Dr. Dobb's Journal

July 2013

Unit Testing in C: Tools and Conventions

Next

ALSO INSIDE

UBM Tech

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

July 2013

COVER ARTICLE

FEATURES

More on DrDobbs.com

C++ Reading List

The new C++11 standard has led to a flood of new books and updates to classics. These are the core books you need.

http://www.drdobbs.com/240155654

Orthodoxy vs. Pragmatism, or How I Became a Better Developer

Few indeed are the practices that can be recommended universally. Wisdom suggests the humility of knowing this and applying it liberally.

http://www.drdobbs.com/240155112

Exceeding Supercomputer Performance with Intel Phi

Using MPI on inexpensive clusters of Intel Xeon Phi coprocessors can produce results that exceed the performance of today's high-end supercomputers.

http://www.drdobbs.com/240155836

Detailed Profiling of SQL Activity in MySQL 5.6

MySQL's latest update to the Performance Schema brings the ability to profile a statement's activity, low-level wait events, and I/O impact. It is the easiest and most detailed way to identify what statements to tune and how.

http://www.drdobbs.com/240154959

Reference Counting and Object Harvesting in Delphi

The latest versions of Delphi reclaim unused object and memory automatically. This feature requires a new perspective on calls to `Free` and to destructors.

http://www.drdobbs.com/240155664

Mailbag

In response to Andrew Binstock's editorial "Orthodoxy versus Pragmatism, or How I Became A Better Developer" (http://www.drdobbs.com/240155112), which discusses the dangers of excessive adherence to coding rules and practices, we received these comments.

"I think this is true in any aspect of life: Wisdom comes from an embrace of universal truths (often truisms, like 'test your code' or 'be kind to your neighbors'), coupled with the knowledge of how to thread the needle between idealism and pragmatism. The latter is what takes years of experience.

Since knowing when to embrace pragmatism is so hard to explain, I try to focus on why embrace pragmatism. With testing, if you have a clear understanding of what errors you should be afraid of, and which ones you shouldn't, you will gravitate toward more meaningful tests. Or to put it another way, a 100% code coverage mentality is an indication that you don't know why you are testing, so you use a scattershot approach.

I think it helps to see programming as technical writing. Most college graduates understand that writing is a creative process, that the rules should be followed except when there's a reason to break them, and that ultimately the measure of success is the impact on the audience. With programming, there are two audiences: the computer (in the service of the user), and the maintainer of the code. If you ask yourself who will read your code, and why, and if you reflect on this when you read code, you will learn to write more legible code. For example, I've found that I tend to read my code with a stack trace in hand-- jumping straight to a line without having bothered to check the javadoc or introductory comments. This has led me to make my code more legible with a stack trace. That means better variable names, more precise

method names, clearer API contracts, and throwing an exception immediately when I get bad data, rather than waiting for a `NullPointerException` to be triggered 100 lines later in an unrelated class."

— dleppik554

"The discipline is not called (by some) Software Engineering for nothing. Quality, Performance, and Cost are talked of as constraints. Be careful here; this is where it all falls apart. Compromise on ANY of these in a safety-critical system and lives are put at risk; in a commercial system, £millions are at risk.

Quality and Performance are the measure of a software engineer and the work he/she produces and should NOT be constrained in any way. Put it another way: How much rubbish software is there out there? How does that reflect on us as a so-called profession?

Some would argue that Software Engineering has yet to become a mature discipline and we are seeing the results of a still adolescent subject. We need to do something about that FAST, and a dogmatic approach to methodology would be a reasonable short-term solution. But we do need to mature as a profession, with high standards AND our own internal monitoring of training and practice. Until that happens, there will always be too much freedom to fall short of our ideals!"

— Richard Taylor
New Milton, UK

Andrew Binstock Responds: "Thanks for your thoughtful comments. I understand your overarching point and tend to agree, but I disagree in the details. For example, you wrote: 'Quality and Performance are the measure of a software engineer and the work he/she produces and should NOT be constrained in any way.'

This suggests you've not managed a product with a tight budget. Budget is a serious constraint on nearly all projects. That's no different from medicine, hard sciences, or any of the engineering disciplines. You always have to work within a budget."

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

News Briefs

By Adrian Bridgwater

Apache CloudStack 4.1.0 Released Into The Wild

The Apache CloudStack project has announced the 4.1.0 release of the CloudStack Infrastructure-as-a-Service (IaaS) cloud orchestration platform. This is the first major release from Apache CloudStack since its graduation from the Apache Incubator on March 20 of this year. New features include an API discovery service that allows an end point to list its supported APIs and their details. The team has added an "events framework" to provide an "event bus" with publish, subscribe, and unsubscribe semantics.

http://www.drdobbs.com/240156253

Enriched ALM In Visual Studio 2013 and Team Foundation Server 2013

Microsoft has this week staged its TechEd technical user conference and announced previews of Visual Studio 2013 and Team Foundation Server 2013 with a specific focus upon the products' Application Lifecycle Management features. Focusing initially on the Application Lifecycle workflow features of VS 2013, we learn that there is continuing focus on Agile project management features (including backlog and sprint management) first seen in TFS 2012 — these efforts join the Kanban support Microsoft added in the TFS 2012 updates. Core coding functionality has been addressed with a new "heads up display" feature in Visual Studio. This is designed to provide much-needed insights into code while working. There is a selection of "indicators" here now and the team says that it will be adding more over time.

http://www.drdobbs.com/240156139

A Blueprint For Requirements Definition and Management

Blueprint's 5.2 version Requirements Definition and Management (RDM) software has arrived with 50 new enhancements and performance improvements. The software features *Blueprint for Enterprise Agile* as a new packaged solution and add-on to help solve what its makers call "the fundamental issues" encountered when transitioning from traditional waterfall-style processes to more agile frameworks.

http://www.drdobbs.com/240156194

Rapid Iteration Is Better Iteration

Skytap Automation Pack for IBM Rational Team Concert is now available as a tool for DevOps teams to provision IBM build machines on-demand in the Skytap cloud. This integration is designed to work with IBM's own SmartCloud Continuous Delivery solution to enable elastic cloud resources that meet sporadic demand for build machine capacity.

http://www.drdobbs.com/240155942

Free Code Review Shop For Ten-In-A-Team Developers

SmartBear is making its code review technology available in two new solutions formats: one for the SoapUI community, and one positioned for small companies and development teams. CodeReviewer will allow software application development teams of up to 10 members to review code at no cost. But of course, for teams of up to 25 developers, CodeReviewer Pro is available with wider code review functionality and full technical support.

http://www.drdobbs.com/240155794

Unit Testing in C: Tools and Conventions

Two lightweight testing frameworks make it easy to unit test C code.

By James W. Grenning

In this article, I look at unit tests using two unit test harnesses that work in C. Along the way, I will also discuss some of the common terminology of automated unit testing. Let me start by discussing the fundamental tool, the test harness.

What Is A Unit Test Harness?

A unit test harness is a software package that allows a programmer to express how production code should behave. A unit test harness's job is to provide these capabilities:

- A common language to express test cases
- A common language to express expected results
- Access to the features of the production code programming language
- A place to collect all the unit test cases for the project, system, or subsystem
- A mechanism to run the test cases, either in full or in partial batches
- A concise report of the test suite success or failure
- A detailed report of any test failures

I'll shortly look at two popular harnesses for testing embedded C. They are both easy to use and are descendants of the xUnit family of unit test harnesses.

First, I'll employ Unity (http://goo.gl/uc6pU), a C-only test harness. Later, I will use CppUTest, a unit test harness written in C++, but not requiring C++ knowledge to use. You'll find that the bulk of the material in this article can be applied using any test harness.

Here are a few terms that will come in handy while reading this explanation:

- Code under test is just like it sounds; it is the code being tested.
- Production code is code that is (or will be) part of the released product.
- Test code is code that is used for testing the production code and is not part of the released product.

- A test case is test code that describes the behavior of code under test. It establishes the preconditions and checks that significant post conditions are met.
- A test fixture is code that provides the proper environment for a series of test cases that exercise the code under test. A test fixture will assist in establishing a common setup and environment for exercising the production code.

To take the mystery out of these terms, let's look at a few tests for something we've all used: `sprintf`. For this first example, `sprintf` is the code under test; it is production code.

`sprintf` is good for a first example because it is a standalone function, which is the most straightforward kind of function to test. The output of a standalone function is fully determined by the parameters passed immediately to the function. There are no visible external interactions and no stored state to get in the way. Each call to the function is independent of all previous calls.

Unity: A C-Only Test Harness

Unity is a straightforward, small unit test harness. It comprises just a few files. Let's get familiar with Unity and unit tests by looking at a couple example unit test cases. If you are a long-time Unity user, you'll notice some additional macros that are helpful when you are not using Unity's scripts to generate a test runner.

A test should be short and focused. Think of it as an experiment that silently does its work when it passes, but makes some noise when it fails. This test checks that `sprintf` handles a format spec with no format operations.

```
TEST(sprintf, NoFormatOperations)
{
    char output[5];
    TEST_ASSERT_EQUAL(3, sprintf(output, "hey"));
    TEST_ASSERT_EQUAL_STRING("hey", output);
}
```

The `TEST` macro defines a function that is called when all tests are run. The first parameter is the name of a group of tests. The second parameter is the name of the test. We'll look at `TEST` in more detail later.

The `TEST_ASSERT_EQUAL` macro compares two integers. `sprintf` should report that it formatted a string of length three, and if it does, the `TEST_ASSERT_EQUAL` check succeeds. As is the case with most unit test harnesses, the first parameter is the expected value.

`TEST_ASSERT_EQUAL_STRING` compares two null-terminated strings. This statement declares that output should contain the string `"hey"`: Following convention, the first parameter is the expected value.

If either of the checked conditions is not met, the test will fail. The checks are performed in order, and the `TEST` will terminate on the first failure.

Notice that `TEST_ASSERT_EQUAL_STRING` could pass by accident; if the output just happened to hold the `"hey"` string, the test would pass without `sprintf` doing a thing. Yes, this is unlikely, but we better improve the test and initialize the output to the empty string.

```
TEST(sprintf, NoFormatOperations)
{
    char output[5] = "";
    TEST_ASSERT_EQUAL(3, sprintf(output, "hey"));
    TEST_ASSERT_EQUAL_STRING("hey", output);
}
```

The next `TEST` challenges `sprintf` to format a string with `%s`.

```
TEST(sprintf, InsertString)
{
    char output[20] = "";
    TEST_ASSERT_EQUAL(12, sprintf(output,
      "Hello %s\n", "World"));
    TEST_ASSERT_EQUAL_STRING("Hello World\n", output);
}
```

A weakness in both the preceding tests is that they do not guard against `sprintf` writing past the string terminator. The following tests watch for output buffer overruns by filling the output with a known value and checking that the character after the terminating null is not changed.

```
TEST(sprintf, NoFormatOperations)
{
    char output[5];
    memset(output, 0xaa, sizeof output);
    TEST_ASSERT_EQUAL(3, sprintf(output, "hey"));
    TEST_ASSERT_EQUAL_STRING("hey", output);
    TEST_ASSERT_BYTES_EQUAL(0xaa, output[4]);
}
TEST(sprintf, InsertString)
{
    char output[20];
    memset(output, 0xaa, sizeof output);
    TEST_ASSERT_EQUAL(12, sprintf(output, "Hello %s\n", "World"));
    TEST_ASSERT_EQUAL_STRING("Hello World\n", output);
    TEST_ASSERT_BYTES_EQUAL(0xaa, output[13]);
}
```

If you're worried about `sprintf` corrupting memory in front of output, we could always make output a character bigger and pass `&output[1]` to `sprintf`. Checking that `output[0]` is still `0xaa` would be a good sign that `sprintf` is behaving itself.

In C, it is hard to make tests totally fool-proof. Errant or malicious code can go way beyond the end or way in front of the beginning of output. It's a judgment call on how far to take the tests. You will see when we get into TDD how to decide which tests to write.

With those tests, you can see some subtle duplication creeping into the tests. There are duplicate output declarations, duplicate initializations, and duplicate overrun checks. With just two tests, this is no big deal, but if you happen to be `sprintf`'s maintainer, there will be many more tests. With every test added, the duplication will crowd out and obscure the code that is essential to understand the test case. Let's see how a test fixture can help `TEST` cases.

Test Fixtures in Unity

Duplication reduction is the motivation for a test fixture. A test fixture helps organize the common facilities needed by all the tests in one place. Notice how `TEST_SETUP` and `TEST_TEAR_DOWN` keep duplication out of the `sprintf` tests.

```
TEST_GROUP(sprintf);

static char output[100];
static const char * expected;
TEST_SETUP(sprintf)
{
    memset(output, 0xaa, sizeof output);
    expected = "";
}

TEST_TEAR_DOWN(sprintf)
{
}
static void expect(const char * s)
{
    expected = s;
```

```
}
static void given(int charsWritten)
{
    TEST_ASSERT_EQUAL(strlen(expected), charsWritten);
    TEST_ASSERT_EQUAL_STRING(expected, output);
    TEST_ASSERT_BYTES_EQUAL(0xaa,
      output[strlen(expected) + 1]);
}
```

The shared data items defined after the `TEST_GROUP` are initialized by `TEST_SETUP` before the opening curly brace of each `TEST`. The data items comprise file scope, accessible by each `TEST` and all the helper functions. For this `TEST_GROUP`, there is no cleanup work for `TEST_TEAR_DOWN`.

The file scope helper functions, `expect` and `given`, help keep the `sprintf` tests clean and low on duplication.

In the end, it's just plain C, so you can do what you want as far as shared data and helper functions. I'm showing the typical way to structure a group of tests with common data and condition checks.

Now these tests are focused, lean, mean, and to the point.

```
TEST(sprintf, NoFormatOperations)
{
    expect("hey");
    given(sprintf(output, "hey"));
}

TEST(sprintf, InsertString)
{
    expect("Hello World\n");
    given(sprintf(output, "Hello %s\n", "World"));
}
```

Notice that once you understand a specific `TEST_GROUP` and have seen a couple examples, writing the next test case is much less work. When there is a common pattern within a `TEST_GROUP`, each test case is easier to read, understand, and evolve, as change becomes necessary.

Installing Unity Tests

It is not evident from the example how the test cases get run with the necessary pre- and post-processing. It's done with another macro: the `TEST_GROUP_RUNNER`. The `TEST_GROUP_RUNNER` can go in the file with the tests or a separate file. To avoid scrolling through the file, I use a separate file. For the two sprintf tests written, the `TEST_GROUP_RUNNER` looks like this:

```
#include "unity_fixture.h"
TEST_GROUP_RUNNER(sprintf)
{
    RUN_TEST_CASE(sprintf, NoFormatOperations);
    RUN_TEST_CASE(sprintf, InsertString);
}
```

Each test case is called through the `RUN_TEST_CASE` macro. Essentially, this `RUN_GROUP_RUNNER` calls the function bodies associated with each of these macros:

```
TEST_SETUP(sprintf);
TEST(sprintf, NoFormatOperations);
TEST_TEAR_DOWN(sprintf);

TEST_SETUP(sprintf);
TEST(sprintf, InsertString);
TEST_TEAR_DOWN(sprintf);
```

Invoking `TEST_SETUP` before each `TEST` means that each test starts out fresh, with no accumulated state. `TEST_TEAR_DOWN` is called to clean up after each test.

Now that the tests are wired into a `TEST_GROUP_RUNNER`, let's see how the `TEST_GROUP_RUNNER`s are called. For this last step, we have to look at `main`. You will have a `main` for your production code and one, or more, for your test code. The Unity test `main` looks like this:

```
#include "unity_fixture.h"

static void RunAllTests(void)
{
    RUN_TEST_GROUP(sprintf);
}

int main(int argc, char * argv[])
{
    return UnityMain(argc, argv, RunAllTests);
}
```

`RUN_TEST_GROUP(GroupName)` calls the function defined by `TEST_GROUP_RUNNER`. Each `TEST_GROUP_RUNNER` you want to run as part of your test `main` has to be mentioned in a `RUN_TEST_GROUP`. Notice that `RunAllTests` is passed to `UnityMain`.

One unfortunate side effect of using a C-only test harness is that you have to remember to install each `TEST` into a `TEST_GROUP_RUNNER`, and the runner is invoked by calling `UnityMain`. If you forget, tests will compile, but not run — potentially giving a false positive.

Because of this opportunity for error, the designers of Unity created a system of code generators that read your test files and produce the needed test runner code. To keep the dependencies low for getting started with Unity, I've opted to not use the code-generating scripts and manually wire all the test code.

When I discuss CppUTest later in this article, you will see another solution to that problem. But before doing that, let's look at Unity's output.

Unity Output

The tests should be run as part the automated test build. A single command builds and runs your test executable. I prefer to build often, with each small change. This is TDD. I set up my development environment to automatically `make all` whenever a file is saved. Test output looks like this:

```
make
compiling SprintfTest.c
Linking BookCode_Unity_tests
Running BookCode_Unity_tests
..
-----------------------
2 Tests 0 Failures 0 Ignored
OK
```

Notice that when all tests are passing, the output is minimal. At quick glance, a single line of text says "`OK`," which means, "All tests passing." In the UNIX style, the test harness follows the "no news is good news" principle. (When a test case fails, as you will see shortly, it reports a specific error message.) It's pretty self-explanatory, but let's decipher the test output and summary line.

Notice also that a dot (`.`) is printed before each test case runs. For a long test run, this lets you know something is happening. The line of hyphens (`- - -`) is just a separator line for the test summary.

- Tests — the total number of `TEST` cases.
- Failures — the total number of `TEST` cases that failed.

- Ignored — a count of the number of tests in ignore state. Ignored tests are compiled but are not run.

Let's add a failing test to see what happens. Look at the test output, and the intentional error in this test case will be evident:

```
TEST(sprintf, NoFormatOperations)
{
    char output[5];
    TEST_ASSERT_EQUAL(4, sprintf(output, "hey"));
    TEST_ASSERT_EQUAL_STRING("hey", output);
}
```

The failure looks like this:

```
make
compiling SprintfTest.c
Linking BookCode_Unity_tests
Running BookCode_Unity_tests
..
TEST(sprintf, NoFormatOperations)
    stdio/SprintfTest.c:75: FAIL
    Expected 4 Was 3
-----------------------
2 Tests 1 Failures 0 Ignored
FAIL
```

The failure reports the filename and line of the failing test case, the name of the test case, and the reason for failure. Also notice the summary line now shows one test failure.

Now, let's look at CppUTest.

CppUTest: A C++ Unit Test Harness

Now that we've seen Unity, I'll quickly describe CppUTest, my preferred unit test harness for C and C++. In full disclosure, I am partial to CppUTest, not only because it is a capable test harness but also be-

cause I am one of its authors. The first examples in this article use Unity. The later examples use CppUTest.

CppUTest was developed to support multiple OS platforms with a specific goal of being usable for embedded development. The CppUTest macros make it so that test cases can be written without knowledge of C++. This makes it easy for C programmers to use the test harness.

CppUTest uses a primitive subset of C++; it's a good choice for embedded development where not all compilers support the full C++ language. You will see that the test cases are nearly identical between Unity and CppUtest. You, of course, can use whichever test harness you prefer for your product development.

This CppUTest test case is equivalent to the second Unity test case found in the section on `sprintf`.

```
TEST(sprintf, NoFormatOperations)
{
    char output[5] = "";

    LONGS_EQUAL(3, sprintf(output, "hey"));
    STRCMP_EQUAL("hey", output);
}
```

Besides the macro names, the test cases are the same.

Let's look at a CppUTest test fixture that is equivalent to the earlier example Unity test fixture (discussed in the "Test Fixtures in Unity" section).

```
TEST_GROUP(sprintf)
{
    char output[100];
    const char * expected;

    void setup()
    {
        memset(output, 0xaa, sizeof output);
        expected = "";
    }
    void teardown()
    {
    }
    void expect(const char * s)
    {
        expected = s;
    }
    void given(int charsWritten)
    {
        LONGS_EQUAL(strlen(expected), charsWritten);
        STRCMP_EQUAL(expected, output);
        BYTES_EQUAL(0xaa, output[strlen(expected) + 1]);
    }
};
```

Again, it is very similar to the previous example, with all the same concepts represented. One formatting difference is that the CppUTest `TEST_GROUP` is followed by a set of curly braces enclosing shared data declarations and functions. Everything between the curly braces is part of the `TEST_GROUP` and is accessible to each `TEST` in the group. The shared data items (`output`, `expected`, and `length`) are initialized by a special helper function called `setup`. As you might guess, `setup` is called before each `TEST`. Another special function, `teardown`, is called after each `TEST`. (In this example, it is not used.) `expect` and `given` are free-form helper functions that are accessible to all `TEST` cases in the `TEST_GROUP`.

These refactored test cases are identical to the Unity test cases:

```
TEST(sprintf, NoFormatOperations)
{
    expect("hey");
    given(sprintf(output, "hey"));
}
```

```
TEST(sprintf, InsertString)
{
    expect("Hello World\n");
    given(sprintf(output, "%s\n", "Hello World"));
}
```

One advantage to CppUTest is that tests self-install. There is no need for an external script to generate a test runner or to manually write and maintain test-wiring code like `RUN_TEST_CASE`, `TEST_GROUP_RUNNER`, and `RUN_TEST_GROUP`. On the minor difference list are the assertion macros; each test harness supports different macros, though there is functional overlap.

You may notice that Unity and CppUTest are suspiciously close in their macros and test structure. Well, there is no real mystery there; they do follow a well-established pattern that I first saw with JUnit, a Java test framework. The more specific similarities are because I contributed the test fixture-related macros to the Unity project.

CppUTest Output

As already explained for Unity, tests run as part of an automated build using `make`. Test output looks like this:

```
make all
compiling SprintfTest.cpp
Linking BookCode_tests
Running BookCode_tests
..
OK (2 tests, 2 ran, 0 checks, 0 ignored, 0 filtered out)
```

Just like with Unity, when all tests are passing, the output is minimal. Here is how to interpret the summary line of the test run:

- tests — the total number of `TEST` cases.
- ran — the total number of `TEST` cases that ran (in this case, they passed, too).
- checks — a count of the number of condition checks made. (Condition checks are calls such as `LONGS_EQUAL`.)
- ignores — a count of the number of tests in ignore state. Ignored tests are compiled but are not run.
- filtered out — a count of the number of tests that were filtered out of this test run. Command-line options select specific tests to run.

Let's insert an error into the test to see what the output looks like:

```
TEST(sprintf, NoFormatOperations)
{
    char output[5];

    LONGS_EQUAL(4, sprintf(output, "hey"));
    STRCMP_EQUAL("hey", output);
}
```

The failure looks like this:

```
make
compiling SprintfTest.cpp
Linking BookCode_Unity_tests
Running BookCode_Unity_tests
...
stdio/SprintfTest.cpp:75: TEST(sprintf, NoFormatOperations)
expected <4 0x2>
but was <3 0x1>
Errors (1 failures, 2 tests, 2 ran, 1 checks, 0 ignored, 0
filtered out, 0 ms)
```

The failure reports the line of the failing condition check, the name of the test case, and the reason for failure. Also notice the summary line includes a count of test failures.

If you ever insert an error on purpose into a test case, make sure you remove it, or you risk baking a bug into your code.

Unit Tests Can Crash

One other possible outcome during a test run is a crash. Generally speaking, C is not a safe language. The code can go off into the weeds, never to return. `sprintf` is a dangerous function. If you pass it an output buffer that is too small, it will corrupt memory. This error might crash the system immediately, but it might cause a crash later. The behavior is undefined. Consequently, a test run may silently exit with an OK, silently exit early showing no errors, or crash with a bang.

When you have a silent failing or crashing test, let the test harness help you confirm what is wrong. Sometimes a production code change will cause a previously passing test to fail, or even crash. So, before chasing the crash, make sure you know which test is failing.

Because the test harness is normally quiet except for test failures, when a test crashes, you probably won't get any useful output. Both Unity and CppUTest have a command-line option for running the test in verbose mode (`-v`). With `-v`, each `TEST` announces itself before running. Conveniently, the last `TEST` mentioned is the one that crashed.

You can also filter tests by test group (`-g testgroup`) and test case (`-n testname`). This lets you get very precise about which test cases are running. These are very helpful for chasing down crashes.

The Four-Phase Test Pattern

In Gerard Meszaros' book, *xUnit Testing Patterns* (http://goo.gl/zCfHb), he describes the Four-Phase Test, which is what I use, too. The goal of the pattern is to create concise, readable, and well-structured tests. If you follow this pattern, the test reader can quickly determine what is being tested. Paraphrasing Gerard, here are the four phases:

- Setup: Establish the preconditions to the test.
- Exercise: Do something to the system.
- Verify: Check the expected outcome.
- Cleanup: Return the system under test to its initial state after the test.

To keep your tests clear, make the pattern visible in your tests. When this pattern is broken, the documentation value of the test is diminished; the reader has to work harder to understand the requirements expressed by the test.

Conclusion

At this point, you should have a good overview of Unity and CppUTest, and understand how test fixtures and test cases allow a set of tests to be defined. Whether you use them to practice TDD on your C projects or just to ensure higher code quality is entirely up to you. *[If you choose to go the TDD route, though, you might find the remainder of the material in the book from which this article is excerpted to be useful in your work. See below. —Ed.]*

— This article is excerpted and adapted from the book* Test-Driven Development for Embedded C (http://goo.gl/zRozg), *published by Pragmatic Programmers (http://www.pragprog.com). You can find the complete code by visiting the book's home page at http://goo.gl/do5Kd. James W. Grenning invented Planning Poker, an Agile estimation technique, and is one of the original authors of the Manifesto for Agile Software Development.

Comment

13 Linux Debuggers for C++ Reviewed

Most time in debuggers is spent doing the same few things: setting breakpoints, stepping through code, looking at variables. Which products make those features supremely accessible and useful? We compare 13 debuggers and find out.

By Howard Rubin

Have you compared debuggers lately? Until recently, I'd been programming using only one debugger — the one supplied by my compiler vendor. Suddenly, with a new job programming on Linux, I find the range of choices in debuggers is dizzying. Wikipedia lists 18 GUI front ends for GDB alone (http://is.gd/kLityg). This article is the result of my effort to choose a debugger with a good GUI front end for my first UNIX/Linux job in several years.

Scope

There are many tools that provide functionality related to debugging. Examples include memory-leak detectors, memory-corruption detectors, profilers, and GDB enhancers such as undoDB (http://is.gd/59DU0G). And with multithreaded programming going mainstream, there are single-purpose tools to help diagnose deadlock and race conditions. Those tools, while valuable, are not part of this review — I'm solely focusing on debuggers, and specifically, the user-friendliness of the interfaces. The focus on the user experience is intentional and specific: I find that the vast majority of the time in the debugger involves two principal activities — stepping through code and examining variables. In the next section, I show a screen capture of each product and I comment on how well it provides these functions in addition to how well it enables me to move and dock tabs so that I can maximize my screen space.

In subsequent sections, I rate the products on other common features, such as setting breakpoints, customizing variable display, and so on. I also explain my thinking on what makes good implementations of some of these features now that I've seen them implemented across these 13 products, some standalone and some embedded in IDEs.

Products

Affinic Technology's **Affinic 0.5.3** (http://www.affinic.com/), recently updated to version 1.0, is a commercial GUI wrapper for GDB. It can be downloaded and tried out for free. A commercial license costs $49. Versions for Linux and Windows (via Cygwin) are available. Affinic uses single-button hotkeys and buttons for stepping through the target program. Its variable display is usable but minimal. Its tabbed docking windows would be good if they remembered their position from one debugging session to the next. The "variable display tooltip" is bare bones: Hovering the cursor over a variable displays as much of its value as will fit in the single status line at the bottom of the main window.

Code::Blocks 12.11 (http://www.codeblocks.org/) is a free, open-source product that uses a plugin model to add various capabilities, which I'll touch on later. It offers tabbed docking windows, but they won't dock together into user-selected groups. Its variable display understands STL vectors, but not STL strings. Its tooltips show variable values for only the simplest data types, and its program console window isn't part of its docking system.

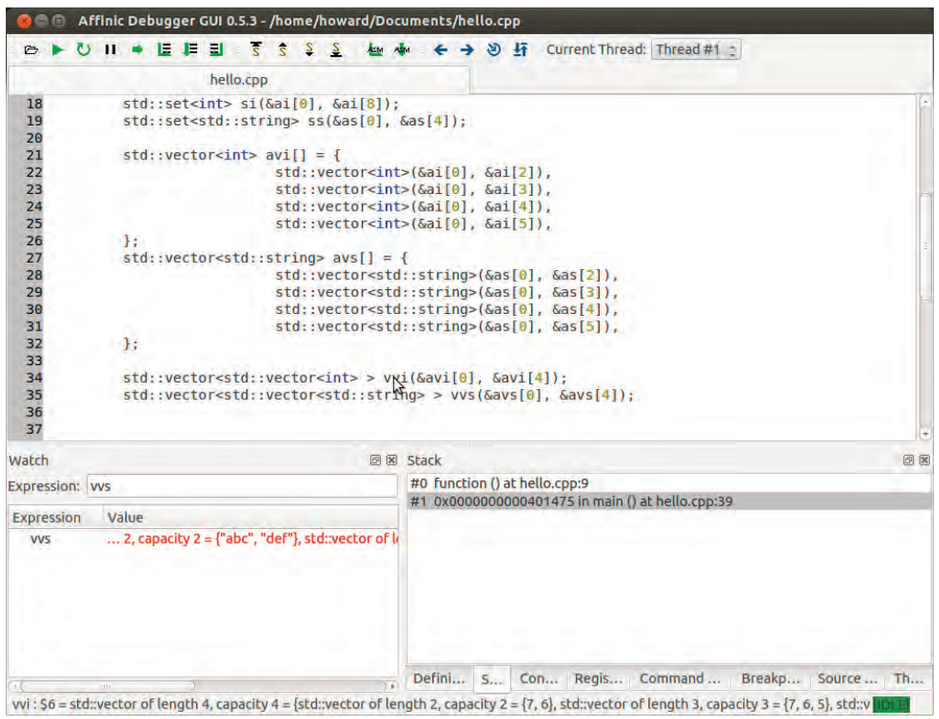

Figure 1: The Affinic debugger.

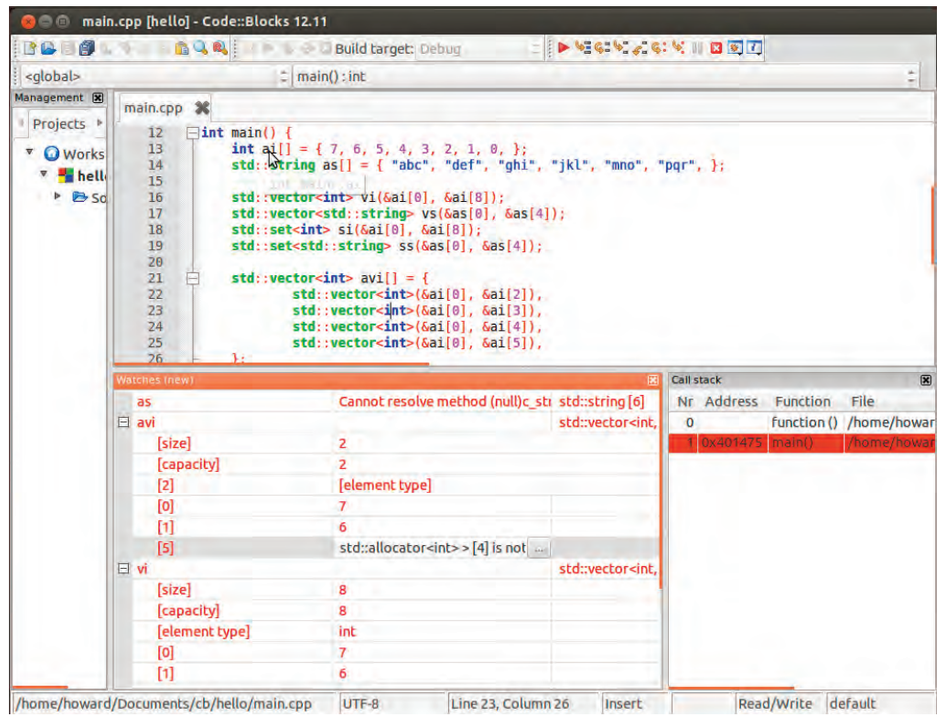

Figure 2: Code::Blocks.

Codelite 5.1 (http://codelite.org) is an open-source C++ IDE that runs on Windows, Linux, Mac OS X, and FreeBSD among other platforms. It has tabbed docking windows, but they resist efforts to form user-selected groups. Also, the product doesn't understand nested data structures like an array of vectors, and its Quick Debug mode doesn't remember breakpoints from one debug session to the next.

DDD 3.3.12 (http://www.gnu.org/software/ddd/) is the GNU project's standard GUI front end to GDB and its other language debuggers. In many ways, it is a minimalist front end. It has single-button hotkeys for stepping through debugged program execution, but it lacks a toolbar with buttons for those commands. It displays nested data structures on a single line, which makes complex data items hard to understand. It sometimes gets stuck starting up, with an hourglass cursor that never changes back to normal.

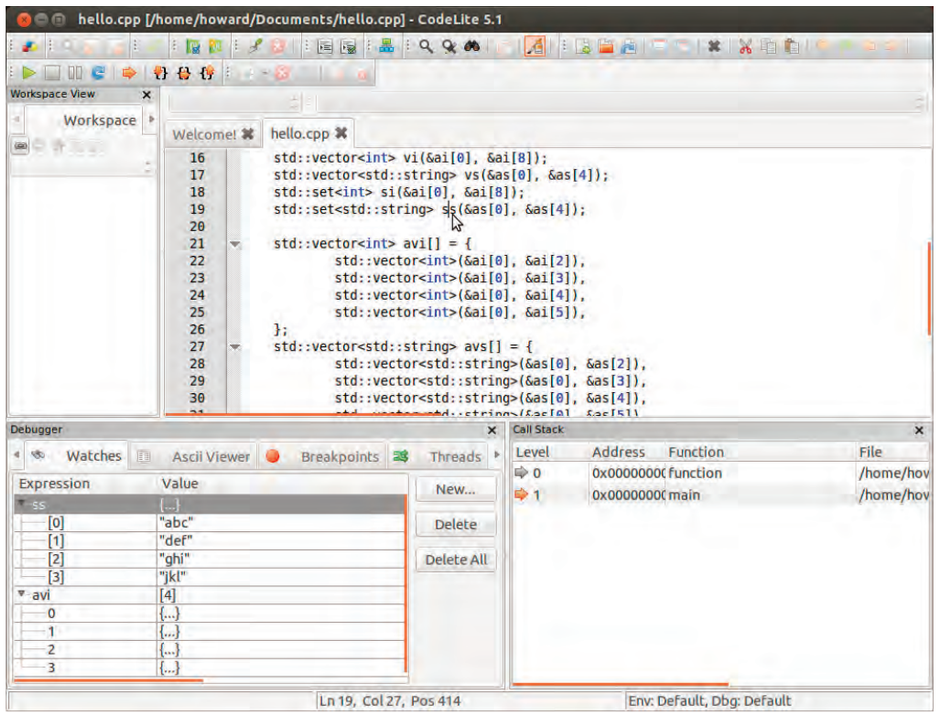

Figure 3: Codelite 5.1.

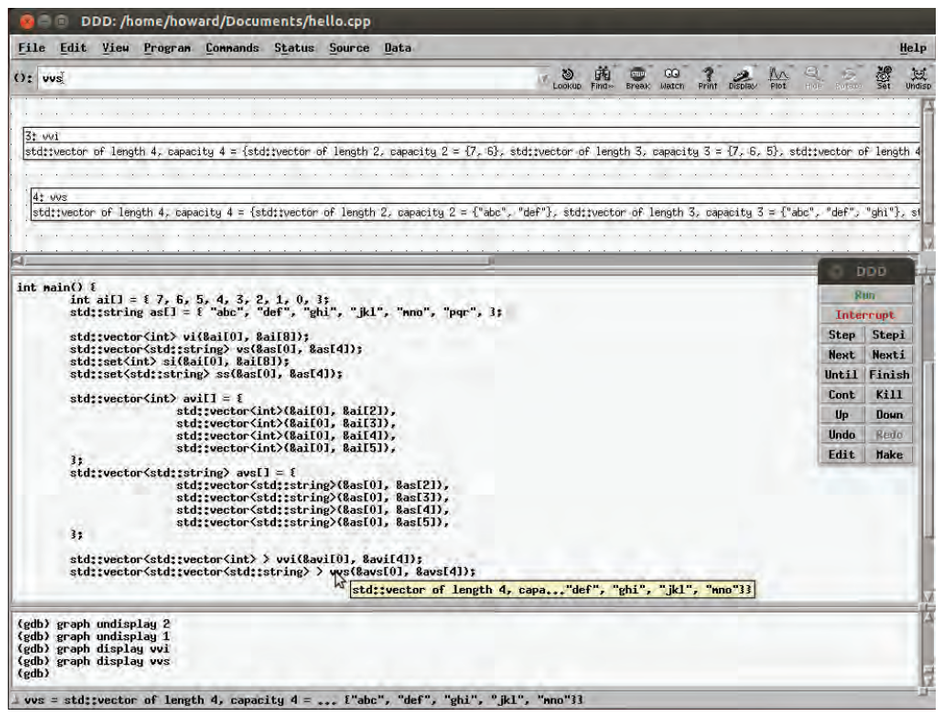

Figure 4: Gnu DDD.

Eclipse (http://www.eclipse.org) is the Java IDE that, through continued development and its extensive ecosystem, has grown into an environment for development in many languages, including C and C++. It has resizable and detachable tabbed docking windows, excellent support for variable display, and an adequate code declaration/definition navigator. Its single button hotkeys and toolbar buttons for stepping through the debugged program are good. It also remembers breakpoints across debug sessions, but to find out how many times a breakpoint has been ignored, you have to issue the "info break" command in the GDB console window. (While this example shows Eclipse using GDB, it supports many other compilers/debuggers: Cross-platform C/C++ development in Eclipse on Linux and Windows (http://is.gd/4yDNBp) is also an option. —*Ed.*)

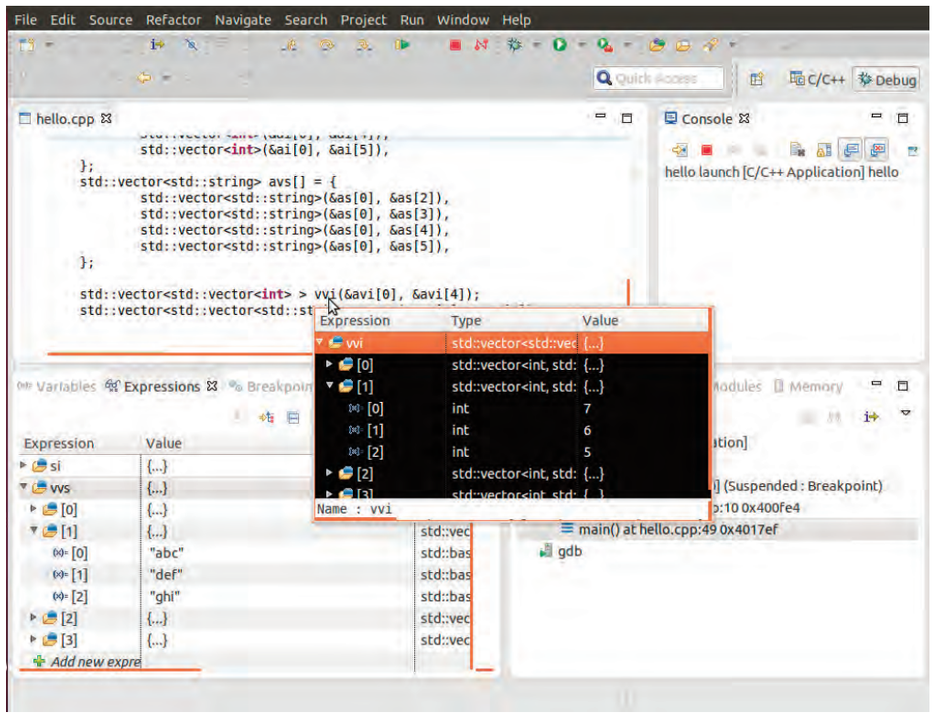

Figure 5: Eclipse (Juno release) front-ending GDB.

GNU Emacs (http://www.gnu.org/software/emacs/): As a long-time Emacs user, I wanted to like **GNU Emacs version 23.3.1's GDB mode** (http://www.emacswiki.org/emacs/GdbMode). I like many of its features, but it's starting to show some age. It has multiple windows, some with tabs (to switch to other windows), but it doesn't support rearranging them. I like its display of single-level STL containers as arrays, but for the nested container: `std::vector<std::vector<int>>` it displays only the unhelpful { . . . }.

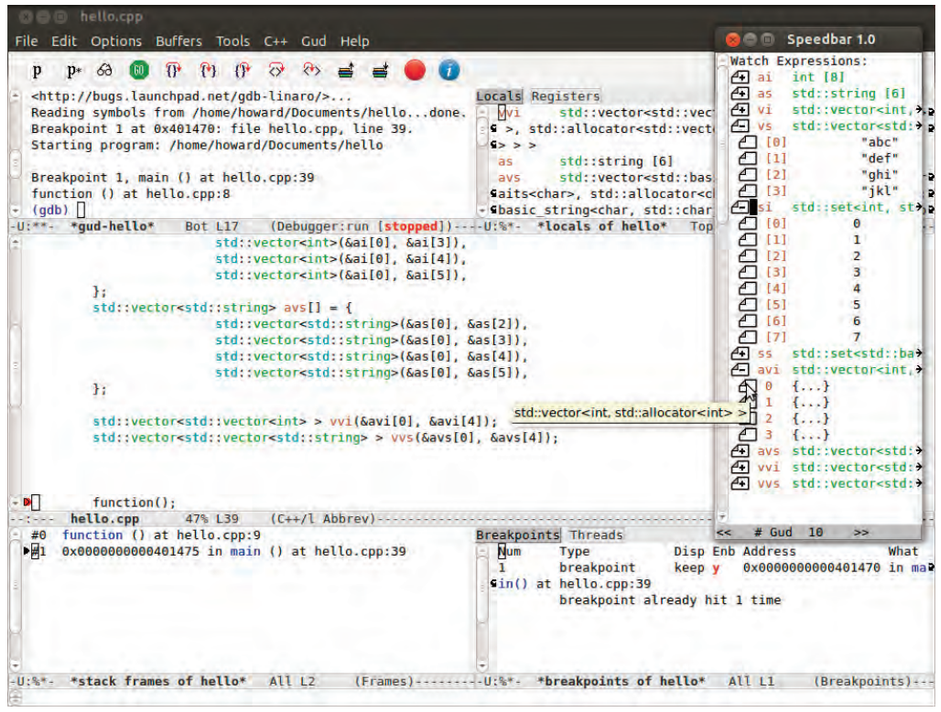

Figure 6: GNU Emacs' GDB mode.

KDevelop 4.3 (http://www.kdevelop.org) is an open-source IDE for Linux, Solaris, FreeBSD, Max OS X, and other UNIX flavors. It has single-button hotkeys, but lacks toolbar buttons for stepping through the debugged program. Its simple variable and single-level STL container support is good, but displays that unhelpful abbreviation { . . . } for nested STL containers (see Figure 7). It does possess an impressive menu of commands to navigate program symbols.

Nemiver 0.9.1 (http://projects.gnome.org/nemiver/) "is an on-going effort to write a standalone graphical debugger that integrates well in the GNOME desktop environment," according to its creators. Predictably, it's open-source. It strives to maximize user friendliness in the debugger interface. It has single-button hotkeys and a toolbar with buttons for stepping through the debugged program. It pops up a convenient window for simple variables and arrays, but has no support for STL containers.

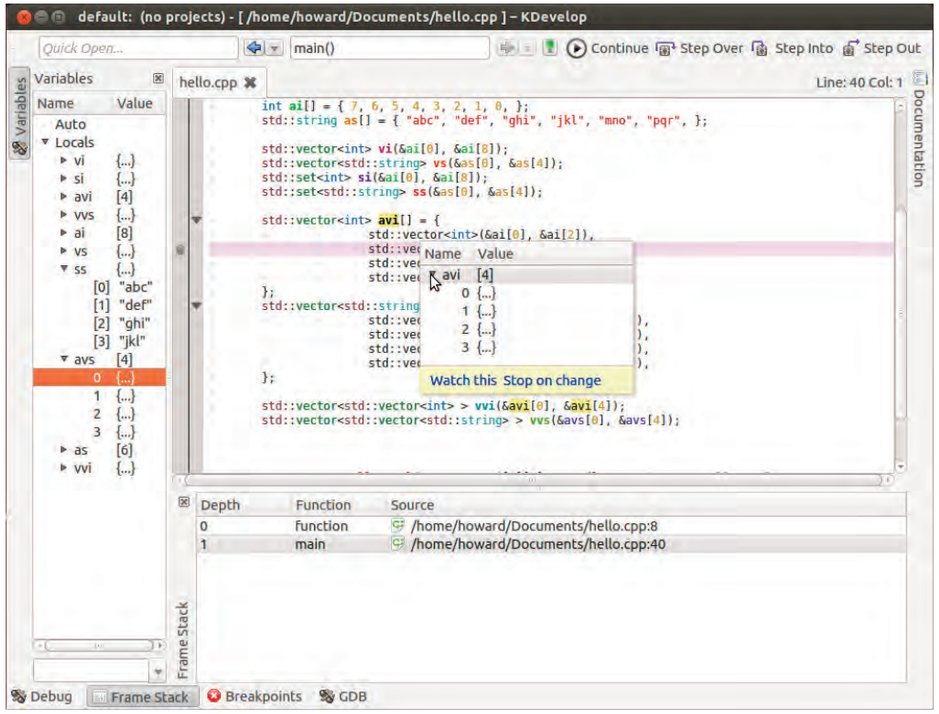

Figure 7: The KDevelop debugger.

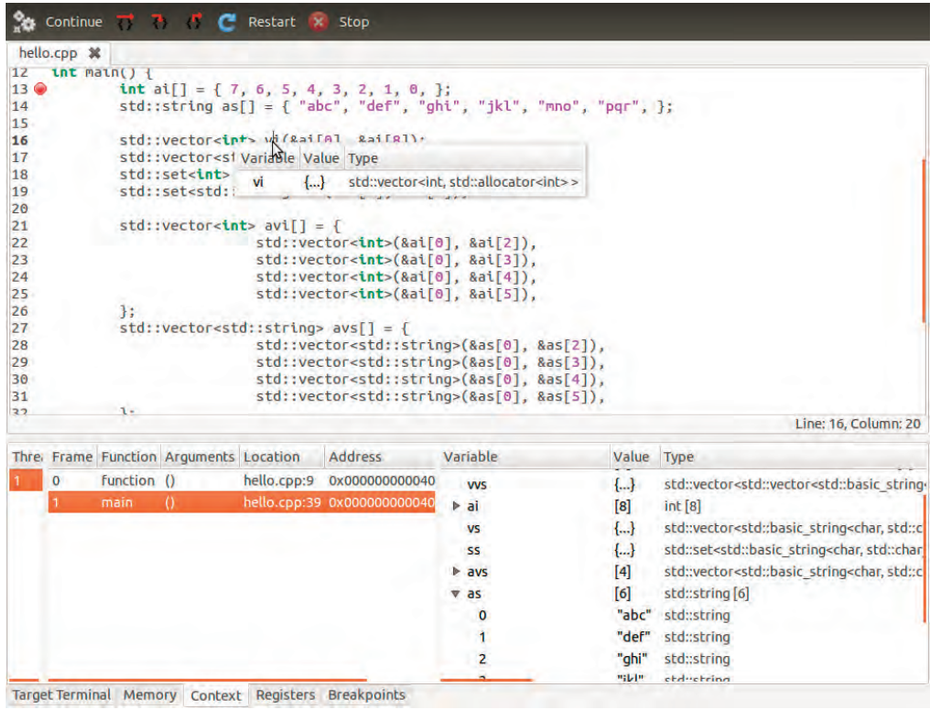

Figure 8: Nemiver.

Oracle's NetBeans 7.3 (http://www.netbeans.org), like Eclipse, started out as a Java IDE and added extensive capabilities for C and C++ programming. Like Eclipse, NetBeans is open-source and actively developed. The IDE offers single-button hotkeys and a toolbar with buttons for stepping through the debugged program. Its resizable detachable tabbed docking windows are excellent. NetBeans remembers breakpoints from one session to the next. Unfortunately, its tooltips for variables display STL container values on unformatted lines, but its Variables window has good navigation controls.

Digia Qt Creator 2.4.1 (http://qt.digia.com/) has single-button hotkeys and buttons for stepping through the target program. Its variable value tooltips sometimes appear as black text on a black background, but moving the mouse pointer over the solid black tooltip corrects this. Its variable display works well for nested data structures, and it has a complete menu of commands to navigate program symbols.

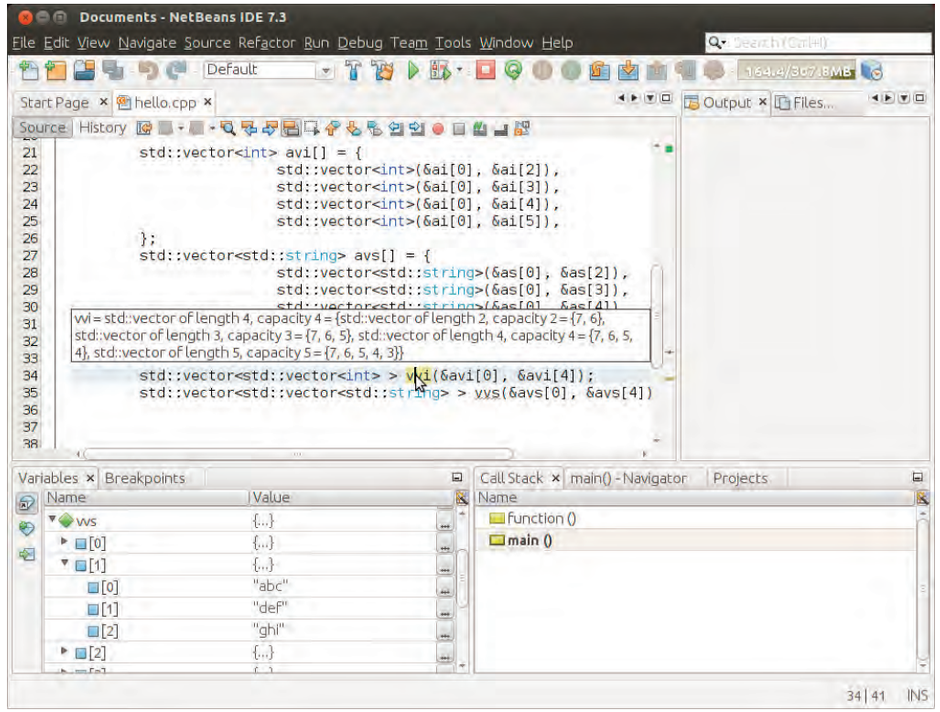

Figure 9: Oracle NetBeans.

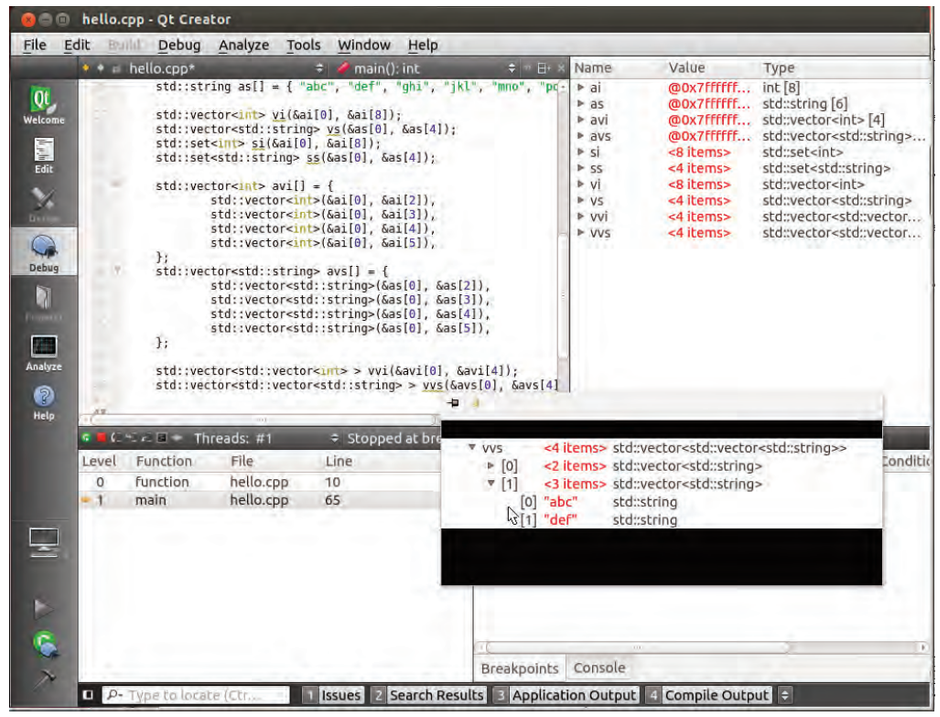

Figure 10: Digia Qt Creator.

SlickEdit 17.0.3.0 (http://www.slickedit.com) is a commercial programming editor with some excellent user interface features. Single-seat licenses start at $299. SlickEdit was recently reviewed in Dr. Dobb's . Its program-stepping toolbar is the best I've seen — each button's tooltip not only tells you what the button does, but also how to do the same action with the keyboard. It has the docking windows I like, but its Watch window doesn't group with its other windows. It remembers breakpoints across debug sessions, but when multiple variable watch windows are open, it is painfully slow to start up (with my test program). SlickEdit's tech support tells me that this problem will be fixed in the next release.

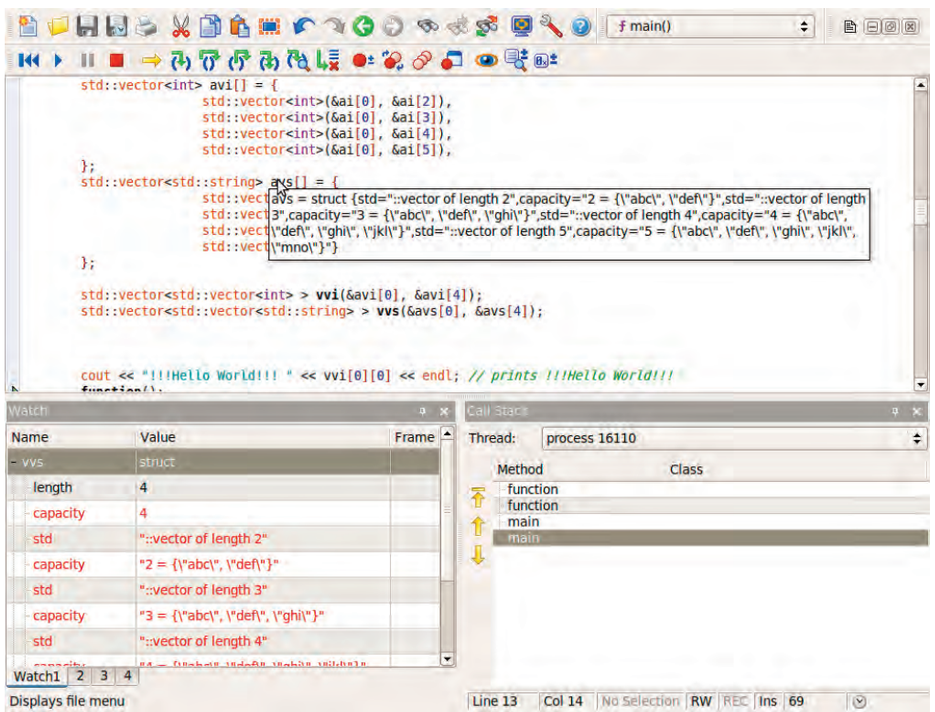

Figure 11: SlickEdit.

Rogue Wave's TotalView 8.11.0-2 (http://www.roguewave.com/) is a high-powered product with the best support for debugging multithreaded and multiprocessor code of the products listed here. Its pure debugging prowess is far ahead of mainstream products, including GDB. It's a commercial product, with single-seat licenses starting at $400. As to the UI, it has the single-keypress hotkeys and toolbar buttons I like. I hear from Rogue Wave that they'll add tabbed docking windows sometime in the next year, but for now, TotalView does not have them. While the UI has room for improvement, the product's superior engineering is apparent everywhere you look.

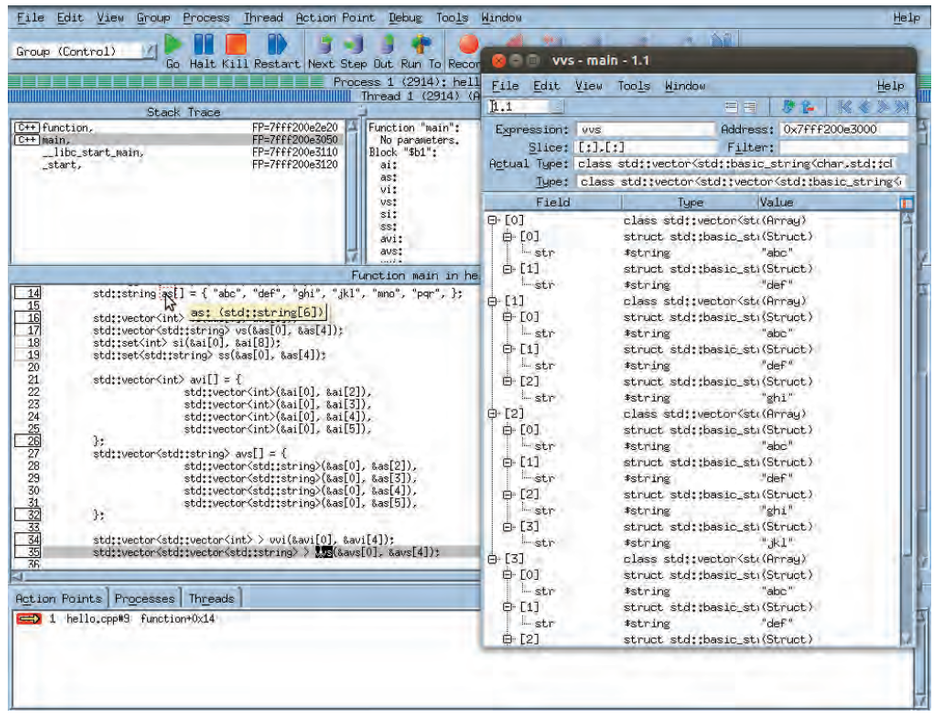

Figure 12: Rogue Wave TotalView.

Zerobugs 1.22.139 (http://zerobugs.codeplex.com) enables debugging for C, C++, and D. It sports a Gtkmm-based GUI, and includes a Python scripting framework. It usually requires you to build it from source. It wouldn't build for me but fortunately they offer a binary installer for 64 bit Ubuntu 11.10 so I tested it there. It doesn't understand STL containers or strings, but it does remember your breakpoints across debug session.

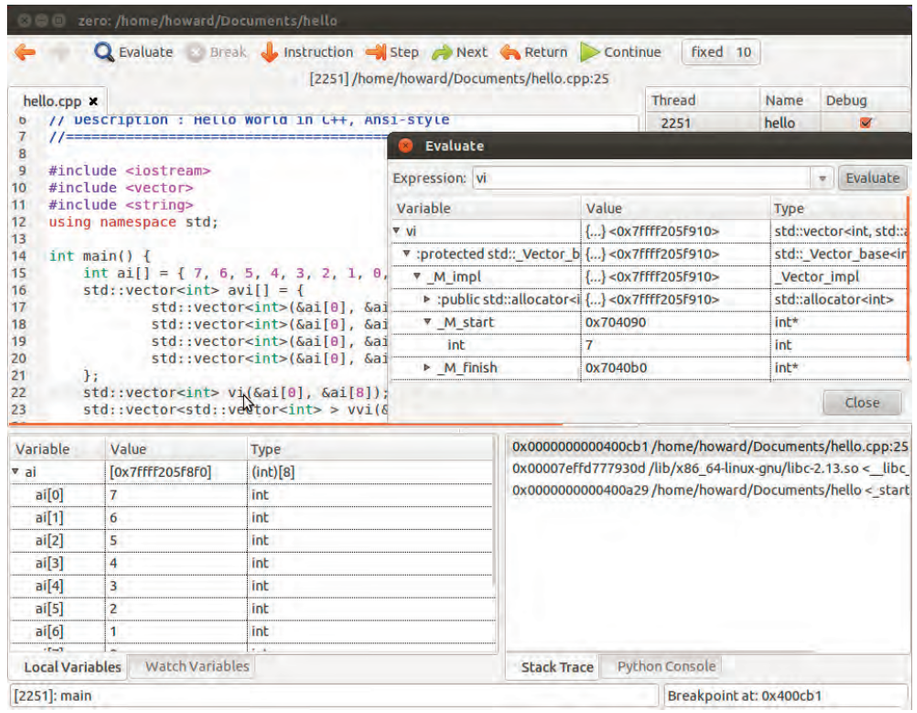

Figure 13: The Zerobugs UI.

This list is by no means complete. I could have included other lesser-known and/or more specialized debuggers and GDB interfaces, but I believe this list presents the best selection of debugging tools commonly used by C and C++ developers for non-embedded work.

Rating Basic Features

This section describes the most important qualities I use to pick a debugger for everyday use, focusing on the operations I do many times in every debug session. They include things like stepping through the program, interrupting program execution, and inspecting variable values. The debugger should make these important operations as easy as possible. Let's examine these in a bit more detail.

When stepping through a program, the four actions we typically perform most frequently are stepping over a function call, stepping into a function, returning from a function, and resuming program execution. Because they're so common, and because I prefer the keyboard over the mouse, my ideal debugger will have a single-button hotkey for each of these actions. The best hotkeys require only a single button press, such as F6 or "s." Combinations like Control-n are acceptable, but navigating through a menu by pressing something like Alt-r-n many times in every debug session is inconvenient. If you prefer the mouse, your ideal debugger will have a one-click toolbar button for each of these commands, such as the one in SlickEdit (Figure 11).

The most common ways to interrupt program execution are the breakpoint on a line, the conditional breakpoint (usually based on a variable value expression), and the breakpoint with a hit (sometimes called "ignore") count. Two other important ways to interrupt a running program are the watchpoint, which interrupts the program when a variable changes, and the catchpoint, which interrupts the program when it throws an exception. Three more useful features are abilities to temporarily disable a breakpoint, to find out how many times a breakpoint with an ignore count has been ignored, and to automatically restore breakpoints from a previous session. My ratings increase with the range of program interruption options.

There are five standard kinds of windows for inspecting variable values — tooltip, watch, locals, autos, and quick-watch. The quickest of these is the tooltip, which is a temporary window that displays a variable's value when the mouse hovers over the variable. It disappears when you press the escape key or move the mouse away. Good tooltip windows have data structure navigation controls as good as those in any of the other variable inspection windows. And the best ones don't waste every other line of screen real estate with something like "(Struct)" or the variable type.

In a variable inspection window, you should be able to navigate complex variables. It's also the most convenient place to call an arbitrary function, so the window should support that use. A common example

of that is a function that dumps the state of a complex or nested variable to the console or a file.

By default, every debugger will display a variable the way its programmer laid it out. For example, inspecting the values of a linked list could require you to manually navigate links until you reach the node you're interested in.

The debugger should automatically display variables in the standard library (such as STL containers) as if they were an array — one element per line. Other variables that internally manage memory, such as `std::strings` have a similar problem, because they're complex and nested; so by default, finding the data they contain requires even more manual navigation than linked lists do. When displaying variable values in any window (such as the tooltip, the watch window, or the local variable list), it's important that the debugger show the data that the programmer stored there, rather than any nested containing structure. This is a nonissue for most variables, but for more complex data structures like those mentioned here, it's very important.

When you've temporarily interrupted program execution, it's essential to be able to navigate up and down the call stack and see the program state. Usually, you're looking at the local variables for each stack frame. This is usually done with a clickable call stack window.

Everybody has "favorite" information they use while debugging. My favorites include variable values and the call stack. And when looking at a program statement, the ability to see as much of its surrounding context as possible can really aid productivity. For these reasons, a GUI and resizable and detachable tabbed docking windows are valuable.

In Table 1, I rate these features using a scale from 1 to 5, with 5 being best. Like all such assessments, they are subjective (but I expect that where I gave low rankings to features, most developers would agree those products need improvement to be comparable to the other listed products).

Product	Stepping keyboard	Stepping mouse	Breakpoints	Tooltips	Variable Inspection Views	Navigate Call Stack	Tab Docking Windows	Totals
Affinic	5	5	4	2	3	5	3	27
Code::Blocks	5	5	4	2	2	5	3	26
Codelite	4	5	4	1	2	5	3	24
DDD	5	5	4	2	2	5	2	25
Eclipse	5	5	4	4	4	5	5	32
Emacs	3	5	2	1	3	5	2	21
KDevelop	5	5	4	3	2	5	2	26
Nemiver	5	5	4	2	2	5	2	25
Netbeans	5	5	5	2	5	5	5	32
QTCreator	5	2	4	3	5	5	2	26
SlickEdit	5	5	5	3	3	5	4	30
Totalview	5	5	4	1	5	5	1	26
Zerobugs	5	3	5	1	3	5	1	23

Table 1: Ratings of basic features, where 5 is best.

Less Used Features

These features I discuss here support operations we perform less often, so while important in choosing a debugger, they are somewhat less important — to me, at least — than the features in the previous section.

There are two frequent actions I used as indicators for this section: toggle breakpoint and run to line. One-button hotkeys are ideal for these, but two-button combinations work well enough here.

Sometimes you step over a function call and then, having seen its return value, you wish you'd stepped in. For that situation, the ability to set the next statement to be executed back one or more lines by moving the instruction pointer saves the trouble of rerunning the program. Moving the instruction pointer forward to skip a program statement is another use of the set-next-statement, though less common.

At other times, you might like to know how the program (or a function in it) would act if a variable had a different value. The ability to change the value of a variable can save you from having to edit, recompile, and perform the sometimes complex steps to duplicate your program's state. Every variable window should support this use.

Some programs implement data structures complex enough that the debugger's default variable value display isn't very helpful. Examples include hand-crafted linked lists and tagged unions. Certainly, you can click-click-click your way through a custom linked list, or look at a union's tag field to see which union member is active, but that extra effort can break the concentration you need to diagnose a really tough bug. For frequently used data, some programmers (including me) find it worthwhile to write a bit of custom code to do that work automatically. Of course, your debugger must support this kind of customization. These customizations must be in separate files, so changes don't have to be merged as new versions of the debugger are released. Also, these customizations are designed for data structures the debuggers couldn't possibly know about.

I don't often use the ability to load a core dump, but it's invaluable for those (thankfully) rare crashes that only seem to happen at customer sites.

One non-debugging feature I like in a debugger is the ability to click on a symbol and have the file and line with the declaration or definition appear. Ideally, the debugger should present you with a choice of declaration or definition.

Customizing the keyboard bindings is something I typically do not do, but some of my coworkers adamantly insist that this is an important feature. There are a few capabilities that, together, make this ability fully featured. In order of importance, they are binding debugger commands to new keys, importing/exporting a complete set of bindings, and emulating other debuggers' key bindings.

Table 2 (on the next page) summarizes how the various products scored on all of these less used features.

Conclusion

None of the debuggers I tested was perfect. Not surprisingly, every product had some unexpected weakness that many of the others didn't share.

However, most of the debuggers supported the bulk of the features I look for. The only debuggers with resizable and detachable tabbed docking windows are Eclipse, Netbeans, and SlickEdit. All are fine choices. For now, I'm using Eclipse. Its engineering isn't as good as TotalView's, but for my uses, it makes the best overall compromise between UI and engineering — it has both docking windows and good variable display.

(5 is best)	Toggle breakpoint	Run to line	Set Next Statement	Change Variable Value	Custom Variable Display	Load Core Dump	Jump to Declaration/ Definition	Easy build and debug	Customize Keyboard	Totals
Affinic	4	3	1	2	1	1	3	1	1	17
Code::Blocks	5	5	5	1	3	1	2	5	1	28
Codelite	5	5	5	1	4	5	5	5	1	36
DDD	4	1	1	1	1	5	3	1	1	18
Eclipse	5	5	5	5	5	5	4	4	4	42
Emacs	4	1	1	3	1	1	2	3	2	18
KDBG	5	5	5	1	1	5	1	1	3	27
KDevelop	4	5	5	1	3	5	1	1	3	28
Nemiver	5	5	5	4	1	5	1	1	1	28
Netbeans	5	5	1	5	5	5	5	5	5	41
QTCreator	5	5	5	5	1	4	5	5	4	39
SlickEdit	5	5	5	5	5	5	5	5	5	45
Totalview	5	5	5	5	5	5	1	1	1	33
Zerobugs	3	1	1	5	2	5	2	1	3	23

Table 2: Ratings of lesser-used features (5 is best).

TotalView gets a special mention for its excellent multi-threaded and multiprocessor support, and when it gets tabbed docking windows, it'll be a top contender. Its multi-thread support is so far ahead of the field that if that were my principal need, I'd look no further.

I hope that if you're using other products, this comparison will provide you incentive to experiment and possibly find a tool that better serves your specific needs.

Test Platforms

At home, I tested on Canonical Ltd. Ubuntu 12.04 64-bit on VMware. At wor,k I tested on Ubuntu 10.04 64 bit on a quad-core AMD system. The Zerobugs binary is available for Ubuntu 11.10, so that's where I tested it.

Ubuntu 10.04 installs GDB version 7.1-1ubuntu2, which is sometimes extremely slow. Fortunately, recent versions correct that I used GDB 7.5.1.

I installed the STL pretty printers for GDB. They require Python support, so if you compile GDB from source, you'll have to install `python-dev` and run GDB's pre-build configure script with the `--with-python` switch.

— Howard Rubin has been programming professionally for more than 20 years and lives in Boulder, Colorado. He is on the C++ development team at Intio (http://www.intio.us/products/clearstart-svm.php), a vendor of 3D medical imaging systems.

Comment

From the Vault

Working with Microsoft PEX Framework

Automated white-box testing of .NET applications.

By Joydip Kanjilal

The PEX Framework is a Visual Studio 2010 powertool for automated white-box testing of .NET applications. Implemented as a Visual Studio add-in, PEX was developed by Microsoft Research and can be downloaded here. Once installed, PEX lets you automatically generate test suites with high code coverage.

Unlike other unit testing tools, PEX suggests fixes to potential bugs it detects in your code. PEX is capable of performing an analysis on your application's code, searching for boundary conditions, generating test cases, searching for assertion failures, and ultimately reducing maintenance costs. In this article, I examine the PEX Framework and show how you can integrate it with Visual Studio to create better unit tests with high code coverage.

To work with the PEX framework, you need the following installed on your system:

- .NET Framework 3.5 or higher
- Visual Studio 2008 or Visual Studio 2008 Team System
- Visual Studio 2010

Alternatively, if for some reason you don't have a version of Visual Studio compatible with PEX installed on your system, you can still use PEX from the command line.

Installing the PEX Framework

To install the PEX framework in your system, follow these steps:

1. Double click on the PEX setup file. The window in Figure 1 appears:

Figure 1: Executing the Microsoft Pex 2010 setup.

2. Click Next
3. Accept the License agreement
4. Specify the drive and folder you want PEX to be installed on
5. Click Finish when PEX has been installed on your system

You can now start Visual Studio 2010 and begin using PEX.

Exploring PEX Using Visual Studio 2010

To begin exploring PEX, I'll create a sample project in Visual Studio 2010 to build an application that displays the sum of two numbers in a `MessageBox`. To do this, open Visual Studio 2010 and create a WPF application. Open `Mainwindow.xaml` and drag-and-drop two `TextBox` controls, two `Label` controls, and a `Button` control. Here's the markup code for the `MainWindow.xaml` file:

```
<Window x:Class="DDJ.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
       presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid Height="205" Width="440">
        <Button Content="Get Result" Height="24"
         HorizontalAlignment="Left" Margin="164,132,0,0"
         Name="btnResult" VerticalAlignment="Top"
         Width="80" Click="btnResult_Click" />
        <TextBox Height="23" HorizontalAlignment="Left"
         Margin="164,43,0,0" Name="txtFirstNumber"
         VerticalAlignment="Top" Width="120" />
        <TextBox Height="23" HorizontalAlignment="Left"
         Margin="164,90,0,0" Name="txtSecondNumber"
         VerticalAlignment="Top" Width="120" />
        <Label Content="Enter First Number     : "
         Height="28" HorizontalAlignment="Left"
         Margin="31,41,0,0" Name="label1"
         VerticalAlignment="Top" Width="127" />
        <Label Content="Enter Second Number : " Height="28"
         HorizontalAlignment="Left" Margin="31,88,0,0"
         Name="label2" VerticalAlignment="Top" Width="127" />
    </Grid>
</Window>
```

Insert the following event handler for the `Button` control into the `MainWindow.xaml.cs` file:

```
private void btnResult_Click(object sender, RoutedEventArgs e)
    {
        Calculate.DisplayResult(txtFirstNumber.Text,
            txtSecondNumber.Text);
    }
```

The `DisplayResult()` method is a static method that belongs to the `Calculate` class. This method would display the sum of two numbers entered by the user when the `Button` is clicked on. The result would be displayed in a `MessageBox`. Here is the complete `MainWindow.xaml.cs` code:

```
using System.Windows;
namespace DDJ
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private void btnResult_Click(object sender,
            RoutedEventArgs e)
        {
            Calculate.DisplayResult(txtFirstNumber.Text,
                txtSecondNumber.Text);
        }
    }
}
```

And, here is the complete code of the `Calculate` class:

```
using System;
namespace DDJ
{
    public class Calculate
    {
        public static void DisplayResult(String firstNumber,
            String secondNumber)
        {
            int sum = GetSum(Int32.Parse(firstNumber),
                Int32.Parse(secondNumber));
            System.Windows.MessageBox.Show(sum.ToString(),
                "Result");
        }
        public static int GetSum(int firstNumber, int
            secondNumber)
        {
            return (firstNumber + secondNumber);
        }
    }
}
```

Using PEX

To execute PEX on your application's code, follow these steps:

- In Visual Studio 2010, open the source file on which you intend to run PEX. In our example, it is the `Calculate.cs` file.

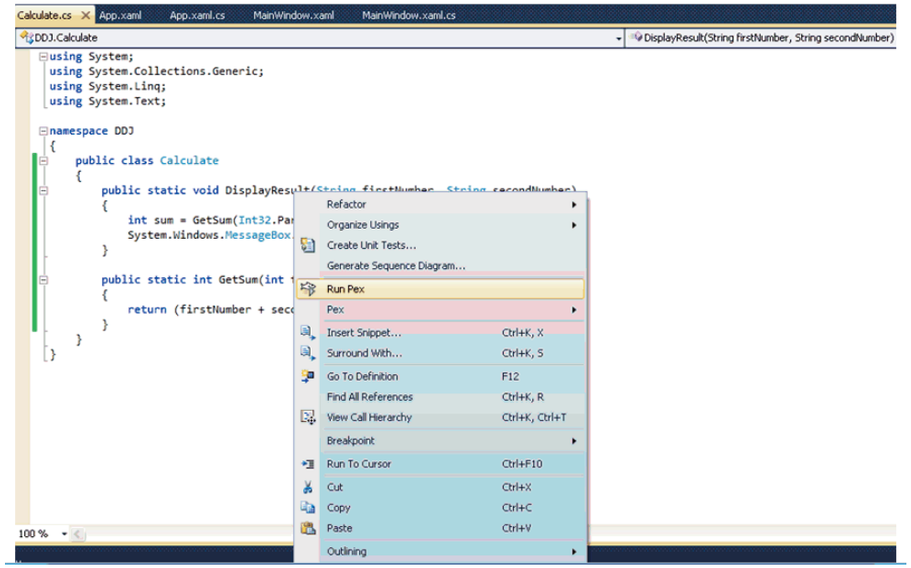

Figure 2: Executing PEX on your application's code.

- Select the class or method on which PEX should be executed. Note that this class or method should be public. We will select the `DisplayResult()` method in this example.
- Next, right click on the class or method and choose "Run Pex." We will run PEX on the `DisplayResult()` method as in Figure 2.

PEX will now start exploring your method or class. This process is known as "PEX Exploration" and the result of this exploration is displayed in a window called "Pex Explorer." After exploration of the `DisplayResult()` method, the output looks like Figure 3.

Note that if you place your cursor inside a particular method and then run PEX, PEX would consider that method only for its explorations. However, if the cursor is placed outside of the method and PEX is executed, PEX exploration would work on the entire class.

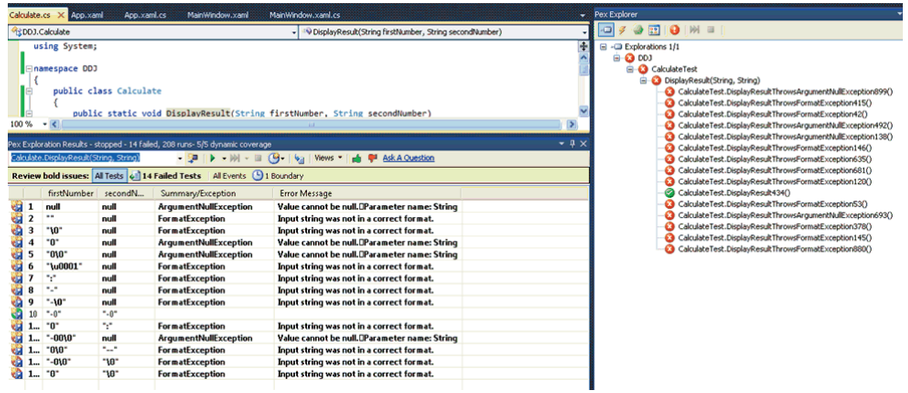

Figure 3: The PEX Explorer Window.

Understanding the PEX Explorer Window

The PEX Explorer is a window that displays the results of a PEX Exploration process. There are four symbols or icons that display the exploration details on the method or class on which PEX has been executed. These symbols include a:

- Clock symbol that signifies that a PEX exploration is still in progress
- Red symbol that denotes failed test cases
- Green symbol used to denote test cases that have passed
- Blue symbol that implies that PEX has failed to generate test cases for that part of the program's code

Fixing the Errors

You now need to fix the test errors that have been displayed in the PEX Explorer on execution of PEX on the `DisplayResult()` method. To do this, you need to add proper boundary conditions in your code. Here is the optimized version of the `DisplayResult()` method:

```
public static void DisplayResult(String firstNumber,
    String secondNumber)
        {
            int x, y;
            try
            {
                x = Int32.Parse(firstNumber);
            }
            catch
            {
                x = 0;
            }
            try
            {
                y = Int32.Parse(secondNumber);
            }
            catch
            {
                y = 0;
            }
            int sum = GetSum(x, y);
            System.Windows.MessageBox.Show(sum.ToString(),
                "Result");
        }
```

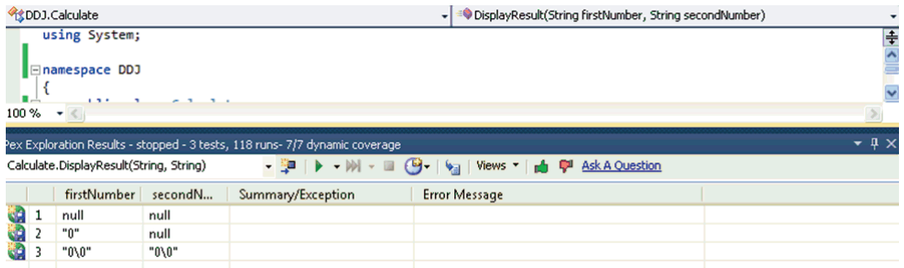

Figure 4: The PEX Explorer Window shows the test cases have passed.

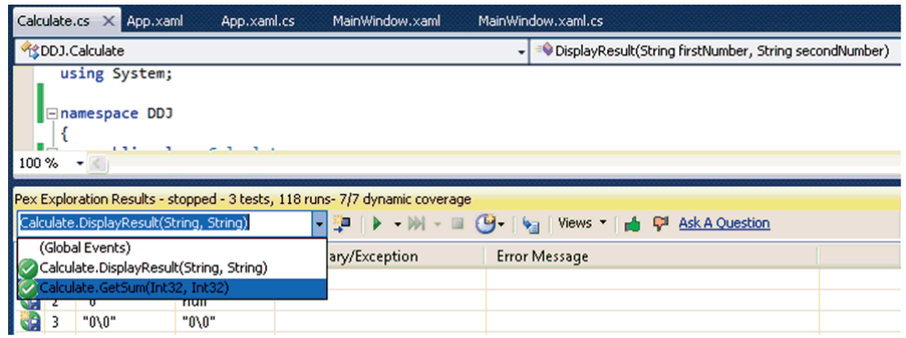

Figure 5: Selecting the method or code block of your choice inside the Pex Explorer Window.

When you run PEX again on this method, you'll see that all the test cases have passed, as in Figure 4.

When you run PEX on the `Calculate` class, you can take a look at the PEX Exploration results for each of the methods of the class. You just need to select the method from the `DropDownList` control as in Figure 5.

Summary

The PEX Framework from Microsoft is a great tool in detecting and fixing bugs in your code. In this article, I've presented the core features of PEX and described how you can make use of this framework to build high-performance, robust applications.

— Joydip Kanjilal is a Microsoft Most Valuable Professional in ASP.NET and author of numerous books on Windows programming. He can be contacted at joydipkanjilal@yahoo.com.

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

ACCESS DATA ITEMS IN ANCESTOR STACK FRAMES SAFELY

Retain/recall semantics enable selected data on the call stack to be accessible from descendant frames.They are similar to dynamic scoping and the structural opposite of throw/catch. For many problems that developers currently use global data to address, retain/recall can be more convenient and flexible. The approach is safe for concurrent, recursive, and re-entrant code.

http://www.drdobbs.com/240155450

MOVING IS NOT COPYING

The C++ standards committee first met at the end of 1989; the notion of moving as a distinct operation from copying did not enter the official C++ Standard until 2011. Therefore, there are some subtleties that have taken more than 20 years to nail down. Like many subtle ideas, the idea of moving data in C++ is built on a simple concept.

http://www.drdobbs.com/240156175

LEONARDO'S CODE

Al Williams discusses his ongoing love/hate relationship with Arduino. He likes that it has launched a whole ecosystem of inexpensive and readily available peripherals. He doesn't like the clunky IDE, the bizarre terms (like sketch), and the fan boy culture surrounding it.

http://www.drdobbs.com/240156256

SO YOU WANT TO WRITE A MOBILE APP? PLACE YOUR BETS!

On top of everything else, you'll have to make big bets on hardware platforms and development approaches. This won't be easy.

http://www.drdobbs.com/240155587

WHY AREN'T THERE BETTER TESTING TOOLS?

Static analysis, UI record and playback, and load testing dominate today's testing tools. Just like they did in the 1990s. Where's the progress?

http://www.drdobbs.com/240155303

Andrew Binstock Editor in Chief, Dr. Dobb's
andrew.binstock@ubm.com

Deirdre Blake Managing Editor, Dr. Dobb's
deirdre.blake@ubm.com

Amy Stephens Copyeditor, Dr. Dobb's
amy.stephens@ubm.com

Sean Coady Webmaster, Dr. Dobb's
sean.coady@ubm.com

Jon Erickson Editor in Chief Emeritus, Dr. Dobb's

DR. DOBB'S EDITORIAL
751 Laurel Street #614
San Carlos, CA
94070
USA

UBM TECH
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000