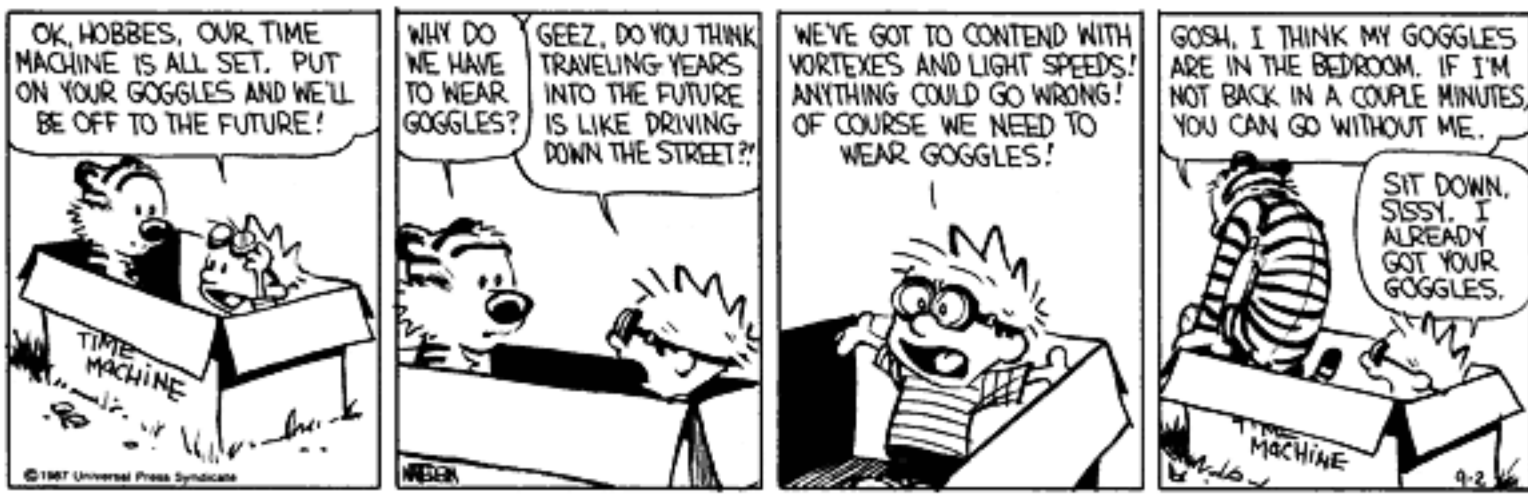
Tools and Techniques

Introduction

Tools you can use individually: Test frameworks, memory checkers

The size of the task: Building software for a collaboration



What do you need to do the job?

I need to calculate the sum of prime numbers in the 1st 100 integers:

```
int sumPrimes() {
    int sum = 0;
    for ( int i=1; i < 100; i++ ) {  // loop over possible primes
        bool prime = true;
        for (int j=1; j < 10; j++) { // loop over possible factors
            if (i % j == 0) prime = false;
        }
        if (prime) sum += i;
    }
    return sum;
}
```

This is quick, throw-away code

- Not well structured, efficient, general or robust
- I understand what I intended, because I wrote it just now

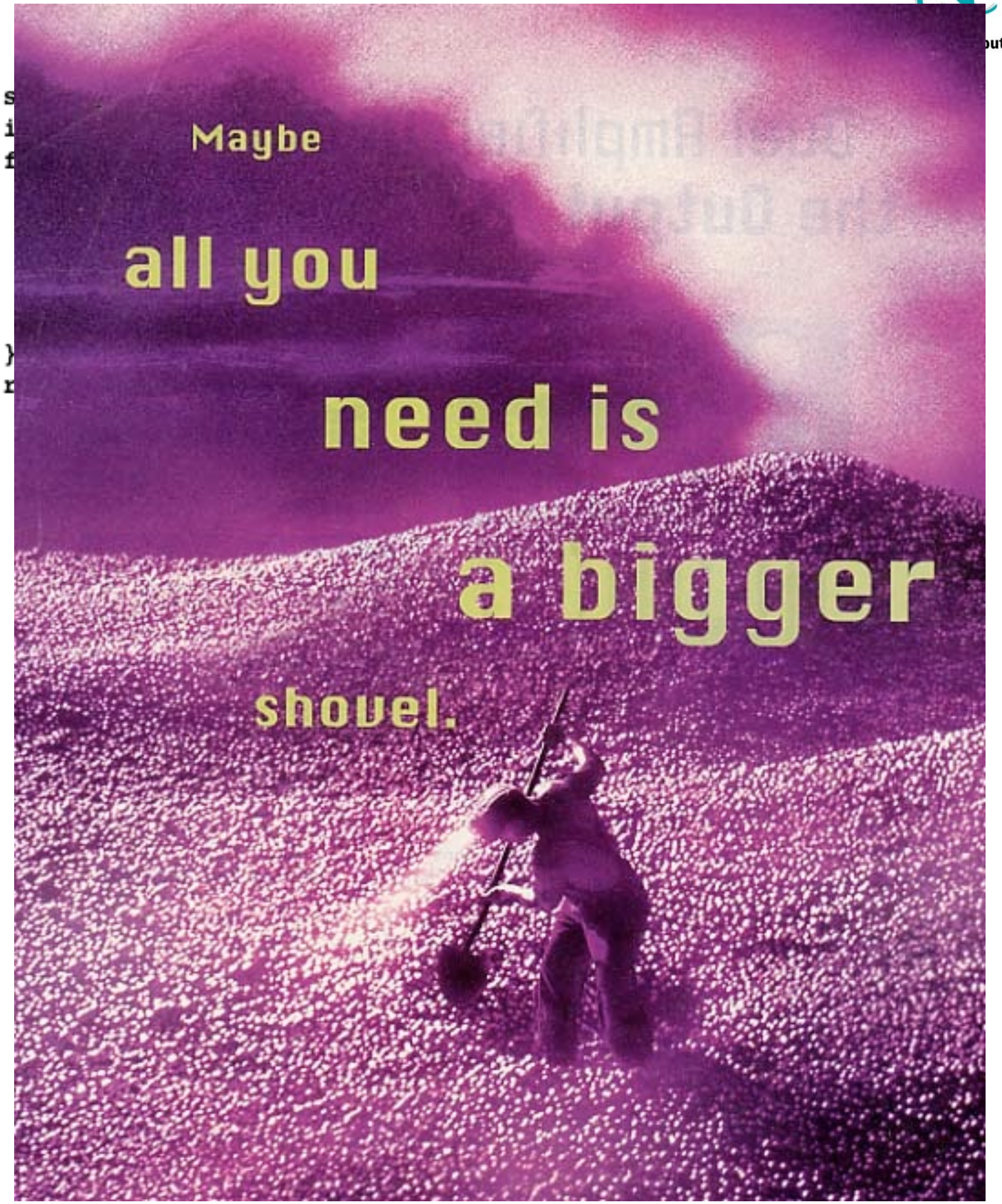
Already, I need an editor, compiler, linker, and probably a debugger

"Don't worry, I'll remember why I wrote it that way."

"The answer looks OK, lets move on."

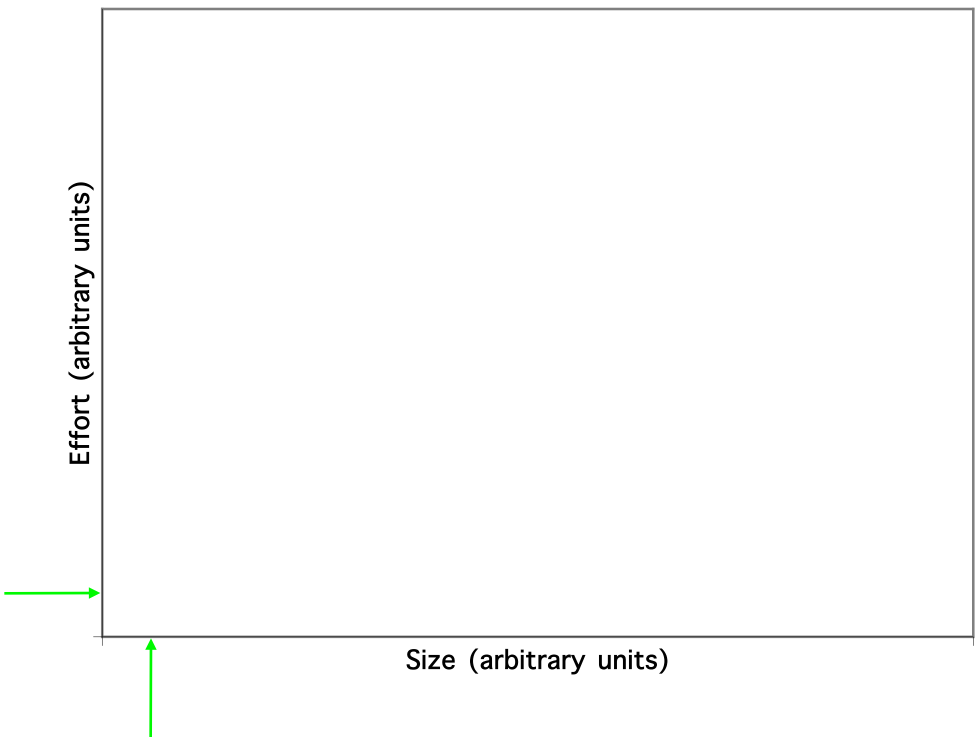
"Does anybody know where this value came from?"

"Your #%@!& code broke again!"



Projects come in different sizes

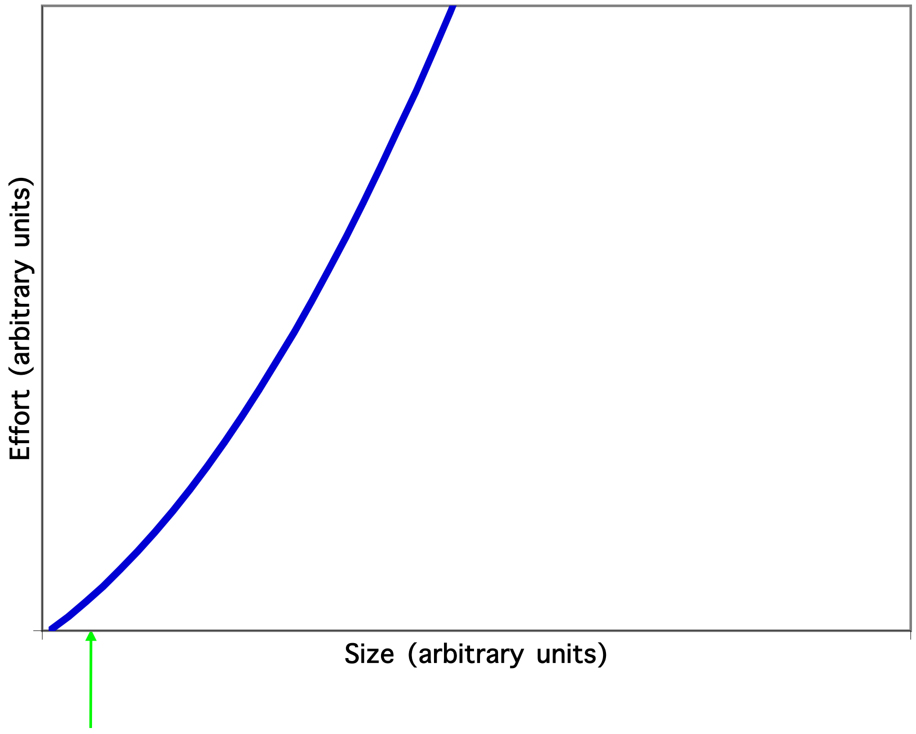
My sample program is a pretty small project!



Projects come in different sizes

My sample program is a pretty small project!

It can be done with a simple technique:



But that won't solve larger problems well

Projects come in different sizes

My sample program is a pretty small project!

It can be done with a simple technique:

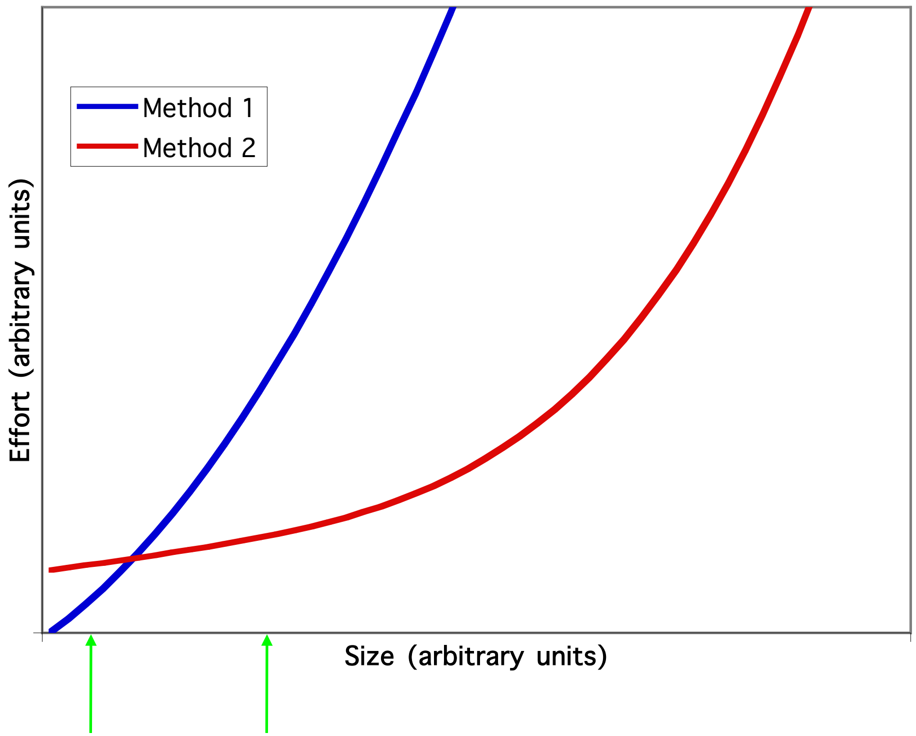
But that won't solve larger problems well

Projects come in different sizes

CERN School of Computing

A larger project may need a different approach

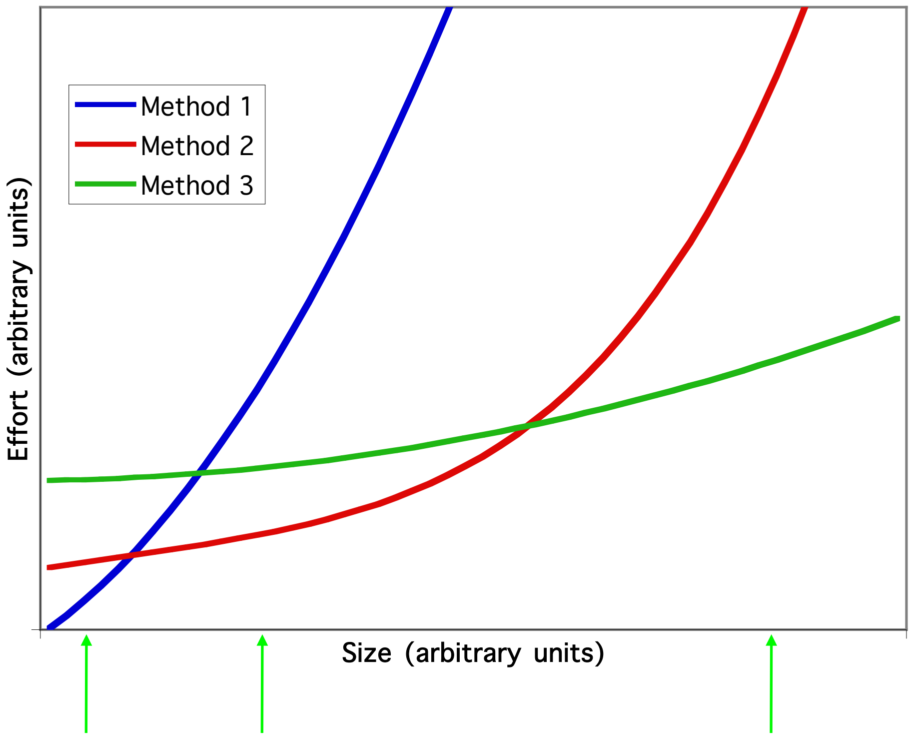
- Those tend to require more effort up front



What do you do when your project grows?

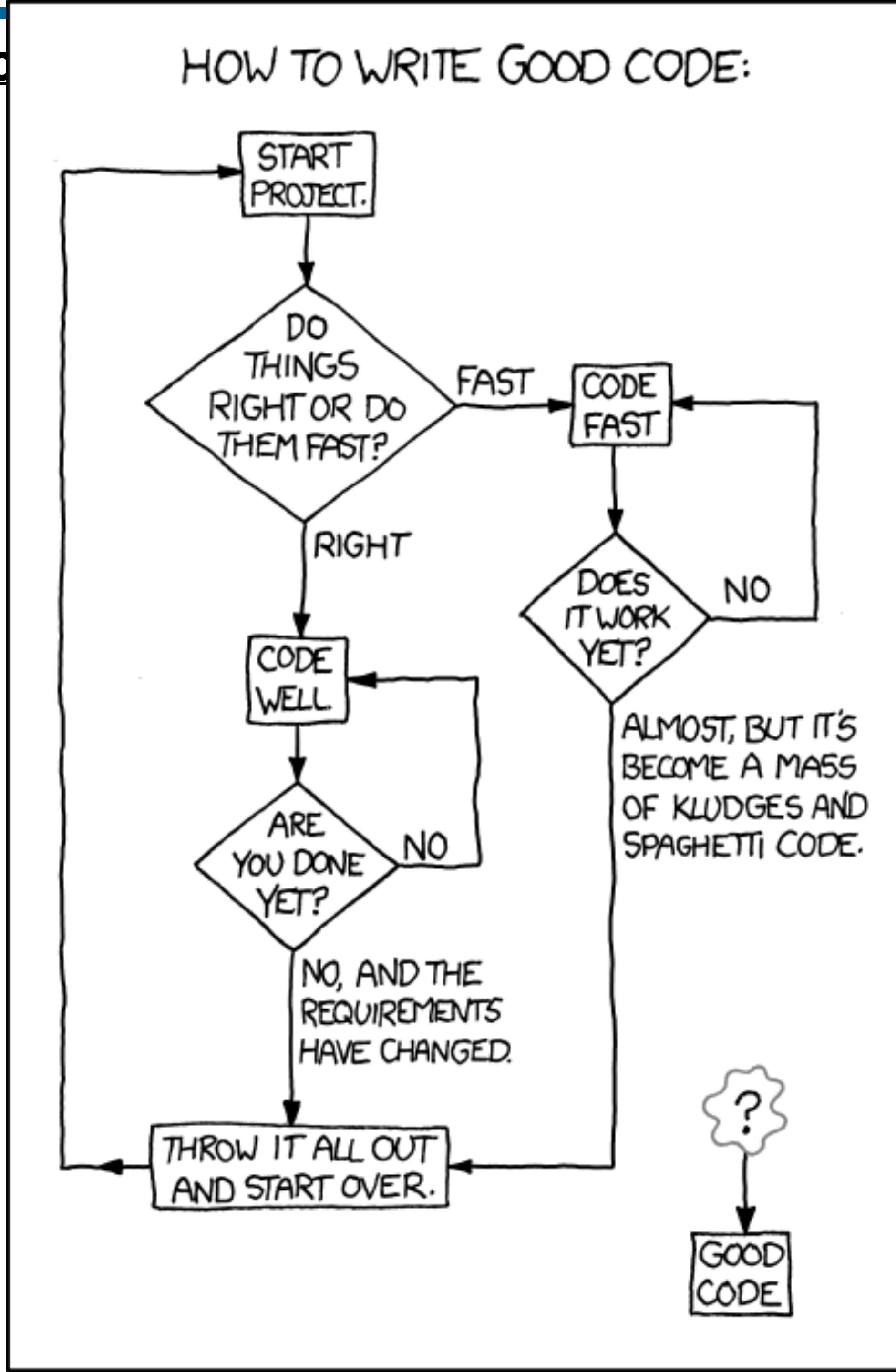
Projects come in different sizes

If you're trying to solve a really large problem:



Projects c

If you're



What has all this to do with us?

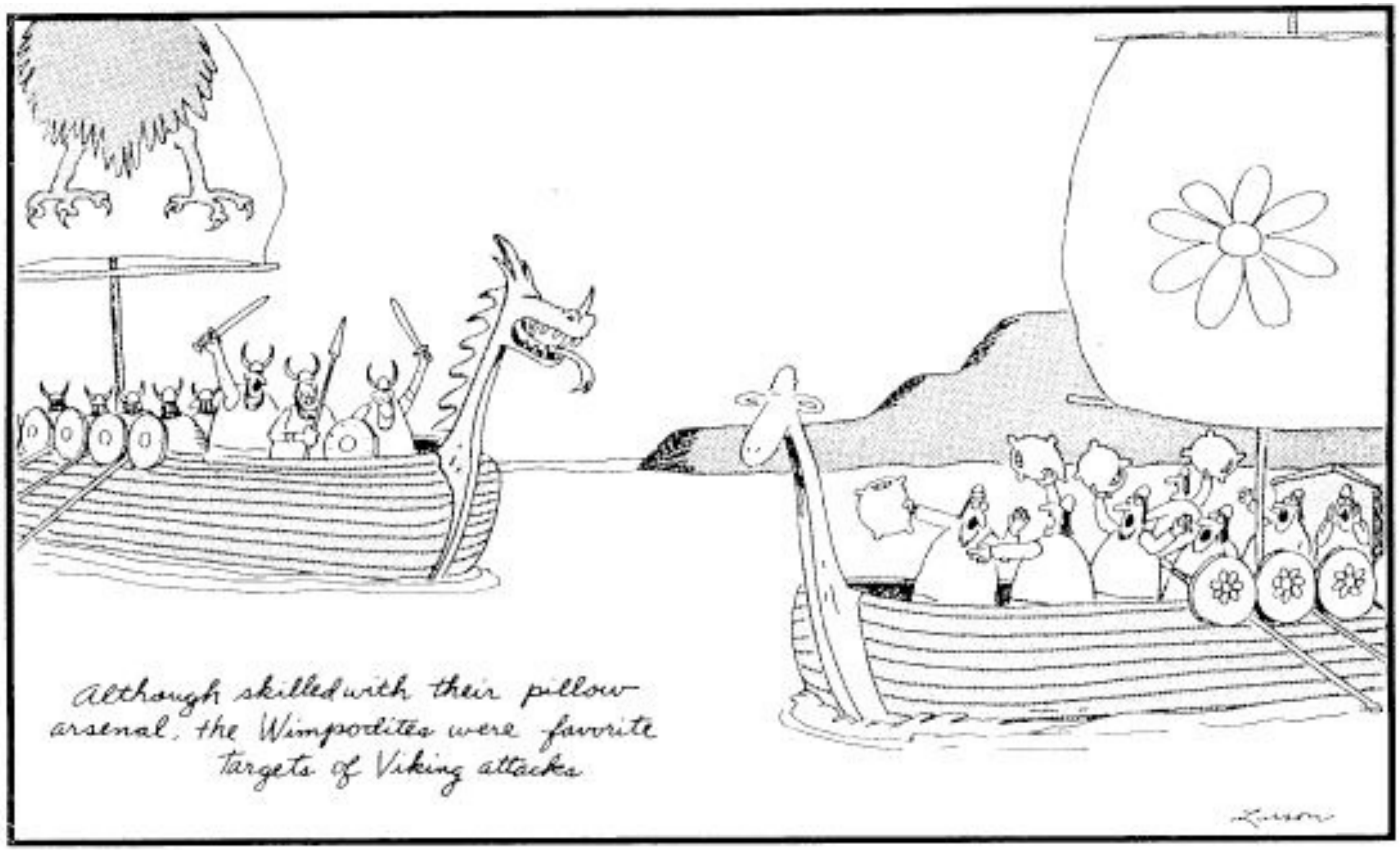
CERN School of Computing

Our systems tend to be complex systems

- HEP tends to work at the limit of what we know how to do

"If you only have a hammer, wood screws look a lot like nails" - ??

"If you only have a screwdriver, nails are pretty useless" - Don Briggs



Larger projects have standard ways of doing things

To make it possible to communicate, you need a shared vocabulary

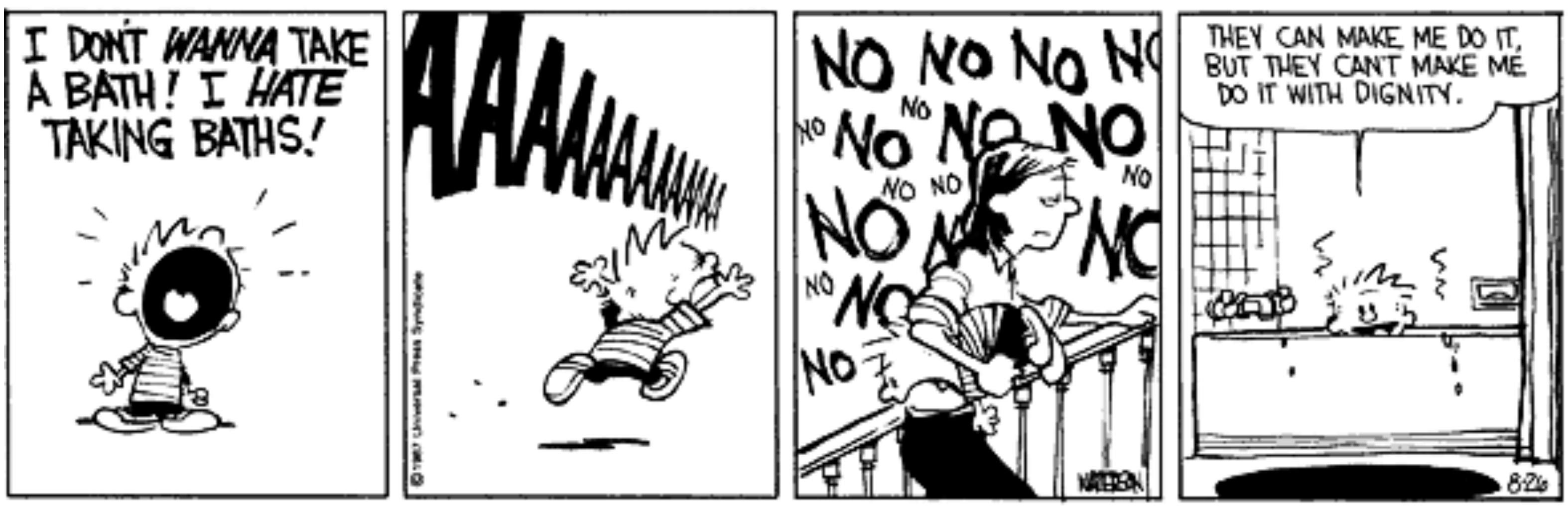
- Standards for languages, data storage, etc.

For people to work together, you have to control integrity of source code

- E.g. Git to provide versioning and control of source code

Just building a large system can be difficult

- Need tools for creating releases, tracking problems, etc.



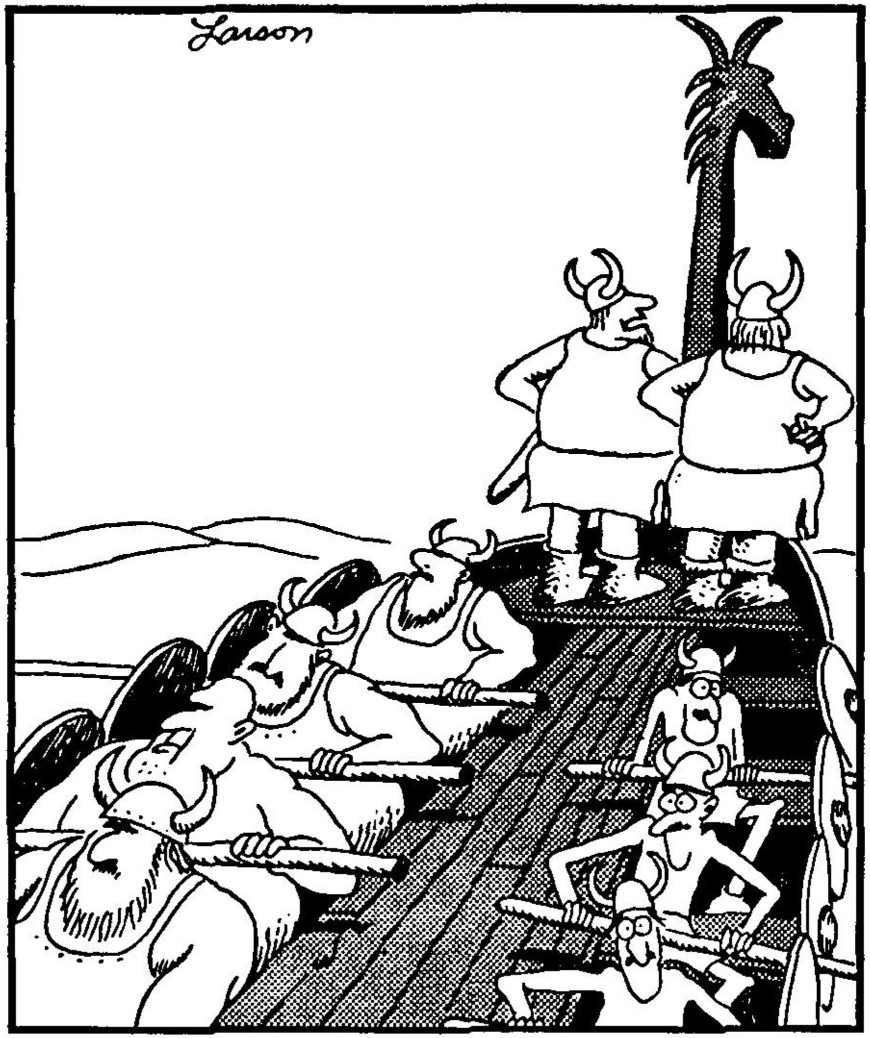
But individual effort is still important!

You can't build a great system from crummy parts

You want your efforts to make a difference

Good tools & technique can help you do a better job

"Whatever you do may seem insignificant, but it is most important that you do it." - Gandhi



"I've got it, too, Omar ... a strange feeling like we've just been going in circles."

Tools you can use

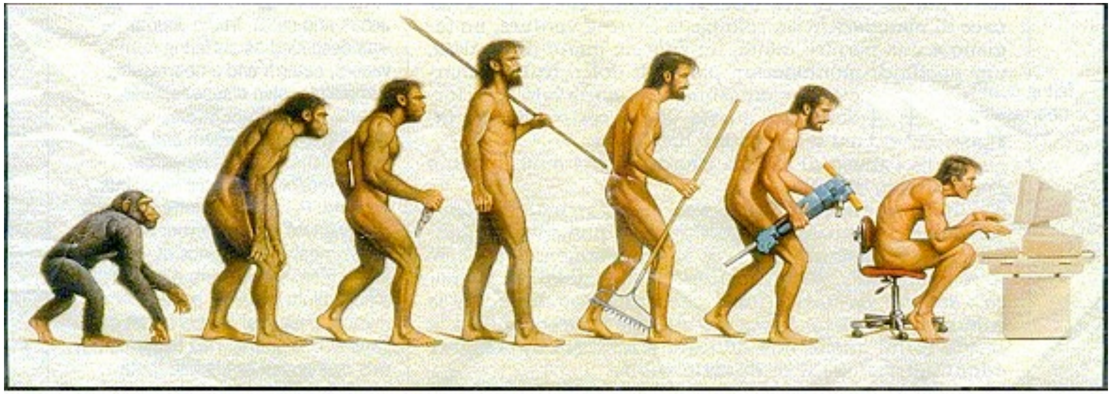
Knowing whether it works - JUnit, CppUnit, PyTest etc

Toward an informed way of experimental working

Progress often comes from small, experimental changes

- Allows you to make quick progress on little updates
- Without risk to the big picture

How do you know those steps are progress?



Somewhere, something went terribly wrong

Testing

But don’t you see Gerson - if the particle is too small and too short-lived to detect, we can’t just take it on faith that you’ve discovered it.”

The role of testing tools

Remember our original example: sum of primes in first 100 integers

- Simple routine, written in a few minutes
- "So simple it must be right"

```
int sumPrimes() {
    int sum = 0;
    for ( int i=1; i < 100; i++ ) {  // loop over possible primes
        bool prime = true;
        for (int j=1; j < 10; j++) { // loop over possible factors
            if (i % j == 0) prime = false;
        }
        if (prime) sum += i;
    }
    return sum;
}
```

Donald Knuth: "I have only proved it correct, I have not tested it"

The role of testing tools

CERN School of Computing

Remember our original example:

- Simple routine, written in a few minutes
- "So simple it must be right"

```
int sumPrimes(int n) {
    int sum = 0;
    for ( int i=1; i < n; i++ ) {  // loop over possible primes
        bool prime = true;
        for (int j=1; j < 10; j++) { // loop over possible factors
            if (i % j == 0) prime = false;
        }
        if (prime) sum += i;
    }
    return sum;
}
```

- (Assumed) valuable enough to reuse and extend

But it's not right...

"Study it forever and you'll still wonder. Fly it once and you'll know."
- Henry Spencer

How to test?

Simplest: Run it and look at the output

- Gets boring fast!
- How often are you willing to do this? Really carefully?

```
std::cout << 2 << " " << sumPrimes(2) << std::endl;
std::cout << 3 << " " << sumPrimes(3) << std::endl;
std::cout << 11 << " " << sumPrimes(11) << std::endl;
std::cout << 13 << " " << sumPrimes(13) << std::endl;
```

- Will you really check the answers? Thousands of them?

How to test?

Simplest: Run it and look at the output

- Gets boring fast!
- How often are you willing to do this? Really carefully?

More realistic: Code test routines to provide inputs, check outputs

```
bool passed = true;
if (sumPrimes(2) != 2) {
    std::cout << " 2 failed with " << sumPrimes(2) << std::endl;
    passed = false;
}

if (sumPrimes(3) != 5) {
    std::cout << " 3 failed with " << sumPrimes(3) << std::endl;
    passed = false;
}
std::cout << (passed ? "All passed!" : "Failed!") << std::endl;
```

- Becomes ungainly - imagine hundreds of developers testing complex code

How to test?

Simplest: Run it and look at the output

- Gets boring fast!
- How often are you willing to do this? Really carefully?

More realistic: Code test routines to provide inputs, check outputs

- Can become ungainly

Most useful: A test framework

- Can invest in great feedback
- Better control over testing

```
•CPPUNIT_ASSERT_EQUAL(0, sumPrimes(1));
•CPPUNIT_ASSERT_EQUAL(2, sumPrimes(2));
•CPPUNIT_ASSERT_EQUAL(5, sumPrimes(3));
```

Testing Frameworks: CppUnit, Junit, PyUnit et al

CERN School of Computing

More

Each time you write a function:

```
public class SumPrimes {
    /** Return sum of primes up through n */
    public int sumPrimes(int n);
}
```

You should write a test:

```
public void testOneIsNotPrime() {
    SumPrimes s = new SumPrimes();
    Assert.assertEquals(0, s.sumPrimes(1) );
}
```

Invoke the function

Check expected result

Plus tests for other cases…

```
public void testTwoIsPrime() {
    SumPrimes s = new SumPrimes();
    Assert.assertEquals(2, s.sumPrimes(2) );
}
```

Embed that in a framework

Gather together all the tests

```
// define test suite
   public static Test suite() {
      // all tests from here down in hierarchy
      TestSuite suite = new TestSuite(TestFindVals.class);
      return suite;
   }
```

Junit uses class name to find tests

Start the testing

- To just run the tests:

```
junit.textui.TestRunner.main(TestFindVals.class.getName());
```

And that's it!

Invoke my test class

Running the tests

```
java TestSumPrimes
..
Time: 0.002

OK (2 tests)
```

```
java TestSumPrimes
..F
Time: 0.003
There was 1 failure:
1) testTwoIsPrime(TestSumPrimes)junit.framework.AssertionFailedError: check
sumPrimes(2) expected:<2> but was:<0>
    at TestSumPrimes.testTwoIsPrime(TestSumPrimes.java:23)

FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```

```
public void testTwoIsPrime() { // 2 is prime
  SumPrimes s = new SumPrimes();
  Assert.assertEquals("check sumPrimes(2)", 2, s.sumPrimes(2));
}
```

CppUnit, PyUnit output similar

```
void TestSumPrimes::testTwoIsPrime() {
  CPPUNIT_ASSERT_EQUAL(2, sumPrimes(2));
}
```

```
TestSumPrimes.cpp:13: Assertion
Test name: TestSumPrimes::testTwoIsPrime
equality assertion failed
- Expected: 2
- Actual  : 0
```

```
def test_sumPrimes(self):
   assert sumPrimes(1) == 0, "1 case"
   assert sumPrimes(2) == 2, "2 case"
```

```
======================================================================
FAIL: test_sumPrimes (__main__.TestSumPrimes)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "TestSumPrimes.py", line 11, in test_sumPrimes
    assert sumPrimes(2) == 2, "2 case"
AssertionError: 2 case
```

Results of testing "SumPrimes"

1 is not a prime, doesn't include

Should "max" be included or not?

All prime numbers are divisible by one, that's OK

```
int sumPrimes(int n) {
    int sum = 0;
    for ( int i=1; i < n; i++ ) {  // loop over possible primes
        bool prime = true;
        for (int j=1; j < 10; j++) { // loop over possible factors
            if (i % j == 0) prime = false;
        }
        if (prime) sum += i;
    }
    return sum;
}
```

If you divide a number by itself, the remainder is zero

Lesson 1: It's not easy to understand somebody else's code

- Assumptions, reasons are hard to see

  "Is one a prime number?"

  "Do I include the end point?" - was originally "sum of 1st 100 numbers"

  Sometimes bugs are hidden by other ones

Lesson 2: Better structure would have helped

- Separate "isPrime" from counting loop to allow separate understanding
  - Makes code checking for primes clearer, easier to test
  - Lets you check counting loop independently

Why?

One test isn't worth very much

- Maybe saves you a couple seconds once or twice

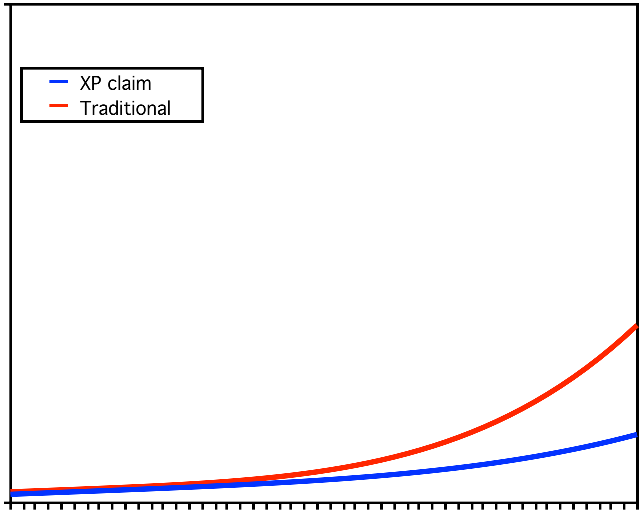
But consistently building the tests as you build the code does have value

- Have you ever broken something while fixing a bug? Adding a feature?
  - Tests remember what the program is supposed to do
- A set of tests is definitive documentation for what the code does
- Alternating between writing tests and code keeps the work incremental
  - Keeping the tests running prevents ugly surprises
- And it's very satisfying!

Extreme Programming advocates writing the tests before the code

More

- Large projects require structure
- Individuals report excellent results



The art of testing

What makes a good test?

- Not worth testing something that's too simple to fail
  - 2+2 really is 4
- Some functionality is too complex to test reliably
- Best to test functionality that you understand, but can imagine failing
  - If you're not sure, write a test
  - If you have to debug, write a test
  - If somebody asks what it does, write a test

How big should a test be?

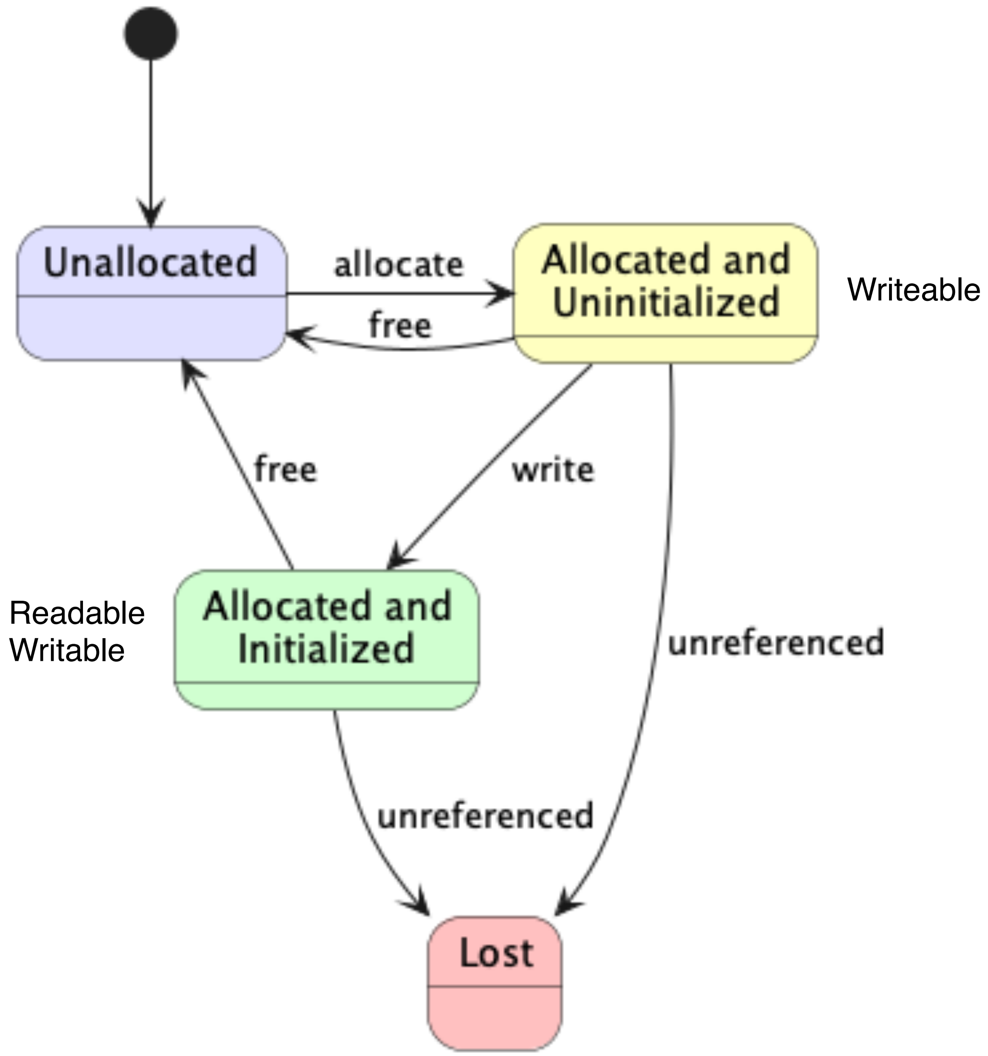
- A *Unit test is a unit of failure
  - When a test fails, it stops and moves to the next test
  - The pattern of failures can tell you what you broke
- Make lots of small tests to check what still works

What about existing code?

- Not practical to write a complete set of tests
- But you can write tests for new code, modifications, when you have a question about what it does, when you have to debug it, etc

Avoiding memory problems

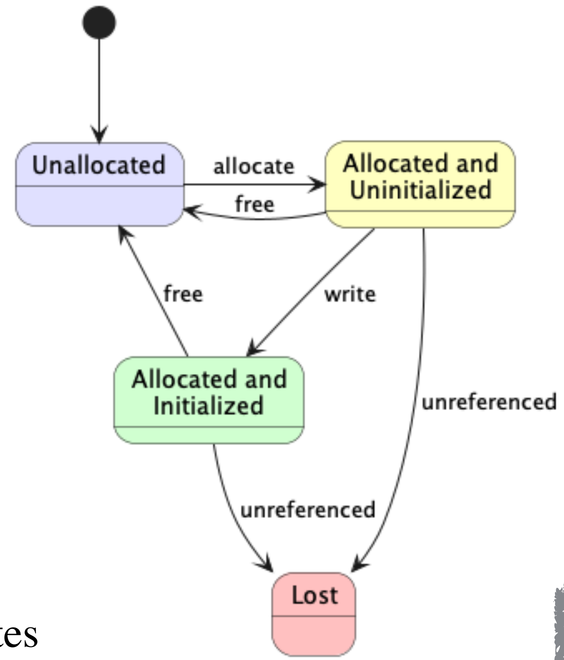
Memory State Machine



Memory-related problems

Read/write incorrectly

- Read from uninitialized memory
- Read/write via uninitialized pointer/ref
- Read/write past the valid range
- Read/write via a stale pointer/reference
  E.g. after deallocating memory



More

Memory management mistakes

- Deallocation of (currently) unowned memory
  Freeing something twice results in later overwrites
- Memory leaks
  Forgetting to free something results in unusable memory

Often cause "really hard to find" bugs

- Crashes, incorrect results - traceback and dump don't show cause
- Occur far from the real cause - breakpoints don't help
- Often intermittent

Note: Language choices reduce these, but don't make them go away!

Allocator (malloc) can find some of these

Standard Linux malloc has limited run-time checking option:

```
$ a.out
free(): invalid pointer 0x8049840!
```

Controlled by "MALLOC_CHECK_" environment variable

```
$ export MALLOC_CHECK_= n
```

Bit 0: print basic message (current default)

Bit 1: terminate and print more

Bit 2: print simplified messages

'man mallopt' for more info

Turning off can save several percent off time of some programs

"Hold my beer" approach to performance…

Tools can find even more of these

`valgrind` as one of many tools:

```
$ valgrind ./five
...
==3029799== Invalid read of size 4
==3029799==    at 0x400989: main (five.cpp:16)
==3029799==  Address 0x5b4e0c0 is 0 bytes inside a block of size 4 free'd
==3029799==    at 0x4C3A299: operator delete(void*, unsigned long)
==3029799==    by 0x400972: main (five.cpp:14)
==3029799==  Block was alloc'd at
==3029799==    at 0x4C378C3: operator new(unsigned long)
==3029799==    by 0x400953: main (five.cpp:10)
```

Read Upward

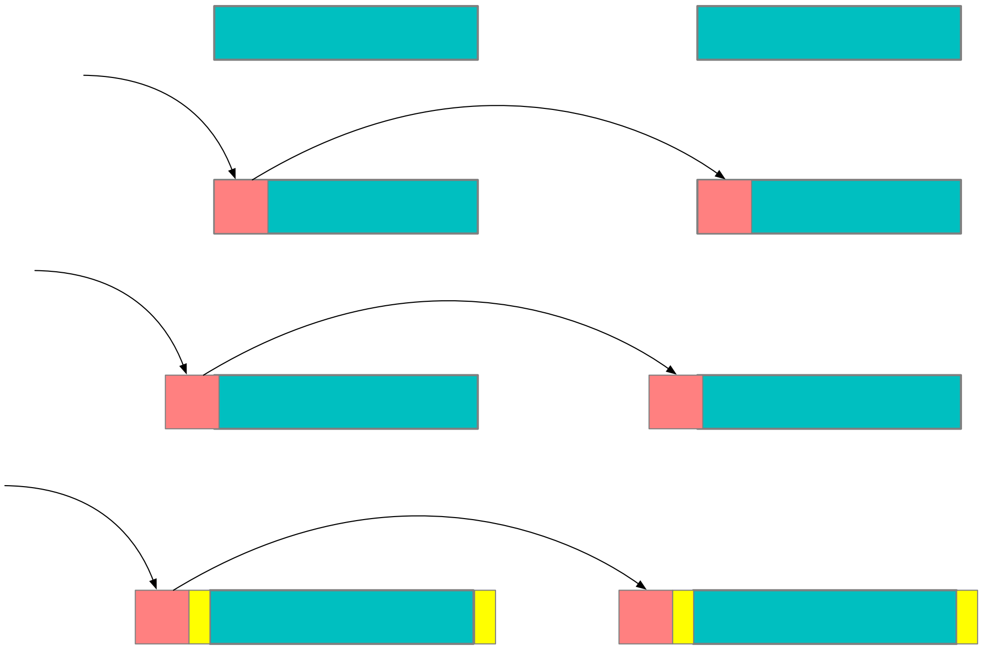
Why not always use it?

- Checking slows program significantly
- Too many errors?
- Only finds a limited number of error types

When to use it?

- Debugging a specific problem
- Run periodically to check for silent bugs
- As part of overall test routine

One example: Access to Heap Memory



Two pieces of memory you've allocated to yourself

When you free them, they're put on a free memory list
Cheapest: Use first few bytes for pointer
But if you write-after-free, you corrupt that chain

Instead, system can allocate some extra in front
That's safer against late writes
But uses more memory

Or allocate even more as "buffer zones"
Changes to those values indicate access
before or after what you have allocated:
myArray[-1], myArray[myArray.length]

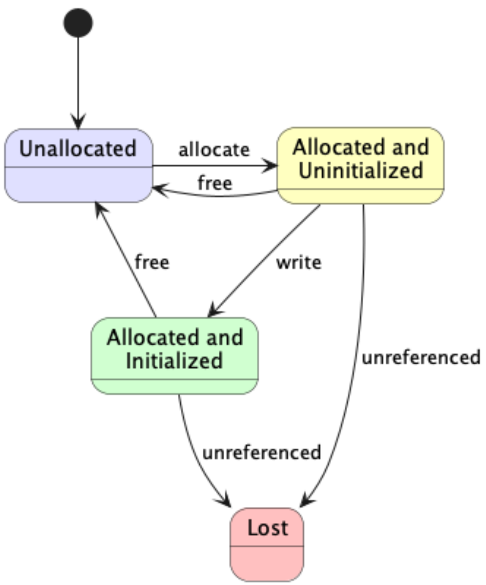
Specialized tools - leak checking

Automated, unambiguous identification of leaks is difficult

- "forgot to free" vs "haven't freed yet" vs "program's ending, don't bother"
- "can no longer reference any part" vs "no references to the beginning"

But reading the code is not a reliable method either

- A leak is a mistake of omission, not commission
- Often requires cooperation to leak memory:
  - Creator of allocated item may have no idea where it goes
  - Consumer may not realize responsible for deallocation
    - Doesn't need to be deallocated
    - Expects some third party to deallocate



Several approaches:

- Print all allocate/free, and let the human reason it out
- List all allocated memory when the program ends, let human reason it out
- Provide a browser, let human reason about status during running
- Provide a suite of heuristics that can be tuned to the code's structure

How do these actually work?

Replacement libraries

- E.g. a more careful malloc, perhaps automatically linked
- Can't check individual load/store instructions

Source code manipulation

- Preprocessor inserts instrumentation before compilation
  - Can know about scope, variable accesses, control flow
  - But requires source code, is language specific

Object code insertion / Instruction emulation

- Process object code to recognize & instrument load/store instructions
  - Can efficiently check every use of memory
  - Specific to both architecture and compiler, hard to port
  - Knows less about scope, variable accesses, control flow

A small catalog of available memory tools

More

CERN School of Computing

Validity tests

- DMalloc - replacement library with instrumentation
- ElectricFence - checks for write outside proper boundaries
- AddressSanitizer - integrated with clang & gcc compiler to check operations
- valgrind - instruction-by-instruction checking

Leak checkers

- Windows Leak Detector - runtime attach
- LeakTracer - compilation based
- Memprof
- MemCheck - part of Valgrind
- ccmalloc

Some IDEs have built-in tools

How do you use these?

Big-bang approach is incredibly depressing

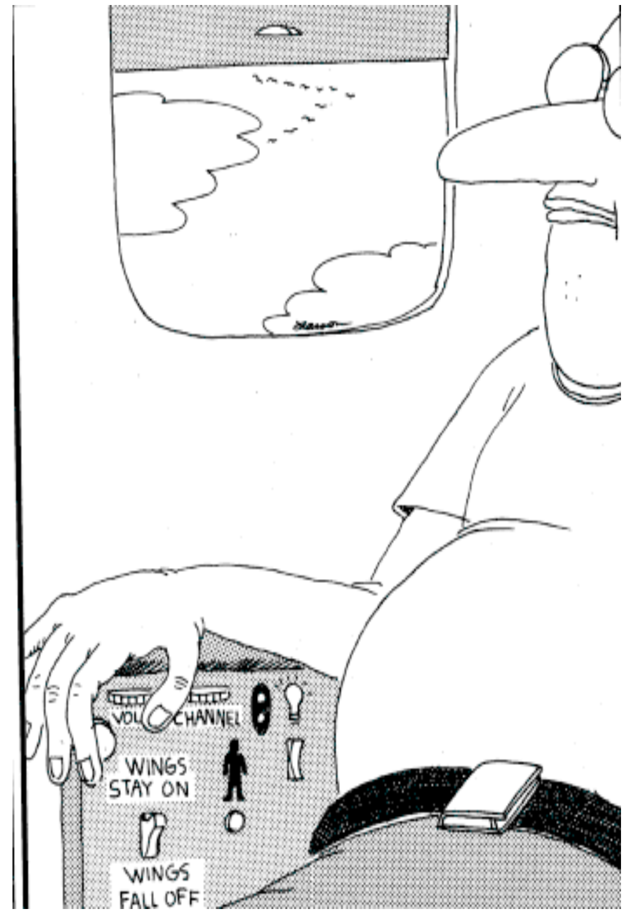
- Many products have lots of (benign?) errors
- These can swamp your own efforts

Better: isolate your own code for initial checks

- Ties in with a test framework:
  "Does it work as expected?"
- Check often, fix incrementally

You still have to test "in the wild"

- Many errors are due to poor interfaces
- Learn from these and fix them!



Goal: "An informed way of experimental working"

CERN School of Computing

Find a way of doing good work

Use tools wisely

Think about what you're doing