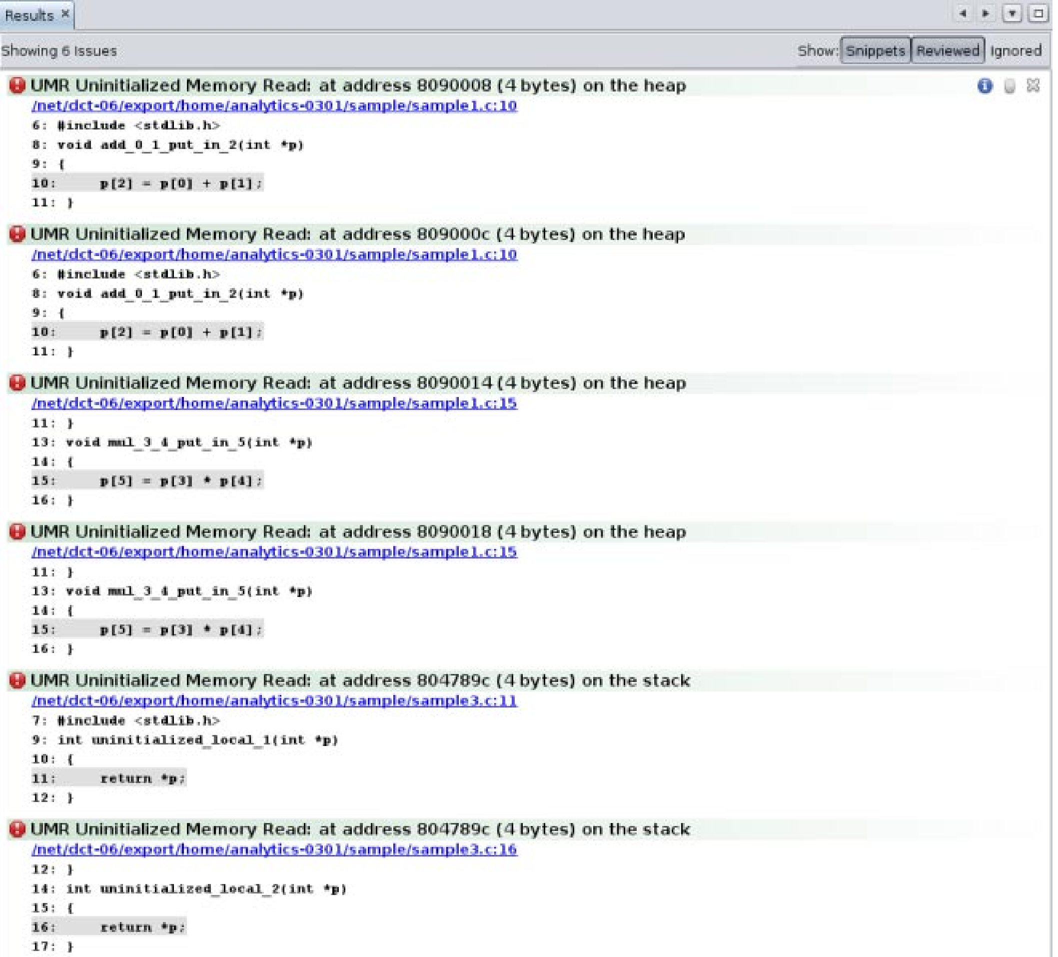

Now the Results tab shows just the three core dynamic memory issues.

---

**Note –** Core issues are the issues that, when fixed, are likely to eliminate the other issues. A core issue usually combines several of the issues listed on the All view because, for example, those issues have a common allocation point, or they occur at the same data address in the same function.

---

7. To see all of the dynamic memory issues, select the All radio button at the top of the Issues tab. Now the Results tab displays six dynamic memory issues.




Look at the three issues that were added to the display and see how they are related to the core issues. It looks as though fixing the cause of the first issue in the display is likely to also eliminate the second and third issues.

To hide the other three dynamic memory access issues while you investigate these first one, click the Ignore button for each of the issues.

---

**Note** – You can later redisplay the closed issues by clicking the Show Ignored Issues button

at the top of the Results tab.

---

8. Investigate the first issue by clicking the error icon to display the stack trace. For this issue, the stack trace includes the Call Stack and the Allocated At Stack.

UMR Uninitialized Memory Read: at address 8090008 (4 bytes) on the heap
/net/dct-06/export/home/analytics-0301/sample/sample1.c:10

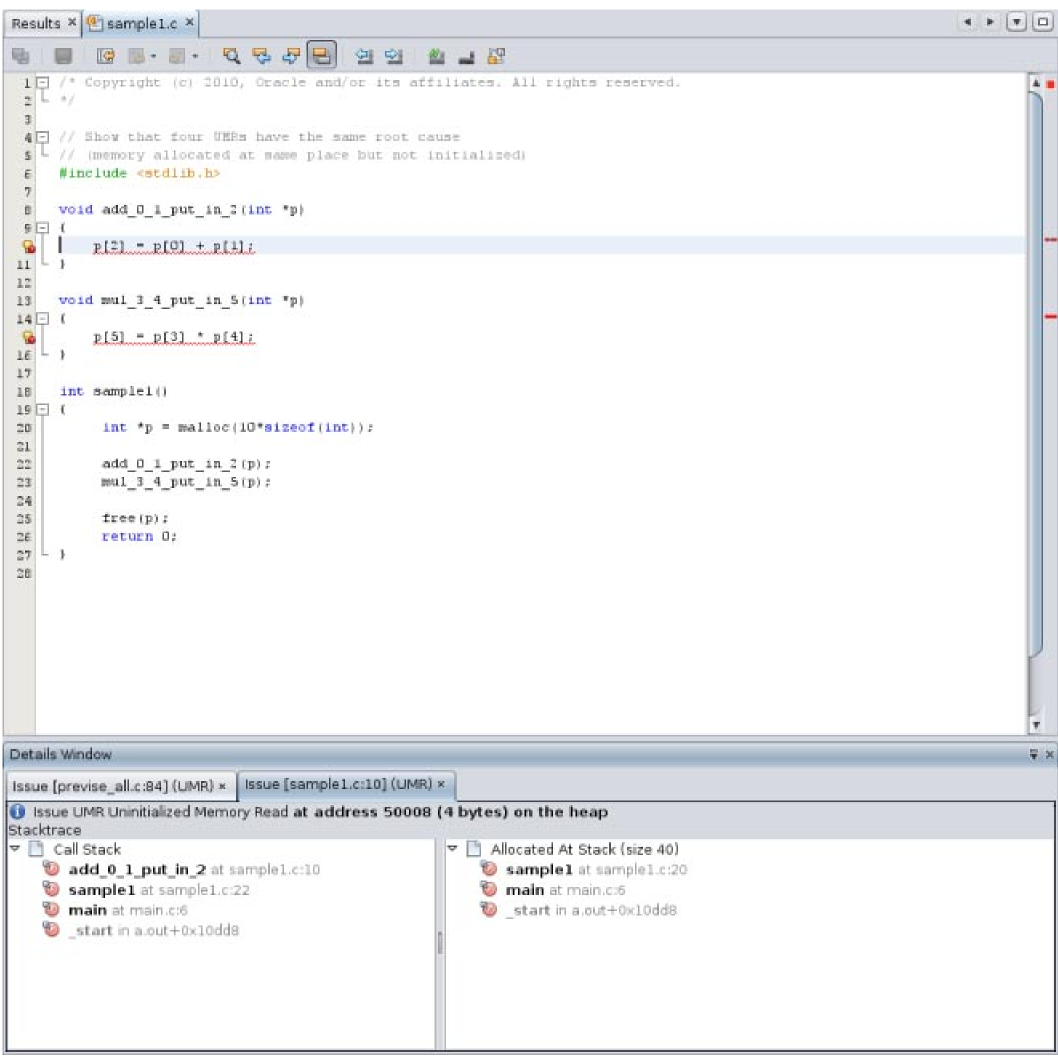
```
6: #include <stdlib.h>
8: void add_0_1_put_in_2(int *p)
9: {
10:     p[2] = p[0] + p[1];
11: }
```

Stacktrace

Call Stack
- add_0_1_put_in_2 at sample1.c:10
- sample1 at sample1.c:22
- main at main.c:6
- _start in a.out+0x8050ecb

Allocated At Stack (size 40)
- sample1 at sample1.c:20
- main at main.c:6
- _start in a.out+0x8050ecb

Double-click function calls in the stacks to see the associated lines in the source file. When the source file opens, the stack trace is displayed in a Details window below the file.




9. Close the Code Analyzer GUI.

## Collecting and Displaying Code Coverage Data

Whether or not you have collected static data or dynamic memory access data, you can compile, instrument, and run your application to collect code coverage data.

1. Since you built the application with the `-g` option before you collected dynamic memory error data and saved a copy of the binary before instrumenting it, you can copy the saved binary to instrument for coverage data collection.

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

2. Instrument the binary with Uncover:

   ```
   uncover a.out
   ```

3. Run the instrumented binary to collect the code coverage data.

   ```
   ./a.out
   ```

   The code coverage data is written to an `a.out.uc` directory in your sample directory.

4. Run Uncover on the `a.out.uc` directory.

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

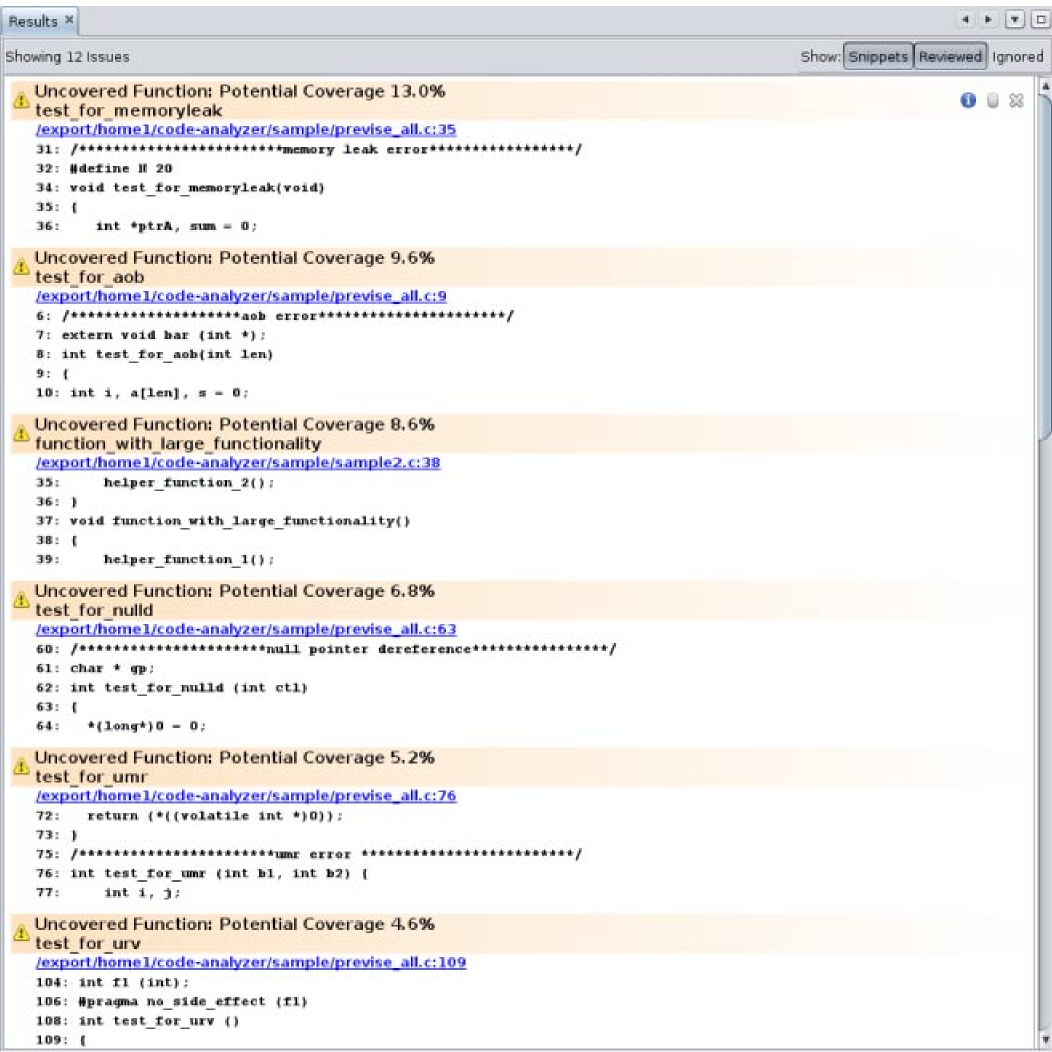
   The code coverage data is written to an uncover subdirectory in the `a.out.analyze` directory in your sample directory.

5. Open the Code Analyzer GUI to view the results:

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

6. The Results tab now shows static issues, dynamic memory issues, and code coverage issues. To filter the results and show just the code coverage issues, select the Coverage checkbox on the Issues tab.

   Now the Results tab shows just the twelve code coverage issues. The description of each issue includes a potential coverage percentage, which indicates the percentage of coverage that will be added to the total coverage for the application if a test covering the relevant function is added.




---

**Note** – To see all of the issues without scrolling up and down, click the Hide Snippets button

at the top of the Results tab to hide the code snippets.

---

7. In the Issues tab, notice that nine of the coverage issues are in the `previse_all.c` source file, three of them are in `sample2.c`, and one is in `previse_1.c`. To further filter the results and show just the issues for the `sample2.c` file, select the checkbox for that file on the Issues tab.

   The Results tab now shows just the three code coverage issues found in `sample2.c`.

```
Results ×
Showing 3 Issues                                              Show: Snippets Reviewed Ignored

Uncovered Function: Potential Coverage 8.9%
function_with_large_functionality
/net/dct-06/export/home/analytics-0301/sample/sample2.c:38
35:      helper_function_2();
36: }
37: void function_with_large_functionality()
38: {
39:      helper_function_1();

Uncovered Function: Potential Coverage 2.0%
function_with_small_functionality
/net/dct-06/export/home/analytics-0301/sample/sample2.c:43
39:      helper_function_1();
40: }
42: void function_with_small_functionality()
43: {
44:      helper_function_5();

Uncovered Function: Potential Coverage 1.2%
report_error
/net/dct-06/export/home/analytics-0301/sample/sample2.c:48
44:      helper_function_5();
45: }
47: void report_error()
48: {
49:      printf("An error has occurred\n");
```

8. Open the source file by clicking on the source file path link in one of issues. Scroll down in the source file until you see the warning icons in the left margin.

```
Results × | sample2.c ×
25      helper_function_4();
26  }
27  void helper_function_2()
28  {
29      printf("helper_function_2\n");
30      helper_function_3();
31  }
32  void helper_function_1()
33  {
34      printf("helper_function_1\n");
35      helper_function_2();
36  }
37  void function_with_large_functionality()
38  {
39      helper_function_1();
40  }
41
42  void function_with_small_functionality()
43  {
44      helper_function_5();
45  }
46
47  void report_error()
48  {
49      printf("An error has occurred\n");
50  }
51
52  int sample2()
53  {
54
55      covered_function();
56      if (error)
57          report_error();
58      return 0;
59  }
60
```

Code that is not covered is marked with a yellow bracket, for example

```
    helper_function_5();
}
```

. The coverage issues found in the file are marked with warning icons

## Using the Issues Found by the Code Analyzer to Improve Your Code

By fixing the core issues found by the Code Analyzer, you should be able to eliminate the other issues found in your code, and make major improvements in its quality and stability.

By doing static error checking, you can find the risky code in your application. But static error checking can generate false positives. Dynamic checking can help verify and eliminate these errors, giving a more accurate picture of the issues in your code. And code coverage checking can help you improve your dynamic test suites.

The Code Analyzer integrates the results these three types of checking to give you the most accurate analysis of your code all in one tool.

## Potential Errors Found by Code Analyzer Tools

The compilers, Discover, and Uncover find static code issues, dynamic memory access issues, and coverage issues in your code. The specific error types that are found by these tools and analyzed by the Code Analyzer are listed in the following sections.

### Static Code Issues

Static code checking finds the following types of errors:

- ABR: beyond Array Bounds Read
- ABW: beyond Array Bounds Write
- DFM: Double Freeing Memory
- ECV: Explicit type Cast Violation
- FMR: Freed Memory Read
- FMW: Freed Memory Write
- FOU: `PM_OUT` use before definition
- INF: INFinite empty loop
- Memory leak
- MFR: Missing Function Return
- MRC: Missing malloc Return value Check
- NFR: uNinitialized Function Return
- NUL: NULl pointer dereference, leaky pointer check
- RFM: Return Freed Memory
- UMR: Uninitialized Memory Read, Uninitialized Memoey Read bit operation
- URV: Unused Return Value
- VES: out-of-scope local Variable usage

### Dynamic Memory Access Issues

Dynamic memory access checking finds the following types of errors:

- ABR: beyond Array Bounds Read
- ABW: beyond Array Bounds Write
- BFM: Bad Free Memory
- BRP: Bad Realloc address Parameter
- CGB: Corrupted Guard Block
- DFM: Double Freeing Memory
- FMR: Freed Memory Read
- FMW: Freed Memory Write
- IMR: Invalid Memory Read
- IMW: Invalid Memory Write
- Memory leak
- OLP: OverLaPping source and destination

- PIR: Partially Initialized Read
- SBR: beyond Stack Bounds Read
- SBW: beyond Stack Bounds Write
- UAR: UnAllocated memory Read
- UAW: UnAllocated memory Write
- UMR: Unitialized Memory Read

Dynamic memory access checking finds the following types of warnings:

- AZS: Allocating Zero Size
- Memory leak
- SMR: Speculative unitialized Memory Read

## Code Coverage Issues

Code coverage checking determines which functions are uncovered. In the results, code coverage issues found are labeled as Uncovered Function, with a potential coverage percentage, which indicates the percentage of coverage that will be added to the total coverage for the application if a test covering the relevant function is added.

Copyright ©2011 This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS. Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

E23337

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