Runtime Error Detection

This topic explains how to use C++test to perform runtime error detection. Included in this section:

Runtime Error Detection Overview

C++test's runtime error detection enables teams to automatically identify serious runtime defects—such as memory leaks, null pointers, uninitialized memory, and buffer overflows—at the unit or application level. It is suitable for both enterprise and embedded development.

The adaptability of this capability makes runtime memory analysis possible for teams working with non-standard memory allocation models—e.g., with embedded systems. Since the instrumentation used for this analysis is lightweight, it can be run on the target board, simulator, or host for embedded testing.

The collected problems are reported with the details required to understand and fix the problem (including memory block size, array index, allocation/deallocation stack trace etc.) Coverage metrics are tracked to help the you measure and increase the scope of your testing efforts.

Performing Runtime Error Detection at the Application Level

To perform runtime error detection on an application run:

- Run one of the built-in configurations from the Application Monitoring group (e.g. **Application Monitoring> Build and Run Application with Memory Monitoring**). These are described in Application Monitoring Group.

C++test will prepare an instrumented version of the application executable and then run it. Depending on the configuration selected, C++test will report coverage statistics and/or memory errors found during the application execution.

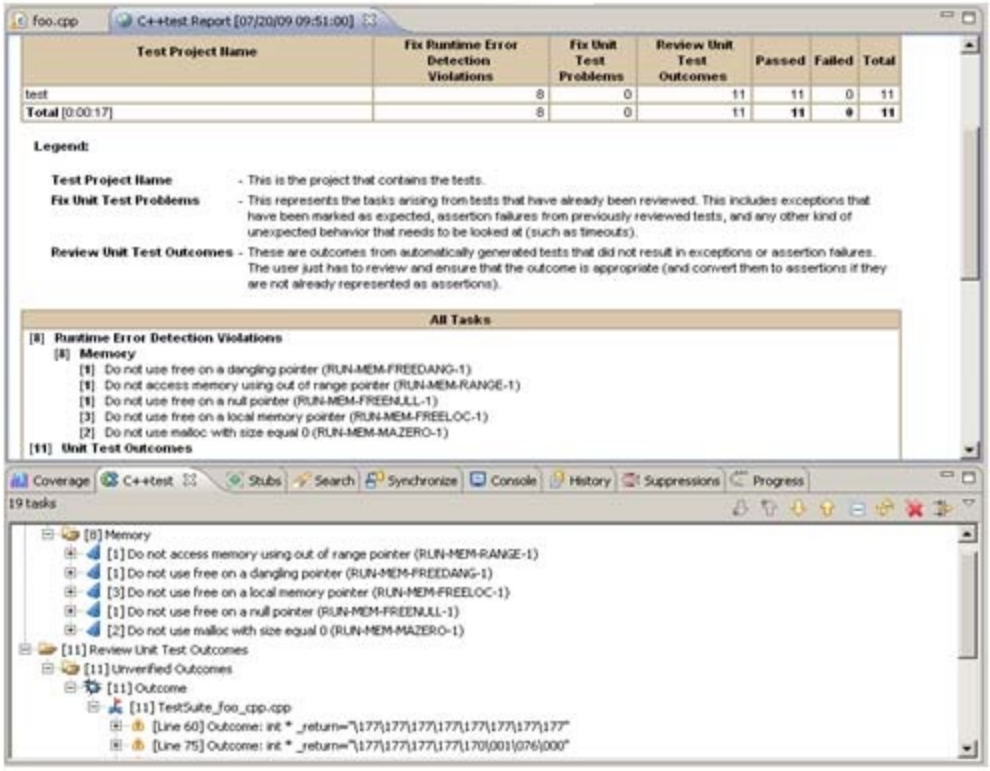
> **Important Note**
>
> For a project that is built into an executable, you can monitor an application run with runtime error detection. To monitor a library project, you need to add additional code with a 'main()' function definition (that simulates some scenario of using the tested library) to the project for C++test purposes. For example, such a 'main()' function can be added with the `#ifdef PARASOFT_CPPTEST` guard into one of the tested library source files.

Performing Runtime Error Detection During Unit Test Execution

To perform runtime error detection during unit test execution:

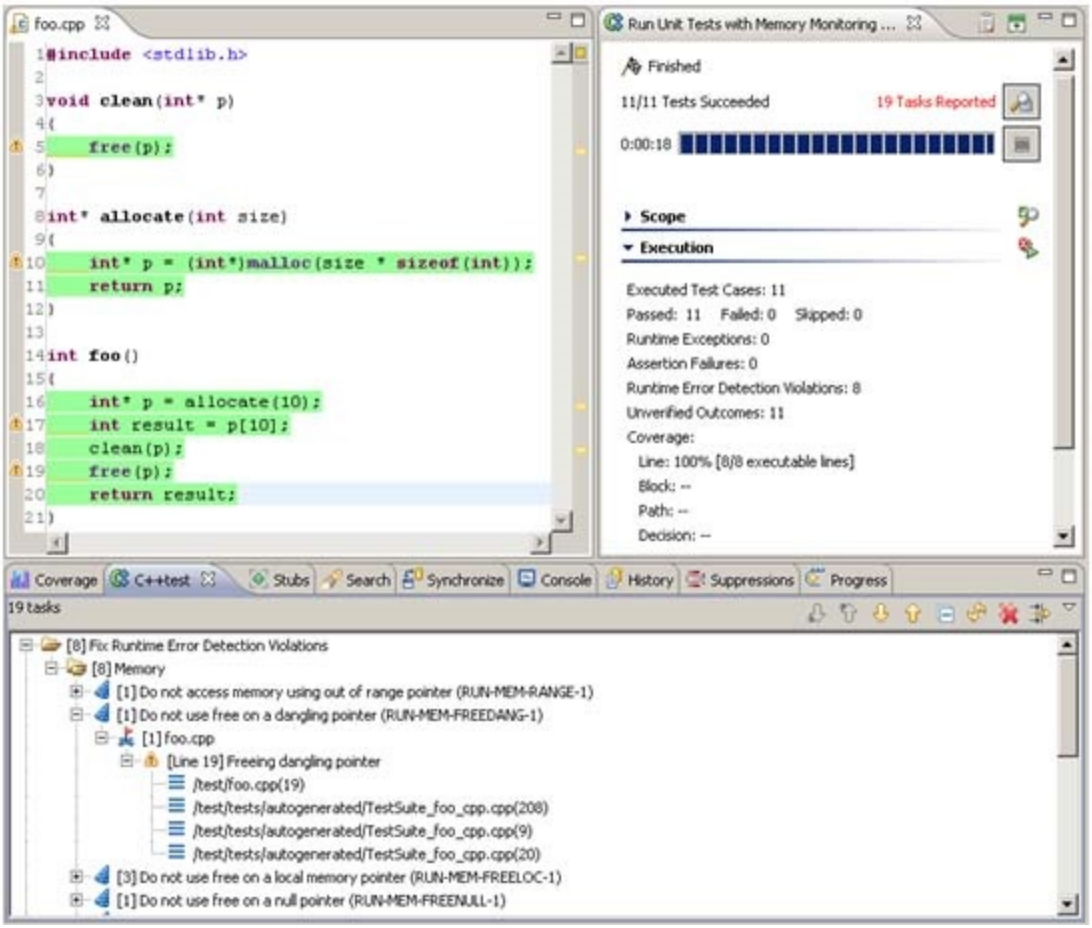
- Run the **"Unit Testing> Run Unit Tests with Memory Monitoring"** Test Configuration.

C++test will produce a test executable with runtime memory analysis enabled and then execute the tests. After test execution completes, C++test will report the memory problems found alongside the regular unit testing tasks.



Each runtime error detection problem is reported as a violation of a given rule (with message, location, and stack trace) in the following places:

- The C++test tasks view in the GUI.
- The code editor in the GUI.
- The C++test console (in both GUI and CLI runs).
- The report generated after the test run.



Suppressing Runtime Error Detection Violations

Runtime error detection violations can be suppressed like static analysis violations: in the GUI, or in the source code (`// parasoft-suppress <ruleid> ["<reason>"]`).

For detailed instructions, see Suppressing the Reporting of Acceptable Violations.

Customizing Runtime Error Detection Options

You can customize runtime error detection by modifying options from the Test Configuration manager's Execution tab.

In the **Execution> General** tab, you can control the following options:

- **Unit Testing or Application Monitoring analysis mode:** This determines whether runtime error detection should be performed when executing unit tests or if a standalone application should be built and executed (without test cases).
  - For Application Monitoring mode, you can (optionally) specify the application binary location/name (**Test application binary**) as well as specify the execution command line (**Application command line**).
- **Test execution flow:**
  - For Application Monitoring mode, you can use the **Build application executable** or **Build and run application executable** test execution flows.
  - For Unit Testing mode, you can use a standard or custom unit test execution flow. For details, see Customizing the Test Execution Flow.
- **Instrumentation mode:**
  - For Application Monitoring mode, you can use **Application full monitoring**, **Application memory monitoring**, or **Application coverage monitoring**.
  - For Unit Testing mode, you can use **Full runtime with memory monitoring**.

In the **Execution> Runtime** tab, you can control the following option:

- **Test executable run directory:** For both Unit Testing and Application Monitoring modes, this determines the directory in which the test executable is created and should execute. If relative paths are used in test case sources or in the original code, C++test will search for the referenced files in this directory.

Runtime Error Detection Rules

C++test provides the following runtime error detection “rules” that find memory-related problems in the tested code:

Rule Identifier	Description
RUN-MEM-DANG	Do not access memory using a dangling pointer This rule finds problems related to using a pointer to an already freed memory.
RUN-MEM-WILD	Do not access memory using a wild pointer This rule finds problems related to using a pointer that does not point to a valid memory buffer.
RUN-MEM-NULL	Do not access memory using a null pointer This rule finds problems related to using a null pointer.
RUN-MEM-RANGE	Do not access memory using an out of range pointer This rule finds problems related to accessing a buffer out of range (e. g. accessing 10th element of 9-elements buffer).
RUN-MEM-UNINIT	Do not read uninitialized memory This rule finds problems related to reading from allocated (but not initialized) memory.
RUN-MEM-FREEDANG	Do not use free on a dangling pointer This rule finds problems related to trying to free an already-freed memory pointer.
RUN-MEM-FREEIL	Do not use free on an illegal pointer This rule finds problems related to trying to use free on a pointer that does not point to the valid memory block allocated with malloc.
RUN-MEM-FREELOC	Do not use free on a local memory pointer This rule finds problems related to trying to use free on a pointer that points to a local memory block.
RUN-MEM-FREEGLOB	Do not use free on a global memory pointer This rule finds problems related to trying to use free on a pointer that points to a global memory block.
RUN-MEM-FREENULL	Do not use free on a null pointer This rule finds problems related to trying to use free on a null pointer.
RUN-MEM-MAZERO	Do not use malloc with a size equal to 0 This rule finds problems related to using malloc to allocate a zero-size buffer.
RUN-MEM-CAZEROELEM	Do not use calloc with a number of elements equal to 0 This rule finds problems related to using calloc to allocate a zero-elements buffer.
RUN-MEM-CAZEROSIZE	Do not use calloc with an element size equal to 0 This rule finds problems related to using calloc to allocate a buffer of zero-sized elements.

RUN-MEM-RAILL	Do not use realloc on an illegal pointer This rule finds problems related to using realloc on a pointer that does not point to the valid memory block allocated with malloc.
RUN-MEM-RALOC	Do not use realloc on a local memory pointer This rule finds problems related to using realloc on a pointer that points to a local memory block.
RUN-MEM-RAGLOB	Do not use realloc on a global memory pointer This rule finds problems related to using realloc on a pointer that points to a global memory block.
RUN-MEM-RAZERO	Do not use realloc with a new size equal to 0 This rule finds problems related to using realloc to allocate a zero-size buffer.
RUN-MEM-LEAK	Avoid memory leaks This rule finds memory leaks. It will report a problem when a pointer to the memory allocated with malloc/realloc/calloc is lost.
RUN-MEM-CORRUPT	Avoid memory corruption This rule finds memory corruption. It will report a problem when a memory block allocated with malloc/realloc/calloc got overwritten unexpectedly during the deallocation process.

Runtime Error Detection Test Configurations

For a description of the available Test Configurations, see Built-in Test Configurations.

We recommend that you use the "Build Application with..." Test Configurations if you want to verify that everything is ready for testing. These are also the Test Configurations to use if the built application will be run externally (e.g., for embedded testing). In this case, the complete flow consists of the Build, Deploy/Run (done manually) and 'Read logs' steps.

"Build and Run Application with..." Test Configurations should be used when you want C++test to build and execute the tested application using the command line specified in the Test Configuration.

More specifically:

- "... Memory Monitoring" configurations are used when you want C++test to perform runtime error detection on the tested application.
- "... Coverage Monitoring" configurations are used when you want C++test to collect coverage information from the tested application run.
- - "... Full Monitoring" configurations are used when you want to perform runtime error detection as well as collect coverage information.

Runtime Error Detection Known Limitations

- Only C-style dynamic memory allocations (using malloc(), calloc(), realloc()) are monitored.
- Only C-style dynamic memory deallocations (using free()) are monitored.
- Only global arrays are monitored (static arrays defined in function are not monitored).
- Memory-related operations have to be done directly in the tested and instrumented compilation unit in order to be monitored.
- Memory-related operations which are being done in C++ templates are not monitored and may influence results.
- Memory-related operations which are being done in external libraries are not monitored and may influence results.
- By default, memory leaks (RUN-MEM-LEAK) are reported only in the Application Monitoring mode.