The Simplest Automated Unit Test Framework That Could Possibly Work

The title of this article is a variation on a theme from Extreme Programming (XP)[1]. XP is a code-centric discipline for getting software done right, on time, within budget, while having fun along the way. Quit laughing[2]. The XP approach is to take the best software practices to the extreme. For example, if code reviews are good, then code should be reviewed constantly, even as it's written, hence the XP practice of Pair Programming, where all code is written by two developers sharing a single workstation. One programmer pilots the keyboard while the other watches to catch mistakes and gives strategic guidance – then they switch roles as needed. The next day they may pair up with other folks. Likewise, if testing is good, then all tests should be automated and run many times per day. An ever-growing suite of unit tests should be executed whenever you create or modify any function, to ensure that the system is still stable. Furthermore, developers should integrate code into the complete, evolving system and run functional tests often (at least daily).

Confront Change

You've probably seen the old cartoon that says, "You guys start coding while I go find out what they want." I spent a number of years as a developer wondering why users couldn't figure out what they wanted before I started coding. I found it very frustrating to attend a weekly status meeting only to discover that what I completed the week before wasn't quite going to fit the bill because the analysts changed their mind. It's hard to reshape concrete while it's drying. Only once in my career have I had the luxury of a "finished spec." to code from[3].

Over the years, however, I've discovered that it is unreasonable to expect mere humans to be able to articulate software requirements in detail without sampling an evolving, working system. It's much better to specify a little, design a little, code a little, test a little, and then, after evaluating the outcome, do it all over again. The ability to develop from soup to nuts in such an iterative fashion is one of the great advances of this object-oriented era in software history. But it requires nimble programmers who can craft resilient (i.e., slow-drying) code. Change is hard.

Ironically, there is another kind of change that good programmers want desperately to perform, but that management has always opposed: improving the physical design of working code. What maintenance programmer hasn't had occasion to curse the aging, flagship company product as a convoluted patchwork of spaghetti, wholly resistant to modification? The fear of tampering with a functioning system, while not totally unfounded, robs code of the resilience it needs to endure. "If it ain't broke, don't fix it" eventually gives way to "We can't fix it – rewrite it." Change is necessary.

Fortunately, in our day there has arisen the discipline of Refactoring, the art of internally restructuring code to improve its design, without changing the functionality visible to the user[4]. Such tweaks include extracting a new function from another, or its inverse, combining methods; replacing a method with an object; parameterizing a method or class; or replacing conditionals with polymorphism. Now that this process of improving a program's internal structure has a name and the support of industry luminaries, we will likely be seeing more of it in the workplace.

It Was Working When I Laid It Down

But should the force for change come from analysts or programmers, there is still the risk that changes today will break what worked yesterday. What we're all after is a way to build code that withstands the winds of change and actually improves over time.

There are many practices that purport to support this quick-on-your-feet motif, of which XP is only one. In this article I explore what I think is the key to making incremental development work: a ridiculously easy-to-use automated unit test framework, which I have implemented in C++, C, and Java.

Unit tests are what developers write to gain the confidence to say the two most important things that any developer can say:

1. I understand the requirements
2. My code meets those requirements

I can't think of a better way ensure that you know what the code you're about to write should do than to write the unit tests first. This simple exercise helps focus the mind on the task ahead, and will likely lead to working code faster than just jumping into coding. Or, to express it in XP terms, Testing + Programming is faster than just Programming. Writing tests first also puts you on guard up front against boundary conditions that might cause your code to break, so your code is more robust right out of the chute.

Once you have code that passes all your tests, you then have the peace of mind that if the system you contribute to isn't working, it's not your fault. The phrase, "All my tests pass" is a powerful trump card in the workplace that cuts through any amount of politics and hand waving.

Writing good unit tests is so important that I'm amazed I didn't discover its value earlier in my career. Let me rephrase that. I'm not really amazed, just disappointed. I still remember what turned me off to formal testing at my first job right out of school. The testing manager (yes, we had one in 1978!) asked me to write a unit test *plan*, whatever *that* was. Being an impatient youth I thought it was silly to waste time writing a plan – why not just write the test? That encounter soured me on the idea of formal test plans for years thereafter.

Automated Testing

I think most developers, like myself, would rather write code than write about code. But what does a unit test look like? Quite often developers just verify that some well-behaved input produces the expected output, which they inspect visually. There are two dangers in this approach. First, programs don't always receive just well-behaved input. We all know that we should test the boundaries of programs input, but it's hard to think about it when you're trying to just get things working. If you write the test for a function first before you start coding, you can wear your QA hat and ask yourself, "What could possibly make this break?" Code up a test that will prove the function you'll write isn't broken, then put on your developer hat and make it happen. You'll write better code than if you hadn't written the test first.

The second danger is inspecting output visually to see if things work. It's fine for toy programs, but production software is too complex for that kind of activity. It is tedious and error-prone to visually inspect program output to see if a test passed. Most any such thing a human can do a computer can do, but without error. It's better to formulate tests as collections of boolean expressions and have the test program report any failures.

As an example, suppose you need to build a Date class in C++ that does the following:

- A date can be initialized with a string (YYYY-MM-DD), 3 integers (Y,M,D), or nothing (today's date).
- A date object can yield its year, month, and day or a string of the form "YYYY-MM-DD".
- All relational comparisons are available, as well as computing the duration between two dates (in years, months, and days), and adding or subtracting a duration.
- Dates need to span an arbitrary number of centuries (e.g., 1600-2200)

Your class could store three integers representing the year, month, and day (just be sure the year is 16 bits or more to satisfy the last bullet above). The interface for your Date class might look like this:

```
// date.h
#include <string>
#include "duration.h"  // a 3-int struct

class Date
{
public:
    Date();
    Date(int year, int month, int day);
    Date(const std::string&);

    int getYear() const;
    int getMonth() const;
    int getDay() const;
    std::string toString() const;
```

```
    friend Duration duration(const Date&, const Date&);
    friend bool operator<(const Date&, const Date&);
    friend bool operator<=(const Date&, const Date&);
    friend bool operator>(const Date&, const Date&);
    friend bool operator>=(const Date&, const Date&);
    friend bool operator==(const Date&, const Date&);
    friend bool operator!=(const Date&, const Date&);
private:
    …
};
```

You can now write tests for the functions you want to implement first, something like the following:

```
int main()
{
    Date mybday(1951,10,1);
    Date today;

    test(mybday < today);
    test(mybday <= today);
    test(mybday != today);
    test(mybday == mybday);

    cout << "Passed: " << getNumPassed() << ", Failed: "
         << getNumFailed() << endl;
}

/* Output:
Passed: 4, Failed: 0
*/
```

In this case you can assume that the function `test` maintains the global variables `nPass` and `nFail`. The only visual inspection you do is to read the final score. If a test failed, then `test` would print out an appropriate message. The framework described below has such a test function, among other things.

As you continue in the test-and-code cycle you'll want to build a suite of tests that are always available to keep all your related classes in good shape through any future maintenance. As requirements change, you add or modify tests accordingly.

The TestSuite Framework

As you learn more about XP you'll discover that there are some automated unit test tools available for download, such as JUnit for Java and CppUnit for C++. These are brilliantly designed and implemented, but I want something even simpler. I want something that I can not only easily use but also understand internally and even tweak if necessary. And I don't need no steenking GUI, thank you very much. So, in the spirit of TheSimplestThingThatCouldPossibleWork, I present the TestSuite Framework, as I call it, which consists of two classes: `Test` and `Suite`. `Test` is an abstract class you derive from to override the `run` method, which should in turn call `test_`[5] for each boolean test condition you define. For the `Date` class above you could do something like the following:

```
// DateTest.h: Use the test class
#include "test.h"
#include "date.h"

class DateTest : public Test
{
    Date mybday;
    Date today;

public:
    DateTest()
        : mybday(1951, 10,1)
```

```
    {}
    void run()
    {
        testOps();
    }
    void testOps()
    {
        test_(mybday < today);
        test_(mybday <= today);
        test_(mybday != today);
        test_(mybday == mybday);
    }
};
```

You can now run the test very easily, like this:

```
// datetest.cpp: Automated Testing (with a Framework)
#include <iostream>
#include "DateTest.h"
using namespace std;

int main()
{
    DateTest test;
    test.run();
    test.report();
}

/* Output:
Test "DateTest":
      Passed: 4    Failed: 0
*/
```

As development continues on the `Date` class, you'll add other tests called from `DateTest::run`, and then execute the main program to see if they all pass.

The `Test` class uses RTTI to get the name of your class (e.g., `DateTest`) for the report[6]. The output stream defaults to `cout`, but there is a `setStream` method that lets you specify the output stream, and `report` sends output to that stream. See Figures 1 and 2 for the definition and implementation of `Test`. No rocket science here. `Test` just keeps track of the number of successes and failures as well as the stream where you want `Test::report` to print the results. `test_` and `fail_` are macros so that they can include filename and line number information available from the preprocessor (which is not available in the Java version, of course).

In addition to `test_` there are the `succeed_` and `fail_` functions for cases where a boolean test won't do. For example, a simple stack class template might have the following specification:

```
// Stack.h
…

template<typename T>
class Stack
{
public:
    Stack(size_t) throw(StackError, bad_alloc);
    ~Stack();

    void push(const T&) throw(StackError);
    T pop() throw(StackError);
    T top() const throw(StackError);
    size_t size() const;
    …
};
```

Before giving any thought at all to implementation it's easy to come up with general categories of tests for this class:

```
class StackTest : public Test
{
    enum {SIZE = 5};
    Stack<int> stk;

public:
    StackTest() : stk(SIZE)
    {}

    void run()
    {
        testUnderflow();
        testPopulate();
        testOverflow();
        testPop();
        testBadSize();
    }
…
```

But to test whether exceptions are working correctly, you have to generate an exception and call `succeed_` or `fail_` explicitly, as `StackTest::testBadSize` class illustrates:

```
    void testBadSize()
    {
        try
        {
            Stack<int> s(0);
            fail_("Bad Size");
        }
        catch (StackError&)
        {
            succeed_();
        }
    }
```

Since a stack of size zero is prohibited, "success" in this case means that a `StackError` exception was caught, so I have to call `succeed_` explicitly. The implementation of the `Stack` class template is in Figure 3 and `StackTest` and its results appear in Figure 4.

Test Suites

Real projects usually contain many classes, so there needs to be a way to group tests together so you can just push a single button to test the entire project. The `Suite` class allows you to collect tests into a functional unit. You add a derived `Test` object to a `Suite` with the `addTest` method, or you can swallow an entire existing `Suite` with `addSuite`. To illustrate, I have four modules that work together to provide real-world date support. `JulianDate` is what I call an API of C functions that implement the basics of Julian Date arithmetic[7]. The `JulianTime` module uses `JulianDate` but adds support for hours, minutes, and seconds. `Date` uses `JulianDate` and adds Nice Things for C++ users – likewise `Time` and `JulianTime`. Since these classes are all interrelated and are packaged as a single library, I want to test them together. After defining each test class (derived from `Test`, of course), I group them into a suite entitled "Date and Time tests". Here's an actual test run:

```
// test Suite for the Date projects
#include <iostream>
#include "suite.h"
#include "JulianDateTest.h"
#include "JulianTimeTest.h"
#include "DateTest.h"
```

```
#include "TimeTest.h"
using namespace std;

int main()
{
    Suite s("Date and Time Tests”, &cout);

    s.addTest(new JulianDateTest);
    s.addTest(new JulianTimeTest);
    s.addTest(new DateTest);
    s.addTest(new TimeTest);
    s.run();
    long nFail = s.report();
    s.free();
    cout << "\nTotal failures: " << nFail << endl;
    return nFail;
}

/* Output:
Suite "Date and Time Tests"
===========================
Test "class MonthInfoTest":
       Passed: 18    Failed: 0
Test "class JulianDate":
       Passed: 36    Failed: 0
Test "class JulianTime":
       Passed: 29    Failed: 0
Test "class Date":
       Passed: 57    Failed: 0
Test "class Time":
       Passed: 84    Failed: 0
===========================

Total failures: 0
*/
```

`Suite::run` calls `Test::run` for each of its contained tests - likewise for `Suite::report`. Individual test results can be written to separate streams, if desired. If the test passed to `addSuite` has a stream pointer assigned already, it keeps it; otherwise it gets its stream from the Suite object. The code for `Suite` is in Figures 5 and 6. As you can see, `Suite` just holds a vector of pointers to `Test`. When it’s time to run each test, it just loops through the tests in the vector calling their `run` method.

No C++? No Problem!

After I showed `TestSuite` to the developers where I used to work, a number of programmers were a little chagrined that they couldn’t use it, because they were developing strictly in C. It took all of one afternoon to change all the classes to structs and rename things accordingly to come up with a C version. A simple test example is in Figure 7. The Java version was even easier (of course!). You can get all the code on the C/C++ Users Journal web site (http://www.cuj.com).

Summary

It takes some discipline to write unit tests before you code, but if you have an automated tool, it’s a lot easier. I just add a project in my IDE for a test suite for each project, and switch back and forth between the test and the real code as needed. There’s no conceptual baggage, no extra test scripting language to learn, no worries – just point, click, and test!

Figure 1 – The Test Class Header

```
// Test.h
#ifndef TEST_H
#define TEST_H
```

```
#include <string>
#include <iostream>
#include <cassert>
using std::string;
using std::ostream;
using std::cout;

// The following have underscores because they are macros.
// (The motivation was to avoid a conflict with ios::fail).
// For consistency, succeed_() also has an underscore.

#define test_(cond) \
  do_test(cond, #cond, __FILE__, __LINE__)
#define fail_(str) \
  do_fail(str, __FILE__, __LINE__)

class Test {
public:
  Test(ostream* osptr = &cout);
  virtual ~Test(){}
  virtual void run() = 0;
  long getNumPassed() const;
  long getNumFailed() const;
  const ostream* getStream() const;
  void setStream(ostream* osptr);
  void succeed_();
  long report() const;
  virtual void reset();
protected:
  void do_test(bool cond, const string& lbl,
               const char* fname, long lineno);
  void do_fail(const string& lbl,
               const char* fname, long lineno);
private:
  ostream* osptr;
  long nPass;
  long nFail;
  // Disallowed:
  Test(const Test&);
  Test& operator=(const Test&);
};

inline Test::Test(ostream* osptr) {
  this->osptr = osptr;
  assert(osptr);
  nPass = nFail = 0;
}

inline long Test::getNumPassed() const {
  return nPass;
}

inline long Test::getNumFailed() const {
  return nFail;
}

inline const ostream* Test::getStream() const {
  return osptr;
}

inline void Test::setStream(ostream* osptr) {
  this->osptr = osptr;
}

inline void Test::succeed_() {
```

```
  ++nPass;
}

inline void Test::reset() {
  nPass = nFail = 0;
}
#endif // TEST_H
```

Figure 2 – The Test Class Implementation

```
// test.cpp
#include "Test.h"
#include <iostream>
#include <typeinfo> // Visual C++ requires /GR""
using namespace std;

void Test::do_test(bool cond,
                   const std::string& lbl,
                   const char* fname,
                   long lineno){
  if (!cond)
    do_fail(lbl, fname, lineno);
  else
    succeed_();
}

void Test::do_fail(const std::string& lbl,
                   const char* fname,
                   long lineno){
  ++nFail;
  if (osptr){
    *osptr << typeid(*this).name()
              << "failure: (" << lbl << ") , "
              << fname
              << " (line " << lineno << ")\n";
  }
}

long Test::report() const {
  if (osptr){
    *osptr << "Test \"" << typeid(*this).name()
              << "\":\n\tPassed: " << nPass
              << "\tFailed: " << nFail
              << endl;
  }
  return nFail;
}
```

Figure 3 – A Simple Stack Template

```
// Stack.h
#include <cassert>
#include <cstddef>
#include <stdexcept>
#include <string>
#include <new>

using std::logic_error;
using std::string;
using std::bad_alloc;

// MS std namespace work-around
#ifndef _MSC_VER
using std::size_t;
#endif

class StackError : public logic_error
```

```
{
public:
    StackError(const string& s)
        : logic_error(s)
    {}
};

template<typename T>
class Stack
{
public:
    Stack(size_t) throw(StackError, bad_alloc);
    ~Stack();

    void push(const T&) throw(StackError);
    T pop() throw(StackError);
    T top() const throw(StackError);
    size_t size() const;

private:
    T* data;
    size_t max;
    size_t ptr;
};

template<typename T>
inline Stack<T>::~Stack()
{
    delete [] data;
    max = ptr = 0;
}

template<typename T>
inline size_t Stack<T>::size() const
{
    return ptr;
}

template<typename T>
Stack<T>::Stack(size_t siz) throw(StackError, bad_alloc)
{
    if (siz == 0)
        throw StackError("bad size in Stack(size_t)");
    data = new T[siz];
    max = siz;
    ptr = 0;
}

template<typename T>
void Stack<T>::push(const T& x) throw(StackError)
{
    if (ptr == max)
        throw StackError("stack overflow");

    assert(ptr < max);
    data[ptr++] = x;
}

template<typename T>
T Stack<T>::pop() throw(StackError)
{
    if (ptr == 0)
        throw StackError("stack underflow");
    return data[--ptr];
}
```

```
template<typename T>
T Stack<T>::top() const throw(StackError)
{
    if (ptr == 0)
        throw StackError("stack underflow");
    return data[ptr - 1];
}
```

Figure 4 – Testing the Stack Template

```
#include "Stack.h"
#include "test.h"
#include <iostream>
using namespace std;

class StackTest : public Test
{
    enum {SIZE = 5};
    Stack<int> stk;

public:
    StackTest() : stk(SIZE)
    {}

    void run()
    {
        testUnderflow();
        testPopulate();
        testOverflow();
        testPop();
        testBadSize();
    }

    void testBadSize()
    {
        try
        {
            Stack<int> s(0);
            fail_("Bad Size");
        }
        catch (StackError&)
        {
            succeed_();
        }
    }

    void testUnderflow()
    {
        test_(stk.size() == 0);

        try
        {
            stk.top();
            fail_("Underflow");
        }
        catch (StackError&)
        {
            succeed_();
        }

        try
        {
            stk.pop();
            fail_("Underflow");
        }
```

```
        catch (StackError&)
        {
            succeed_();
        }
    }

    void testPopulate()
    {
        try
        {
            for (int i = 0; i < SIZE; ++i)
                stk.push(i);
            succeed_();
        }
        catch (StackError&)
        {
            fail_("Populate");
        }

        test_(stk.size() == SIZE);
        test_(stk.top() == SIZE-1);
    }

    void testOverflow()
    {
        try
        {
            stk.push(SIZE);
            fail_("Overflow");
        }
        catch (StackError&)
        {
            succeed_();
        }
    }

    void testPop()
    {
        for (int i = 0; i < SIZE; ++i)
            test_(stk.pop() == SIZE-i-1);
        test_(stk.size() == 0);
    }
};

int main()
{
    StackTest t;
    t.setStream(&cout);
    t.run();
    t.report();
}

/* Output:
Test "class StackTest":
       Passed: 14    Failed: 0
*/
```

Figure 5 – The File Suite.h

```
// suite.h
#ifndef SUITE_H
#define SUITE_H
#include "../TestSuite/Test.h"
#include <vector>
using std::vector;
```

```
class TestSuiteError;

class Suite {
public:
  Suite(const string& name, ostream* osptr = &cout);
  string getName() const;
  long getNumPassed() const;
  long getNumFailed() const;
  const ostream* getStream() const;
  void setStream(ostream* osptr);
  void addTest(Test* t) throw (TestSuiteError);
  void addSuite(const Suite&)
    throw(TestSuiteError);
  void run();  // Calls Test::run() repeatedly
  long report() const;
  void free(); // Deletes tests
private:
  string name;
  ostream* osptr;
  vector<Test*> tests;
  void reset();
  // Disallowed ops:
  Suite(const Suite&);
  Suite& operator=(const Suite&);
};

inline
Suite::Suite(const string& name, ostream* osptr)
   : name(name) {
  this->osptr = osptr;
}

inline string Suite::getName() const {
  return name;
}

inline const ostream* Suite::getStream() const {
  return osptr;
}

inline void Suite::setStream(ostream* osptr) {
  this->osptr = osptr;
}
#endif
```

Figue 5 – The File Suite.cpp

```
// suite.cpp
#include "Suite.h"
#include <iostream>
#include <stdexcept>
#include <cassert>
using namespace std;

class TestSuiteError : public logic_error {
public:
  TestSuiteError(const string& s = "")
    : logic_error(s) {}
};

void Suite::addTest(Test* t)
  throw(TestSuiteError) {
  // Make sure test has a stream:
  if (t == 0)
    throw TestSuiteError(
      "Null test in Suite::addTest");
```

```
  else if (osptr != 0 && t->getStream() == 0)
    t->setStream(osptr);
  tests.push_back(t);
  t->reset();
}

void Suite::addSuite(const Suite& s)
  throw(TestSuiteError) {
  for (size_t i = 0; i < s.tests.size(); ++i)
    addTest(s.tests[i]);
}

void Suite::free() {
  // This is not a destructor because tests
  // don't have to be on the heap.
  for (size_t i = 0; i < tests.size(); ++i) {
    delete tests[i];
    tests[i] = 0;
  }
}

void Suite::run() {
  reset();
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    tests[i]->run();
  }
}

long Suite::report() const {
  if (osptr) {
    long totFail = 0;
    *osptr << "Suite \"" << name
             << "\"\n=======";
    size_t i;
    for (i = 0; i < name.size(); ++i)
      *osptr << '=';
    *osptr << "=\n";
    for (i = 0; i < tests.size(); ++i) {
      assert(tests[i]);
      totFail += tests[i]->report();
    }
    *osptr << "=======";
    for (i = 0; i < name.size(); ++i)
      *osptr << '=';
    *osptr << "=\n";
    return totFail;
  }
  else
    return getNumFailed();
}

long Suite::getNumPassed() const {
  long totPass = 0;
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    totPass += tests[i]->getNumPassed();
  }
  return totPass;
}

long Suite::getNumFailed() const {
  long totFail = 0;
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
```

```
    totFail += tests[i]->getNumFailed();
  }
  return totFail;
}

void Suite::reset() {
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    tests[i]->reset();
  }
}
```

Figure 7 – A C test for a complex number function

```
#include <stdio.h>
#include <assert.h>
#include "ctest.h"  /* the Test "class" */

/* Stuff to test (usually #include'd) */
typedef struct
{
    double real, imag;
} complex;

complex c_add(complex c1, complex c2)
{
    complex r;
    r.real = c1.real + c2.real;
    r.imag = c1.imag + c2.imag;
    return r;
}

complex c1 = {1.0, 1.0};
complex c2 = {2.0, 2.0};
complex c3 = {3.0, 3.0};

void testEqual(Test* pTest)
{
    complex z = {1.0, 1.0};
    ct_test(pTest, z.real == c1.real && z.imag == c1.imag);
    ct_test(pTest, z.real != c2.real && z.imag != c2.imag);
}

void testAdd(Test* pTest)
{
    complex r = c_add(c1, c2);
    ct_test(pTest, r.real == c3.real && r.imag == c3.imag);
}

int main()
{
    Test* pTest = ct_create("Complex", NULL);
    bool rc = ct_addTestFun(pTest, testEqual);
    rc = ct_addTestFun(pTest, testAdd);
    assert(rc);
    ct_setStream(pTest, stdout);
    ct_run(pTest);
    ct_report(pTest);
    ct_destroy(pTest);
    return 0;
}

/* Output:
Test "Complex":
       Passed: 3
       Failed: 0
```

```
*/
```

---

[1] See Kent Beck's book, *eXtreme Programming Explained: Embrace Change* (Addison-Wesley, 2000, ISBN 0-201-61641-6), or visit www.Xprogramming.com for more information on XP. The XP theme from which this article derives its title is DoTheSimplestThingThatCouldPossiblyWork.

[2] "Stop laughing" are Kent's own words. See ibid, p. xvi.

[3] Yes, you guessed it. It was a government project. For all the paperwork, at least I knew what to do.

[4] The seminal work on this subject is of course Martin Fowler's *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 2000, ISBN 0-201-48567-2). See www.refactoring.com.

[5] The trailing underscore is necessary so as not to conflict with ios::fail.

[6] If you're using Microsoft Visual C++, you need to specify the compile option `/GR""`. If you don't, you'll get an access violation at runtime.

[7] These functions hold a Julian Day as a `long`. The `JulianTime` functions hold a `JulianDate` plus hour, minute, and second all in one `double`. These are free functions because I want them to support both C and C++. The `Date` and `Time` classes wrap these modules and add more functionality. For more on Julian day arithmetic, see Chapter 19 of my book, C & C++ Code Capsules, P-H, 1998.

Chuck Allison

Chuck Allison is Assistant Professor of Computer Science at Utah Valley State College. He is a contributing member of the C++ Standards Committee, Senior Editor for the C/C++ Users Journal, and author of the book "C & C++ Code Capsules: A Guide for Practitioners" (Prentice-Hall, 1998). He has taught C++ and Java extensively at corporations throughout the United States. You can contact Chuck through his website www.freshsources.com.