Hi, I’m Ian. I’m here to talk about dirty tricks in software construction.
What’s a dirty trick?

By “dirty trick,” I mean a practice that may be necessary in some contexts, but a terrible idea in others. Something like a small amount of technical debt we undertake mindfully.
e.g., the Ghost Typo

For example, a tester attending this conference told me her co-workers were ignoring e-mails about a draft document. But when she told them, “There’s a typo in your draft,” they suddenly paid attention to the substantive concerns she was raising.
Your examples?

These practices aren’t something you can get away with every time—but they may be just the trick to get you or your team unblocked.
Why “Dirty Tricks?”

I proposed this talk because our team just finished a big project. Afterwards, we reflected on what practices worked or didn’t work. A third category emerged: things that perhaps shouldn’t have worked, but did... this time.
This isn’t one of those talks that features a case study or rich data. This talk is mainly a set of variations on a theme, loosely clustered into a few broad groups.
The Meta-Dirty Trick:

The first and most important trick—and the only one I’ll express in the imperative—is...
DON’T GET FIRED

...don’t take the things in this talk as career advice. Take them as either entertaining or cringe-inducing war stories.
I. Blunt Code

The first category of dirty tricks relates to writing code.
The Code Crowbar

This trick relates to adding unit tests to difficult-to-test legacy code.

http://www.flickr.com/photos/toasty/4903485751
class WidgetTest
{
    public:
        void TestPizazz()
        {
            Widget w;
            w.vim = 2;
            w.vigor = 5;
            assert(w.Pizazz() == 10);
        }
};

Imagine we’d like to write this test for a Widget class.
class Widget
{
    public:
        // ...

    private:
        int vim;
        int vigor;
        int Pizazz() { return vim * vigor; }
};

But we can’t, because the data and method we’d like to test are private members.
Eventually, we need to move that stuff to a separate class. But we want working tests before we rearrange. So, we could do this before we include widget.h in our tests.
That’s horrible and ugly, right? Breaking open the Widget class (temporarily) is slightly less evil.
class WidgetConfig
{
public:
    WidgetConfig(int vim, int vigor)
        : vim(vim), vigor(vigor) {}

    int Pizazz() { return vim * vigor; }
}

private:
    int vim;
    int vigor;
};

Once the tests pass, we can safely move the configuration to its own more testable class...
class Widget
{
    public:
        // ...
    
    private:
        WidgetConfig config;
};

...and out of the Widget class.
class WidgetConfigTest
{
public:
    void TestPizazz()
    {
        WidgetConfig wc(2, 5);
        assert(wc.Pizazz() == 10);
    }
};

Our test now compiles and runs without any shenanigans.
Next up: macro abuse. I’m as suspicious of macro-heavy code as the next developer. For entertainment sometime, try looking up “macro abuse” on Stack Overflow.
/*
 * Your horror stories?
 */

#define TRUE 0

We won’t be talking about examples this egregious.
someComplicatedFunction(
    LONG_NAME_FOR_FOO,
    WITH_FOO_AND_BAR,
    BAR_VALUE);

someComplicatedFunction(
    LONG_NAME_FOR_QUUX,
    WITH_QUUX_AND_BAZ,
    BAR_VALUE);

Let’s look instead at a legitimate (if stilted) use of C macros. Do you see the line of code here that isn’t like the others?
someComplicatedFunction(
    LONG_NAME_FOR_FOO,
    WITH_FOO_AND_BAR,
    BAR_VALUE);

someComplicatedFunction(
    LONG_NAME_FOR_QUUX,
    WITH_QUUX_AND_BAZ,
    BAR_VALUE);

Without knowing anything about these functions, we might guess that the last line should say BAZ_VALUE instead.
#define DO_SOMETHING_WITH(k, v) {
   someComplicatedFunction(LONG_NAME_FOR_ ## k,      
                          WITH_ ## k ## _AND_ ## v,     
                          v ## _VALUE);
}

If we were to write an ugly C macro,...
DO_SOMETHING_WITH(FOO, BAR);
DO_SOMETHING_WITH(QUUX, BAZ);

...then our intent would be much clearer in the code, and the particular error from earlier would be difficult to make.
Testing? Testing!

A related (ab–)use of macros is walling off code for testing purposes.

http://www.flickr.com/photos/av_hire_london/5579137689
if (systemBatteryLevel() < LOW_BATTERY_THRESHOLD)
{
    puts("Low battery");
}
#ifndef TESTING

if (systemBatteryLevel() < LOW_BATTERY_THRESHOLD)
{
    puts("Low battery");
}

#endif

...we can wall it off when we’re compiling our test. This technique isn’t useful if the behavior we’re walling off is actually part of the function’s contract (as we’ll see later).
As long as we’re being too cute with C, let’s take a look at some more compile-time chicanery.

http://www.flickr.com/photos/pagedooley/2128892824
Imagine a program whose statistics-gathering phase will fall over for large inputs. For now, we’re safe—the input will never be large enough to break the algorithm.
#define MAX_SALESPEOPLE 1000

int someStatistic = 30000 / (1000 - numSalespeople);

But if a later developer comes along and raises the max input size (perhaps in a feature race with the competition), the statistics code may break at runtime.
We could use the assert() macro to catch the problem at runtime. But the compiler can actually check some of these kinds of constraints.
There are multiple ways to implement compile-time assertions, even without the Boost library. Here’s one technique from Jon Jagger.
The wheel, reinvented

I’d rather do almost anything than roll my own TCP stack or PNG image processor for work. (For fun on the weekend is another story.) On the other hand, if the task is narrow and the risk low (e.g., test frameworks), reinventing the wheel is sometimes okay. http://www.flickr.com/photos/29225114@N08/3094190643
II. Test Practices

The next few dirty tricks have to do with code-level testing done by the developer.
The first trick in this category is stubbing out functions.
#ifndef TESTING

if (systemBatteryLevel() < LOW_BATTERY_THRESHOLD) {
    puts("Low battery");
}

#endif

In our earlier example, we used the blunt instrument of macros to snip out hardware dependencies during testing.
#ifndef TESTING

if (systemBatteryLevel() < LOW_BATTERY_THRESHOLD)
{
    TurnWarningLightOn();
}

#endif

But what if this function did something we actually needed to check during testing?
if (batteryLevelFunc && (*batteryLevelFunc)() < LOW_BATTERY_THRESHOLD)
{
    TurnWarningLightOn();
}

In that case, we can’t just skip over that code during testing. Instead, we might use a function pointer or inheritance to let us rewire the hardware-dependent code while the test runs.
double fakeLowBatteryLevel() {
    return LOW_BATTERY_THRESHOLD / 2.0;
}

void testAimWithLowBattery() {
    MindControlLaser l;
    l.batteryLevelFunc = &fakeLowBatteryLevel;
    l.Aim();
    assert(l.WarningLightOn());
}

Here's a fake function that simulates a low-battery condition, and a test that uses it.
Copy-and-paste coding is one of the things I hate most in this world. And yet...
describe Bucket do
  context 'when full' do
    subject { Bucket.new }
    before { subject.fill 10.blocks }

    {:capacity => 0,
     :weight => 20}.each do |name, value|
      its(name) { should == value }
    end
  end
end

...sometimes the cure is worse than the disease. This Ruby code contains two test assertions, but they’re hidden by the torturous loop we used to try to avoid repetition.
describe Bucket do
  context 'when full' do
    subject { Bucket.new }
    before { subject.fill 10.blocks }

    its(:capacity) { should == 0.blocks }
    its(:weight) { should == 20.pounds }
  end
end

If we’d just cut, pasted, and modified the assertions, we’d have much clearer test code. See David Chelimsky’s writing for more discussion on repetition vs. legibility. One other area where cut and paste can result in less confusion and better functionality: makefiles.
Telling your compiler to report all warnings as errors can feel like a dirty trick—especially to your teammates. But in the context of this particular project, doing so resulted in many caught errors and almost no false positives.
Here, I’m using “manualization” to mean “making an automated process manual.” Frequent integration and testing doesn’t have to happen on a server. Teams who constantly merge one another’s changes throughout the day are doing a form of manual CI. wiktionary.com; http://www.flickr.com/photos/philaaronson/2485460766
Inspiration from elsewhere

There have been books on refactoring C++ for years, but the topic didn’t really click with me until I saw it in a different environment. What programming communities do you look to for inspiration? (Picture from informit.com)
III. Social Beings

The last category deals with how we relate to our jobs and our teammates.
Yes, official company templates are there for a reason. But breaking them happens for a reason, too. Sometimes, the right thing to do is write your documentation in whatever format stays out of your way, then port it to the official format later. http://www.flickr.com/photos/facilitybikeclub/3197419294
Our companies pay us not just for what’s in the job description, but for the lifetime of creativity, expertise, and experience that led to our coming here. Playing hooky and coming to conferences can make us better at what we do. http://www.flickr.com/photos/reinis/3454438258
Sometimes you need a laptop with a particular Linux distro for a couple of weeks of testing, or a Mac with a specific Photoshop version to decode a client’s files. Sometimes we need to get scrappy with our resource allocation. Just be above board about whatever you do. http://www.flickr.com/photos/rmgimages/4882451072
Job transformers

I’d go crazy if I had to do nothing but write embedded C all day. If instead I think of my job as helping engineers make electrical measurements, that transforms my job into a programmer + document writer + tester + support technician. So much more variety! http://www.flickr.com/photos/62109962@N07/6142290405
My hope is that hearing these tales has reminded you of a few groan–worthy war stories of your own, and therefore kept you entertained for the last few minutes. I also hope the talk has been a reminder that every practice has a context. I’ve enjoyed talking to and learning from you over the past two days—cheers!