IBM

developerWorks®

Open source C/C++ unit testing tools, Part 1: Get to know the Boost unit test framework

Arpan Sen

December 08, 2009

It's a no-brainer: Every software product needs a regression suite. Traditionally, unit testing frameworks have been developed by testing teams on an ad hoc basis. Not only does this make maintenance of the test suite tough, things like monitoring program execution for time/memory performance become non-portable across operating systems. Taking such considerations into account, this series introduces you to the choices available for creating sophisticated regression frameworks using open source software. This article, part 1 in the series, explains the Boost unit testing framework for C/C++-based products.

View more content in this series

What is unit testing?

It's quite likely that a complex piece of `C/C++` code will have bugs, and trying to test for them after the code's written is akin to finding a needle in a haystack. A more prudent approach is to test individual pieces of code as they are written by adding small (unit) tests that specifically target certain areas—for example, some computationally intensive `C` function or some `C++` class claiming to model a certain data structure, like a queue. A regression suite built with this philosophy will then have a collection of unit tests and a test driver program that runs the tests and reports the results.

Generating a test for a specific function or class

For a complex piece of code like a text editor, an external tester cannot generate tests targeting specific routines—the tester won't have much idea about internal code organization. Where Boost comes in so handy is in *white box testing*: As a developer, you code the tests that do the semantic checks for your classes and functions. This process is of paramount importance, because future maintainers of your code are bound to tamper with the original logic at some point, and the unit tests will fail the moment something breaks. By using white box tests, it's often easier to see what's going wrong without having to use a debugger.

Consider the simple string class in Listing 1. This class is not robust, and you would use Boost to test it.

Listing 1. An uninspiring string class

```
#ifndef _MYSTRING
#define _MYSTRING

class mystring {
  char* buffer;
  int length;
  public:
    void setbuffer(char* s) { buffer = s; length = strlen(s); }
    char& operator[ ] (const int index) { return buffer[index]; }
    int size( ) { return length; }
 };

#endif
```

Some of the typical string-related checks would validate whether an empty string has 0 length, accessing out-of-index results in an error message or exception, and so on. Listing 2 shows some of the tests worth creating for any string implementation. To run the sources in Listing 2, you simply compile it with g++ (or any other standards-compliant `C++` compiler). Notice that no separate main function is needed, nor does the code use any link library: The unit_test.hpp header that's part of the Boost installation contains all the necessary definitions.

Listing 2. Unit tests for the string class

```
#define BOOST_TEST_MODULE stringtest
#include <boost/test/included/unit_test.hpp>
#include "./str.h"

BOOST_AUTO_TEST_SUITE (stringtest) // name of the test suite is stringtest

BOOST_AUTO_TEST_CASE (test1)
{
  mystring s;
  BOOST_CHECK(s.size() == 0);
}

BOOST_AUTO_TEST_CASE (test2)
{
  mystring s;
  s.setbuffer("hello world");
  BOOST_REQUIRE_EQUAL ('h', s[0]); // basic test
}

BOOST_AUTO_TEST_SUITE_END( )
```

The `BOOST_AUTO_TEST_SUITE` and `BOOST_AUTO_TEST_SUITE_END` macros indicate the start and end of the test suite, respectively. Individual tests reside between these macros, and in that sense their semantics are like `C++` namespaces. Each individual unit test is defined using the `BOOST_AUTO_TEST_CASE` macro. Listing 3 shows the output from the code in Listing 2.d

Listing 3. Output from the code in Listing 2

```
[arpan@tintin] ./a.out
Running 2 test cases...
test.cpp(10): error in "test1": check s.size() == 0 failed

*** 1 failure detected in test suite "stringtest"
```

Let's take a detailed look at the creation of the unit tests from the previous listings. The basic idea is to test individual class features using Boost-provided macros. `BOOST_CHECK` and `BOOST_REQUIRE_EQUAL` are some of the predefined macros (also known as *test tools*) that Boost provides to validate code output.

Boost test tools

Boost has a whole host of test tools, which are basically macros used to validate expressions. The three main categories of test tools are `BOOST_WARN`, `BOOST_CHECK`, and `BOOST_REQUIRE`. The difference between `BOOST_CHECK` and `BOOST_REQUIRE` is that in the former case, testing continues even if the assertion has failed, while in the latter case, it is deemed a critical error and testing stops. Listing 4 uses a trivial `C++` snippet to drive home the difference between these tool categories.

Listing 4. Using the three variants of the Boost testing tools

```
#define BOOST_TEST_MODULE enumtest
#include <boost/test/included/unit_test.hpp>

BOOST_AUTO_TEST_SUITE (enum-test)

BOOST_AUTO_TEST_CASE (test1)
{
  typedef enum {red = 8, blue, green = 1, yellow, black } color;
  color c = green;
  BOOST_WARN(sizeof(green) > sizeof(char));
  BOOST_CHECK(c == 2);
  BOOST_REQUIRE(yellow > red);
  BOOST_CHECK(black != 4);
}

BOOST_AUTO_TEST_SUITE_END( )
```

The first `BOOST_CHECK` fails, and so does the first `BOOST_REQUIRE`. However, the second `BOOST_CHECK` is not reached, because the code exits when the `BOOST_REQUIRE` fails. Listing 5 shows the output of the Listing 4 code.

Listing 5. Understanding the difference between BOOST_REQUIRE and BOOST_CHECK

```
[arpan@tintin] ./a.out
Running 1 test case...
e2.cpp(11): error in "test1": check c == 2 failed
e2.cpp(12): fatal error in "test1": critical check yellow > red failed

*** 2 failures detected in test suite "enumtest"
```

On similar lines, if you need to check some individual functions or class methods for corner cases, the easiest thing would be to create a new test and call that routine with arguments and expected values. Listing 6 provides an example.

Listing 6. Using a Boost test to check for function and class methods

```
BOOST_AUTO_TEST(functionTest1)
{
  BOOST_REQUIRE(myfunc1(99, ‘A’, 6.2) == 12);
  myClass o1(“hello world!\n”);
  BOOST_REQUIRE(o1.memoryNeeded( ) < 16);
}
```

Pattern matching

It is common to test the output generated from some function against a "golden log." `BOOST_CHECK` comes in handy here, too, and you also need to use the Boost library's `output_test_stream` class. The `output_test_stream` is initialized with the golden log file (run.log in the example below). The output from the `C/C++` function is fed onto this `output_test_stream` object, and then the `match_pattern` routine of this object is called. Listing 7 provides the details.

Listing 7. Pattern matching against a golden log file

```
#define BOOST_TEST_MODULE example
#include <boost/test/included/unit_test.hpp>
#include <boost/test/output_test_stream.hpp>
using boost::test_tools::output_test_stream;

BOOST_AUTO_TEST_SUITE ( test )

BOOST_AUTO_TEST_CASE( test )
{
    output_test_stream output( "run.log", true );
    output << predefined_user_func( );
    BOOST_CHECK( output.match_pattern() );
}

BOOST_AUTO_TEST_SUITE_END( )
```

Floating point comparison

One of the trickiest checks in regression setups is doing floating point comparisons. Looking at the code in Listing 8, all seems to have gone well—at least on the face of it.

Listing 8. Floating point comparison that does not work

```
#define BOOST_TEST_MODULE floatingTest
#include <boost/test/included/unit_test.hpp>
#include <cmath>

BOOST_AUTO_TEST_SUITE ( test )

BOOST_AUTO_TEST_CASE( test )
{
  float f1 = 567.0102;
  float result = sqrt(f1); // this could be my_sqrt; faster implementation
                                      // for some specific DSP like hardware
  BOOST_CHECK(f1 == result * result);
}

BOOST_AUTO_TEST_SUITE_END( )
```

On running this test, the `BOOST_CHECK` macro fails despite the fact that you're using the `sqrt` function provided as part of the standard library. So what is going wrong? The issue with floating point comparisons is that of precisions—`f1` and `result*result` start differing from a couple of places after the decimal point. To fix this situation, Boost test utilities provide the `BOOST_WARN_CLOSE_FRACTION`, `BOOST_CHECK_CLOSE_FRACTION`, and `BOOST_REQUIRE_CLOSE_FRACTION` macros. To use any of these three macros, you must include the predefined Boost header floating_point_comparison.hpp. The syntax for all three macros is the same, so this article discusses only the check variant (see Listing 9).

Listing 9. Syntax for the BOOST_CHECK_CLOSE_FRACTION macro

```
BOOST_CHECK_CLOSE_FRACTION (left-value, right-value, tolerance-limit);
```

Instead of using `BOOST_CHECK` in Listing 9, try `BOOST_CHECK_CLOSE_FRACTION` with a tolerance limit of 0.0001. Listing 10 shows how the code now looks.

Listing 10. Floating point comparison that works

```
#define BOOST_TEST_MODULE floatingTest
#include <boost/test/included/unit_test.hpp>
#include <boost/test/floating_point_comparison.hpp>
#include <cmath>

BOOST_AUTO_TEST_SUITE ( test )

BOOST_AUTO_TEST_CASE( test )
{
  float f1 = 567.01012;
  float result = sqrt(f1); // this could be my_sqrt; faster implementation
                                      // for some specific DSP like hardware
  BOOST_CHECK_CLOSE_FRACTION (f1, result * result, 0.0001);
}

BOOST_AUTO_TEST_SUITE_END( )
```

This code runs just fine. Now, replace the tolerance limit in Listing 10 with 0.0000001. Listing 11 shows the output.

Listing 11. An example where the comparison failed because of an unacceptable tolerance limit

```
[arpan@tintin] ./a.out
Running 1 test case...
sq.cpp(18): error in "test": difference between f1{567.010132} and
    result * result{567.010193} exceeds 1e-07

*** 1 failure detected in test suite "floatingTest"
```

Another common (and vicious) problem that keeps repeating itself in production software is the comparison of variables of type `double` and `float`. A nice feature of `BOOST_CHECK_CLOSE_FRACTION` is that it does not let you make such comparisons. The left and right values in the macro must be of the same type—`float` or `double`. In Listing 12, if `f1` were a double and `result` a float, you would see an error during compilation.

Listing 12. Error: the types of left and right arguments to BOOST_CHECK_CLOSE_FRACTION differ

```
[arpan@tintin] g++ sq.cpp -I/u/c/lib/boost
/u/c/lib/boost/boost/test/test_tools.hpp:
   In function
       `bool boost::test_tools::tt_detail::check_frwd(Pred,
       const boost::unit_test::lazy_ostream&,
   boost::test_tools::const_string, size_t,
   boost::test_tools::tt_detail::tool_level,
   boost::test_tools::tt_detail::check_type,
   const Arg0&, const char*,
   const Arg1&, const char*, const Arg2&, const char*)
   [with Pred = boost::test_tools::check_is_close_t, Arg0 = double,
   Arg1 = float, Arg2 = boost::test_tools::fraction_tolerance_t<double>]':
sq.cpp:18:   instantiated from here
/u/c/lib/boost/boost/test/test_tools.hpp:523: error: no match for call to
 `(boost::test_tools::check_is_close_t) (const double&, const float&,
     const boost::test_tools::fraction_tolerance_t<double>&)'
```

Custom predicate support

Boost test tools validate Boolean conditions. You can augment the test tools to support more complicated checks—say, to determine whether the contents of two lists are the same or whether a certain condition is valid on all elements of a vector. You can also extend the `BOOST_CHECK` macro to perform custom predicate support. Let's perform a custom check on the contents of a list generated from user-defined `C` function: Check whether the result has all elements greater than 1. The custom check function needs to return the type `boost::test_tools::predicate_result`. Listing 13 shows the details.

Listing 13. Validating complex predicates using Boost test tools

```
#define BOOST_TEST_MODULE example
#include <boost/test/included/unit_test.hpp>

boost::test_tools::predicate_result validate_list(std::list<int>& L1)
{
  std::list<int>::iterator it1 = L1.begin( );
  for (; it1 != L1.end( ); ++it1)
   {
     if (*it1 <= 1) return false;
   }
  return true;
}

BOOST_AUTO_TEST_SUITE ( test )

BOOST_AUTO_TEST_CASE( test )
{
    std::list<int>& list1 = user_defined_func( );
    BOOST_CHECK( validate_list(list1) );
}

BOOST_AUTO_TEST_SUITE_END( )
```

The `predicate_result` object has an implicit constructor that accepts a Boolean value, which explains why the code works fine even though the expected and actual return types of `validate_list` differ.

There is yet another way by which you could test complex predicates with Boost: the `BOOST_CHECK_PREDICATE` macro. The good part about using this macro is that it does not use the `predicate_result`. On the flip side, though, the syntax is a bit rough. The user needs to pass the function name and the argument(s) to the `BOOST_CHECK_PREDICATE` macro. Listing 14 has the same functionality as Listing 13 except for the use of different macros. Notice that the return type of `validate_result` is now Boolean.

Listing 14. The BOOST_CHECK_PREDICATE macro

```
#define BOOST_TEST_MODULE example
#include <boost/test/included/unit_test.hpp>

bool validate_list(std::list<int>& L1)
{
  std::list<int>::iterator it1 = L1.begin( );
  for (; it1 != L1.end( ); ++it1)
   {
     if (*it1 <= 1) return false;
   }
  return true;
}

BOOST_AUTO_TEST_SUITE ( test )

BOOST_AUTO_TEST_CASE( test )
{
    std::list<int>& list1 = user_defined_func( );
    BOOST_CHECK_PREDICATE( validate_list, list1 );
}

BOOST_AUTO_TEST_SUITE_END( )
```

Having multiple test suites in a file

It is possible to include multiple test suites into a single file. Each test suite must have a pair of `BOOST_AUTO_TEST_SUITE... BOOST_AUTO_TEST_SUITE_END` macros defined inside the file. Listing 15 shows two different test suites defined inside the same file. When running the regressions, run the executable with the predefined `–log_level=test_suite` option. As you can see from Listing 16, the output generated using this option is much more verbose and amenable to quick debugging.

Listing 15. Using multiple test suites in a single file

```
#define BOOST_TEST_MODULE Regression
#include <boost/test/included/unit_test.hpp>

typedef struct {
                 int c;
                 char d;
                 double e;
                 bool f;
               } Node;

typedef union  {
                 int c;
                 char d;
                 double e;
                 bool f;
               } Node2;

BOOST_AUTO_TEST_SUITE(Structure)
```

```
BOOST_AUTO_TEST_CASE(Test1)
{
    Node n;
    BOOST_CHECK(sizeof(n) < 12);
}

BOOST_AUTO_TEST_SUITE_END()

BOOST_AUTO_TEST_SUITE(Union)

BOOST_AUTO_TEST_CASE(Test1)
{
    Node2 n;
    BOOST_CHECK(sizeof(n) == sizeof(double));
}

BOOST_AUTO_TEST_SUITE_END()
```

Here's the output from the code in Listing 15:

Listing 16. Running multiple test suites with the –log_level option

```
[arpan@tintin] ./a.out --log_level=test_suite
Running 2 test cases...
Entering test suite "Regression"
Entering test suite "Structure"
Entering test case "Test1"
m2.cpp(23): error in "Test1": check sizeof(n) < 12 failed
Leaving test case "Test1"
Leaving test suite "Structure"
Entering test suite "Union"
Entering test case "Test1"
Leaving test case "Test1"
Leaving test suite "Union"
Leaving test suite "Regression"

*** 1 failure detected in test suite "Regression"
```

Understanding test suite organization

So far, this article has discussed Boost test utilities with test suites that have no hierarchy. Now, let's try to make a test suite using Boost that tests a software product the way an external tool user typically sees it done. Within the test framework itself, there typically will be multiple suites, each checking certain product features. For example, regression framework of a word processor should have suites that check for font support, different file formats, and so on. Each individual test suite will have multiple unit tests. Listing 17 provides an example of a test framework. Note that the entry point to the code must be the routine (aptly) named `init_unit_test_suite`.

Listing 17. Creating a master test suite for running regressions

```
#define BOOST_TEST_MODULE MasterTestSuite
#include <boost/test/included/unit_test.hpp>
using boost::unit_test;

test_suite*
init_unit_test_suite( int argc, char* argv[] )
{
    test_suite* ts1 = BOOST_TEST_SUITE( "test_suite1" );
    ts1->add( BOOST_TEST_CASE( &test_case1 ) );
    ts1->add( BOOST_TEST_CASE( &test_case2 ) );

    test_suite* ts2 = BOOST_TEST_SUITE( "test_suite2" );
    ts2->add( BOOST_TEST_CASE( &test_case3 ) );
    ts2->add( BOOST_TEST_CASE( &test_case4 ) );

    framework::master_test_suite().add( ts1 );
    framework::master_test_suite().add( ts2 );

    return 0;
}
```

Each test suite (for example, `ts1` in Listing 17) is created by using the macro `BOOST_TEST_SUITE`. The macro expects a string that is the name of the test suite. All the test suites are eventually added to the master test suite using the `add` method. Likewise, you create each test using the macro `BOOST_TEST_CASE` and add each test to the test suite, again using the `add` method. You can also add unit tests to the master test suite, although it is not a recommended practice. The `master_test_suite` method is defined as part of the `boost::unit_test::framework` namespace: It implements a singleton internally. The code in Listing 18, which comes from the Boost sources itself, explains how it works.

Listing 18. Understanding the master_test_suite method

```
master_test_suite_t&
master_test_suite()
{
    if( !s_frk_impl().m_master_test_suite )
        s_frk_impl().m_master_test_suite = new master_test_suite_t;

    return *s_frk_impl().m_master_test_suite;
}
```

The unit tests that are created using the `BOOST_TEST_CASE` macros accept function pointers as their input arguments. So in Listing 17, `test_case1`, `test_case2`, and so on, are void functions that the user is free to code the way he or she prefers. Note, however, that the Boost test setup uses a fair bit of memory on the heap; every call to `BOOST_TEST_SUITE` boils down to a new `boost::unit_test::test_suite(<test suite name>)`.

Fixtures

Conceptually, a *test fixture* is meant to set up an environment before a test is executed and clean up when the test is complete. Listing 19 provides a simple example.

Listing 19. A basic Boost fixture

```
#define BOOST_TEST_MODULE example
#include <boost/test/included/unit_test.hpp>
#include <iostream>

struct F {
    F() : i( 0 ) { std::cout << "setup" << std::endl; }
    ~F()          { std::cout << "teardown" << std::endl; }

    int i;
};

BOOST_AUTO_TEST_SUITE( test )

BOOST_FIXTURE_TEST_CASE( test_case1, F )
{
    BOOST_CHECK( i == 1 );
    ++i;
}

BOOST_AUTO_TEST_SUITE_END()
```

Listing 20 shows the output.

Listing 20. Output from boost fixture usage

```
[arpan@tintin] ./a.out
Running 1 test case...
setup
fix.cpp(16): error in "test_case1": check i == 1 failed
teardown

*** 1 failure detected in test suite "example"
```

Instead of using the `BOOST_AUTO_TEST_CASE` macro, this code uses `BOOST_FIXTURE_TEST_CASE`, which takes in an additional argument. The `constructor` and `destructor` methods of this object do the necessary setup and cleanup. A sneak peek into the boost header unit_test_suite.hpp confirms this (see Listing 21).

Listing 21. Boost fixture definition from header unit_test_suite.hpp

```
#define BOOST_FIXTURE_TEST_CASE( test_name, F )       \
struct test_name : public F { void test_method(); };                   \
                                                                        \
static void BOOST_AUTO_TC_INVOKER( test_name )()       \
{                                                                       \
    test_name t;                                                        \
    t.test_method();                                                    \
}                                                                       \
                                                                        \
struct BOOST_AUTO_TC_UNIQUE_ID( test_name ) {};       \
                                                                        \
BOOST_AUTO_TU_REGISTRAR( test_name )(                     \
    boost::unit_test::make_test_case(                       \
        &BOOST_AUTO_TC_INVOKER( test_name ), #test_name ),                 \
    boost::unit_test::ut_detail::auto_tc_exp_fail<                  \
        BOOST_AUTO_TC_UNIQUE_ID( test_name )>::instance()->value() );   \
                                                                        \
void test_name::test_method()                                           \
```

Internally, Boost is publicly deriving a class from the `struct F` (see Listing 19), then creating an object out of that class. As per the rules of public inheritance in `C++`, all protected and public variables of the `struct` class are directly accessible in the function that follows. Note that in Listing 19, the variable `i` being modified is one that belongs to the internal object `t` of type `F` (see Listing 20). It is quite okay to have a regression suite in which only couple of tests need some sort of explicit initialization and therefore use the fixture feature. Listing 22 has a test suite in which only one out of three tests is using fixture.

Listing 22. Boost test suite with a mix of fixture and non-fixture tests

```
#define BOOST_TEST_MODULE example
#include <boost/test/included/unit_test.hpp>
#include <iostream>

struct F {
    F() : i( 0 ) { std::cout << "setup" << std::endl; }
    ~F()          { std::cout << "teardown" << std::endl; }

    int i;
};

BOOST_AUTO_TEST_SUITE( test )

BOOST_FIXTURE_TEST_CASE( test_case1, F )
{
    BOOST_CHECK( i == 1 );
    ++i;
}

BOOST_AUTO_TEST_CASE( test_case2 )
{
    BOOST_REQUIRE( 2 > 1 );
}

BOOST_AUTO_TEST_CASE( test_case3 )
{
    int i = 1;
    BOOST_CHECK_EQUAL( i, 1 );
    ++i;
}

BOOST_AUTO_TEST_SUITE_END()
```

Listing 22 has fixtures that are defined and used on an individual test case. Boost also lets the user define and use global fixtures with the macro `BOOST_GLOBAL_FIXTURE (<Fixture Name>)`. You can define any number of global fixtures, which allows splitting up of initialization code. Listing 23 uses a global fixture.

Listing 23. Using global fixtures for initializing regression

```
#define BOOST_TEST_MODULE example
#include <boost/test/included/unit_test.hpp>
#include <iostream>

struct F {
    F()           { std::cout << "setup" << std::endl; }
    ~F()         { std::cout << "teardown" << std::endl; }
};

BOOST_AUTO_TEST_SUITE( test )
BOOST_GLOBAL_FIXTURE( F );
BOOST_AUTO_TEST_CASE( test_case1 )
{
    BOOST_CHECK( true );
}
BOOST_AUTO_TEST_SUITE_END()
```

For multiple fixtures, the setup and tear-down occurs in the order you declare them. In Listing 24, the constructor of `F` is called before `F2`; likewise for the destructor.

Listing 24. Using multiple global fixtures in regression

```
#define BOOST_TEST_MODULE example
#include <boost/test/included/unit_test.hpp>
#include <iostream>

struct F {
    F()           { std::cout << "setup" << std::endl; }
    ~F()         { std::cout << "teardown" << std::endl; }
};

struct F2 {
    F2()           { std::cout << "setup 2" << std::endl; }
    ~F2()         { std::cout << "teardown 2" << std::endl; }
};

BOOST_AUTO_TEST_SUITE( test )
BOOST_GLOBAL_FIXTURE( F );
BOOST_GLOBAL_FIXTURE( F2 );

BOOST_AUTO_TEST_CASE( test_case1 )
{
    BOOST_CHECK( true );
}
BOOST_AUTO_TEST_SUITE_END()
```

Note that you cannot use global fixtures as objects in individual tests. Nor can you directly access their public/protected non-static methods or variables inside a test.

Conclusion

That's it. This article has introduced you to one of the most powerful open source regression frameworks: Boost. You learned about the basic Boost checks, pattern matching, floating point comparison, custom checks, test suite organization—both manual and automatic—and fixtures. Be sure to look into the Boost documentation for further information. Further articles in this series will introduce other open source regression frameworks, like cppUnit.

Related topics

- Boost documentation: Check out this comprehensive guide to Boost test tools documentation.
- Boost policies and protocols: Be sure to check out this guide to the Boost test policies and protocols.
- Boost mailing list: Find a wealth of information, including fixes to common problems during installation and test.
- Boost Test: Download the Boost framework.
- IBM product evaluation versions: Get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Evaluate XL C/C++ for AIX