CSCI 1300

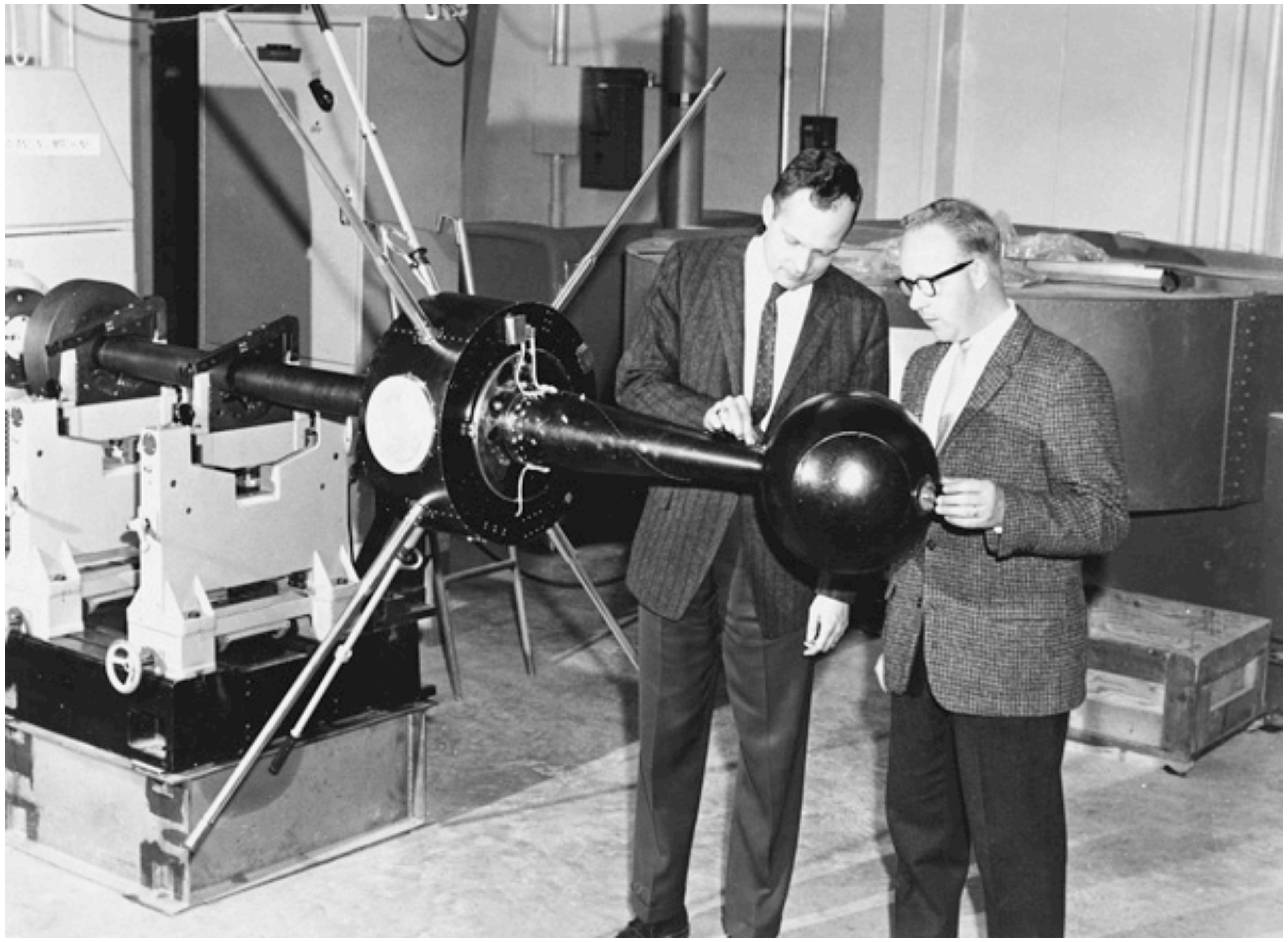
Intro to Computing

Gabe Johnson

Lecture 43 May 1, 2013

Software Engineering
Testing, Measuring, Collaborating

Enginering != Hacking



You don’t get to space by messing around until it looks like it might work.

Enginering != Hacking

This talk is about **Software Engineering**.

To be clear, there's lots of different mindsets and philosophical approaches to writing code, and software engineering is just one of them.

I'll talk more about these differences on Friday, but for today, we're talking strictly about engineering.

Engineering, n:

The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation or safety to life and property.

—American Engineers' Council for Professional Development

Engineering, n:

The **creative** application of **scientific** principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation or safety to life and property.

—American Engineers' Council for Professional Development

Engineering, n:

The creative application of scientific principles to **design or develop** structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation or safety to life and property.

—American Engineers' Council for Professional Development

Engineering, n:

The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or **to construct or operate** the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation or safety to life and property.

—American Engineers' Council for Professional Development

Engineering, n:

The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or **to forecast** their behavior under specific operating conditions; all as respects an intended function, economics of operation or safety to life and property.

—American Engineers' Council for Professional Development

Engineering, n:

The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an **intended function**, economics of operation or safety to life and property.

—*American Engineers' Council for Professional Development*

Engineering, translated:

Creative and scientific.

About designing what could be.

Building and using what we designed.

Use measurements to build and predict.

All for some purpose.

... a pretty big topic, eh?

Design: Next Time

I’ll talk in much greater depth about design next time.

For this talk: deciding why we should build something, and what generally it should do, will be taken as a given.

You boss says “make this!” and you do. But how?

Building Software

This class has given you a *very high overview* of the fundamental task of writing code.

Our programs have been dozens, or maybe hundreds of lines of code.

A real application involves hundreds of thousands of lines of code (maybe more!) written by teams of people, often in several languages using different toolkits and technologies.

Measuring Software

As we build software (and after we've built it) we have to be able to measure it in various ways:

- Does it work?
- Does it work fast enough?
- Does it cause unforeseen problems?
- Can we use it to make something else?
- Where are the bugs?
- How severe are the bugs?
- Does it handle extreme conditions X, Y, Z?

Testing Saves Lives

This is a medical device that is supposed to dose patients with a relatively low amount of radiation in most use cases.

Most. Not all.

Therac Design Flaws

Therac 25 is often given as an example of deeply flawed software design *and* user interface design.

For our purposes: the software that controls this thing was *not* designed or developed in a way that controlled software testing could be performed.

The safeties were supposedly done in software. But there were conditions where the safeties didn't work. This caused severe radiation poisoning and deaths.

Test-Driven Design Helps

One way to reduce (not eliminate) buggy software is to determine in advance **how** something should work, what the data structure and algorithmic **invariants** are, what the possible **use cases** are, and what environmental or situational **conditions** might alter performance.

Avoid Hubris

Requirements are hard.

Coming up with this list is a job of itself, so many programmers skip this step because they think they are Teh Best Coderz Evah™, but they're wrong.

Requirements → Tests

When we have some requirements (like function and data structure invariants), we have a great starting place for writing tests:

```
/**
 * This function returns the sum of squares of the input array.
 **/
 int get_sum_of_squares(int[] input) { ... }
```

Without looking at this code, we can gather some requirements based on the documentation, and what we know from 7th grade math class.

1. What's it do?

```
/**
 * This function returns the sum of squares of the input array.
 **/
 int get_sum_of_squares(int[] input) { ... }
```

We know it is going to compute some squares, and add them together.

2. What's the input?

```
/**
 * This function returns the sum of squares of the input array.
 **/
 int get_sum_of_squares(int[] input) { ... }
```

We see from the function signature that it takes an array of integers, conveniently called 'input'.

Think about what could be *wrong* with this input. Empty? Very big? Negative numbers? Zeros? Some of these might be problems, others might not be. You have to engage your brain.

3. What’s the output?

```
/**
 * This function returns the sum of squares of the input array.
 **/
 int get_sum_of_squares(int[] input) { ... }
```

We see from the function signature that it returns a single integer. So we know that inside this function definition we’ll need a *return* statement, and we have to guarantee that it actually runs.

Also: we know that squares are all positive, so the result should also be positive.

4. What potential problems?

```
/**
 * This function returns the sum of squares of the input array.
 **/
 int get_sum_of_squares(int[] input) { ... }
```

Say we get input that is very large. Maybe one of those numbers is so large that when we take its square, we don't have enough room inside this computer's *int* data type to store the result. Or, maybe we get that problem when we start adding the squares together.

5. Faulty Definition?

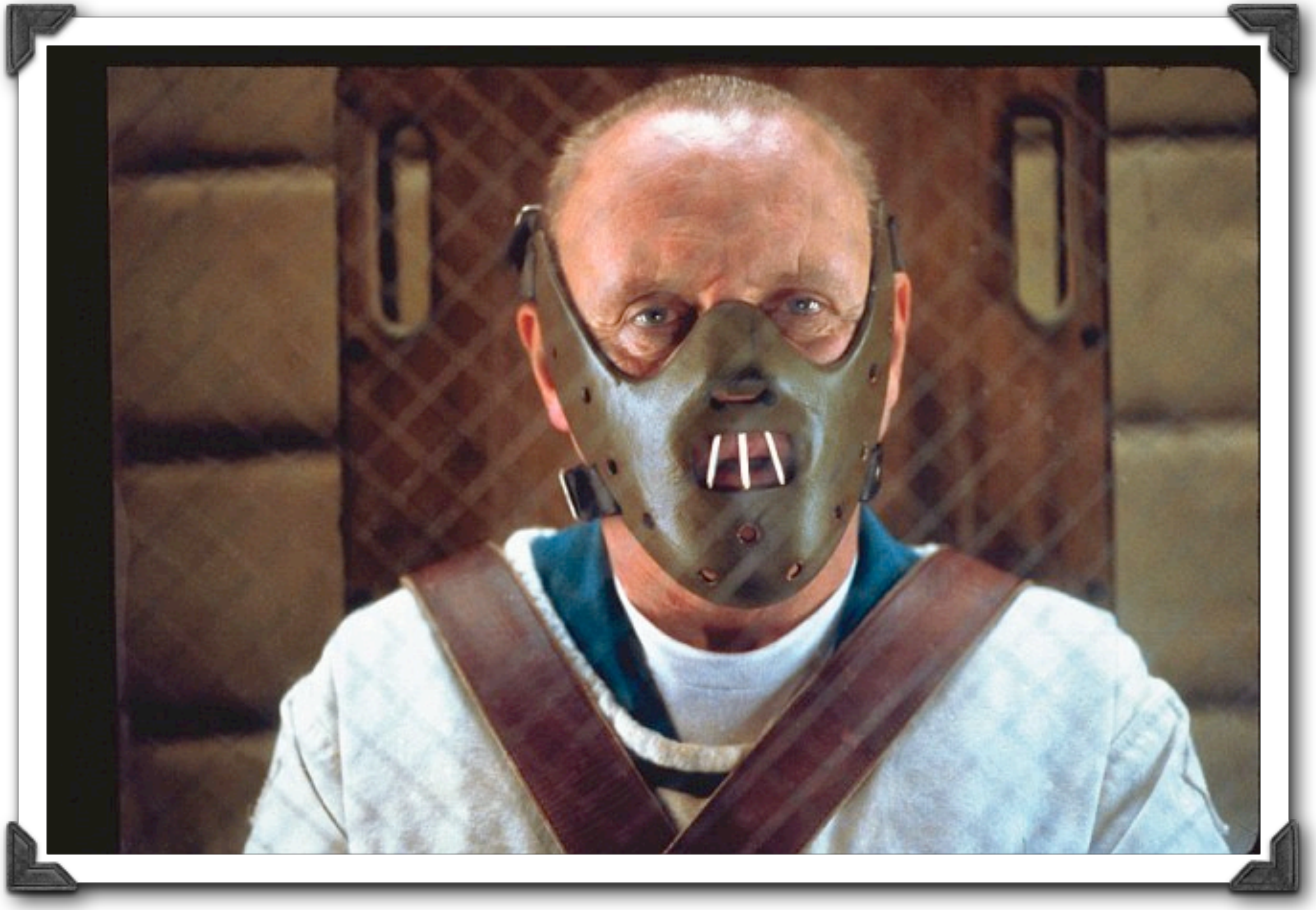
```
/**
 * This function returns the sum of squares of the input array.
 **/
 int get_sum_of_squares(int[] input) { ... }
```

We don’t even know how much data is contained in the *input* array. In some languages we don’t need this info passed in explicitly (Java, Python) but others we must be given the input size. Maybe the person who gave you this definition messed up? Maybe they spend their days hacking Java and it didn’t occur to them that C++ is different. This is *always* a possibility.

Write Some Tests

```
/**
 * This function returns the sum of squares of the input array.
 **/
 int get_sum_of_squares(int[] input) { ... }
```

All of the observations we just made could potentially be turned into an external regression test (also known as a unit test). Other observations could be the basis of sanity checks inside your code.



When it comes to writing unit tests, it pays to be creative, slightly crazy, and maybe a bit insane.

Tests based on assertions

Most testing is based on the idea of assertions.

You assert that something is true, and if it is, don't say anything. Only make noise when something is wrong. This makes it easy to spot problems.

In many languages/tools assertions can pinpoint problems by line number or by function.

```
assert 7 < 10
```
this says nothing because it is ok.

```
assert 7 > 10
```
this will tell you about a problem

Write Tests First

```
test_negative_sum()
    data = array [5, 3, 0, 9, -24]
    result = get_sum_of_squares(data)
    assert(result >= 0)

test_empty_input()
    result = get_sum_of_squares(empty array)
    assert result is 0

test_right_answer()
    result = get_sum_of_squares(array [3, 4, 5])
    assert result is 50
```

Then Write Code

```
/**
 * This function returns the sum of squares of the input array.
 **/
 int get_sum_of_squares(int[] input) {
    // happy code goes here, because if we screw up, our
    // tests will tell us where and what.
 }
```

Unit Testing

Software Engineers use this technique to a pretty extreme degree. We call it variously *Unit Testing, Regression Testing,* and there are loads of frameworks to help you do this. The most famous is called JUnit, by Kent Beck and Erich Gamma.

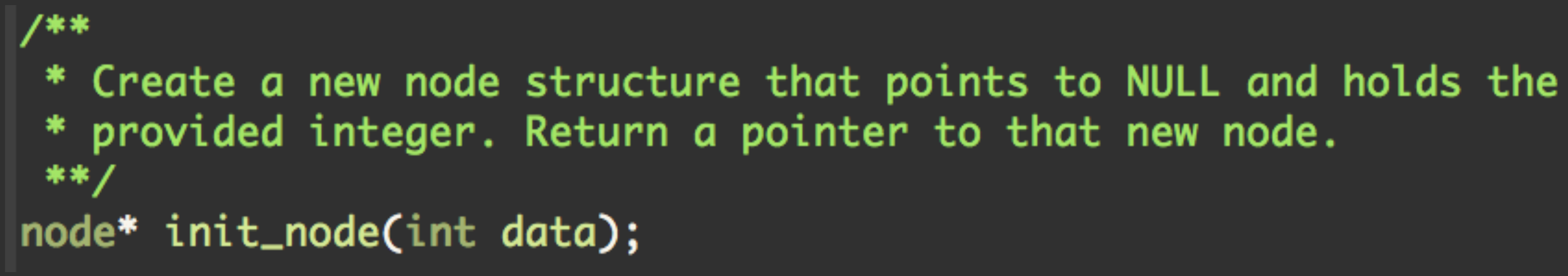
You can find C++ unit testing frameworks sitting around. Google has one. There's CPPUnit (cousin of JUnit). And the C++ unit testing we use in class was written by Alec Thilenius, a 2270 student.

TDD is happiness

Lots of people *love* test-driven design.

It works well if you know
—how something should work,
—what proper and improper inputs are,
—what proper and improper side effects are

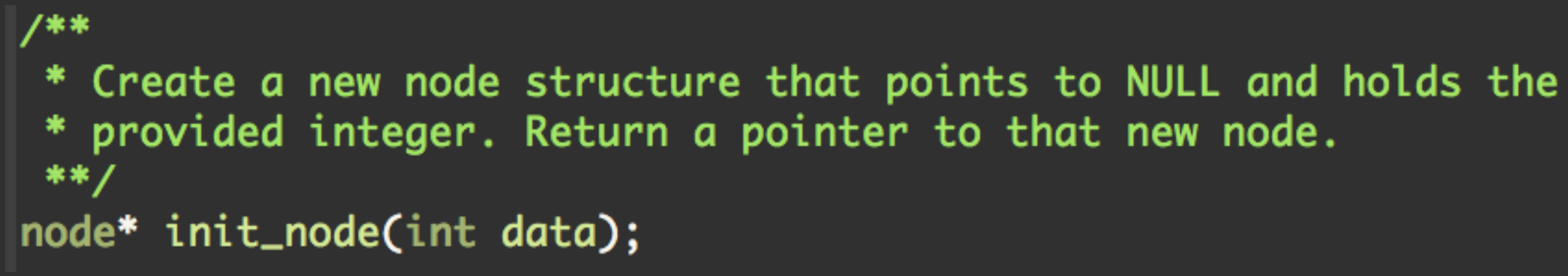
If you don't: you can still do it but it might be more trouble than it is worth. Also, if you don't know these things, *are you really engineering?*



```
/**
 * Create a new node structure that points to NULL and holds the
 * provided integer. Return a pointer to that new node.
 **/
node* init_node(int data);
```

For example:

This is directly from the HW 9 header file. The requirements are *right there*. The comment specifies the function's *invariants:* what should be true on input and exit?



```c
/**
 * Create a new node structure that points to NULL and holds the
 * provided integer. Return a pointer to that new node.
 **/
node* init_node(int data);
```

We’re going to write a unit test for this. It should:

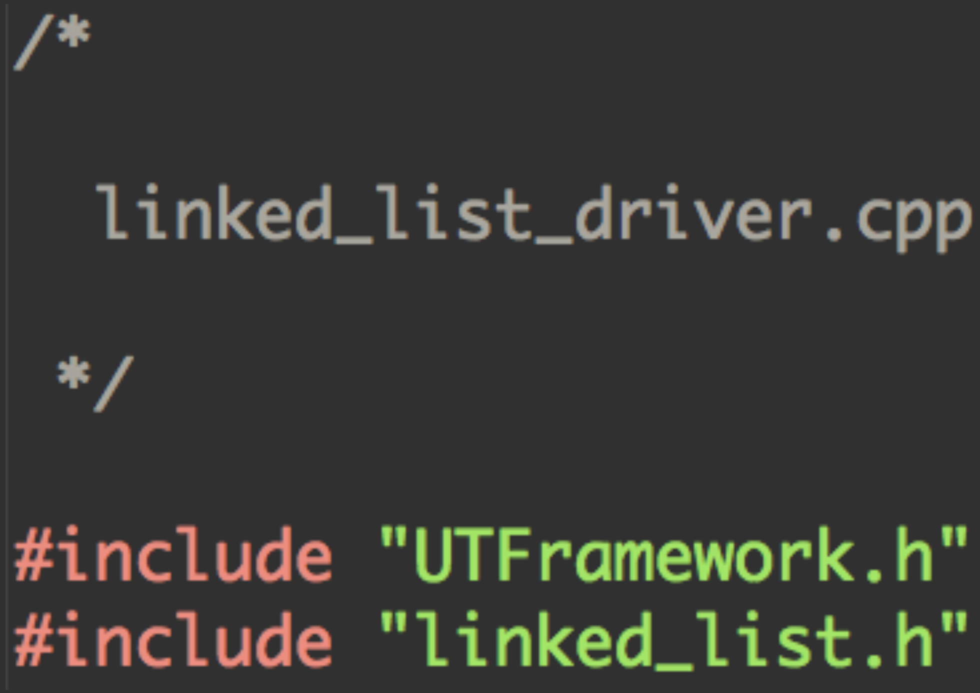
1. Return a pointer to a node.
2. The node should have a NULL ‘next’ member.
3. The node’s ‘data’ member should be equal to the provided integer.

Which Framework?

I mentioned several unit testing frameworks already:

CPPUnit
Google Testing Framework
JUnit
PyUnit

We'll use the simple but effective framework made by Alec. Look at UTFramework.cpp for dirty details. I'll show you how to use it.



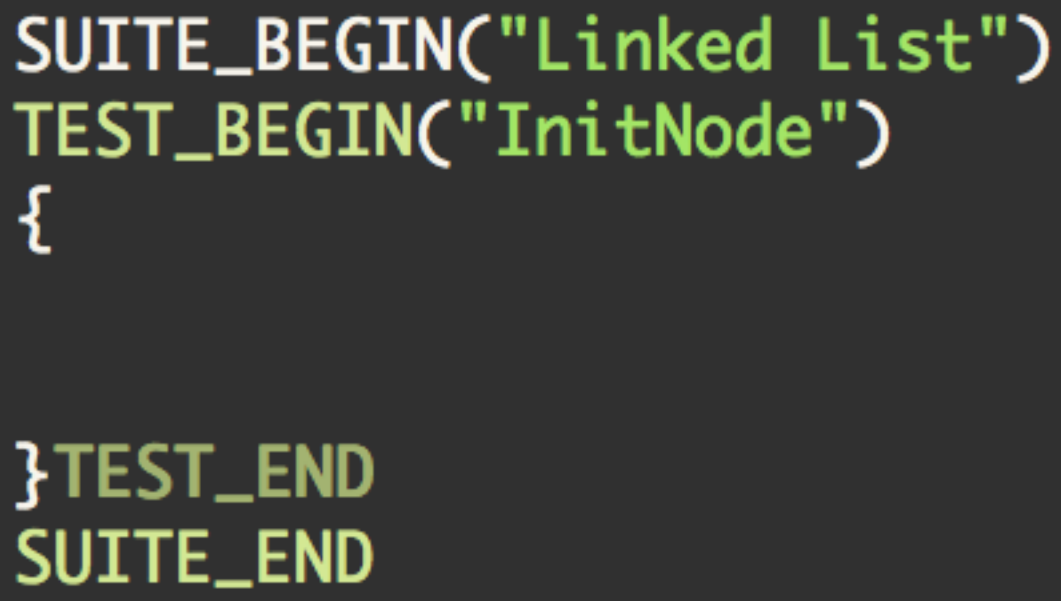
```
/*

  linked_list_driver.cpp

 */

#include "UTFramework.h"
#include "linked_list.h"
```

First thing to do is make a new file for our unit tests. Then include both the header file for the framework (in this case UTFramework.h) and the relevant headers for whatever we are testing.

Configure the Test

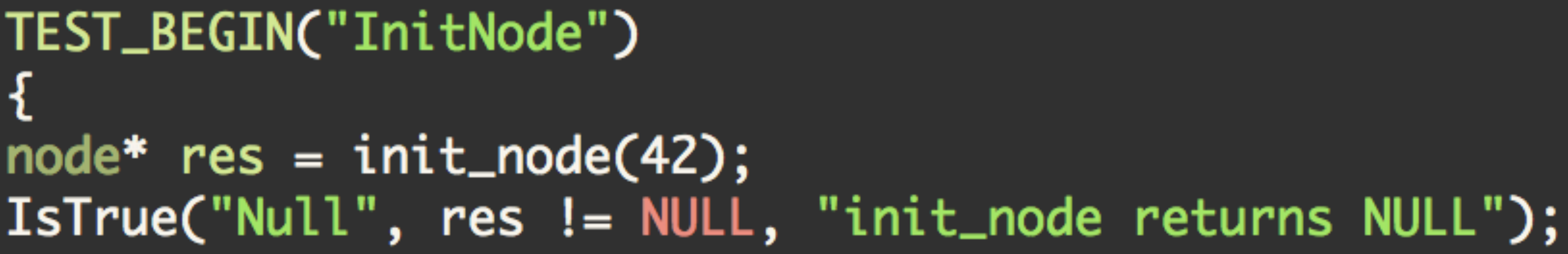
Now we'll need to configure the unit test framework in some way. This is (unfortunately) different for all of them. In C++ they all tend to be based on something called *pre-processor macros*.

This is not a topic we learned in CS 1300. You can treat them sort of like standard functions. As long as you follow the pattern I demonstrate you should be OK.



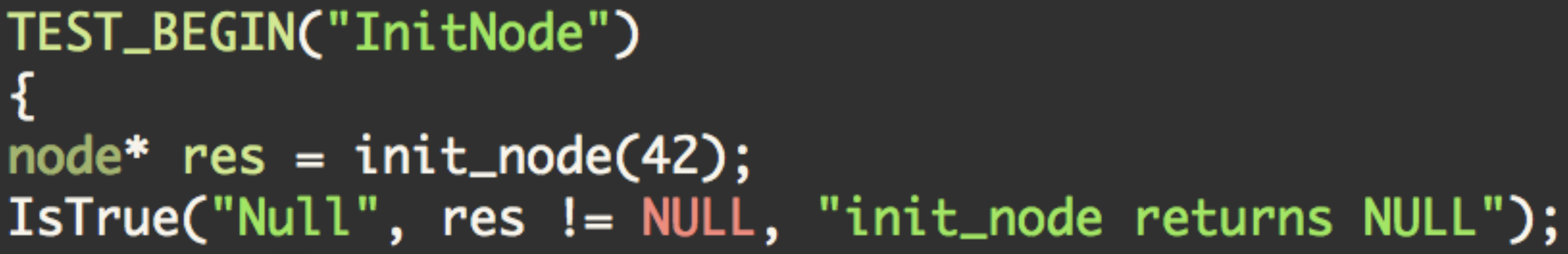
```
SUITE_BEGIN("Linked List")
TEST_BEGIN("InitNode")
{


}TEST_END
SUITE_END
```

Have SUITE_BEGIN and SUITE_END statements on the outside. Between those, you'll have one or more tests, indicated with TEST_BEGIN() { .. } TEST_END.



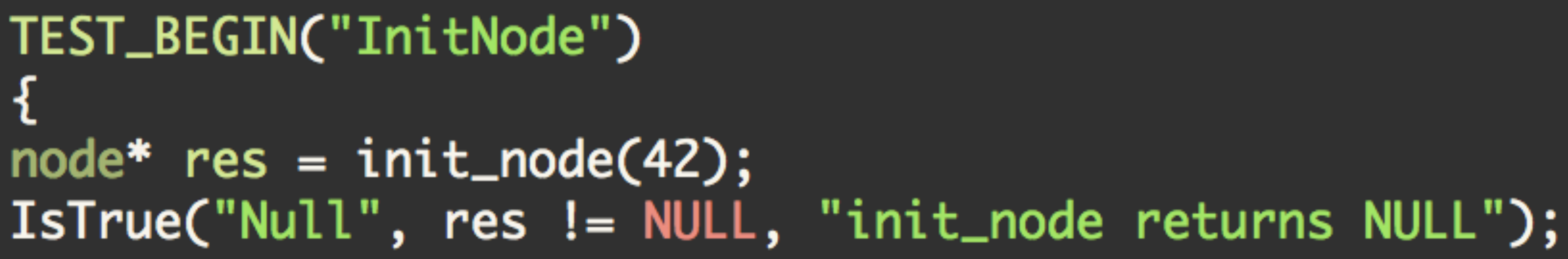
```
TEST_BEGIN("InitNode")
{
node* res = init_node(42);
IsTrue("Null", res != NULL, "init_node returns NULL");
```

Inside each test, you will make a series of *assertions*. In Alec's UTFramework, there is only one assertion: **IsTrue**.



```
TEST_BEGIN("InitNode")
{
node* res = init_node(42);
IsTrue("Null", res != NULL, "init_node returns NULL");
```

The first parameter is the label of the test.
This will print every time the assertion runs.

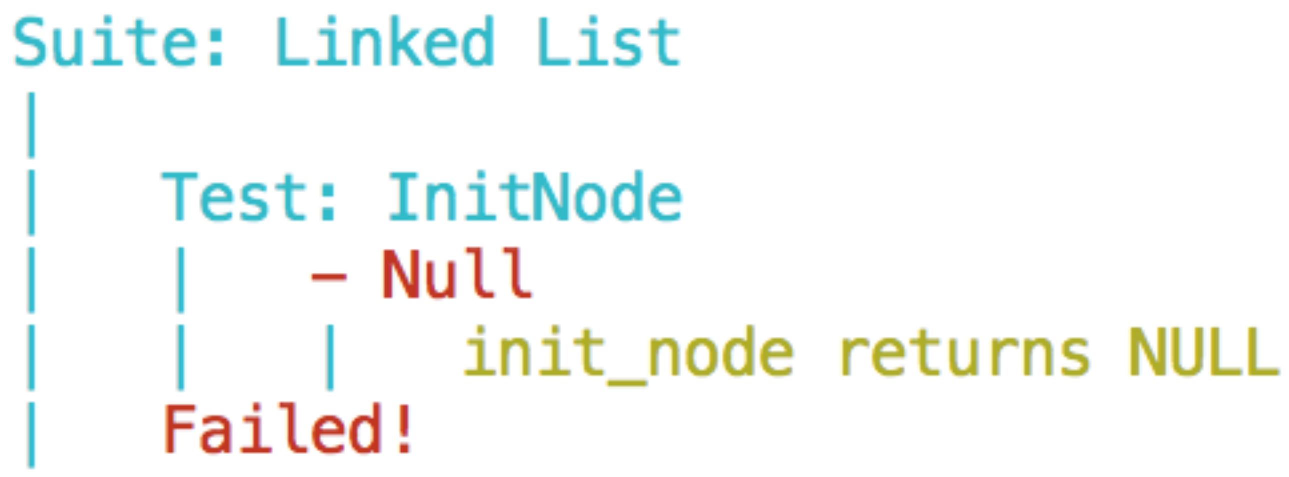


```
TEST_BEGIN("InitNode")
{
node* res = init_node(42);
IsTrue("Null", res != NULL, "init_node returns NULL");
```

The second parameter is a boolean statement. This is where you test to see if some condition holds or not. In this case we are expecting `res` to be non-null. So, if `res` *is* null, the assertion will fail.

```
TEST_BEGIN("InitNode")
{
node* res = init_node(42);
IsTrue("Null", res != NULL, "init_node returns NULL");
```

The last parameter is a string that will be displayed *only if the assertion fails*.

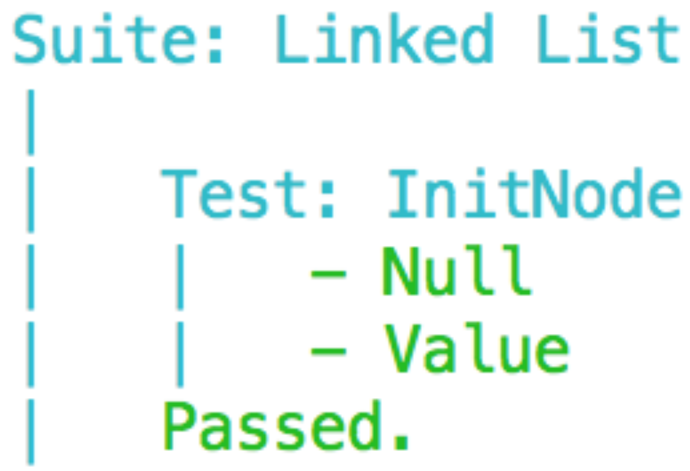


```
Suite: Linked List
|
|   Test: InitNode
|   |   - Null
|   |   |   init_node returns NULL
|   Failed!
```

You are probably pretty familiar with this by now.

When you run your unit test, individual test cases (like InitNode) will run. They are composed of several assertions (like the one we labeled ‘Null’).

When they fail, they give you error text like this.



```
Suite: Linked List
|
|   Test: InitNode
|   |    - Null
|   |    - Value
|   Passed.
```

When an assertion passes, it does *not* display an error. In general, unit test frameworks only complain when there's a problem. In this case, it also prints out in happy green text.

Tests help you sleep.

If you have a **giant project** with a team of engineers all working on different parts, unit tests will give you a way to determine if some new change will cause things to fail.

This is of course limited to the things that you can measure, and unit tests give you a solid way of measuring those things.

This is why...

This is why it is a *good thing* to have over-the-top unit tests that prove that crazy situations—weird input, misconfigured environments, etc—do or do not work. It gives you a measurable degree of certainty. Much better than “I think it works!”

Teamwork

In a real software engineering situation there will be teams of engineers and managers:

Some define the requirements,
Some define the test structure,
Some define individual tests,
Some write the application code,
Some identify bugs exposed by tests,
Some fix those bugs,
Some measure project progress,
And so on.

Next Time

Friday is the last lecture. It is *optional*, but don't let that turn you away.

I just found out that **my company will receive seed money**, so this is probably my last lecture at CU for the foreseeable future. I will make it count!

"HOW TO WIN AT SOFTWARE"