55 Lines of C++

Automated Unit Testing On The Cheap

Chuck Allison, Utah Valley University — February 20, 2014

Automated Unit Testing

Unit Testing is a crucial component of developing quality software. A "unit" is a small, logically indivisible software component, such as a function, class or module—any executable artifact that the developer perceives as a building block in constructing software.

Automated unit tests typically consist of a sequence of boolean expressions that are expected to be **true**. For example, suppose you want to test the basic operations of a stack class—**push**, **pop**, **top**, and **size**. Since the standard stack class template in C++ does not throw exceptions, we will test the following wrapper of **std::stack**, Stack, which does throw exceptions, to illustrate how to test exceptions as well.

```
#include <stack>
#include <stdexcept>

template<typename T>
class Stack {
    std::stack<T> data;
public:
    T& top() {
        if (data.size() == 0)
            throw std::logic_error("underflow");
        return data.top();
    }
    const T& top() const {
        if (data.size() == 0)
            throw std::logic_error("underflow");
        return data.top();
    }
    void push(const T& t) {
        data.push(t);
    }
    void pop() {
        if (data.size() == 0)
            throw std::logic_error("underflow");
        data.pop();
    }
    size_t size() const {
        return data.size();
    }
};
```

The test framework described in this article provides the following functions in the header file, **test.h**:

test_(<expr>)
throw_(<expr>,<exc_type>)
nothrow_( )
succeed( )
fail( )
report_( )

If the expression passed to **test_** is true (or non-zero), a "success counter" is incremented. If the expression evaluates to false (or zero), a "fail counter" is incremented and a message is printed. Execution continues in either case. Users call **fail_** to arbitrarily increment the fail counter (rarely needed). A **succeed_** function that increment the success counter is also provided.

The **throw_** function is used to make sure that exceptions are thrown for corner cases. The first argument is the expression intended to evoke an exception, and the second argument is the type of the exception. If the exception occurs, the success counter is incremented, otherwise a message is printed and the fail counter increases, just like with **test_**. The **nothrow_** function considers an exception a failure, and updates the fail count.

To review the final results, users call **report_**. All output operations go to standard output (**cout**).

The following test driver shows how to use **test.h** with the Stack class above.

```
#include "test.h"

int main() {
    Stack<int> stk;
    test_(stk.size() == 0);

    // Test Exceptions (top and pop are invalid on an empty stack)
    throw_(stk.top(),logic_error);
    throw_(stk.pop(),logic_error);
    nothrow_(stk.size());

    // Test push and top
    stk.push(1);
    test_(stk.top() == 1);
    test_(stk.size() == 1);
    stk.push(2);
    test_(stk.top() == 2);
    test_(stk.size() == 2);

    // Test pop
```

```
    stk.pop();
    test_(stk.top() == 1);
    test_(stk.size() == 1);
    stk.pop();
    test_(stk.size() == 0);
    throw_(stk.top(),logic_error);
    throw_(stk.pop(),logic_error);

    report_();
}
```

The output reported is:

```
Test Report:

  Number of Passes = 13
  Number of Failures = 0
```

To see what output looks like with errors, suppose we change the exception type expected in the first call to **throw_** to **runtime_error**, and the expression in the first call to test to **stk.top( ) == 10**. The output then becomes:

```
THROW FAILURE: stk.top() in file /Users/chuck/UVU/3370/stack_test.cpp on line 7
FAILURE: stk.top() == 10 in file /Users/chuck/UVU/3370/stack_test.cpp on line 12

Test Report:

  Number of Passes = 11
  Number of Failures = 2
```

The Code

Except for **succeed_**, the "functions" in test.h are actually macros that take advantage of the power of the preprocessor. In particular, this is how the name of the file and line numbers are obtained. Here's the code.

```
#ifndef TEST_H
#define TEST_H
#include <cstddef>
#include <iostream>
using std::size_t;

// Unit Test Scaffolding: Users call test_, fail_, succeed_, throw_, nothrow_, and report_
// AUTHOR: Chuck Allison (Creative Commons License, 2001 - 2014)

namespace {
    size_t nPass = 0;
    size_t nFail = 0;
    void do_fail(const char* text, const char* fileName, long lineNumber) {
```

```
        std::cout << "FAILURE: " << text << " in file " << fileName
                  << " on line " << lineNumber << std::endl;
        ++nFail;
    }
    void do_test(const char* condText, bool cond, const char* fileName, long lineNumber) {
        if (!cond)
            do_fail(condText, fileName, lineNumber);
        else
            ++nPass;
    }
    void succeed_() {
        ++nPass;
    }
    void report_() {
        std::cout << "\nTest Report:\n\n";
        std::cout << "\tNumber of Passes = " << nPass << std::endl;
        std::cout << "\tNumber of Failures = " << nFail << std::endl;
    }
}
#define test_(cond) do_test(#cond, cond, __FILE__, __LINE__)
#define fail_(text) do_fail(text, __FILE__, __LINE__)
#define throw_(expr,T)                      \
    try {                                   \
        expr;                               \
        std::cout << "THROW ";              \
        do_fail(#expr,__FILE__,__LINE__);   \
    } catch (T) {                           \
        ++nPass;                            \
    } catch (...) {                         \
        std::cout << "THROW ";              \
        do_fail(#expr,__FILE__,__LINE__);   \
    }

#define nothrow_(expr)                      \
    try {                                   \
        expr;                               \
        ++nPass;                            \
    } catch (...) {                         \
        std::cout << "NOTHROW ";            \
        do_fail(#expr,__FILE__,__LINE__);   \
    }
#endif
```

As you can see, the **test_** macro "string-izes" (via the # operator) the expression and passes the string representation, the evaluated expression, the file name (via the built-in **__FILE__** macro) and line number (the built-in **__LINE__** macro) to another function, **do_test**, which processes the expression appropriately. The function **do_fail** exists only to process an explicit user call to **fail_**.

The **throw_** macro attempts to catch an exception of the indicated type. If no exception occurs, or if an exception of an incompatible type occurs, a failure is flagged.

That's it! 55 lines of C++.