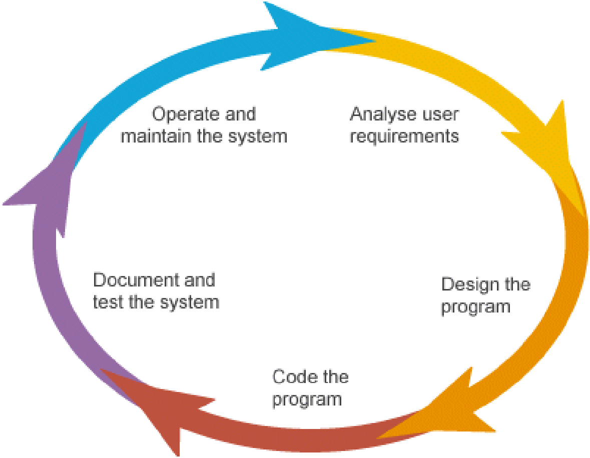
Introduction to Unit Testing

Ryan Houlihan

March 18, 2013

Software Development Lifecycle

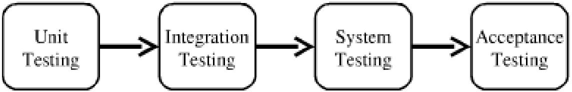


Traditional Testing

- Test the system as a whole
    - Higher level of complexity
    - Individual components are rarely tested
    - Isolating errors is problematic
- Testing Strategies
    - Print Statements
    - Use of Debugger
    - Test Scripts

Proper Testing

1. Unit Testing (Developers)
    - Individual components (class or subsystem)
2. Integration Testing (Developers)
    - Aggregates of subsystems
3. System Testing (Developers)
    - Complete integrated system
    - Evaluates system's compliance with specified requirements
4. Acceptance Testing (Client)
    - Evaluates system delivered by developers



Benefits

- Limits complexity of issues
  - Many bugs and errors are eliminated during development, not afterwards.
- Easier to extend code in future
  - Confirms added functionalities compliance with specified requirements
- Saves time in the development cycle
  - Limits extra system testing and debugging
  - Eliminates logical errors that would otherwise require re-writing of large sections of software

Test Driven Development Cycle

- The process for developing software under these guide lines is the following:
    - Add a test
    - Run the automated tests
        - See test fail
    - Fix test
    - Run automated tests
        - See test succeed
    - Refactor code for general case

Who Uses The TDDC?

- I MUST STRESS - This is the standard practice in NON-SCIENTIFIC software development
- Necessity for larger projects
  - Provides proof that contracted work was completed and completed correctly

What is a Unit?

- Smallest testable part of an application
- Definition differs depending on the type of programming under discussion
    - **Procedural Programming:** an individual function or procedure
    - **Object-Oriented Programming:** an interface such as a class

Unit Testing

- Static Testing
  - Focuses on prevention
  - Done at compile time
  - Tests code only not output of the code
  - Can find: syntax errors, ANSI violations, code that does not conform to coding standards, etc.
  - 100% statement coverage in short time
- Dynamic Testing
  - Focuses on elimination of logical errors
  - Performed during run time
  - Two kinds: White and black box testing
  - Finds bugs in executed pieces of software
  - Limited statement coverage with long run time

Black Box Testing

- Tests functionality without knowledge of internal structure
  - Input is valid across a range of values
    - Below, within and above range
  - Input is valid only if it is a member of a discrete set
    - Tests valid discrete values and invalid discrete values
    - Example:

```
public class Calender {
  void printMonthYear(int month, int year){
    throws InvalidMonthException
    ....
  }
}
```

    - Valid Input: Month → 1-12 ; Year → Any Integer

White Box Testing

- Tests internal structures and workings of an application
- Also assumes knowledge of the internal structure of the application
- Described by metrics:
    - Statement coverage
        - execution of all statements at least once
    - Branch coverage
        - testing of all decision outcomes
    - Path coverage
        - testing across each logical path independently

Unit Testing Heuristics

1. Create test when object design is complete
2. Develop tests cases using effective number of test cases
3. Cross check test cases to eliminate duplicates
4. Create test harness
    - Test drivers and stubs needed for integration testing
5. Describe test oracle (usually first successful test)
6. Execute test cases - Re-execute when change is made (regression testing)
7. Compare results with test oracle (automate where possible)

Assertion and Exceptions

- Assertion:
    - Used to test a class or function by making an assertion about its behavior
    - Fatal if assertion not met
- Exception:
    - Same as an Assertion except it is non-fatal if the expectation is not meet

Objects: Stubs, Fakes and Mocks

- We use a variety of modified objects to test the behavior of other objects
- Allow us to isolate one unit at a time for testing
- By isolating one unit at a time we can narrow down bugs and other issues in our software

Stubs

- Class/Object which provides a valid but static response. All input will result in the same response
- Can not verify whether a method has been called or not.
- Example:

```
public class TestingScheduler extends Scheduler {
  public TestingScheduler(int timeInMillisForTest){
    this.timeInMillisForTest = timeInMillisForTest;
  }
}
```

Fake Object

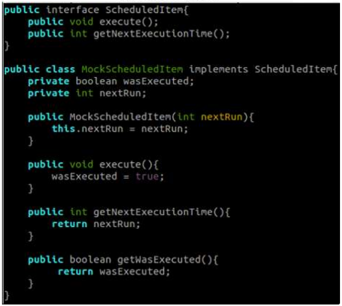
- Class/Object which implements an interface but contains fixed data and no logic
- Simply returns "good" or "bad" data depending on the implementation.

Mock Object

- Simulated objects which mimic the behavior of real objects in controlled ways.
- Used to test the behavior of some other object through assertions.
- Similar to fakes except they keep track of which of the mock object's methods are called, with what kind of parameters, and how many times.
- Can verify that the method under test, when executed, will:
    - Call certain functions on the mock object (or objects) it interacts wit
    - react in an appropriate way to whatever the mock objects do (same as fake/stub)

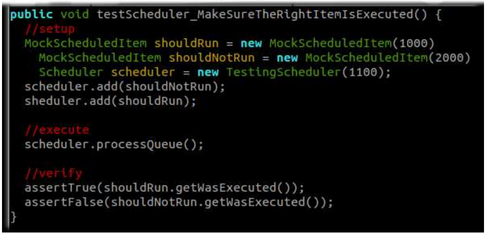
Mock Example

Mock Class



```
public interface ScheduledItem{
    public void execute();
    public int getNextExecutionTime();
}

public class MockScheduledItem implements ScheduledItem{
    private boolean wasExecuted;
    private int nextRun;

    public MockScheduledItem(int nextRun){
        this.nextRun = nextRun;
    }

    public void execute(){
        wasExecuted = true;
    }

    public int getNextExecutionTime(){
        return nextRun;
    }

    public boolean getWasExecuted(){
         return wasExecuted;
    }
}
```

Test Case



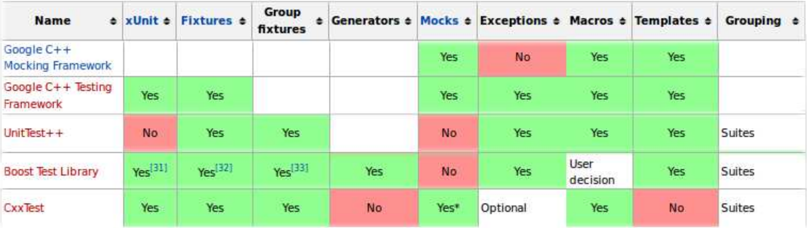
```
public void testScheduler_MakeSureTheRightItemIsExecuted() {
  //setup
  MockScheduledItem shouldRun = new MockScheduledItem(1000)
    MockScheduledItem shouldNotRun = new MockScheduledItem(2000)
    Scheduler scheduler = new TestingScheduler(1100);
  scheduler.add(shouldNotRun);
  sheduler.add(shouldRun);

  //execute
  scheduler.processQueue();

  //verify
  assertTrue(shouldRun.getWasExecuted());
  assertFalse(shouldNotRun.getWasExecuted());
}
```

C++ Frameworks

- All support a variety of platforms out of box
- **Google Test**: http://code.google.com/p/googletest/
  - Rich set of assertions, death test, fatal and non-fatal failures, value and type parameterized tests, XML test report generation
  - Has a powerful mocking framework
- **UnitTest++**: http://unittest-cpp.sourceforge.net/
  - Simple, portable, fast and has a small footprint.
  - No support for mocking
- **Boost Test**: http://www.boost.org/libs/test/
  - Uses boost but provides no mocking

C++ Frameworks

- **CxxTest**: http://cxxtest.com/
  - Uses a C++ parser and code generator for test registration
  - Framework for generating mocks of global functions but not of objects



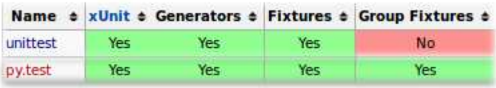
Name	xUnit	Fixtures	Group fixtures	Generators	Mocks	Exceptions	Macros	Templates	Grouping
Google C++ Mocking Framework					Yes	No	Yes	Yes	
Google C++ Testing Framework	Yes	Yes			Yes	Yes	Yes	Yes	
UnitTest++	No	Yes	Yes		No	Yes	Yes	Yes	Suites
Boost Test Library	Yes[31]	Yes[32]	Yes[33]	Yes	No	Yes	User decision	Yes	Suites
CxxTest	Yes	Yes	Yes	No	Yes*	Optional	Yes	No	Suites

Python Frameworks

- pyUnit (unittest): http://pyunit.sourceforge.net/pyunit.html
  - Easy support for distributed testing and good plugin architecture
  - Tests are quite structured and easy to read
  - Part of Pythons standard library
  - More complicated to use then py.test but contains many more features

Python Frameworks

- py.test: http://pytest.org
  - Easy support for distributed testing and good plugin architecture
  - Simple to use. I.e Easy assertions (assert x == 42 no assertEqual()), etc.
  - Framework can lead to unstructured hard to read unit tests
  - Lacks the features of pyUnit



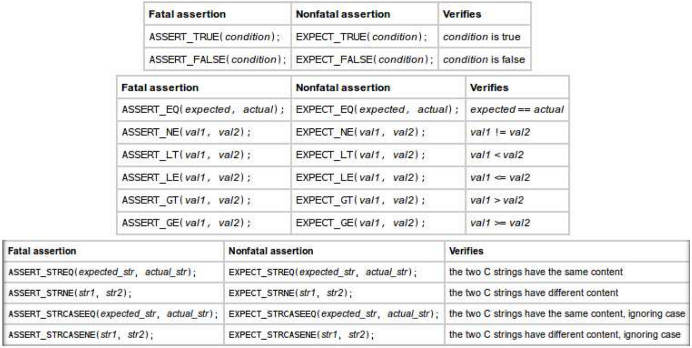
Name	xUnit	Generators	Fixtures	Group Fixtures
unittest	Yes	Yes	Yes	No
py.test	Yes	Yes	Yes	Yes

What Google Test Provides

- Easy to use Assertions and Exceptions
- Very powerful mocking capabilities
- Fairly easy and straightforward to use
- Works on all common platforms
- Contains a GUI for testing

Assertions and Exceptions

- Assertions and Exceptions in Google Test are quite easy and straight forward



Fatal assertion	**Nonfatal assertion**	**Verifies**
ASSERT_TRUE(*condition*);	EXPECT_TRUE(*condition*);	*condition* is true
ASSERT_FALSE(*condition*);	EXPECT_FALSE(*condition*);	*condition* is false

Fatal assertion	**Nonfatal assertion**	**Verifies**
ASSERT_EQ(*expected*, *actual*);	EXPECT_EQ(*expected*, *actual*);	*expected* == *actual*
ASSERT_NE(*val1*, *val2*);	EXPECT_NE(*val1*, *val2*);	*val1* != *val2*
ASSERT_LT(*val1*, *val2*);	EXPECT_LT(*val1*, *val2*);	*val1* < *val2*
ASSERT_LE(*val1*, *val2*);	EXPECT_LE(*val1*, *val2*);	*val1* <= *val2*
ASSERT_GT(*val1*, *val2*);	EXPECT_GT(*val1*, *val2*);	*val1* > *val2*
ASSERT_GE(*val1*, *val2*);	EXPECT_GE(*val1*, *val2*);	*val1* >= *val2*

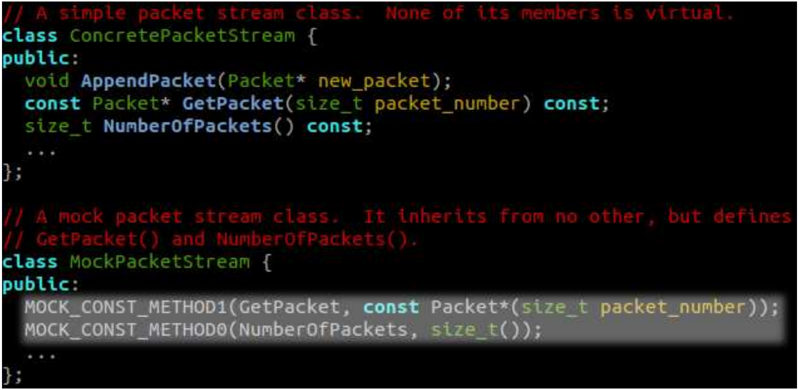
Fatal assertion	**Nonfatal assertion**	**Verifies**
ASSERT_STREQ(*expected_str*, *actual_str*);	EXPECT_STREQ(*expected_str*, *actual_str*);	the two C strings have the same content
ASSERT_STRNE(*str1*, *str2*);	EXPECT_STRNE(*str1*, *str2*);	the two C strings have different content
ASSERT_STRCASEEQ(*expected_str*, *actual_str*);	EXPECT_STRCASEEQ(*expected_str*, *actual_str*);	the two C strings have the same content, ignoring case
ASSERT_STRCASENE(*str1*, *str2*);	EXPECT_STRCASENE(*str1*, *str2*);	the two C strings have different content, ignoring case

Mocking Virtual Methods

- Mocking method must always be in a public section

```
class Foo {
public:
  ...
virtual bool Transform(Gadget* g) = 0;

protected:
  virtual void Resume();

private:
  virtual int GetTimeOut();
};

class MockFoo : public Foo {
public:
  ...
MOCK_METHOD1(Transform, bool(Gadget* g));

  // The following must be in the public section, even though the
  // methods are protected or private in the base class.
  MOCK_METHOD0(Resume, void());
  MOCK_METHOD0(GetTimeOut, int());
};
```

Mocking Non-Virtual Methods



```
// A simple packet stream class.  None of its members is virtual.
class ConcretePacketStream {
public:
  void AppendPacket(Packet* new_packet);
  const Packet* GetPacket(size_t packet_number) const;
  size_t NumberOfPackets() const;
  ...
};

// A mock packet stream class.  It inherits from no other, but defines
// GetPacket() and NumberOfPackets().
class MockPacketStream {
public:
  MOCK_CONST_METHOD1(GetPacket, const Packet*(size_t packet_number));
  MOCK_CONST_METHOD0(NumberOfPackets, size_t());
  ...
};
```

Implementing Unit Testing in LAMMPS

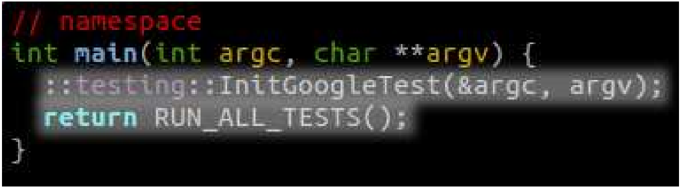
- LAMMPS is a large and complicated piece of software
- Work is distributed through different classes
- LAMMPS is designed in a modular fashion so to be easy to modify and extend with new functionality. 75% of the source code was added in this fashion
- With unit testing adding new features to LAMMPS becomes much simpler and safer.

Bond Styles

- LAMMPS contains many bond styles: Harmonic, Hybrid, Morse, fene, fene expanded, etc.
- We implemented a unit testing frame work and test cases for all the bond styles
- Linked Google test and LAMMPS as external libraries to our unit testing library.
    - Required no modification to LAMMPS

Creating a Test Case

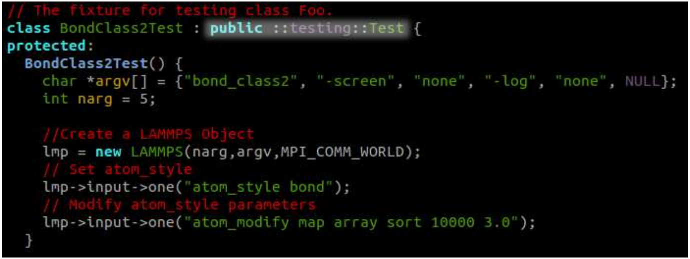
- Different main for each test case
- First initialize the framework then run all the tests which automatically detects and runs all the tests



```
// namespace
int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Creating a Test Case

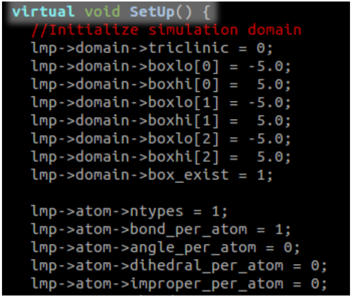
- Create a class which is derived from the ::testing::test class
- Class creation is called once at the beginning of the test



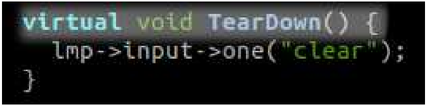
```
// The fixture for testing class Foo.
class BondClass2Test : public ::testing::Test {
protected:
  BondClass2Test() {
    char *argv[] = {"bond_class2", "-screen", "none", "-log", "none", NULL};
    int narg = 5;

    //Create a LAMMPS Object
    lmp = new LAMMPS(narg,argv,MPI_COMM_WORLD);
    // Set atom_style
    lmp->input->one("atom_style bond");
    // Modify atom_style parameters
    lmp->input->one("atom_modify map array sort 10000 3.0");
  }
```

Framework

- SetUp() is run at the beginning of each individual test case
- TearDown() is run at the end of each individual test case and before the next SetUp() is run

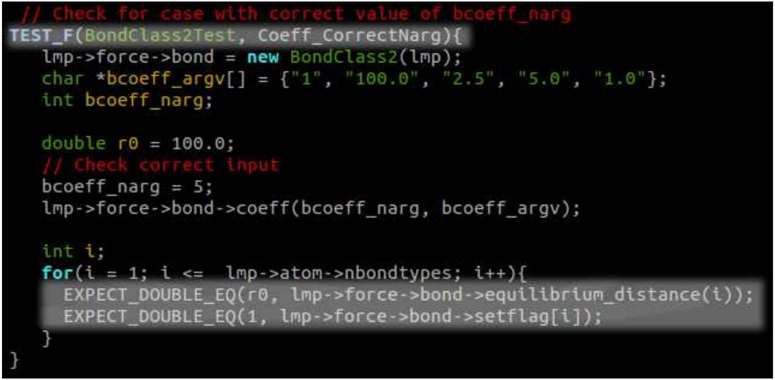


```
virtual void SetUp() {
   //Initialize simulation domain
   lmp->domain->triclinic = 0;
   lmp->domain->boxlo[0] = -5.0;
   lmp->domain->boxhi[0] =  5.0;
   lmp->domain->boxlo[1] = -5.0;
   lmp->domain->boxhi[1] =  5.0;
   lmp->domain->boxlo[2] = -5.0;
   lmp->domain->boxhi[2] =  5.0;
   lmp->domain->box_exist = 1;

   lmp->atom->ntypes = 1;
   lmp->atom->bond_per_atom = 1;
   lmp->atom->angle_per_atom = 0;
   lmp->atom->dihedral_per_atom = 0;
   lmp->atom->improper_per_atom = 0;
```



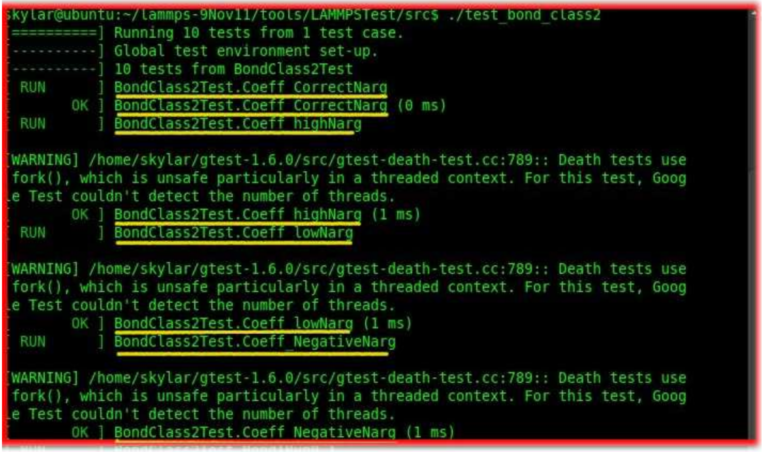
```
virtual void TearDown() {
   lmp->input->one("clear");
}
```

Sample Test



```
// Check for case with correct value of bcoeff_narg
TEST_F(BondClass2Test, Coeff_CorrectNarg){
    lmp->force->bond = new BondClass2(lmp);
    char *bcoeff_argv[] = {"1", "100.0", "2.5", "5.0", "1.0"};
    int bcoeff_narg;

    double r0 = 100.0;
    // Check correct input
    bcoeff_narg = 5;
    lmp->force->bond->coeff(bcoeff_narg, bcoeff_argv);

    int i;
    for(i = 1; i <=  lmp->atom->nbondtypes; i++){
      EXPECT_DOUBLE_EQ(r0, lmp->force->bond->equilibrium_distance(i));
      EXPECT_DOUBLE_EQ(1, lmp->force->bond->setflag[i]);
    }
}
```

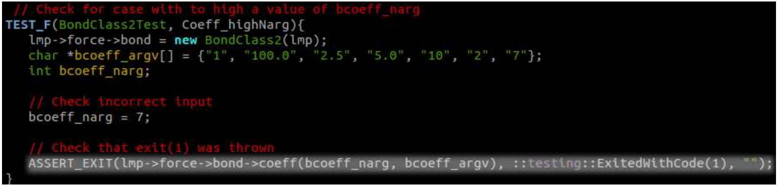
Sample Output



```
kylar@ubuntu:~/lammps-9Nov11/tools/LAMMPSTest/src$ ./test_bond_class2
[==========] Running 10 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 10 tests from BondClass2Test
[ RUN      ] BondClass2Test.Coeff_CorrectNarg
[       OK ] BondClass2Test.Coeff_CorrectNarg (0 ms)
[ RUN      ] BondClass2Test.Coeff_highNarg

[WARNING] /home/skylar/gtest-1.6.0/src/gtest-death-test.cc:789:: Death tests use
 fork(), which is unsafe particularly in a threaded context. For this test, Goog
e Test couldn't detect the number of threads.
[       OK ] BondClass2Test.Coeff_highNarg (1 ms)
[ RUN      ] BondClass2Test.Coeff_lowNarg

[WARNING] /home/skylar/gtest-1.6.0/src/gtest-death-test.cc:789:: Death tests use
 fork(), which is unsafe particularly in a threaded context. For this test, Goog
e Test couldn't detect the number of threads.
[       OK ] BondClass2Test.Coeff_lowNarg (1 ms)
[ RUN      ] BondClass2Test.Coeff_NegativeNarg

[WARNING] /home/skylar/gtest-1.6.0/src/gtest-death-test.cc:789:: Death tests use
 fork(), which is unsafe particularly in a threaded context. For this test, Goog
e Test couldn't detect the number of threads.
[       OK ] BondClass2Test.Coeff_NegativeNarg (1 ms)
```

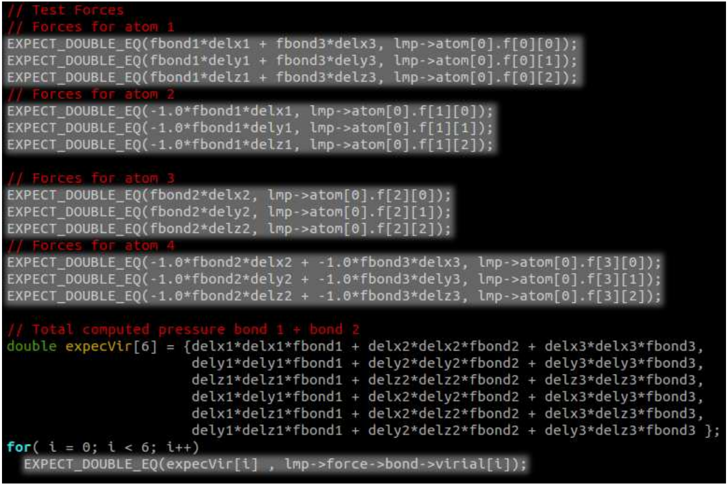
Another Sample Test

- Every test is called TEST_F(). The first argument is the class the second argument is the test label.



```
 // Check for case with to high a value of bcoeff_narg
TEST_F(BondClass2Test, Coeff_highNarg){
   lmp->force->bond = new BondClass2(lmp);
   char *bcoeff_argv[] = {"1", "100.0", "2.5", "5.0", "10", "2", "7"};
   int bcoeff_narg;

   // Check incorrect input
   bcoeff_narg = 7;

   // Check that exit(1) was thrown
   ASSERT_EXIT(lmp->force->bond->coeff(bcoeff_narg, bcoeff_argv), ::testing::ExitedWithCode(1), "");
}
```

Even More Sample Tests



```
// Test Forces
// Forces for atom 1
EXPECT_DOUBLE_EQ(fbond1*delx1 + fbond3*delx3, lmp->atom[0].f[0][0]);
EXPECT_DOUBLE_EQ(fbond1*dely1 + fbond3*dely3, lmp->atom[0].f[0][1]);
EXPECT_DOUBLE_EQ(fbond1*delz1 + fbond3*delz3, lmp->atom[0].f[0][2]);
// Forces for atom 2
EXPECT_DOUBLE_EQ(-1.0*fbond1*delx1, lmp->atom[0].f[1][0]);
EXPECT_DOUBLE_EQ(-1.0*fbond1*dely1, lmp->atom[0].f[1][1]);
EXPECT_DOUBLE_EQ(-1.0*fbond1*delz1, lmp->atom[0].f[1][2]);

// Forces for atom 3
EXPECT_DOUBLE_EQ(fbond2*delx2, lmp->atom[0].f[2][0]);
EXPECT_DOUBLE_EQ(fbond2*dely2, lmp->atom[0].f[2][1]);
EXPECT_DOUBLE_EQ(fbond2*delz2, lmp->atom[0].f[2][2]);
// Forces for atom 4
EXPECT_DOUBLE_EQ(-1.0*fbond2*delx2 + -1.0*fbond3*delx3, lmp->atom[0].f[3][0]);
EXPECT_DOUBLE_EQ(-1.0*fbond2*dely2 + -1.0*fbond3*dely3, lmp->atom[0].f[3][1]);
EXPECT_DOUBLE_EQ(-1.0*fbond2*delz2 + -1.0*fbond3*delz3, lmp->atom[0].f[3][2]);

// Total computed pressure bond 1 + bond 2
double expecVir[6] = {delx1*delx1*fbond1 + delx2*delx2*fbond2 + delx3*delx3*fbond3,
                      dely1*dely1*fbond1 + dely2*dely2*fbond2 + dely3*dely3*fbond3,
                      delz1*delz1*fbond1 + delz2*delz2*fbond2 + delz3*delz3*fbond3,
                      delx1*dely1*fbond1 + delx2*dely2*fbond2 + delx3*dely3*fbond3,
                      delx1*delz1*fbond1 + delx2*delz2*fbond2 + delx3*delz3*fbond3,
                      dely1*delz1*fbond1 + dely2*delz2*fbond2 + dely3*delz3*fbond3 };
for( i = 0; i < 6; i++)
  EXPECT_DOUBLE_EQ(expecVir[i] , lmp->force->bond->virial[i]);
```