Week 10

- Unit Testing with Ceedling
- How to make sure you didn’t screw up

Where do we have problems?

- Write Code
- Write more code
- Change something
- Make sure everything works as expected

What could happen?

- Your codebase grows
- Several people work with you
- Someone changes some small detail
- The program crashes with exit code 11
- What did go wrong?

Unit Tests

- Test your units
- A unit may be a function
- A unit may be a struct/class (higher language)
- Test the behavior of those units

How to structure a test?

- Setup everything you need to test a unit
- Test only one specific thing per test
- Clean up resources

```
void addToList(List* l, int element){
if(l->size == l->capacity){
    resizeList(l);
  }
  l->elements[l->size] = element;
  l->size++;
}

void test_lotsOfStuff(){
  List* l = createList();
  TEST_ASSERT_EQUAL_INT(0, l->size);
  TEST_ASSERT_EQUAL_INT(1, l->capacity);
  TEST_ASSERT_EQUAL_INT(0, l->elements[0]);

  addToList(l, 11);
  addToList(l, 12);
  TEST_ASSERT_EQUAL_INT(2, l->size);
  TEST_ASSERT_EQUAL_INT(2, l->capacity);
  TEST_ASSERT_EQUAL_INT(11, l->elements[0]);
  TEST_ASSERT_EQUAL_INT(12, l->elements[1]);
}
```

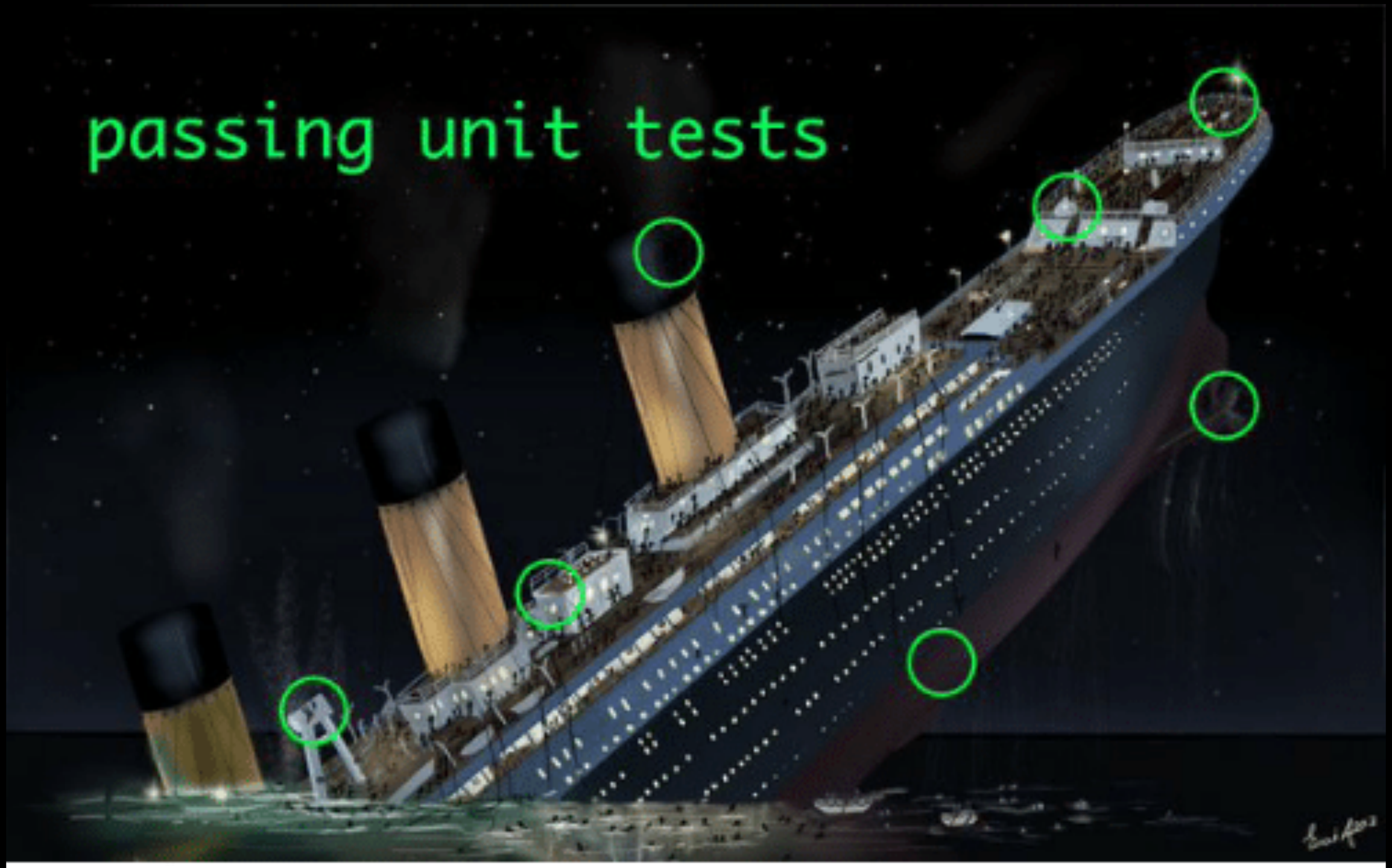
Does this look like a clean test?

Why bother?

- Change something in your code
- Run all unit tests
- Each failing test will tell you what doesn’t work
- Search efficiently for bugs

How not to do unit tests

- Write only some unit tests for the happy path
- Write code first, tests later to support your code
- „It just works, I don’t need tests for this“
- I am a good programmer I won’t need tests
- I know that I am drunk. Therefore I will drive more carefully



Hits too close to home

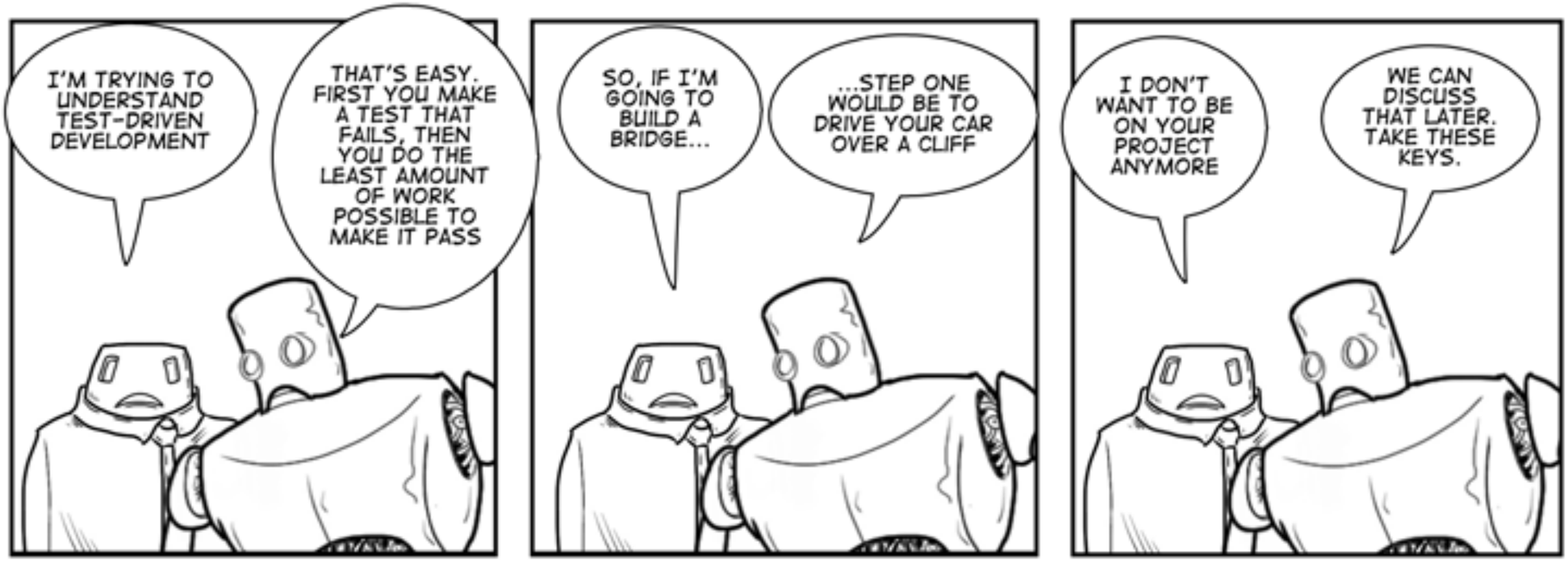
Test Driven Development

- TDD is the IoT of CS
- There are myriads of different guidelines
- Let me share my experience

1) Write a Failing Unit Test

2) Make this test pass with as little code as possible

3) Refactor


I'M TRYING TO UNDERSTAND TEST-DRIVEN DEVELOPMENT
THAT'S EASY. FIRST YOU MAKE A TEST THAT FAILS, THEN YOU DO THE LEAST AMOUNT OF WORK POSSIBLE TO MAKE IT PASS
SO, IF I'M GOING TO BUILD A BRIDGE...
...STEP ONE WOULD BE TO DRIVE YOUR CAR OVER A CLIFF
I DON'T WANT TO BE ON YOUR PROJECT ANYMORE
WE CAN DISCUSS THAT LATER. TAKE THESE KEYS.

Write a Failing Unit Test

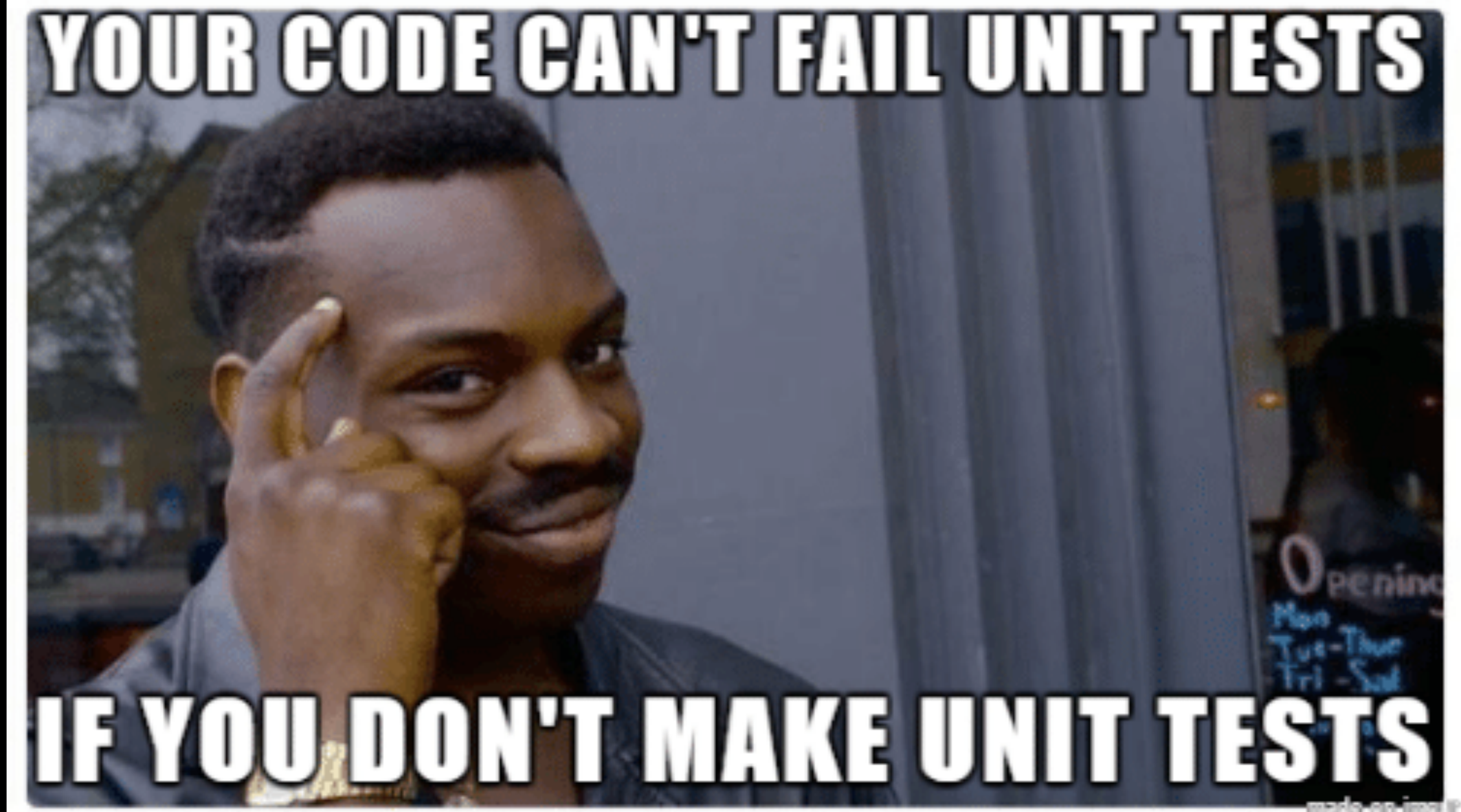
- You establish that as of time of writing your code doesn't implement that behavior
- You can now tell exactly when your code starts working
- You can test whether your toolchain works (!)

Make this test pass with as little code as possible

- Not more code than necessary
- You must assume that everything you write is wrong
- Only tests can tell whether something is right
- Any code you write that isn’t tested, is danger

Refactor

- Remove any unidiomatic code
- After making sure your code passes all tests, make the code clean
- Cleaning code is refactoring
- Write additional tests for all behavior you want your code to have



Unit what?

Ceedling

- Ceedling is a build system for C
- It’s done by the guys who also make Unity (not the game engine)
- Unity is a unit testing toolchain for C
- Search for Ceedling and follow the installation instructions

Exercises for next week

- Write a unit tested data structure of your choice:
  Tree, List, Queue, LinkedList, Hashmap, Bloomfilter

- Use TDD for this process

Example: Stack

- ceedling new stack
- cd stack
- ceedling

```
~/Desktop/stack
➜ ceedling

-------------------
OVERALL TEST SUMMARY
-------------------

No tests executed.
```

Create a Module

- ceedling module:create\[stack\]

```
~/Desktop/stack
→ ceedling module:create\[stack\]
File src/stack.c created
File src/stack.h created
File test/test_stack.c created
Generate Complete

~/Desktop/stack
→ ceedling


Test 'test_stack.c'
-------------------
Generating runner for test_stack.c...
Compiling test_stack_runner.c...
Compiling test_stack.c...
Compiling unity.c...
Compiling stack.c...
Compiling cmock.c...
Linking test_stack.out...
Running test_stack.out...

--------------------
IGNORED TEST SUMMARY
--------------------
[test_stack.c]
  Test: test_stack_NeedToImplement
  At line (14): "Need to Implement stack"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  1
PASSED:  0
FAILED:  0
IGNORED: 1
```

```c
#include "unity.h"
#include "stack.h"

void setUp(void)
{
}

void tearDown(void)
{
}

void test_stack_NeedToImplement(void)
{
    TEST_IGNORE_MESSAGE("Need to Implement stack");
}
```

For all asserts (e.g. TEST_ASSERT_EQUAL_INT(2, 1+1);

Executed before each test

Executed after each test

Tests are like:

void test_blabla(void){}

The test_ is important

First test

```
#ifndef _STACK_H
#define _STACK_H

typedef struct stack{

} stack;

#endif // _STACK_H
```

```
#include "unity.h"
#include "stack.h"

stack *s;

void setUp(void)
{
}

void tearDown(void)
{
}

void test_stack(void){
    TEST_ASSERT_NOT_NULL(s);
}
```

```
-------------------
FAILED TEST SUMMARY
-------------------
[test_stack.c]
  Test: test_stack
  At line (15): "Expected Non-NULL"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  1
PASSED:  0
FAILED:  1
IGNORED: 0
```

We need to allocate memory

- How do we test that all memory is freed at the end of a test?
- Write a wrapper around malloc and free that increases/ decreases a counter for each allocation
- Difficult to test otherwise

```c
//TEST_STACK.c
#include <stdint.h>
#include <stdlib.h>

uint16_t allocationCount = 0;

void* safe_malloc(size_t size){
    allocationCount++;
    return malloc(size);
}

void safe_free(void* ptr){
    allocationCount--;
    free(ptr);
}

void tearDown(void)
{
    TEST_ASSERT_EQUAL(0, allocationCount);
}
```

```c
void setUp(void)
{
    s = safe_malloc(sizeof(stack));
}

void tearDown(void)
{
    TEST_ASSERT_EQUAL(0, allocationCount);
}

void test_stack(void){
    TEST_ASSERT_NOT_NULL(s);
}
```

```
FAILED TEST SUMMARY
--------------------
[test_stack.c]
  Test: test_stack
  At line (25): "Expected 0 Was 1"

----------------------
OVERALL TEST SUMMARY
----------------------
TESTED:  1
PASSED:  0
FAILED:  1
IGNORED: 0
```

```
stack *s = NULL;

void setUp(void)
{
    s = safe_malloc(sizeof(stack));
}

void tearDown(void)
{
    safe_free(s);
    TEST_ASSERT_EQUAL(0, allocationCount);
}

void test_stack(void){
    TEST_ASSERT_NOT_NULL(s);
}
```

```
-----------------------
OVERALL TEST SUMMARY
-----------------------
TESTED:  1
PASSED:  1
FAILED:  0
IGNORED: 0
```

Now let’s test the size of the stack

- It should be 0 at the beginning
- Add a member to the struct

```
typedef struct stack{
    int size;
} stack;
```

```c
void setUp(void)
{
    s = safe_malloc(sizeof(stack));
}

/*
..
*/

void test_stackSize0(void){
    TEST_ASSERT_EQUAL_INT(0, s->size);
}
```

```
-----------------------
OVERALL TEST SUMMARY
-----------------------
TESTED:  2
PASSED:  2
FAILED:  0
IGNORED: 0
```

Test will fail on some systems

Why?

Add a constructor

- Add a function that receives a pointer and sets up all values
- Simplest way
- Set the size manually to 0

```c
#ifndef _STACK_H
#define _STACK_H
#include <stdint.h>

typedef struct stack{
    int size;
} stack;

void stack_create(stack* s);

#endif // _STACK_H
```

stack.h

```c
#include "stack.h"

void stack_create(stack* s){
    s->size = 0;
}
```

stack.c

```c
void setUp(void)
{
    s = safe_malloc(sizeof(stack));
    stack_create(s);
}
```

test_stack.c

Our stack needs to have space for elements

- Pass safe_malloc to constructor
- Pass maximum size to constructor
- Let constructor allocate its own memory
- What is the first step?
- Add a int* memory to your stack struct

Function pointer

```
void stack_create(stack** s, int maxSize, void* (*allocation)(size_t));
```

```
void stack_create(stack** s, int maxSize, void* (*allocation)(size_t)){
    s->size = 0;
}
```

```
void setUp(void)
{
    s = safe_malloc(sizeof(stack));
    stack_create(&s, 256, safe_malloc);
}
```

What did we change so far?

- Function declarations

What did we not change?

- Code

What do we do now?

- Refactoring
- We want to allocate memory in the constructor

```
void stack_create(stack** s, int maxSize, void* (*allocation)(size_t)){
    *s = (stack *)allocation(sizeof(stack));
    stack *stk = *s;
    stk->size = 0;
}
```

Why pass stack**?

- stack* is an Address
- You want stack* to point to heap allocated memory
- You change stack* in the function
- The original stack isn’t modified
- Pass a reference to the pointer

And now code?

- WRONG!
- Write a test

```
void test_allocationMemory(void){
    TEST_ASSERT_NOT_NULL(s->memory);
}
```

Will fail

```
typedef struct stack{
    int size;
    int* memory;
} stack;
```

We need to allocate a buffer

- How to test that the buffer has the right size?
- Difficult!

```c
void stack_create(stack** s, int maxSize, void* (*allocation)(size_t)){
    *s = (stack *)allocation(sizeof(stack));
    stack *stk = *s;
    stk->size = 0;
    stk->memory = allocation(maxSize * sizeof(int));
}
```

```
------------------
FAILED TEST SUMMARY
------------------
[test_stack.c]
  Test: test_stackNotNull
  At line (26): "Expected 0 Was 1"

  Test: test_stackSize0
  At line (26): "Expected 0 Was 2"

  Test: test_allocationMemory
  At line (26): "Expected 0 Was 3"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  3
PASSED:  0
FAILED:  3
IGNORED: 0

---------------------
BUILD FAILURE SUMMARY
---------------------
Unit test failures.
```

What do we miss?

Free the buffer

```
void tearDown(void)
{
    safe_free(s->memory);
    safe_free(s);
    TEST_ASSERT_EQUAL(0, allocationCount);
}
```

Now lets push something

- We want to test several things
- First things first
- After adding something to the stack, the size should increase

```c
void stack_push(stack* s, int element);
```

```c
void stack_push(stack* s, int element){

}
```

```c
void test_stackSizeIncreasesAfterPush(void){
    stack_push(s, 1);
    TEST_ASSERT_EQUAL_INT(1,s->size);
}
```

```
-----------------
FAILED TEST SUMMARY
-------------------
[test_stack.c]
  Test: test_stackSizeIncreasesAfterPush
  At line (44): "Expected 0 Was 1"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  4
PASSED:  3
FAILED:  1
IGNORED: 0

---------------------
BUILD FAILURE SUMMARY
---------------------
Unit test failures.
```

Let’s make it pass

```
void stack_push(stack* s, int element){
    s->size = 1;
}
```

```
→ ceedling


Test 'test_stack.c'
-------------------
Compiling stack.c...
Linking test_stack.out...
Running test_stack.out...

---------------------
OVERALL TEST SUMMARY
---------------------
TESTED:  4
PASSED:  4
FAILED:  0
IGNORED: 0
```

WRITE MORE TESTS

```c
void test_stackSizeIncreasesAfterPush(void){
    stack_push(s, 1);
    TEST_ASSERT_EQUAL_INT(s->size, 1);
}

void test_stackSizeIncreasesAfterPushTwice(void){
    stack_push(s, 1);
    stack_push(s, 2);
    TEST_ASSERT_EQUAL_INT(s->size, 2);
}
```

```
-------------------
FAILED TEST SUMMARY
-------------------
[test_stack.c]
  Test: test_stackSizeIncreasesAfterPushTwice
  At line (50): "Expected 1 Was 2"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  5
PASSED:  4
FAILED:  1
IGNORED: 0

--------------------
BUILD FAILURE SUMMARY
--------------------
Unit test failures.
```

REFACTOR

```
void stack_push(stack* s, int element){
    s->size++;
}
```

If the maximum is reached, no further adding is possible

- Add a maxSize to the struct
- Set it in the constructor

```c
typedef struct stack{
    int size;
    int maxSize;
    int *memory;
} stack;
```

```c
void test_maxStackAdd(void){
    for(int i = 0; i < 300; i++){
        stack_push(s, i);
    }
    TEST_ASSERT_EQUAL_INT(s->size, s->maxSize);
}
```

```
--------------------
FAILED TEST SUMMARY
--------------------
[test_stack.c]
  Test: test_maxStackAdd
  At line (57): "Expected 300 Was 256"
```

REFACTOR

```
void stack_push(stack* s, int element){
    if(s->size < s->maxSize){
        s->size++;
    }
}
```

When we push the first element we expect it at first position

```c
void test_addFirst(void){
    stack_push(s, 15);
    TEST_ASSERT_EQUAL_INT(15, s->memory[0]);
}
```

```
--------------------
FAILED TEST SUMMARY
--------------------
[test_stack.c]
  Test: test_addFirst
  At line (62): "Expected 15 Was 0"
```

```c
void stack_push(stack* s, int element){
    s->memory[0] = 15;
    if(s->size < s->maxSize){
        s->size++;
    }
}
```

```
Test 'test_stack.c'
------------------
Compiling stack.c...
Linking test_stack.out...
Running test_stack.out...

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  7
PASSED:  7
FAILED:  0
IGNORED: 0
```

```c
void test_addFirstSecond(void){
    stack_push(s, 15);
    stack_push(s, 17);
    TEST_ASSERT_EQUAL_INT(15, s->memory[0]);
    TEST_ASSERT_EQUAL_INT(17, s->memory[1]);
}
```

Is this test good?
How to make it better?

```
-------------------
FAILED TEST SUMMARY
-------------------
[test_stack.c]
  Test: test_addFirstSecond
  At line (69): "Expected 17 Was 0"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  8
PASSED:  7
FAILED:  1
IGNORED: 0
```

REFACTOR

```
void stack_push(stack* s, int element){
    s->memory[s->size] = element;
    if(s->size < s->maxSize){
        s->size++;
    }
}
```

If maximum elements are added, no further elements should be added

```
void test_maxStackAddElements(void){
    for(int i = 0; i < 300; i++){
        stack_push(s, i);
    }
    TEST_ASSERT_EQUAL_INT(255, s->memory[255]);
}
```

```
void stack_push(stack* s, int element){
    if(s->size < s->maxSize){
        s->memory[s->size] = element;
        s->size++;
    }
}
```

Popping an element decreases the size

```
void test_popSize(void){
    stack_push(s, 15);
    stack_push(s, 17);
    TEST_ASSERT_EQUAL_INT(2, s->size);
    stack_pop(s);
    TEST_ASSERT_EQUAL_INT(1, s->size);
}
```

How to make this test better?

```
-------------------
FAILED TEST SUMMARY
-------------------
[test_stack.c]
  Test: test_popSize
  At line (84): "Expected 1 Was 2"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  10
PASSED:   9
FAILED:   1
IGNORED:  0
```

```c
int stack_pop(stack* s){
    s->size--;
    return 0;
}
```

Popping Empty Stack lets size on 0

```c
void test_popEmptyStack(void){
    stack_pop(s);
    TEST_ASSERT_EQUAL_INT(0, s->size);
}
```

```
--------------------
FAILED TEST SUMMARY
--------------------
[test_stack.c]
  Test: test_popEmptyStack
  At line (89): "Expected 0 Was -1"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  11
PASSED:  10
FAILED:   1
IGNORED:  0
```

```c
int stack_pop(stack* s){
    if(s->size>0){
        s->size--;
    }
    return 0;
}
```

Popping Stack returns top element

```
void test_popReturnsTopElement(void){
    stack_push(s, 17);
    TEST_ASSERT_EQUAL_INT(17, stack_pop(s));
}
```

```
--------------------
FAILED TEST SUMMARY
--------------------
[test_stack.c]
  Test: test_popReturnsTopElement
  At line (94): "Expected 17 Was 0"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  12
PASSED:  11
FAILED:   1
IGNORED:  0
```

```c
int stack_pop(stack* s){
    if(s->size>0){
        s->size--;
    }
    return 17;
}
```

Popping multiple elements

```
void test_popReturnsTopElements(void){
    stack_push(s, 17);
    stack_push(s, 15);
    TEST_ASSERT_EQUAL_INT(15, stack_pop(s));
    TEST_ASSERT_EQUAL_INT(17, stack_pop(s));
}
```

How can we improve this test?

```
--------------------
FAILED TEST SUMMARY
--------------------
[test_stack.c]
  Test: test_popReturnsTopElements
  At line (100): "Expected 15 Was 17"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  13
PASSED:  12
FAILED:   1
IGNORED:  0
```

```c
int stack_pop(stack* s){
    if(s->size>0){
        s->size--;
    }
    return s->memory[s->size];
}
```

Popping empty stack returns 0

```c
void test_popOnEmptyStackReturns0(void){
    TEST_ASSERT_EQUAL_INT(0, stack_pop(s));
}
```

```
-------------------
FAILED TEST SUMMARY
-------------------
[test_stack.c]
  Test: test_popOnEmptyStackReturns0
  At line (105): "Expected 0 Was 17"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  14
PASSED:  13
FAILED:   1
IGNORED:  0
```

```c
int stack_pop(stack* s){
    if(s->size>0){
        s->size--;
        return s->memory[s->size];
    }else{
        return 0;
    }
}
```

stack.h

```c
#ifndef _STACK_H
#define _STACK_H
#include <stdint.h>
#include <stdlib.h>

typedef struct stack{
    int size;
    int maxSize;
    int *memory;
} stack;

void stack_create(stack** s, int maxSize, void* (*allocation)
(size_t));
void stack_push(stack* s, int element);
int stack_pop(stack* s);

#endif // _STACK_H
```

stack.c

```c
#include "stack.h"

void stack_create(stack** s, int maxSize, void* (*allocation)(size_t)){
    *s = (stack *)allocation(sizeof(stack));
    stack *stk = *s;
    stk->size = 0;
    stk->maxSize = maxSize;
    stk->memory = allocation(maxSize * sizeof(int));
}
```

```c
void stack_push(stack* s, int element){
    if(s->size < s->maxSize){
        s->memory[s->size] = element;
        s->size++;
    }
}

int stack_pop(stack* s){
    if(s->size>0){
        s->size--;
        return s->memory[s->size];
    }else{
        return 0;
    }
}
```

test_stack.c

```c
#include "unity.h"
#include "stack.h"
#include <stdint.h>
#include <stdlib.h>

uint16_t allocationCount = 0;
void* safe_malloc(size_t size){
    allocationCount++;
    return malloc(size);
}
void safe_free(void* ptr){
    allocationCount--;
    free(ptr);
}

stack *s = NULL;

void setUp(void)
{
    stack_create(&s, 256, safe_malloc);
}

void tearDown(void)
{
    safe_free(s->memory);
    safe_free(s);
    TEST_ASSERT_EQUAL(0, allocationCount);
}

void test_stackNotNull(void){
    TEST_ASSERT_NOT_NULL(s);
}
```

```c
void test_stackSize0(void){
    TEST_ASSERT_EQUAL_INT(0, s->size);
}

void test_allocationMemory(void){
    TEST_ASSERT_NOT_NULL(s->memory);
}

void test_stackSizeIncreasesAfterPush(void){
    stack_push(s, 1);
    TEST_ASSERT_EQUAL_INT(s->size, 1);
}

void test_stackSizeIncreasesAfterPushTwice(void){
    stack_push(s, 1);
    stack_push(s, 2);
    TEST_ASSERT_EQUAL_INT(s->size, 2);
}

void test_maxStackAdd(void){
    for(int i = 0; i < 300; i++){
        stack_push(s, i);
    }
    TEST_ASSERT_EQUAL_INT(s->size, s->maxSize);
}

void test_addFirst(void){
    stack_push(s, 15);
    TEST_ASSERT_EQUAL_INT(15, s->memory[0]);
}

void test_addFirstSecond(void){
    stack_push(s, 15);
    stack_push(s, 17);
    TEST_ASSERT_EQUAL_INT(15, s->memory[0]);
    TEST_ASSERT_EQUAL_INT(17, s->memory[1]);
}
```

```c
void test_maxStackAddElements(void){
    for(int i = 0; i < 300; i++){
        stack_push(s, i);
    }
    TEST_ASSERT_EQUAL_INT(255, s->memory[255]);
}

void test_popSize(void){
    stack_push(s, 15);
    stack_push(s, 17);
    TEST_ASSERT_EQUAL_INT(2, s->size);
    stack_pop(s);
    TEST_ASSERT_EQUAL_INT(1, s->size);
}

void test_popEmptyStack(void){
    stack_pop(s);
    TEST_ASSERT_EQUAL_INT(0, s->size);
}

void test_popReturnsTopElement(void){
    stack_push(s, 17);
    TEST_ASSERT_EQUAL_INT(17, stack_pop(s));
}

void test_popReturnsTopElements(void){
    stack_push(s, 17);
    stack_push(s, 15);
    TEST_ASSERT_EQUAL_INT(15, stack_pop(s));
    TEST_ASSERT_EQUAL_INT(17, stack_pop(s));
}

void test_popOnEmptyStackReturns0(void){
    TEST_ASSERT_EQUAL_INT(0, stack_pop(s));
}
```

Your Homework

- Literally implement any data structure using TDD
- Ask me if you encounter problems
- This will be very counterintuitive at first
- This is how you write great code

Upload your results to Github if you like, you can reuse these structures