CSCI 104 Lab 4:

Unit Testing

Why Unit Test

- Whole program testing (what you've mostly done so far) is meant to verify that your entire program works by verifying its output for different inputs
- This is fine for small programs, but as your program gets larger, there are more and more things inside it that can interact in different ways to potentially cause failures
- Also, many internal features of the program (such as data structures) may not be accessible from the "interface" of the program at all
- Goal of unit testing: test *each piece* of the program separately

Unit Testing Rationale

- Create a *test suite* of tests for a given component of your program
- Create *test cases* that each test one feature or use case for this feature
- Each test case includes one or more *assertions* that check the result of a program's behavior

What to Unit Test

- Figuring out *what* input to test your program on is always the hardest part of testing!
- In general, there are four different categories of test input to try:
  - Nominal
  - Boundary
  - Off-Nominal
  - Stress

Nominal Tests

- Tests that provide the program with standard, expected input
- Example: for a config file parser, nominal input would be a short, correctly formatted file with no unusual constructs
- What would be a nominal test case for a List?

Boundary Tests

- Tests that explore the “edges” of possible input, and/or data values that require special handling
- Often these cases are where programs fail when they would otherwise work for most data
- Example: for a config file parser, boundary input could be lines with comments adjacent to a piece of data, or with a value on the last line of the file
- What would be a boundary test case for a List?

Off-Nominal Tests

- Tests that provide invalid or nonsensical input
- It's easy for unhandled bad input to crash a program or make it spew out garbage
- However: also important to only pass bad input that your programs are supposed to be able to handle! Read the assignment carefully!
- Example: for a config file parser, off-nominal input would be a file with syntax errors, or a blank or nonexistent file
- What would be an off-nominal test case for a List?

Stress Tests

- Tests that provide huge amounts of input (the most it's expected to see in real use)
- If a program has optimization problems or too poor of a runtime, this will catch it pretty quick!
- Example: for a config file parser, a stress test would be a config file with thousands of lines
- What would be a stress test case for a List?

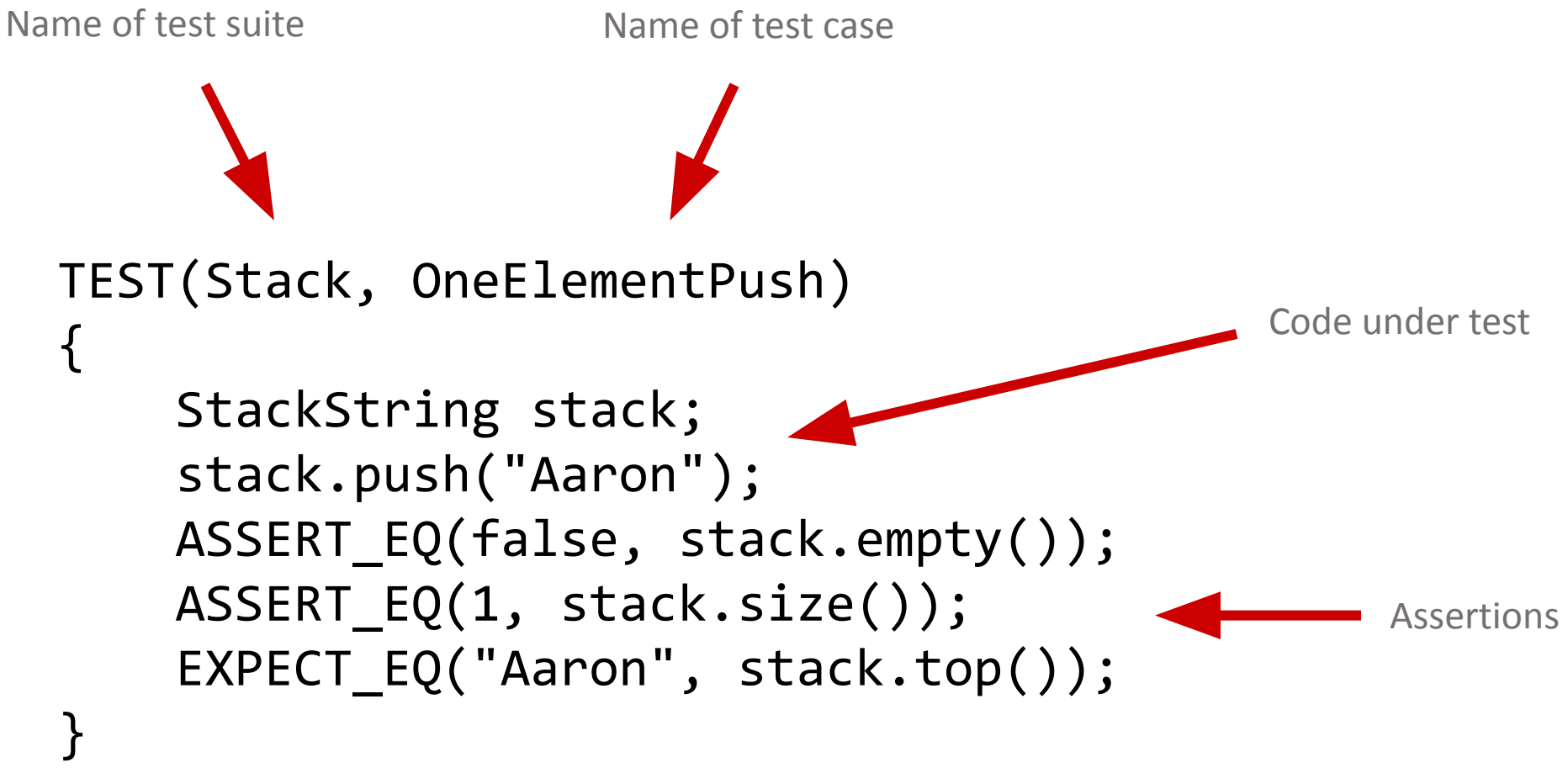
Planning Test Cases

- Carefully scan the assignment sheet and skeleton code for requirements!
  - What is the expected input for this code?
  - What kinds of input are invalid?
  - What should your program do when it hits invalid input?
- Always think about what ways you can test when writing your program.
  - Keep an eye out for what would be good boundary and off-nominal cases!
- Creating these tests gets easier as you get more practice, we promise!
- One method: Test Driven Development, where you write the tests from the requirements *before* you write the code!
  - This is a lot of extra work, but it can help you plan out your code and what it needs to do

How to Unit Test

- Here in CS104 we use a library called Google Test (GTest) to help write unit tests quickly and easily
- GTest provides macros to declare test cases, and a built-in main() function to execute them
- Requires specific flags added to your Makefile:
  - Compile with "`-I /usr/include/gtest/`"
  - Link with "`-l gtest -l gtest_main -pthread`"

Anatomy of a Test Case



The TEST macro

- The TEST() statement is clearly not standard C++
- It is actually a preprocessor macro that gets replaced with another piece of code (a function declaration, plus some setup code)
- Luckily, you don’t actually need to worry about how it works! Just pass it the names of your test suite and test case, and it will take care of setting it up

Assertions

- Assertions are added using the EXPECT_XXX() and ASSERT_XXX() functions
- Assertions compare the first argument (actual) with the second argument (expected).
- EXPECT generates non-fatal failures: the test will print as a failure, but the code in the test case will keep running
- ASSERT generates fatal failures: the test will print as a failure, and the code in the test case will stop immediately
- In what situations would we want to use a fatal failure or a nonfatal one?

Types of Assertions

Comparisons

Assertion	Verifies
EXPECT_EQ(val1, val2);	val1 == val2
EXPECT_NE(val1, val2);	val1 != val2
EXPECT_LT(val1, val2);	val1 < val2
EXPECT_LE(val1, val2);	val1 <= val2
EXPECT_GT(val1, val2);	val1 > val2
EXPECT_GE(val1, val2);	val1 >= val2

(source: GTest Primer)

Logical

Assertion	Verifies
EXPECT_TRUE(condition);	condition is true
EXPECT_FALSE(condition);	condition is false

Exceptions

Assertion	Verifies
EXPECT_THROW(statement, exception_type);	statement throws an exception of the given type
EXPECT_ANY_THROW(statement);	statement throws an exception of any type
EXPECT_NO_THROW(statement);	statement doesn't throw any exception

Looking Back at a Test Case

```
TEST(Stack, OneElementPush)
{
    StackString stack;
    stack.push("Aaron");
    ASSERT_EQ(false, stack.empty());
    ASSERT_EQ(1, stack.size());
    EXPECT_EQ("Aaron", stack.top());
}
```

What is being tested here?

The push() function of this stack implementation

Looking Back at a Test Case

```
TEST(Stack, OneElementPush)
{
    StackString stack;
    stack.push("Aaron");
    ASSERT_EQ(false, stack.empty());
    ASSERT_EQ(1, stack.size());
    EXPECT_EQ("Aaron", stack.top());
}
```

Judging by the test cases, what do we expect to happen when we call push()?

Empty() will return false, size() will return 1, and top() will return the element that was just added

Looking Back at a Test Case

```
TEST(Stack, OneElementPush)
{
    StackString stack;
    stack.push("Aaron");
    ASSERT_EQ(false, stack.empty());
    ASSERT_EQ(1, stack.size());
    EXPECT_EQ("Aaron", stack.top());
}
```

Why are the first two assertions fatal?

The top() function of this stack implementation is not allowed to be called if there are no elements in the stack.

Part 1 (guided)

- Please clone the latest lab repository now
- We will work together to complete a few GTests for a Fibonacci number calculator in Part 1

Advanced Topics: GTest and Operators

- GTest comparisons act like if statements, and will use custom comparison operators if you've defined them
- Also, GTest will try to print the values of things used in assertions that fail
- To let it print your custom types you can define a custom operator<<, e.g.:

```
std::ostream operator<<(std::ostream & stream,
Commit const & commit)
```

Advanced Topics: Custom Checker Functions

- Have to check something that is too complicated for just a few assertions?
- You can also define a custom checker function that does something more complex
- Method 1: add EXPECTs directly into the function body (ASSERTs will not stop the test since those only exit the current function)

Advanced Topics: Custom Checker Functions

- Method 2: Make your function return an AssertionResult and then pass that to EXPECT_TRUE()
- This lets you create completely custom and extremely detailed error messages!
- Example:

```
testing::AssertionResult checkComplicatedCondition(CustomObject & customObj)
{
    if(!customObj.someSpecificCondition)
    {
        return testing::AssertionFailure() << “Specific condition has failed!”;
    }
    …
    return testing::AssertionSuccess();
}
```

Part 2: Testing a List

- You don’t have the implementation for this class, and you need to find two bugs with it.
- Good luck!