Unit Testing

Unit testing lets you test your code piece-by-piece, instead of all at once at the end of development. It also automates the testing process so you know when you introduce bugs into your code. Some go so far as to write tests before writing code. This is called test-driven development (TDD).

- Boost Unit Test Framework (UTF)
- Gcov: Code coverage tool

Basic Testing

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

int add(int x, int y) {
  return x + y;
}

BOOST_AUTO_TEST_CASE(addition) {
  BOOST_CHECK(add(3,3) == 6);
}
```

Test Assertions

- `BOOST_CHECK(bool)` :
  Error if the condition is false.
- `BOOST_CHECK_EQUAL(thing1, thing2)` :
  Error if the things are not equal.
- `BOOST_CHECK_NE(thing1, thing2)` :
  Error if the things are equal.

More Test Assertions

- `BOOST_CHECK_GE(thing1, thing2)`:
  Error if thing1 $<$ thing2.
- `BOOST_CHECK_GT`
- `BOOST_CHECK_LE`
- `BOOST_CHECK_LT`

Assertions Example

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

int add(int x, int y) {
  return x + y;
}

BOOST_AUTO_TEST_CASE(addition) {
  BOOST_CHECK(add(3,3) == 6);
  BOOST_CHECK_EQUAL(add(3,3), 6);
}

BOOST_AUTO_TEST_CASE(universe_is_sane) {
  BOOST_CHECK_GT(5,4);
  BOOST_CHECK_LE(3,65);
}
```

Testing Floating Point Values

- `#include<boost/test/floating_point_comparison.hpp>`
- `BOOST_CHECK_CLOSE(num1, num2, tolerance)`
- `BOOST_CHECK_CLOSE_FRACTION(num1, num2, percentage)`
- `BOOST_CHECK_SMALL(num, tolerance)`

Testing Exceptions

- `BOOST_THROW(expr, exception_class)`:
  Error if expr does not throw an exception of type exception_class
- `BOOST_EXCEPTION(expr, exception_class, check)`:
  Like `BOOST_THROW`, but calls check with the exception and errors if check returns false.

Exception Example

```
#define BOOST_TEST_MODULE "main"
#include<boost/test/included/unit_test.hpp>
#include<stdexcept>
using namespace std;

BOOST_AUTO_TEST_CASE(throw_exception) {
  BOOST_CHECK_THROW(throw runtime_error("whoops"),
      runtime_error);
}

bool check_error(const runtime_error& ex) {
  return strcmp(ex.what(), "whoops") == 0;
}

BOOST_AUTO_TEST_CASE(check_exception) {
  BOOST_CHECK_EXCEPTION(throw runtime_error("whoops"),
      runtime_error, check_error);
}
```

Test Assertion Levels

Boost has three levels of test assertions: `WARN`, `CHECK`, and `REQUIRE`.

- `WARN`: Print a message, but do not count as an error.
- `CHECK`: Print a message and count as an error.
- `REQUIRE`: Print a message, count as an error, and halt testing.

Organizing Tests

- If you have a lot of things to test, you can organize your tests into test cases.
- `BOOST_AUTO_TEST_SUITE(name)` makes a new test suite.
- `BOOST_AUTO_TEST_SUITE_END()` ends a test suite.

Organizing Tests

- If you have a lot of things to test, you can organize your tests into test cases.
- `BOOST_AUTO_TEST_SUITE(name)` makes a new test suite.
- `BOOST_AUTO_TEST_SUITE_END()` ends a test suite.
- Tip: For faster compiles, split out a `test_main.cpp` that includes the entire test library and then just include the test headers in each of your test files.
- Example: `test_funcs.cpp`

Test Fixtures

- Sometimes, especially when testing a class, you need to set up test objects in each test function.
- You can make a test fixture to do this automatically for you.

Test Fixtures

- Sometimes, especially when testing a class, you need to set up test objects in each test function.
- You can make a test fixture to do this automatically for you.
- A test fixture is just a class (or struct). You can do setup in the constructor and teardown in the destructor.
- Each test case is automagically made a member function, so you have access to all the fixture's member variables in your tests.
- You must use `BOOST_FIXTURE_TEST_CASE(name, fixture_class_name)` for your tests.
- Example: `test_vector.cpp`

Code Coverage

- Compile with `g++ -fprofile-arcs -ftest-coverage` and run your executable
- Run `gcov list_of_cpp_files`
- `gcov` command line options:
  - `-r` Only generate coverage for files in the current directory
  - `-n` Don't generate detailed output files
- If `gcov` generates detailed output files, it will show call counts for every line of the source files.