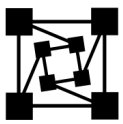
ParaSoft®
c++test™

Dramatically Improving C and C++ Application Quality With Unit Testing



ParaSoft®

Introduction

If you do not perform unit testing you are sacrificing a prime opportunity to simultaneously improve application quality and slash development time and cost. While such a sacrifice is not desirable, it is certainly understandable... at least for C and C++ developers. Until now, C/C++ developers have not had a feasible way to perform unit testing. While Java™ developers could automate unit testing with ParaSoft's Jtest®, C/C++ developers had to perform unit testing manually, which is both difficult and labor-intensive. Now the release of C++Test™, an automatic C/C++ unit testing tool, makes C/C++ unit testing as fast and easy as Jtest has made Java unit testing.

This paper explores the process, advantages, and disadvantages of performing unit testing, then describes how C++Test can offer C/C++ developers all of the benefits of unit testing without submitting them to any of the drawbacks.

What is Unit Testing?

We define unit testing as testing the smallest possible unit of an application; in terms of C and C++, unit testing involves testing a class as soon as it is compiled. The goal of C/C++ unit testing should be to exercise every method or function of every class and detect all existing functionality problems, errors, and construction weaknesses. Finding such problems typically involves three types of testing: black-box testing, white-box testing, and regression testing. Black-box testing checks a class's functionality by determining whether or not the class's public interface performs according to specification; this type of testing is performed without knowledge of implementation details. White-box testing checks that all of a class's methods or functions (including protected and private methods or functions) are robust by determining if the class performs incorrectly when it encounters unexpected input; this type of testing must be performed with full knowledge of the class's implementation details. Regression testing checks whether class modifications have introduced errors into previously correct code.

What Are the Advantages of Unit Testing?

Unit testing is universally recognized as an essential component of the software development process. Unit testing practitioners enjoy such benefits as easier error detection, which has the very desirable end result of increasing software quality at the same time that it reduces development time and cost.

The first way that unit testing facilitates error detection is by bringing you closer to the errors. Figures 1 and 2 demonstrate how unit testing does this.

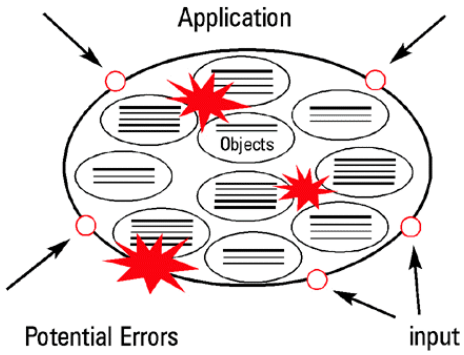


Figure 1: Application Testing

Figure 1 shows a model of testing an application containing many instances of multiple objects. The application is represented by the large oval, and the objects it contains are represented by the smaller ovals. External arrows indicate inputs. Starred regions show potential errors.

To find errors in this model, you need to modify inputs so interactions between objects will force the object to hit the potential errors. This is incredibly difficult. Imagine standing at a pool table with a set of billiard balls in a triangle at the middle of the table, and having to use a cue ball to move the triangle's center ball into a particular pocket— with one stroke. This is how difficult it can be to design an input that finds an error within an application. As a result, developers that rely only on application testing may never reach many of the classes, let alone uncover the errors that they contain.

Testing at the unit level offers a more effective way to find errors. This is demonstrated by Figure 2.

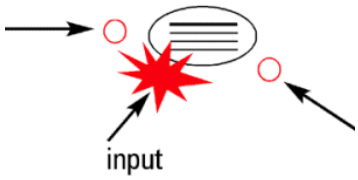


Figure 2: Unit Testing

As Figure 2 illustrates, when you test one object apart from all other objects, you can reach potential errors much easier because you are much closer to the errors. The difficulty of reaching the potential errors when the class is tested as an isolated unit is com-

parable to the difficulty of hitting one billiard ball into a particular pocket with a single stroke.

The second way that unit testing facilitates error detection is by preventing bugs from spawning more bugs, which relieves you from having to wade through problem after problem to remedy what began as a single, simple error. Because bugs build upon and interact with one another, if you leave a bug in your code, chances are it will lead to additional bugs. If you delay testing until the later stages of development, you will probably have to fix more bugs, spend more time finding and fixing each bug, and change more code in order to remove each bug. If you test as you go, it will be easier to find and fix each bug and you'll minimize the chances of bugs spawning more bugs. The result: a significant reduction in debugging time and cost.

Study after study confirms that the time and cost involved in finding software errors rises dramatically the later a problem is detected. Consider the following data reported by Watts Humphrey:

- IBM: An unpublished IBM rule of thumb for the relative costs to identify software defects: during design, 1.5; prior to coding, 1; during coding, 1.5; prior to test, 10; during test, 60; in field use, 100.
- TRW: The relative times to identify defects: during requirements, 1; during design, 3 to 6; during coding 10; in development test, 15 to 40; in acceptance test, 30 to 70; during operation, 40 to 1000 [Boehm 81].
- IBM: The relative time to identify defects: during design review, 1; during code inspections, 20; during machine test, 82 [Remus].
- JPL: Bush reports an average cost per defect: $90 to $120 in inspections and $10,000 in test [Bush].
- Freedman and Weinberg: They report that projects that used review and inspections had a ten-fold reduction in the number of defects found in test and 50% to 80% reductions in test costs, including the costs of the reviews and inspections [Freedman].

What Are the Disadvantages of Unit Testing?

Based on the above information, unit testing sounds like a panacea. If so, why doesn’t every C/C++ developer perform unit testing on every class as soon as he or she compiles it? When performed using the currently available technologies for C/C++, unit testing is difficult, tedious, and time-consuming; until now, C and C++ developers did not have access to tools that could automate enough of the process to make it a component of tight development schedules. A brief look at what is involved in unit testing reveals why so many C and C++ developers shy away from unit testing.

The first step in performing unit testing is making the class testable. This requires two main actions:

- Designing a harness that will run the class.
- Designing stubs that return values for any external resources that are referenced by the class under test, but that are not available or accessible.

Creating a harness involves creating a new class that cannot be used for anything other than testing the original class. According to Hunt and Thomas, test harnesses should include the following features:

- A standard way to specify setup and cleanup.
- A method for selecting individual tests or all available tests.
- A means of analyzing output for expected (or unexpected) results.
- A standard form of failure reporting.

In order to test the class thoroughly and accurately, you need to design a harness that fully exercises the class under test; several modifications or rewrites may be required to create such a harness. Once the harness is created, you must examine it carefully to ensure that it does not contain any errors. An error in the harness can sabotage the test, but because you cannot test a class in isolation (the original problem), you cannot test the harness either.

If your class references any external resources (such as external files, databases, and CORBA objects) that are not yet available or accessible, you must then create stubs that return values similar to those that the actual external resource could return. When creating these stubs, you need to choose stub return values that will test the class's functionality and provide thorough coverage of the class.

The next step is designing and building appropriate test cases. In order to thoroughly test the class’s construction and functionality, you should design two types of test cases: black-box and white-box.

Black-box test cases should be based on the specification document. Specifically, at least one test case should be created for each entry in the specification document; preferably, these test cases should test the various boundary conditions for each entry. Additional black-box test cases should be added for each error that is uncovered, and for any other tests that you deem necessary.

White-box test cases should uncover defects by fully exercising the class’s methods with a wide variety of inputs. However, this is incredibly difficult to do manually. To create effective white-box test cases, you must examine the class’s internal structure, then write

test cases that will cover all of the class's methods as fully as possible, and uncover inputs that will cause the class to crash. Achieving the scope of coverage required for effective white-box testing mandates that a significant number of paths are executed. For example, in a typical 10,000 line program, there are approximately 100 million possible paths; manually generating input that would exercise all of those paths is infeasible.

After these test cases are created, you should execute the entire test suite and analyze the results to determine where errors, crashes, and weaknesses occur. You should also gauge coverage to determine how thoroughly the class was tested and to determine what additional test cases are necessary.

Any time that the class is modified, you should perform regression testing to ensure that no new errors were introduced and/or that previous errors were corrected. Regression testing involves running the same white-box and black-box test cases that were run during the initial test, and analyzing results to determine whether or not changes have caused this class's quality to regress.

C++Test: An Automatic Unit Testing Solution

Because ParaSoft recognized both the value and difficulty inherent in C/C++ unit testing, we added C++Test, an automatic unit testing tool for C and C++, to our line of automatic error prevention and error detection tools. C++Test automates every part of unit testing that can possibly be automated. Specifically, it automatically:

- Builds a harness for each class that it tests.
- Creates any necessary stubs, and allows you to customize these stubs' return values or enter your own stubs.
- Performs all steps involved in white-box testing with the single click of a button.
- Generates test cases that can be used as a base set of black-box test cases.
- Runs black-box test cases.
- Generates outcomes for black-box inputs.
- Performs regression testing.
- Tracks test coverage.

C++Test tests all types of C and C++ projects; C++Test even supports COM objects, allowing you to perform automatic unit testing on classes and methods that call to COM object methods.

C++Test is also highly customizable; for example, you can alter test case generation parameters, prevent certain files, classes, or methods from being tested, and test at any level from a project to a single method or test case.

In addition, C++Test can easily be incorporated into your existing development process. C++Test installs directly into DevStudio so that you can instantly test any class that you are working on. Just click the **Test File** or **Test Project** button in the Developer Studio tool bar, then C++Test will automatically open your file(s) in the C++Test GUI, build a harness for each class, and automatically test each class.

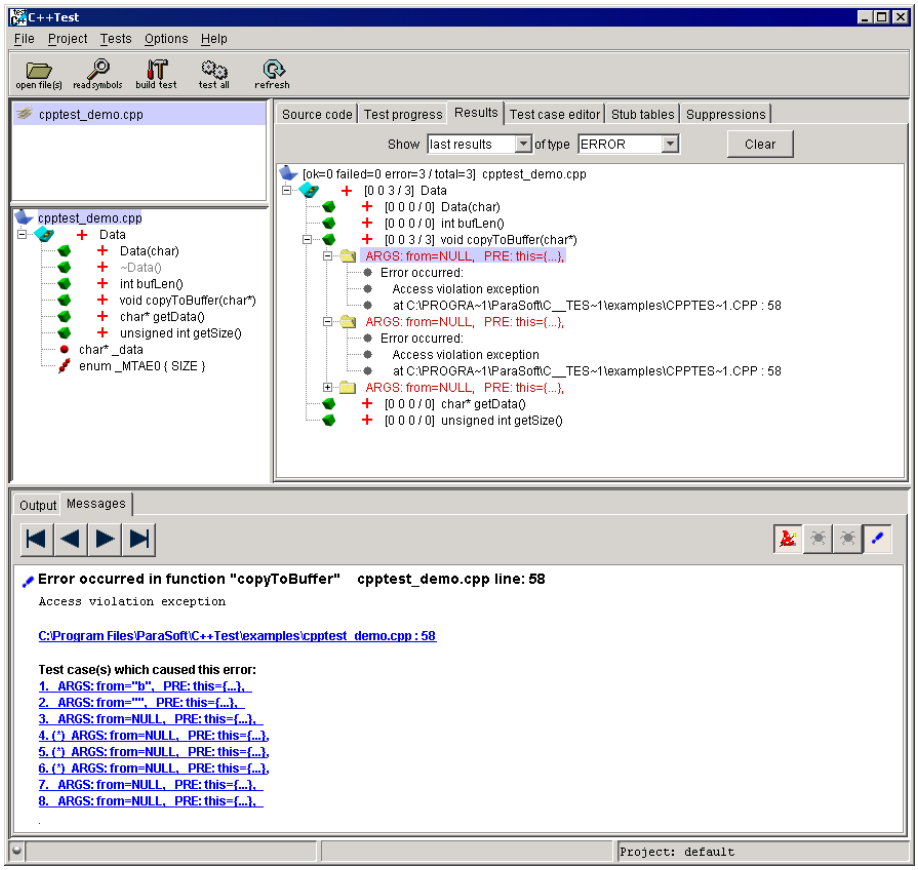


Figure 3: C++Test GUI

By automating the unit testing process, C++Test makes unit testing feasible for even the most time-starved developers and affords greater precision and accuracy than manual unit testing. This translates to higher quality applications, in less time, with lower development, support, and maintenance costs.

What Does C++Test Do?

I. Build a Harness and Generate Stubs

C++Test automatically builds a test harness that is designed to maximize class coverage and error detection. To build a harness for a class, you simply have to open the class, then click the **Build Test** button. C++Test will then automatically create a harness for the class.

In addition, C++Test automatically generates stub functions when a method under test calls a function that is not available or accessible; this allows it to test that interactions with external resources operate correctly and do not contain any hidden weaknesses. Rather than actually calling the function, C++Test calls the stub and uses the return values that the stub provides. If you want to control what return values are used, you can create a stub table that specifies input/outcome correlations.

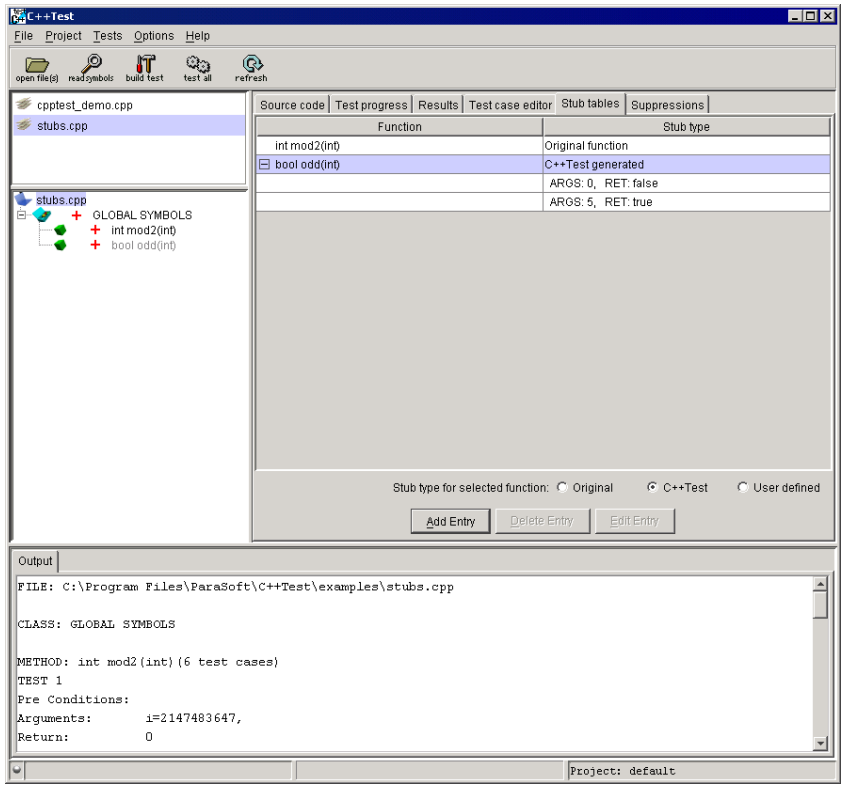


Figure 4: Creating a Stub Table

You can also enter user-defined stubs (for example, if you want to use the original function and this function is defined in a different file, or if you want to simulate the behavior of the original function by replacing the original function with a simplified one).

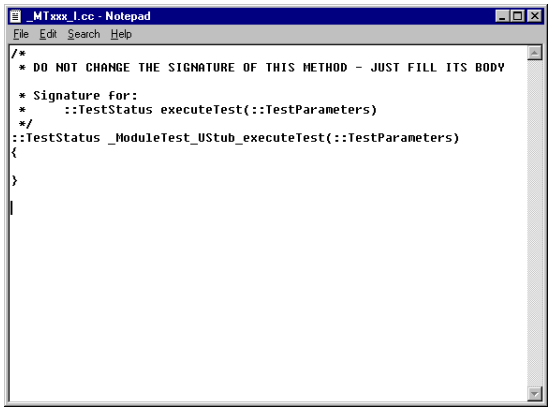


Figure 5: Entering User-Defined Stubs

This ability to automatically generate harnesses and stubs for C/C++ classes allows C++Test to test C/C++ classes as soon as they are compiled, without requiring any user intervention. This lets you automatically detect coding errors as soon as possible, when they are the easiest, cheapest, and fastest to pinpoint and repair. Without such automation, unit testing would likely be sacrificed because of the time and resources it would consume, and all potential benefits of unit testing would be lost.

II. White-Box Testing

C++Test provides an effective and efficient way to perform white-box testing. C++Test completely automates all phases of white-box testing; with the click of a button, C++Test automatically generates and executes test cases designed to thoroughly test the specified class. Any crashes found are automatically flagged and presented in a simple graphical structure. These test cases are then automatically saved so that they can easily be used for regression testing.

Because C++Test automatically generates stubs (or lets you enter your own stubs) when classes reference unavailable or inaccessible external resources, it can even test the construction of classes that reference external objects. In other words, you can run any class or set of classes through C++Test, and C++Test will automatically generate and execute a sophisticated set of test cases designed to uncover as many defects as possible.

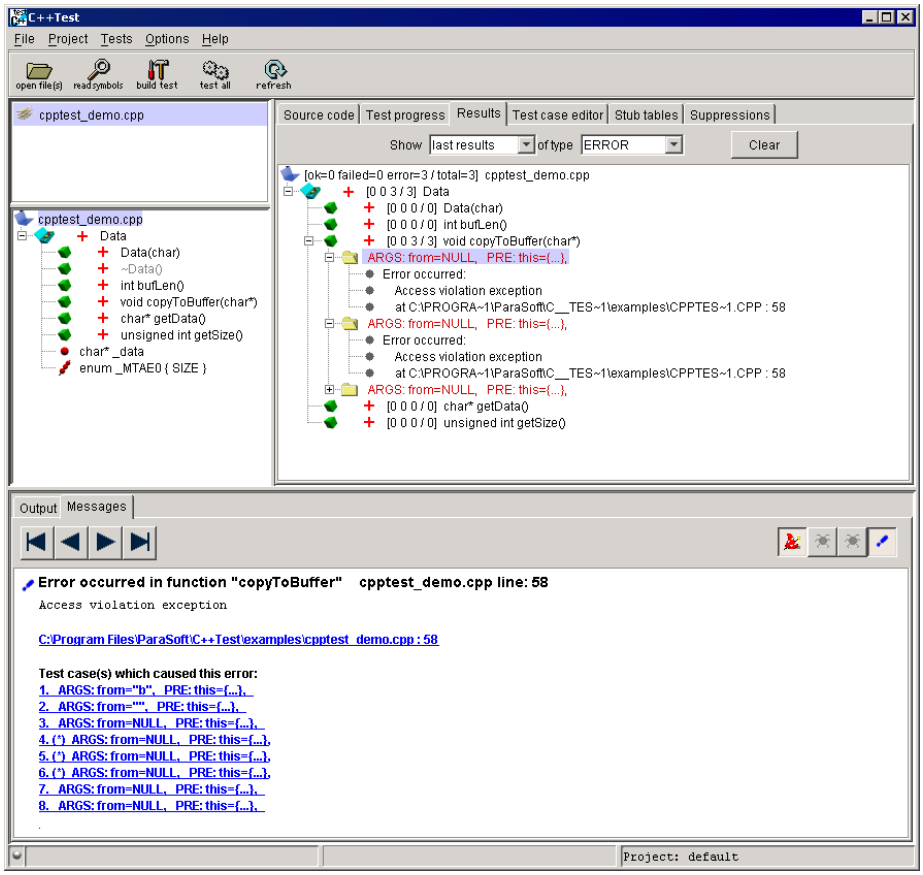


Figure 6: Performing Automatic White-Box Testing

C++Test lets you customize how white-box test cases are generated, and at what level the test is performed (project, file, class, or method).

III. Black-Box Testing

C++Test reduces the burden of black-box testing by automating many of the processes involved in this type of testing. C++Test partially automates the first phase of black-box testing— building test cases— in two ways:

1. **By relieving you from having to enter the outcome for each test case**.

   You can simply enter test case inputs, then have C++Test run the test cases and automatically determine the actual outcomes. If an outcome is correct, no action is required. If an outcome is incorrect, you can enter the expected outcome. This is much faster and easier than manually entering the outcomes for every test case.

2. **By automatically generating a core set of test cases**.

   C++Test automatically designs a wide spectrum of test cases when it performs white-box testing. To use these test cases for black-box testing, you simply need to look at the actual outcomes, then enter the expected values for any incorrect outcomes.

When entering or modifying test cases, you have the convenience of simply typing values into the test case skeleton that C++Test generates automatically. This significantly speeds up the tedious process of entering test case after test case.

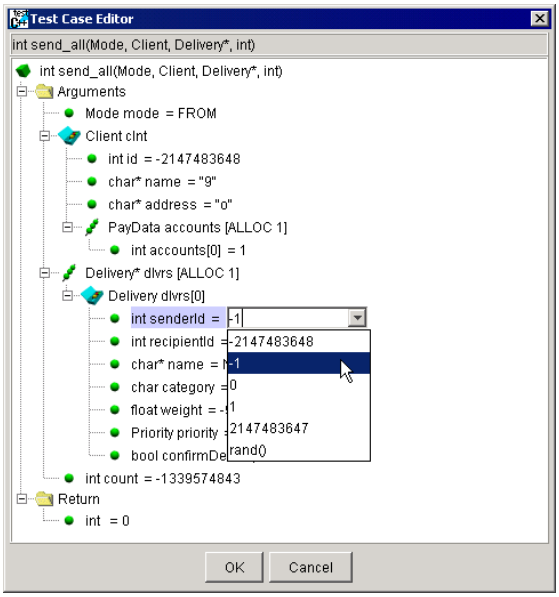


Figure 7: Entering a Test Case

In addition to automating many of the steps involved in building a black-box test suite, C++Test fully automates the subsequent steps of black-box testing. With the click of a button, you can run tests at the project, file, class, or method level, or run a single test case. C++Test then automatically executes all of the test cases, reports all input/outcome correlations, and flags any test cases whose actual outcome differs from the expected outcome or that result in a crash.

III. Regression Testing

C++Test completely automates all steps involved in and related to regression testing. The first time that C++Test tests a class, it saves the tests and test parameters. When you are ready to perform regression testing, you can run all previous white-box and black-box test cases by opening the appropriate project or file(s) and clicking a button;

C++Test then automatically runs the exact same test cases, with the exact same test parameters, and alerts you to any problems that it finds. This means that you can instantly tell whether or not modifications introduced any errors.

IV. Coverage Monitoring

To help you gauge the effectiveness of your current test suite and to give you the information that you need to achieve the greatest possible coverage, C++Test automatically monitors test coverage.

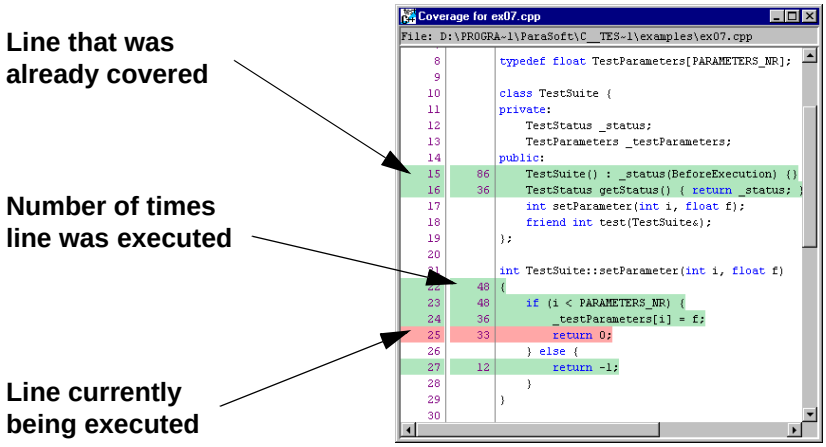


Figure 8: Viewing Test Coverage

C++Test tracks coverage in real-time, then creates a summary coverage report. The coverage window graphically specifies the line currently being executed, the lines that were already executed, and the number of times each line was executed. Thus, it not only indicates whether or not a line has been tested, but also how thoroughly it has been tested. This information is helpful in determining what areas of the code would benefit from additional testing.

V. Single-Stepping With a Debugger

C++Test also makes it easy to single-step through a method as soon as its class is compiled; if you choose to attach a debugger to a method's test, C++Test will automatically launch the Microsoft Visual C++ debugger so that you can easily single-step through any method that you are testing in C++Test.

VI. Error Prevention

You can prevent errors by using ParaSoft's CodeWizard® to automatically enforce industry-wide and custom static coding standards— rules designed to reduce the possibility of introducing errors, as well as increase code portability and maintainability. CodeWizard integrates seamlessly with C++Test: C++Test can automatically run every file that it builds though CodeWizard, then report coding standard violations found within the C++Test GUI.

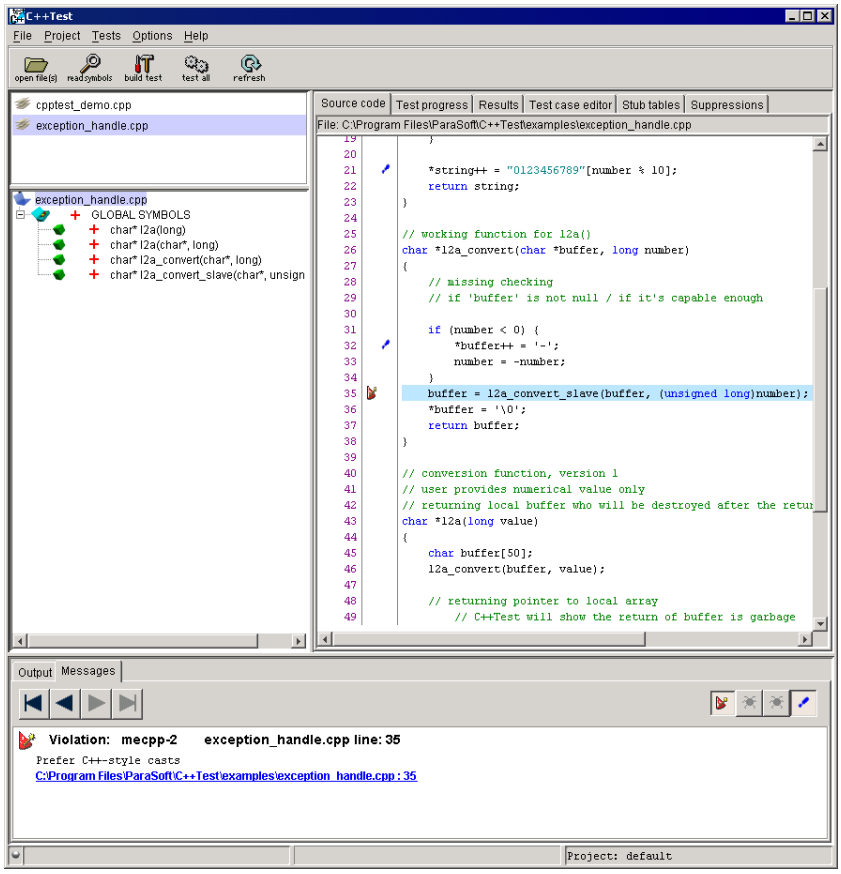


Figure 9: Preventing Errors With CodeWizard

For more information on CodeWizard, see our "Preventing Errors in C/C++" paper.

VII. Runtime Error Detection

C++Test can also help you perform automatic runtime error detection at the class level. If ParaSoft's Insure++® is installed on your system, C++Test can be configured to automatically run classes and methods through Insure++ as they are being tested by C++Test. You will then be alerted to the following types of errors in your classes:

- Memory corruption/uninitialized memory
- Memory leaks
- Memory allocation errors
- Variable definition conflicts
- I/O errors
- Pointer errors
- Library errors
- Logic errors
- Algorithmic errors

This means that you can automatically perform thorough runtime error detection on a class as soon as it is compiled. Like construction and specification errors, runtime errors can then be fixed at the point where it is easiest, fastest, and cheapest to do so.

For more information on Insure++, see our "Insure++: A Tool to Support Total Quality Software" paper.

Conclusion

By performing unit testing, you can prevent many errors from ever occurring, detect existing errors as early as possible, and detect errors more effectively than you could with other testing techniques and technologies. The main barrier preventing developers from performing C/C++ unit testing has been the time and resources that are consumed by performing unit testing with the currently available tools. This barrier is removed with the release of C++Test. C++Test does what developers always wanted to do, but did not believe could be done: automate C/C++ unit testing. C++Test's revolutionary ability to automate the C/C++ unit testing process makes it an incredibly powerful development tool that belongs in every serious developer's tool set.

Availability

C++Test is available now at www.parasoft.com/products/ctest. To learn more about how C++Test and other ParaSoft development tools can help your department prevent and detect errors, talk to a Software Quality Specialist today at 1-888-305-0041, or visit www.parasoft.com.

Contacting ParaSoft

USA

2031 S. Myrtle Ave.
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626)305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 1 64 89 26 00
UK: Tel: +44 171 288 66 00
Germany: Tel: +49 (0) 78 05 95 69 60
Email: info-europe@parasoft.com

References

W. Humphrey. *A Discipline for Software Engineering*. Reading, MA: Addison Wesley, 1995.

A. Hunt and D. Thomas. *The Pragmatic Programmer.* Reading, MA: Addison Wesley, 2000.

Last Updated 4/27/01