CS106L Optional Lecture: Unit Testing & C++ Iceberg

Thomas Poimenidis, Rachel Fernandez

Attendance

https://forms.gle/uGPxuzbvoeppcMXe6

Plan

1. What is Unit Testing?
2. Google’s C++ Test Suite
3. Exploring the C++ Iceberg!

Plan

1. **What is Unit Testing?**
2. Google's C++ Test Suite
3. Exploring the C++ Iceberg!

What is Unit Testing?

“Unit testing is a test-driven development (TDD) method for evaluating software that pays special attention to an individual component or unit of code—the smallest increment possible” (IBM)

Unit Testing vs. Other Testing Types

	Unit Tests	Integration/System Tests
Scope	Small scope (typically a single class or method).	Can be small or large scope (integration of classes/methods, entire systems).
Isolation	Must isolate the unit being tested (no external dependencies, network/file system reads/writes). Dependencies are typically mocked.	Do not necessarily need to be isolated. Typically, integrated within the rest of the application/system and tested within that context.
Speed	Typically, very quick.	Longer, more extensive tests.
Purpose	Verify functionality of individual unit.	Validate functionality of a unit/system/application in the context of a larger subset of the entire system.

How to write Unit Tests?

To write unit test (according to IBM):

1. Identify the Unit

How to write Unit Tests?

To write unit test (according to IBM):

1. Identify the Unit
2. Select an Approach

How to write Unit Tests?

To write unit test (according to IBM):

1. Identify the Unit
2. Select an Approach
3. Establish the Environment (testing framework, testing data, etc.)

How to write Unit Tests?

To write unit test (according to IBM):

1. Identify the Unit
2. Select an Approach
3. Establish the Environment (testing framework, testing data, etc.)
4. Create and use test cases

How to write Unit Tests?

To write unit test (according to IBM):

1. Identify the Unit
2. Select an Approach
3. Establish the Environment (testing framework, testing data, etc.)
4. Create and use test cases
5. Debug and resolve issues (force your tests to find issues)

Why write Unit Tests?

1. Catch bugs early: Unit tests are quick and test small units. Can run frequently when developing).

Why write Unit Tests?

1. Catch bugs early: Unit tests are quick and test small units. Can run frequently when developing).
2. Safety net for changes: If unit tests are already properly written, additional changes to the unit can instantly be tested.

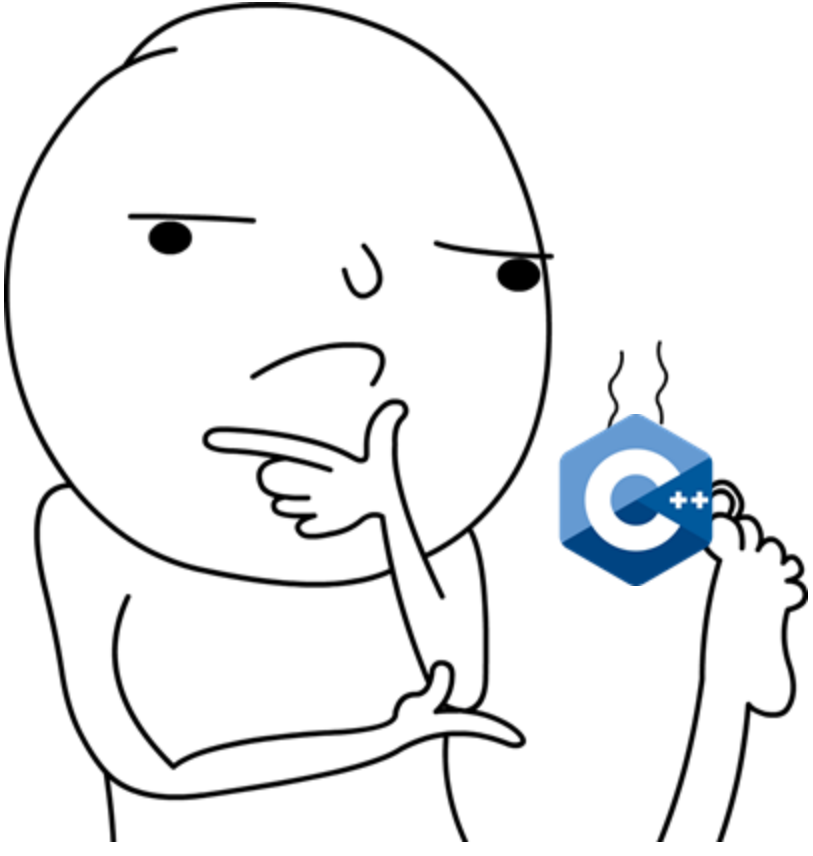
Why write Unit Tests?

1. Catch bugs early: Unit tests are quick and test small units. Can run frequently when developing).
2. Safety net for changes: If unit tests are already properly written, additional changes to the unit can instantly be tested.
3. Documentation: Unit tests can act as documentation for what has and hasn’t been tested.

Why write Unit Tests?

1. Catch bugs early: Unit tests are quick and test small units. Can run frequently when developing).
2. Safety net for changes: If unit tests are already properly written, additional changes to the unit can instantly be tested.
3. Documentation: Unit tests can act as documentation for what has and hasn't been tested.
4. Many more reasons!

What questions do we have?



Plan

1. What is Unit Testing?
2. **Google's C++ Test Suite**
3. Exploring the C++ Iceberg!

C++ Testing Frameworks

Catch2

Boost.Test

CppTest

CppTest

C++ Testing Frameworks

googletest

Google C++ Testing Framework

GoogleTest

What is GoogleTest?

- C++ Testing framework developed by Google!

What is GoogleTest?

- C++ Testing framework developed by Google!
- Provides test fixtures, parameterized tests, and other testing features.

What is GoogleTest?

- C++ Testing framework developed by Google!
- Provides test fixtures, parameterized tests, and other testing features.
- Under the hood: a collection of macros and assertions that are inserted into your code by the preprocessor.

What is GoogleTest?

- C++ Testing framework developed by Google!
- Provides test fixtures, parameterized tests, and other testing features.
- Under the hood: a collection of macros and assertions that are inserted into your code by the preprocessor.
- Test discovery: Automatically registers tests at compile-time, No need to keep track of tests manually.

What is GoogleTest?

- C++ Testing framework developed by Google!
- Provides test fixtures, parameterized tests, and other testing features.
- Under the hood: a collection of macros and assertions that are inserted into your code by the preprocessor.
- Test discovery: Automatically registers tests at compile-time, No need to keep track of tests manually.
- Integration with Google Mock (use mock objects in testing frameworks)

What is GoogleTest?

- C++ Testing framework developed by Google!
- Provides test fixtures, parameterized tests, and other testing features.
- Under the h rtions that are inserted
- Test discov ompile-time, No need to keep track of tests manually.
- Integration with Google Mock (use mock objects in testing frameworks)

A LOT.

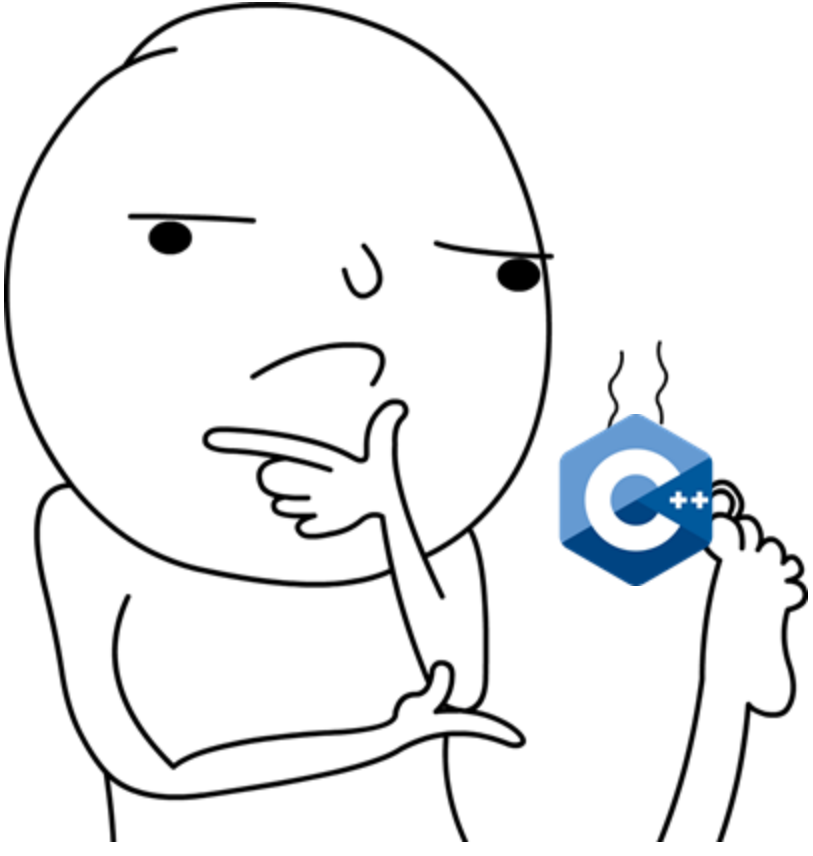
What is GoogleTest?

- C++ Testing framework developed by Google!
- Provides test fixtures, parameterized tests, and other testing features

Let’s Break it Down

- Test discovery: Automatically registers tests at compile-time, No need to keep track of tests manually.
- Integration with Google Mock (use mock objects in testing frameworks)

What questions do we have?



Let’s test this BankAccount class

bank_account.h

```h
#ifndef __BANK_ACCOUNT_H__
#define __BANK_ACCOUNT_H__

struct BankAccount {

    // member variables
    double balance;

    // constructors
    BankAccount();
    explicit BankAccount(const double initial_balance);

    // member functions
    void deposit(double amount);
    bool withdraw(double amount);

};

#endif  // __BANK_ACCOUNT_H__
```

bank_account.cpp

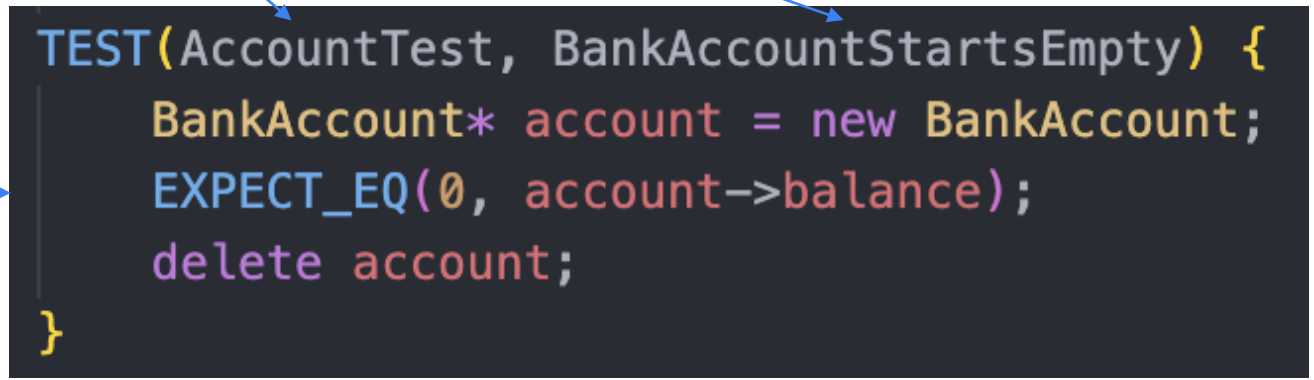
```cpp
#include "bank_account.hpp"

// default constructor initializes to balance of 0
BankAccount::BankAccount()
    : balance(0) {}

// custom constructor that initializes balance to initial_balance
BankAccount::BankAccount(const double initial_balance)
    : balance(initial_balance) {}

// deposit amount into the account
void BankAccount::deposit(double amount) {
    balance += amount;
}

// withdraws amount from balance if funds exist.
bool BankAccount::withdraw(double amount)
{
    if (amount <= balance) {
        balance -= amount;
        return true;
    }
    return false;
}

```

GoogleTest: TEST

TEST

```
TEST(TestSuiteName, TestName) {
  ... statements ...
}
```

Defines an individual test named `TestName` in the test suite `TestSuiteName`, consisting of the given statements.



GoogleTest: TEST_F

TEST_F

```
TEST_F(TestFixtureName, TestName) {
  ... statements ...
}
```

Defines an individual test named `TestName` that uses the test fixture class `TestFixtureName`. The test suite name is `TestFixtureName`.

```
TEST_F(BankAccountTest, BankAccountStartsEmpty) {
    EXPECT_EQ(0, account->balance);
}
```

GoogleTest: TEST_F

TEST_F

```
TEST_F(TestFixtureName, TestName) {
  ... statements ...
}
```

Defines an individual test named `TestName` that uses the test fixture class `TestFixtureName`. The test suite name is `TestFixtureName`.

```
TEST_F(BankAccountTest, BankAccountStartsEmpty) {
    EXPECT_EQ(0, account->balance);
}
```

Hm... this looks the same.
What changed?

GoogleTest: TEST vs. TEST_F

```
TEST(AccountTest, BankAccountStartsEmpty) {
    BankAccount* account = new BankAccount;
    EXPECT_EQ(0, account->balance);
    delete account;
}
```

Just a test name.

```
TEST_F(BankAccountTest, BankAccountStartsEmpty) {
    EXPECT_EQ(0, account->balance);
}
```

This is a test fixture class!

GoogleTest: TEST vs. TEST_F

```
struct BankAccountTest : testing::Test{
    BankAccount* account;

    BankAccountTest() {
        account = new BankAccount;
    }

    virtual ~BankAccountTest() {
        delete account;
    }
};
```

Here's the Test Fixture Class!

- Inherited from testing::Test (GoogleTest base fixture class)
- Acts as a wrapper for initializing the BankAccount object.
- Automatically constructs at the beginning of the test fixture.
- Automatically destructs at the end of the test fixture.
- **First parameter of TEST_F has to be the fixture class name.**

```
TEST_F(BankAccountTest, BankAccountStartsEmpty) {
    EXPECT_EQ(0, account->balance);
}
```

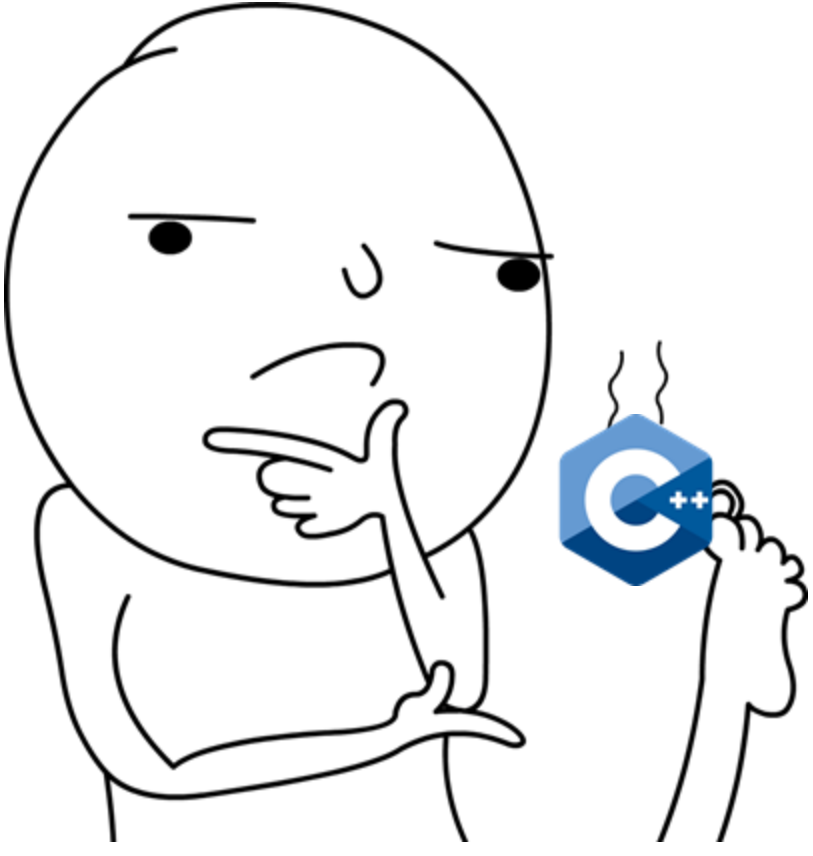
GoogleTest: TEST vs. TEST_F

```
TEST(AccountTest, BankAccountStartsEmpty) {
    BankAccount* account = new BankAccount;
    EXPECT_EQ(0, account->balance);
    delete account;
}
```

```
TEST_F(BankAccountTest, BankAccountStartsEmpty) {
    EXPECT_EQ(0, account->balance);
}
```

Fixtures help remove the initialization and clean-up work required when testing class objects.

What questions do we have?



GoogleTest: TEST_P

```
struct account_state {
    int initial_balance;
    int withdraw_amount;
    int final_balance;
    bool success;

    // operator overload to print nicer error messages. Try making a WithdrawAccountTest
    // fail with and without this overload to see the difference in error message outputs.
    friend std::ostream& operator<<(std::ostream& os, const account_state& obj) {
        return os
            << "initial_balance: " << obj.initial_balance
            << " withdraw_amount: " << obj.withdraw_amount
            << " final_balance: " << obj.final_balance
            << " success: " << obj.success;
    }
};
```

```
struct WithdrawAccountTest : BankAccountTest, testing::WithParamInterface<account_state> {
    WithdrawAccountTest() {
        account->balance = GetParam().initial_balance;
    }
};
```

For parameterized tests, we need a special kind of test fixture.

Need to inherit from testing:WithParamInterface<T>

Where T is a struct containing our parameters.

GoogleTest: TEST_P

```
struct account_state {
    int initial_balance;
    int withdraw_amount;
    int final_balance;
    bool success;

    // operator overload to print nicer error messages. Try making a WithdrawAccountTest
    // fail with and without this overload to see the difference in error message outputs.
    friend std::ostream& operator<<(std::ostream& os, const account_state& obj) {
        return os
            << "initial_balance: " << obj.initial_balance
            << " withdraw_amount: " << obj.withdraw_amount
            << " final_balance: " << obj.final_balance
            << " success: " << obj.success;
    }
};
```

Can overload the << operator for the parameter inputs for better error messages!

```
struct WithdrawAccountTest : BankAccountTest, testing::WithParamInterface<account_state> {
    WithdrawAccountTest() {
        account->balance = GetParam().initial_balance;
    }
};
```

GoogleTest: TEST_P

```
TEST_P(WithdrawAccountTest, FinalBalance){
    auto as = GetParam();
    auto success = account->withdraw(as.withdraw_amount);
    EXPECT_EQ(as.final_balance, account->balance);
    EXPECT_EQ(as.success, success);
}
```

Here is our parameterized test!

GoogleTest: TEST_P

```
TEST_P(WithdrawAccountTest, FinalBalance){
    auto as = GetParam();
    auto success = account->withdraw(as.withdraw_amount);
    EXPECT_EQ(as.final_balance, account->balance);
    EXPECT_EQ(as.success, success);
}
```

But hold on...
What are the parameters?
Where are they coming from?
Who is defining them?

GoogleTest: TEST_P

```
INSTANTIATE_TEST_SUITE_P(DEFAULT, WithdrawAccountTest,
    testing::Values(
        account_state{100,50,50,true},
        account_state{100,200,100,false}
    ));
```

Parameterized tests need to be instantiated!

Using a DEFAULT name, create a parameterized test on the WithdrawAccountTest test suite, using the parameters testing::Values(...)

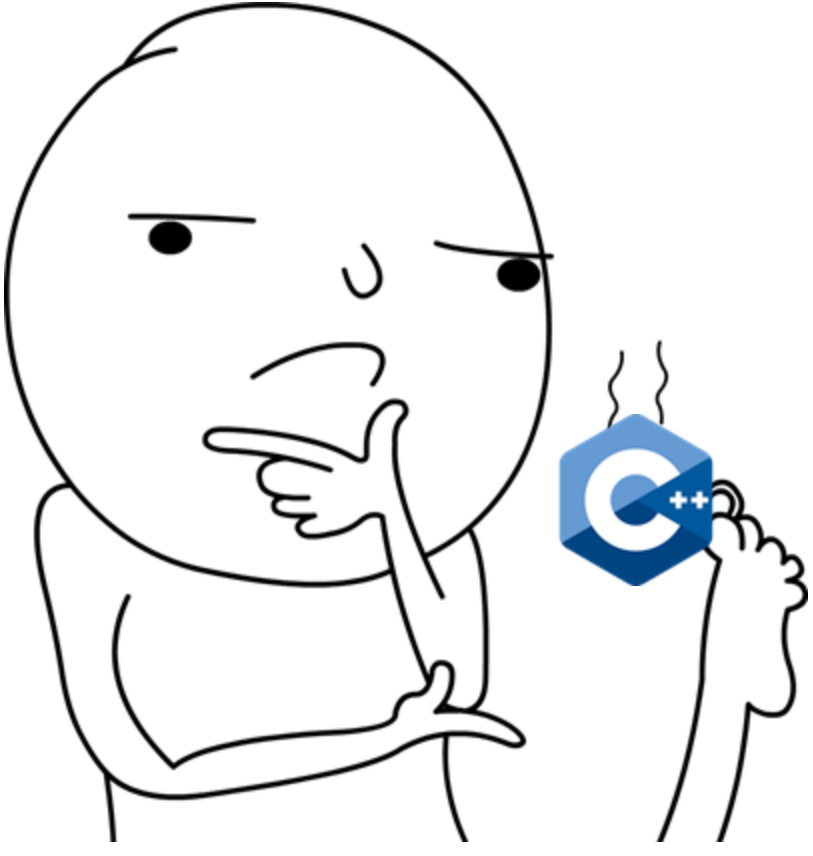
GoogleTest: TEST_P

```
INSTANTIATE_TEST_SUITE_P(DEFAULT, WithdrawAccountTest,
    testing::Values(
        account_state{100,50,50,true},
        account_state{100,200,100,false}
    ));
```

Providing multiple sets of values results in multiple tests!

Super clean way of instantiating lots of tests!

What questions do we have?

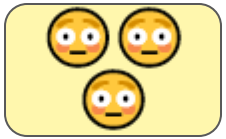


GoogleTest: Behind the Scenes!

```
#define TEST_P(test_suite_name,test_name) class GTEST_TEST_CLASS_NAME_(test_suite_name, test_name) : public
test_suite_name, private ::testing::internal::GTestNonCopyable { public:
GTEST_TEST_CLASS_NAME_(test_suite_name, test_name)() {} void TestBody() override; private: static int
AddToRegistry() { ::testing::UnitTest::GetInstance() ->parameterized_test_registry()
.GetTestSuitePatternHolder<test_suite_name>( GTEST_STRINGIFY_(test_suite_name),
::testing::internal::CodeLocation(__FILE__, __LINE__)) ->AddTestPattern( GTEST_STRINGIFY_(test_suite_name),
GTEST_STRINGIFY_(test_name), new ::testing::internal::TestMetaFactory<GTEST_TEST_CLASS_NAME_( test_suite_name,
test_name)>(), ::testing::internal::CodeLocation(__FILE__, __LINE__)); return 0; } [[maybe_unused]] static int
gtest_registering_dummy_; }; int GTEST_TEST_CLASS_NAME_(test_suite_name, test_name)::gtest_registering_dummy_ =
GTEST_TEST_CLASS_NAME_(test_suite_name, test_name)::AddToRegistry(); void
GTEST_TEST_CLASS_NAME_(test_suite_name, test_name)::TestBody()
```

Expands to:

```
class WithdrawAccountTest_FinalBalance_Test : public WithdrawAccountTest, private
::testing::internal::GTestNonCopyable { public: WithdrawAccountTest_FinalBalance_Test() {} void TestBody()
override; private: static int AddToRegistry() { ::testing::UnitTest::GetInstance() -
>parameterized_test_registry() .GetTestSuitePatternHolder<WithdrawAccountTest>( "WithdrawAccountTest",
::testing::internal::CodeLocation("/Users/thomas/Desktop/Stanford/cs106l/cs106l-lecture-
code/unit_testing/main.cpp", 103)) ->AddTestPattern( "WithdrawAccountTest", "FinalBalance", new
::testing::internal::TestMetaFactory<WithdrawAccountTest_FinalBalance_Test>(),
::testing::internal::CodeLocation("/Users/thomas/Desktop/Stanford/cs106l/cs106l-lecture-
code/unit_testing/main.cpp", 103)); return 0; } [[maybe_unused]] static int gtest_registering_dummy_; }; int
WithdrawAccountTest_FinalBalance_Test::gtest_registering_dummy_ =
WithdrawAccountTest_FinalBalance_Test::AddToRegistry(); void WithdrawAccountTest_FinalBalance_Test::TestBody()
```

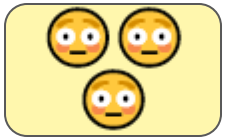


GoogleTest: Behind the Scenes!

```
#define TEST_P(test_suite_name,test_name) class GTEST_TEST_CLASS_NAME_(test_suite_name, test_name) : public
test_suite_name, private ::testing::internal::GTestNonCopyable { public:
GTEST_TEST_CLASS_NAME_(test_suite_name, test_name)() {} void TestBody() override; private: static int
AddToRegistry() { ::testing::UnitTest::GetInstance() ->parameterized_test_registry()
.GetTestSuitePatternHolder<test_suite_name>( GTEST_STRINGIFY_(test_suite_name),
::testing::internal::CodeLocation(__FILE__, __LINE__)) ->AddTestPattern( GTEST_STRINGIFY_(test_suite_name),
GTEST_STRINGIFY_(test_name), new ::testing::internal::TestMetaFactory<GTEST_TEST_CLASS_NAME_( test_suite_name,
test_name)>(), ::testing::internal::CodeLocation(__FILE__, __LINE__)); return 0; } [[maybe_unused]] static int
gtest_registering_dummy_; }; int GTEST_TEST_CLASS_NAME_(test_suite_name, test_name)::gtest_registering_dummy_ =
GTEST_TEST_CLASS_NAME_(test_suite_name, test_name)::AddToRegistry(); void
GTEST_TEST_CLASS_NAME_(test_suite_name, test_name)::TestBody()
```

Expands to:

```
class WithdrawAccountTest_FinalBalance_Test : public WithdrawAccountTest, private
::testing::internal::GTestNonCopyable { public: WithdrawAccountTest_FinalBalance_Test() {} void TestBody()
override; private: static int AddToRegistry() { ::testing::UnitTest::GetInstance() -
>parameterized_test_registry() .GetTestSuitePatternHolder<WithdrawAccountTest>( "WithdrawAccountTest",
::testing::internal::CodeLocation("/Users/thomas/Desktop/Stanford/cs106l/cs106l-lecture-
code/unit_testing/main.cpp", 103)) ->AddTestPattern( "WithdrawAccountTest", "FinalBalance", new
::testing::internal::TestMetaFactory<WithdrawAccountTest_FinalBalance_Test>(),
::testing::internal::CodeLocation("/Users/thomas/Desktop/Stanford/cs106l/cs106l-lecture-
code/unit_testing/main.cpp", 103)); return 0; } [[maybe_unused]] static int gtest
WithdrawAccountTest_FinalBalance_Test::gtest_registering_dummy_ =
WithdrawAccountTest_FinalBalance_Test::AddToRegistry(); void WithdrawAccountTest_
```

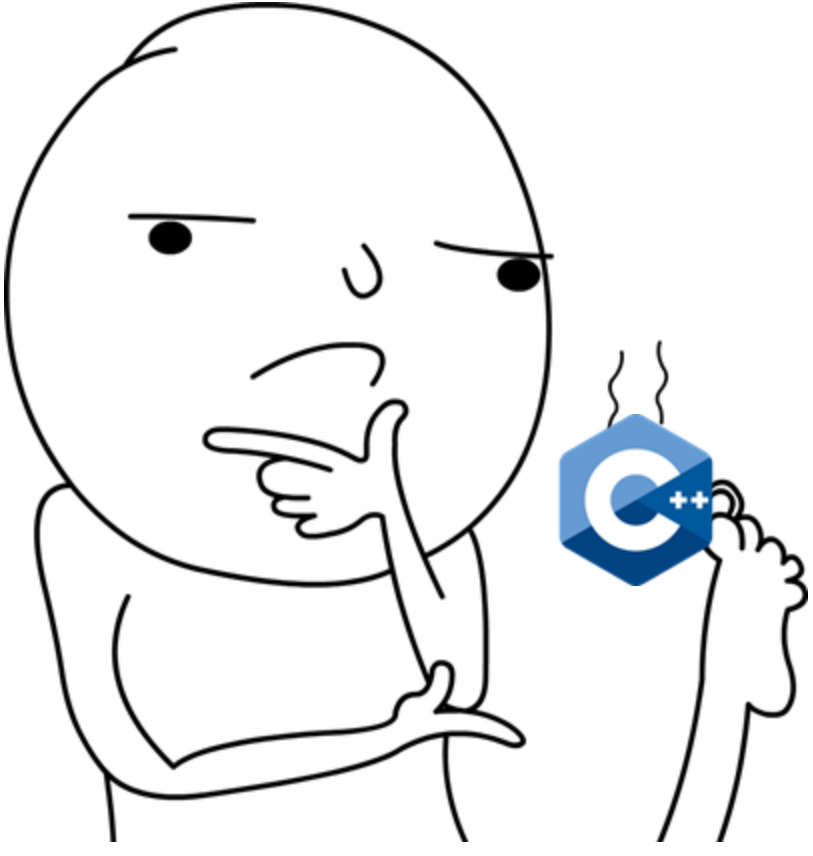


Thankfully, we don't have to deal with any of this!

Code Demo!

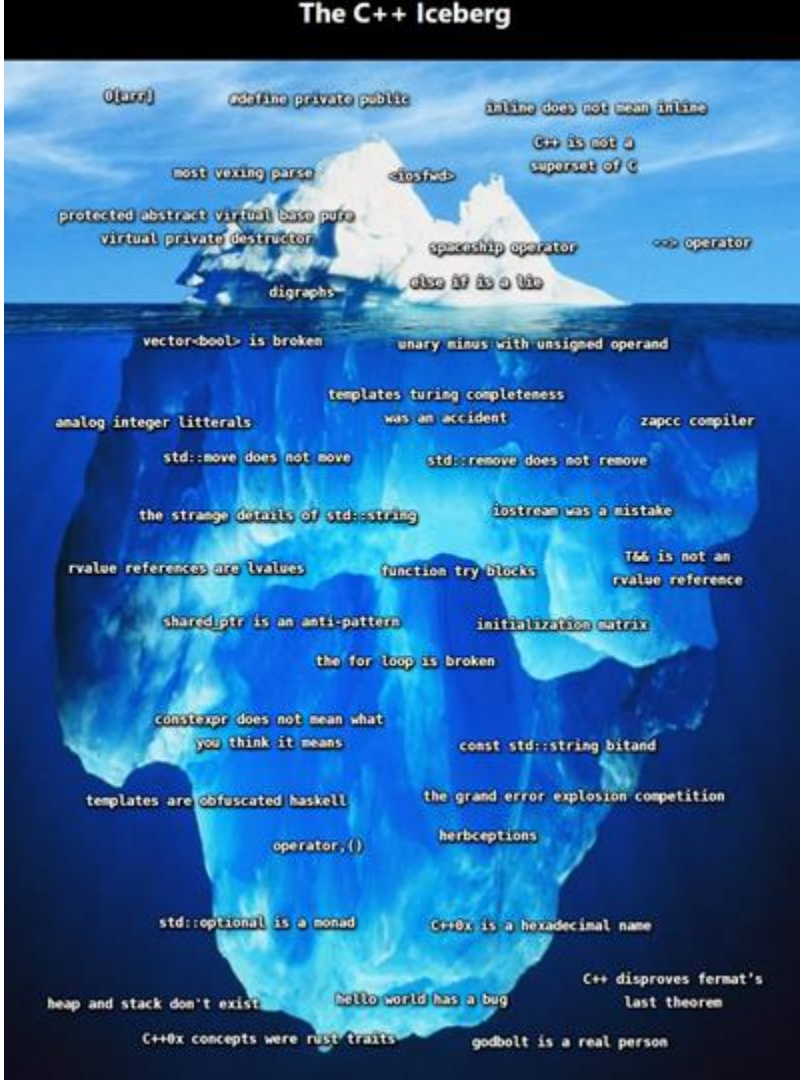
Let’s try using GoogleTest!

What questions do we have?



Plan

1. What is Unit Testing?
2. Google's C++ Test Suite
3. **Exploring the C++ Iceberg!**


The C++ Iceberg
O[arr]
#define private public
inline does not mean inline
C++ is not a superset of C
most vexing parse
<iosfwd>
protected abstract virtual base pure
virtual private destructor
spaceship operator
--> operator
digraphs
else if is a lie
vector<bool> is broken
unary minus with unsigned operand
templates turing completeness
was an accident
analog integer litterals
zapcc compiler
std::move does not move
std::remove does not remove
the strange details of std::string
iostream was a mistake
T&& is not an rvalue reference
rvalue references are lvalues
function try blocks
shared_ptr is an anti-pattern
initialization matrix
the for loop is broken
constexpr does not mean what you think it means
const std::string bitand
templates are obfuscated haskell
the grand error explosion competition
herbceptions
operator,()
std::optional is a monad
C++0x is a hexadecimal name
C++ disproves fermat's last theorem
heap and stack don't exist
hello world has a bug
C++0x concepts were rust traits
godbolt is a real person

[source]

Exploring the C++ Iceberg!

Get in groups and explore the C++ iceberg!

https://victorpoughon.github.io/cppiceberg/

We'll come back together and share interesting things we find!

Announcements

- Assignment 7 due Saturday, 12/6
- Thursday will be used as extra office hours. Come stop by if you have the time!
- Check your grade on the website and let us know if something looks off. This is what we use to determine C/NC for this course.
- You’re almost done with CS106L!!!

Thank you for a great quarter!

tpoimen@stanford.edu
LinkedIn

rfern@stanford.edu
LinkedIn