Concepts and Terms

In this section:

Static Code Analysis

C++test performs static code analysis by statically analyzing code to check compliance with specified static code analysis rules. This analysis is aimed at preventing errors and improving code quality by:

- Detecting definite or potential defects in the source.
- Preventing the usage of code that could compromise security
- Enforcing organizational design guidelines and specifications (application-specific, use-specific, or platform-specific) and error-prevention guidelines abstracted from known specific bugs.
- Improving code maintainability by improving class design and code organization.
- Enhancing code readability by applying common formatting, naming, and other stylistic conventions.

C++test is preconfigured to perform static code analysis with built-in rule sets. The rules enabled by default in most static code analysis-checking Test Configurations have been shown to make an immediate and significant improvement to code. Code that follows this core set of guidelines will be faster, more secure, easier to maintain, and less likely to experience functional problems.

Along with providing preconfigured rule sets, C++test allows you to define your own rule sets, including custom rules, to implement a coding policy specific to your organization. To help teams determine which rules to comply with, rules are categorized by topic (for instance, security, optimization, initialization, and so on) as well as ranked by severity (the likelihood that the detected problem will lead to a bug).

To learn more about the static code analysis rules that are included with C++test, choose **Parasoft> Help**, open the **C++test Static Analysis Rules** book, then browse the available rule description files.

C++test can also check any number of custom rules that you design with the RuleWizard module. With RuleWizard, rules are created graphically (by creating a flow-chart-like representation of the rule) or automatically (by providing code that demonstrates a sample rule violation). By creating and checking custom rules, teams can verify unique project and organizational requirements, as well as prevent their most common errors from recurring.

For details on performing static code analysis, see Static Code Analysis.

Suppressions

Suppressions are used to prevent C++test from reporting additional occurrences of a specific static analysis task (multiple tasks might be reported for a single rule). Suppressed messages will be sent to a special Suppressions view instead of the Quality Tasks view; this allows you to monitor those violations as needed while keeping your main results areas focused on the other errors.

You use suppressions for situations when you generally want to follow a rule, but decide to ignore that rule in an isolated number of exceptional situations. With suppressions, you can continue checking whether your code follows that rule without receiving repeated messages about your intentional rule violations. If you do not want to receive error messages for any violations of a specific rule, we recommend that you modify your Test Configurations so that they no longer check that rule.

Suppressions can be entered in the GUI or in the code. The ones entered in the GUI can be stored in the code, or saved on Parasoft Team Server (so that are shared across the team).

Note that suppression settings are independent of Test Configurations. To avoid confusion, keep in mind that:

- The Test Configuration defines the set of rules that are checked during static analysis.
- Suppressions define which static analysis *results* should be visible in the Quality Tasks view and reports.

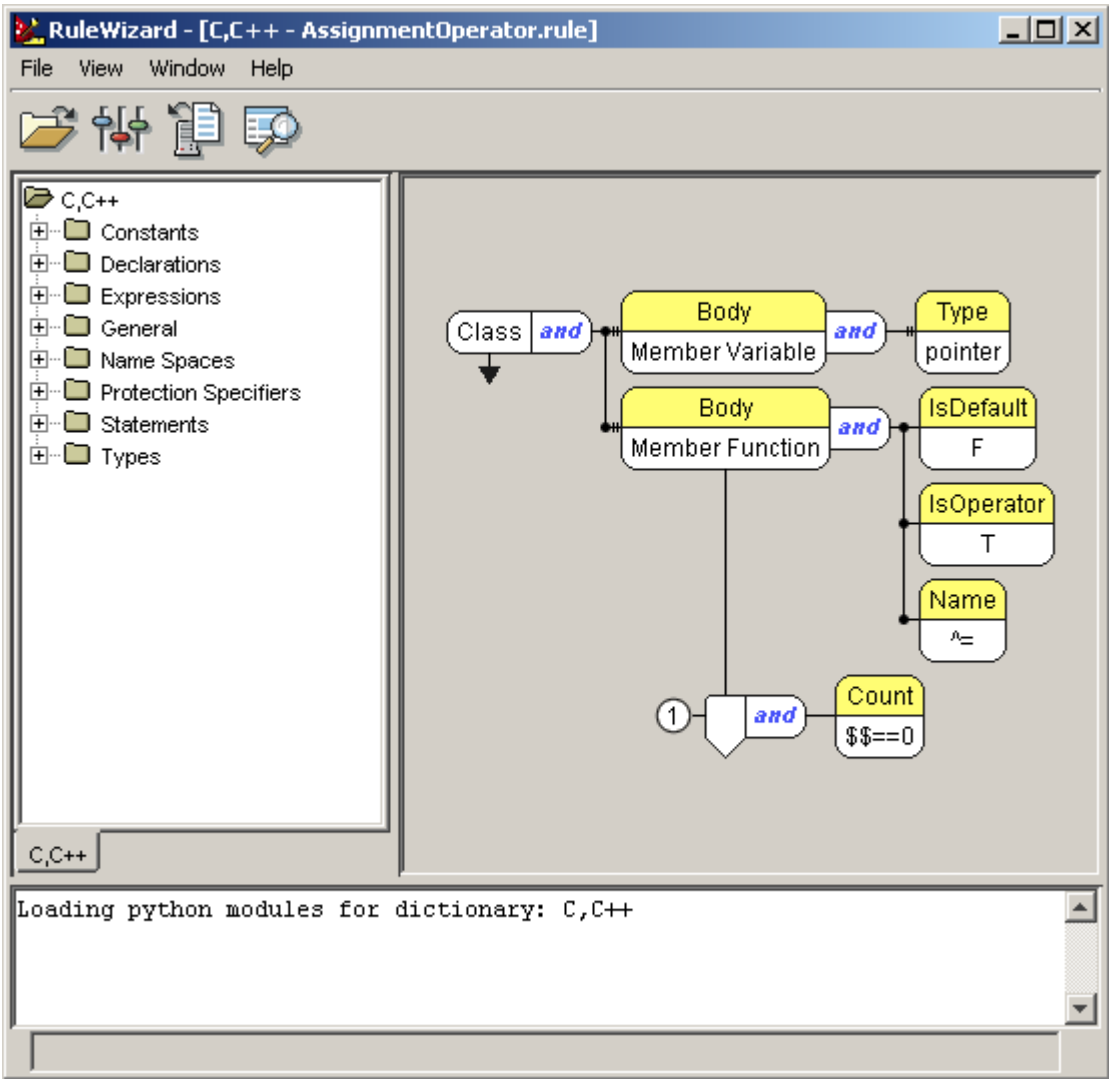
This means that rules selected in the Test Configuration will be checked during analysis, but results that match the suppression criteria will not be displayed.

For details on entering suppressions, see Suppressing the Reporting of Acceptable Violations.

RuleWizard Graphical Rule Editor

RuleWizard allows you to create custom static code analysis rules for C/C++ code and formatting issues. C++test can automatically enforce any valid rule created in RuleWizard. By creating and checking custom rules, teams can verify unique project and organizational requirements, as well as prevent their most common errors from recurring.

With RuleWizard, rules can be created graphically (by creating a flow-chart-like representation of the rule) or automatically (by providing code that demonstrates a sample rule violation). No coding or knowledge of the parser is required to write or modify a rule.



There are two ways to open RuleWizard:

- Choose **Parasoft> Launch RuleWizard**.
- Click the **New** button in the Test Configurations panel's **Static** tab.

The RuleWizard GUI will then open. The RuleWizard User's Guide (accessible by choosing **Help>Documentation** in the RuleWizard GUI) contains information on how to modify, create, and save custom rules.

For details on creating and using custom rules, see Customizing Existing Rules and Creating New Rules.

Flow Analysis

> **Important Note**
>
> An additional license is required to use this functionality. Contact your Parasoft representative.

Flow is a new type of static analysis technology that uses several analysis techniques, including simulation of application execution paths, to identify paths that could trigger runtime defects. Defects detected include use of uninitialized memory, null pointer dereferencing, division by zero, memory and resource leaks.

Since this analysis involves identifying and tracing complex paths, it exposes bugs that typically evade static code analysis and unit testing, and would be difficult to find through manual testing or inspection. Flow Analysis' ability to expose bugs without executing code is especially valuable for users with legacy code bases and embedded code (where runtime detection of such errors is not effective or possible).

This unique breed of static analysis starts analyzing the source code under test by searching for "suspicious points" in the code. A suspicious point is a point where there is potential for a bug. These suspicious points are defined in Flow Analysis rules. Whenever a suspicious point is identified, Flow Analysis investigates possible execution paths which lead to that point and checks if any of those paths actually violates the Flow Analysis rule. If such a path is found, a violation is reported.

For example, the rule that detects possible division by zero says that any point where “/” or “%” operator is used is suspicious. It then checks whether the variable in the denominator can hold zero value at that point on any of the possible execution paths leading to it. If so, an error is reported.

For each bug found, hierarchical flow path data precisely detail the complete execution path that leads to the identified bug, ending with the exact line of code where the bug manifests itself. To reduce the time and effort required to diagnose and correct each problem found, flow path details are supplemented with extensive annotations (for example, an ”Avoid null pointer dereferencing” violation description contains annotations describing which variables contain null values at which point in the flow path).

To make the analysis process more flexible and tailored to your unique project needs, some rules can be parameterized. As a result, Flow Analysis can even be used to detect violations bound to usage of very specific APIs.

Using Flow Analysis, development teams gain the following key benefits:

- **Perform more comprehensive testing with existing resources:** Flow Analysis complements other testing techniques by allowing you to find problems that would otherwise require the development, execution, and maintenance of complex test cases. Flow Analysis investigates various branching possibilities in a program, providing a level of path coverage that is difficult to accomplish with traditional testing. As a result, Flow Analysis often identifies problems that occur while handling rare situations that are typically not covered during testing. Moreover, if the code functionality changes, you can search for bugs in the modified version without updating or regenerating test cases.
- **Automatically identify bugs that pass through multiple units:** Traditional automated unit test generation helps you identify the bugs within a single compilation unit. This is critical. Yet, most developers have performed thorough unit-level testing, corrected all apparent problems, integrated the code, then later encountered problems, such as null pointer dereferencing, that took days to diagnose because they resulted from an obscure or complex execution path that passed through multiple functions or even multiple compilation units Using Flow Analysis, the same problem could be identified automatically
- **Focus on actual bugs and design flaws:** Flow Analysis automatically identifies data-dependent or flow-dependent bugs with reasonable certainty. When Flow Analysis reports a violation, it is often the case that there was a design flaw and this manifested itself as simple violation such as a division by zero or a resource leak.

  For example, Flow Analysis would not report a violation for the following code unless there was a method calling calculateBufferLength and passing it a null pointer:

  ```
  int calculateBufferLength(char* str)
  {
              return strlen(str) + 1;
  }
  ```

- **Find API misuses:** Many of the bugs in practice are due to calling some API with wrong arguments or not properly handling the values returned by the API. For example, an API may be expecting a non-null argument for parameter 2, when parameter 1 is true, or the API may potentially set some field in an object to null. By performing interprocedural analysis, Flow Analysis can point out inconsistencies in the usage of such API.

For details on using BugDetective, see Flow Analysis.

Unit Testing

"Unit testing" refers to testing software code at the simplest functional point, which is typically a single class or a function. Unit testing is typically performed by developers inside a development cycle, rather than in the QA phase. Using unit testing, you can ensure that the application building blocks are solid before they are integrated, thus improving the quality of the entire application. When you test early, it is typically less difficult and time-consuming to identify and fix defects at this point.

Manual unit testing typically involves writing test harnesses by hand, specifying input data, and supplying stubs for missing functions and variables. C++test automates these tasks to make the unit testing process more efficient and consistent.

Generally, unit testing can include:

- Exception testing (also known as white-box testing, stress testing, construction testing, or reliability testing), which is used to confirm that code is structurally sound and can handle the full range of feasible inputs and their combinations without throwing an unexpected exception.

- Functional testing, which is used to verify that software building blocks conform to their specification and that all of the intended functionality is included and working correctly. Creating functional tests at the unit level involves human input to specify particular input and state conditions, and expected output. Functional testing can be implemented as white box testing (testing with knowledge of internals and implementation of a unit under test), or black box testing, which is solely based on external behavior of a unit under test.
- Regression testing, which is used to verify that existing code behavior does not change as the code base evolves. This is typically done by developing a set of tests, verifying their correctness, and running them after code changes to catch deviations in code behavior. Regression testing may rely on both exception testing and functional testing.

C++test can perform all of described types of unit testing; the level and scope of testing performed can be customized to reflect your requirements and test preferences.

Exception testing is performed when you run the test cases that C++test generates automatically. This type of testing exposes unexpected exceptions and checks that the class is structurally sound. The success of reliability testing hinges upon thorough coverage of the code, so you might want to extend the automatically-generated test cases as needed to increase code coverage. C++test measures test coverage to help you assess coverage and determine where additional tests are necessary.

Functional testing is performed when you extend the automatically-generated test cases to verify whether the class's public interface operates as described in the specification.

Regression testing involves regularly testing the evolving code base by running all available test cases and checking if the expected outcomes have changed. C++test reports error messages when test case outcomes from the current test run do not match the expected test case outcomes.

Test cases can be implemented and saved in C or C++ source code. You can extend and modify generated test cases within the IDE's text editor. These test cases use a format similar to the popular CppUnit format. C++test's tests offer more extensive capabilities than CppUnit, including the ability to test C code and provide programmatic access to private and protected data and member functions within the test framework. Existing CppUnit test cases can be imported into C++test and used in concert with automatically-generated test cases.

For details on performing unit testing, see Test Creation and Execution.

Test Case Generation

Writing unit tests is an important task that ensures the quality of the code. Unit tests not only expose bugs and functionality problems, but can also be run as regular regression tests to help you determine whether code additions/modifications break existing functionality or cause unexpected changes.

However, writing tests is time-consuming and, when not done carefully, can miss many important situations. The unit test support provided by C++test helps the developers and testers to create good unit tests very rapidly. It automatically generates many unit tests and allows users to customize the test generation and execution to suit their preferences and needs. C++test automatically generates a large number of test cases which try to exercise all of the different paths through the code, then saves the actual test case outcomes. These test cases essentially take an x-ray of the code's current state, capturing a snapshot of how it was operating prior to the modification. They also help identify potential exceptions that could impact code reliability and security.

C++test can generate test cases for any C/C++ code from a single function to an entire project. By running these test cases in C++test, you verify class robustness and identify inputs that could cause the program to enter an inconsistent state or terminate. You can customize the preconfigured test modes, as well as specific test generation settings.

Test cases are implemented and saved in C or C++ source code (the language used depends on the original source being tested). These test cases use a format similar to the popular CppUnit format. C++test tests offer more extensive capabilities than CppUnit, including the ability to test C code and programmatic access to private and protected data and member functions within the test framework. Existing CppUnit test cases can be imported into C++test and used in concert with automatically-generated test cases The test suite can be extended with user-defined test cases to improve test coverage and verify specific functionality; these tests can be added by modifying automatically-generated test cases, or by defining new ones. Any available test case can be automatically verified and configured for regression testing. By saving all available test cases and leveraging them for automated regression testing, you establish a regression testing infrastructure that immediately identifies unexpected functionality changes and exceptions introduced by code modifications.

Automated test case generation allows you to create a more effective test suite in less time. Test case development is traditionally the most time-consuming part of the unit testing process. With this generation, you do not need to write any code to generate a foundational set of test cases that exercise each class, and you can create more test cases by adding a minimal amount of code to the automatically-generated test cases. You typically do not need to worry about writing test cases for simple methods, and can focus resources on extending/adding tests for the more complex methods.

Moreover, test case generation helps you prevent errors in two key ways:

- It enables you to instantly generate and execute test cases as soon as you are done writing or modifying a class. This helps you find and fix problems before you (or a team member) unwittingly introduce additional errors by adding code that builds upon or interacts with the problematic code.
- It enables fast, thorough reliability testing by automatically generating the required range and type of test cases, which is impractical to achieve by manually designing test cases. C++test tries to create test cases that execute every possible branch of each method it tests. For example, if the method contains a conditional statement (such as an `if` block), C++test will try to generate test cases that test the `true` and `false` outcomes of the `if` statement.

For details on performing automated test case generation, see Generating Test Cases for Regression Testing and Exception Finding.

Stubs

Stubs serve as replacement implementations for functions so that tests can be isolated from dependencies outside of the set of tested functionality. When using stubs, the execution flow is redirected to a stub function.

There are two main reasons to use stubs:

- To isolate the code under test from the integrated environment.
- To test when affecting the behavior of a function is not possible, requiring an alternative implementation.

You can define your own stubs for any test case—automatically-generated test cases as well as user-defined test cases. When you use user-defined stubs, you have complete control over what values an external function returns to the class under test—without having to have the actual external function available.

If a user-defined stub is available, it will always be used during test execution—even if the original function is available. If the original definition was not initially available, but is added later, C++test will continue using the user-defined stub. If you want it to use the original definition, you need to remove the stub (or comment it out).

User-defined stubs are implemented in the form of a function definition with the "CppTest_Stub_" prefix in the function name. For example:

```
/* C++test user stub definition for int doSomething(int i) */
int ::CppTest_Stub_doSomething(int i)
{
    return i + 10;
}
```

The stubbed function's declaration must be accessible in the stub file. In most cases, it is connected by including the appropriate header file into the stub file.

C++test's stubs (except for constructor stubs) take the same values as the original functions.

Stubs that have been automatically generated or created using the Stub wizard can be dynamically configured from test cases without requiring modifications to the stubs' bodies. You can dynamically configure stubs in the Test Case Editor (see Adding Test Suites and Test Cases with the Test Case Editor) or directly from the test case source code using the Dynamic Stubs Configuration (see Dynamic Stubs Configuration).

When using the Dynamic Stubs Configuration is insufficient, you can replace the bodies of generated stubs with a custom logic implementation and utilize C++test API functions for user-defined stubs (see C++test API Functions for User-Defined Stubs).

To learn more about the automatically-generated stubs, see Understanding and Customizing Automated Stub Generation.

To learn how to use user-defined stubs for external resources that you cannot (or do not want to) access during testing, see Adding and Modifying Stubs.

Factory Functions

Factory functions are the user-defined methods used to initialize objects of a given type. Such methods can then be used when generating test cases automatically and when creating test case using the Test Case Wizard.

Factory functions can be used to:

- Provide complex initializers for user-defined types (by writing a factory function that does some additional initialization after the actual object is created).
- Reduce the number of possible object creation methods (by writing a factory function that uses just one of the available constructors for a given class).

Factory functions are functions with a "CppTest_Factory_" prefix in the function name. The function return type defines which type of objects will be created using the given factory function. Factory functions can take arguments that makes them parameterizable. For example, here is a factory function for std::vector<string> containing a number of strings:

```
std::vector<std::string> CppTest_Factory_vector_of_strings(unsigned int size, const std::string&value)
{
    std::vector<std::string> vec;
    for (int i = 0; i < size; i++) {
          vec.push_back(value);
    }
    return vec;
}
```

For details, see Using Factory Functions.

Runtime Error Detection

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.

For details, see Runtime Error Detection.

Application Monitoring

Application monitoring is designed to provide test results from the actual application run. The instrumentation used to perform application monitoring is lightweight and suitable for running on the target board for embedded testing.

C++test can be used to monitor the execution of an application executable prepared with C++test instrumentation. It is usually performed by both developers and QA during their regular application test sessions—along with other complementary quality practices (peer review, static analysis, unit testing, etc.)

With application monitoring enabled during testing, you gain access to details not normally available from QA sessions— without requiring any additional work from the testers. The only preparation required is that application needs to be built by C++test.

Application Monitoring analysis can provide:

- **Coverage analysis**, which gives information about which parts of the application code were covered when the application was executed. This helps the team determine which source code elements implement the specific functionalities or use cases that were not covered by the application-level tests.
- **Runtime error detection**, which can be used to detect memory errors such as memory access errors, memory leaks, memory corruption, and more.

For details on performing application monitoring, see Runtime Error Detection. For details on coverage analysis, see Coverage Analysis.

Test Configurations

A Test Configuration is a collection of settings that define a test scenario that you want to run with C++test. Each time C++test runs a test—in the GUI or from the command line interface—it uses the designated Test Configuration (or the Favorite Test Configuration, if no Test Configuration was explicitly selected). The Test Configuration determines all test parameters. For example, it determines parameters such as:

- The type of tests (static analysis, test case generation, test case execution, etc.)
- The rules checked during static analysis
- The parameters for test case generation
- The scope of each test (what lines to cover, what cutoff date to use, etc.)

C++test includes a set of preconfigured "built-in" Test Configurations representing most common test scenarios. You can further customize these configurations as needed by copying and modifying the built-in configurations, or by creating new user-defined configurations from scratch. User-defined Test Configurations can be placed in the User-defined or Team category. User-defined Test Configurations are stored on the local machine and are available for all tests performed by the local C++test installation. Team Test Configurations are stored on the team's Team Server and can be accessed by all team members.

If C++test is connected to DTP, you can run analysis with test configurations that are stored in DTP.

For general procedures related to configuring and sharing Test Configurations, see Configuring Test Configurations and Rules for Policies. For details on C++test-specific Test Configuration options, see Configuring Test Configuration.

Command Line Interface (cli)

C++test's command line interface (`cpptestcli`) allows you to perform static analysis and unit testing from command line shells and to run C++test from automated build utilities such as batch files, scripts, make, and Ant. Command line mode is available for the Automation Edition of C++test.

`cpptestcli` can send results to the Parasoft DTP, send comprehensive reports to the team manager and to the Parasoft Team Server, and send focused reports to each team developer. Reports can be generated in HTML, PDF, and custom XSL format. Details such as reporting preferences (who should reports be sent to, how should those reports be labelled, what mail server and domain should be used, etc.) Team Server settings, Parasoft by optionss, and email settings, license settings, etc. can be controlled by options files.

The optimal team configuration is to have one C++test (Automation Edition) on the team build machine, C++test on every developer workstation, one C++test on the architect's machine, and one installation of Team Server on the team build machine or another team machine.

Developers use their local installations of C++test to test the code that they write or modify, make the necessary corrections, then check the code and test cases in to source control. Every night the
C++test runs in cli mode on a team machine to verify the checked-in code base; here, it executes all of the tests that developers have created (through automated test generation and manual test definition/customization) and added to source control. After the test completes, team developers can import test result into the C++test GUI to facilitate error examination and correction. Additionally, C++test sends results to the Parasoft

Throughout the process, Team Server manages the sharing and updating of test settings and test files; this standardizes tests across the team and helps team members leverage one another's work. The standardized test settings and custom team rules are configured and maintained by the team architect.

For details about using the command line interface, see Testing from the Command Line Interface.

Parasoft Team Server

The Team Server (formerly named Team Configuration Manager [TCM]) component of Parasoft ensures that all team members have access to the appropriate team Test Configurations, suppressions, rule files, and test case files. Team Server is available and licensed separately. This version of C++test works with Team Server 2.0 and higher, which is distributed as part of Parasoft Server Tools.

After Team Server is installed and deployed as a Web service, the team architect or manager can configure the appropriate team settings and files on one C++test installation, then tell Team Server where to access the settings and related test files. Developers can then point their machines to the Team Server location, and Team Server will ensure that all developer machines access the appropriate settings and files. When the master version of a file is modified, added, or removed, Team Server makes the appropriate updates on all of the team's C++test installations.

Before you can use Team Server to share C++test files, Team Server must be installed and deployed on one of your team's machines. For information on obtaining, installing, and deploying Team Server, refer to the Team Server documentation or contact your Parasoft representative.

For details on how to connect to Team Server, see Connecting to Parasoft Team Server.

Parasoft DTP

Parasoft DTP is a decision support system that provides development teams the on-going visibility and measurement of the software development process necessary to help keep software projects on track. Collecting and consolidating metrics generated during the development process, DTP turns these data points into meaningful statistics and dashboards that provide development managers and team members with the ability to continuously and objectively assess the quality and readiness of the code base, status of the coding process, and the effectiveness of the development team. With DTP, development teams can more readily identify, respond to and manage risks in the code or coding process that threaten project schedules and quality. Parasoft DTP provides the metrics by which management can more effectively assess and direct resources, set and monitor development targets, communicate, guide and measure conformance to development policies, and ensure successful project outcomes.

Once C++test is configured to send information to DTP, developers, architects, and managers can use the DTP dashboard to access role-based reports on quality, progress, and productivity.

For details on how to connect to DTP, see Connecting to DTP.