Testing

CMSC 240
All examples borrowed/modified from
C++ Crash Course by Josh Lospinoso
No Starch Press
Along the Way

- These slides are about unit testing and TDD
- But along the way, we’ll pick up:
  - Function objects
  - Lambda expressions
  - `std::function`
Unit Tests

- Unit tests verify that a focused collection of code (e.g., function or class) behave as intended
  - Want these tests to isolate unit being tested from its depend dependencies (though this may be difficult)
  - If tested unit depends on other unit, sometimes use *mocks* (fake objects) as stand in during tests
    - Mocks are only use for testing
Mocks

- Can be used to simulate fine-grained control over how the dependencies behave during test
- Can also test how unit is interacting with mocks, to ensure this is correct
- Can use mocks to simulate rare events (e.g., out of memory) by programming them to throw exceptions
Types of Unit Tests

• Integration Tests: Test a collection of units together
  ◦ Can also refer to testing interactions between software and hardware
  ◦ NOT a replacement for individual unit tests, but complement them
Types of Unit Tests

- Acceptance Tests: Verify that software meets customer requirements
- Can be used to guide development
- Once acceptance tests passed, software is deliverable
- These tests become part of code base, so built-in protection against refactoring or feature regression
  - Recall: breaking an old feature when adding new...
Types of Unit Tests

• Performance Tests: Just what it sounds like
  ◦ Does code meet speed requirements?
  ◦ Does code meet memory requirements?
  ◦ Does code meet power consumption requirements?

• Typically have an idea where problems will occur, but can’t be sure without testing
Types of Unit Tests

- Performance Tests
- Can’t know whether optimizations are working unless you measure after implementing
- Instrumentation: instrument code to provide relevant measures
  - Also detect errors, log program execution
Intrumentation

- Often part of customer requirements
  - E.g., procedure must execute in under 100ms and/or use less than 1MB of memory
  - By making this part of code, can automate checks as further optimizations are implemented
Before a Working Example…

- Some C++ concepts that we’ll need for this example
  - Function objects
  - Lambda expressions
Function Objects

- One can make user-defined types callable or invocable
  - Done by overloading the function-call operator `operator()`

- Such a type is called a *function type*
  - Instances of a function type are *function objects*

- The function-call operator permits any combination of argument types, return types, and modifiers (except `static`)
Function Objects

Why would you want to do this?

- Might need to interoperate with code that expects function objects
  - Many libraries, including `stdlib` use the function call operator as interface to function-like objects (we’ll see one later)
  - Ex. Creating asynchronous task with `std::async` function, which accepts arbitrary function object that can execute on a separate thread
Function Objects

- Why would you want to do this?
  - The designers of `std::async` could have required coder to expose a run method
  - But function call operator allows generic code to use identical notation to invoke a function or a function-object
#include <cstdint>
#include <cstring>

struct CountIf {
    CountIf(char x)
        : x(x) {}

    size_t operator()(const char* str) const {
        size_t index{}, result{};
        while (str[index]) {
            if (str[index] == x)
                result++;
            index++;
        }
        return result;
    }

    private:
    const char x;
};

int main() {  

    CountIf s_counter{ 's' };  
    auto sally = s_counter("Sally sells seashells by the seashore.");
    printf("Sally: %zd\n", sally);
    auto sailor = s_counter("Sailor went to sea to see what he could see.");
    printf("Sailor: %zd\n", sailor);
    auto buffalo = CountIf{ 'f' }("Buffalo buffalo Buffalo buffalo " "buffalo buffalo Buffalo buffalo.");
    printf("Buffalo: %zd\n", buffalo);
}
```c
#include <cstdint>
#include <stdio>

struct CountIf {
  CountIf(char x)
    : x(x) {} 
    size_t operator()(const char* str) const {
      size_t index{}, result{};
      while(str[index]) {
        if(str[index] == x)
          result++;
        index++;
      }
      return result;
    }

private:
  const char x;
};

int main() {
  CountIf s_counter{ 's' };
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Sally: %zd\n", sally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
  auto buffalo = CountIf{ 'f' }("Buffalo buffalo Buffalo buffalo "
                   "buffalo buffalo Buffalo buffalo.");
  printf("Buffalo: %zd\n", buffalo);
}
```

Output:

Sally: 7
Sailor: 3
Buffalo: 16
Lambda Expressions

- **Lambda expressions** construct unnamed function objects succinctly
  - The function object implies the function type
    - Quick way to create a function object
- Can’t do anything a plain old function declaration can’t do
  - But in specific contexts can be very convenient
    - Declaring function objects can be verbose. Lambda expressions much more succinct
Lambda Expressions: Usage

- Five components
  - **captures**: member variables of the function object
  - **parameters**: arguments required to invoke function object
  - **body**: function object’s code
  - **specifiers**: E.g., constexpr, noexcept
  - **return type**: just what you think
Lambda Expressions: Usage

• Syntax:

  [captures] (parameters) modifiers \(\rightarrow\) return type \{ body \}

• Only capture and body required
  ✷ So everything else is optional

• Each lambda component has direct analogue to part of function object…
```cpp
struct CountIf {
  CountIf(char x)
  : x{ x } {}
  size_t operator()(const char* str) const {
    --snip--
  }

  private:
  const char x;
};
```
Lambda Expressions: Usage

```cpp
struct CountIf {
    CountIf(char x) :
        x{ x } {}
    size_t operator() (const char* str) const {
        --snip--
    }

private:
    const char x;
};
```

- **capture**: The variable `x` is captured by reference.
- **parameters**: The parameter `const char* str` is passed to the function.
- **return type**: The return type of the function is `size_t`.
- **body**: The body of the function contains some code that is not shown.
- **specifiers**: The specifier `const` is used for the parameter and the return type.
**Lambda Parameters and Bodies**

- Lambda expressions produce function objects, and thus are callable
  - You’ll often want the function object to accept parameters upon invocation
- Lambda expression body is just like a function body – all parameters have function scope
- Declare lambda parameters and bodies using essentially same syntax as for functions
Lambda Parameters and Bodies

• Example:

```c
[](int x){return x*x;}
```

• This lambda takes a single `int x` and uses it in the body to perform squaring
#include <cassert>
#include <cstddef>

// Lambda Example

template <typename Fn>
void transform(Fn fn, const int* in, int* out, size_t length) {
    for(size_t i{}; i < length; i++) {
        out[i] = fn(in[i]);
    }
}

int main() {
    constexpr size_t len{ 3 };
    int base[]{ 1, 2, 3 }, a[len], b[len], c[len];
    transform([](int x) { return 1; }, base, a, len);
    transform([](int x) { return x; }, base, b, len);
    transform([](int x) { return 10 * x + 5; }, base, c, len);
    for(size_t i{}; i < len; i++) {
        printf("Element %zd: %d %d %d\n", i, a[i], b[i], c[i]);
    }
}
Don’t be fooled. No different than `typename T`
Don’t be fooled. No different than `typename T`

Except you better provide a type that can be invoked, because of how it’s used.
```cpp
#include <cstddef>
#include <cstdio>

template <typename Fn>
void transform(Fn fn, const int* in, int* out, size_t length) {
    for(size_t i{}; i < length; i++) {
        out[i] = fn(in[i]);
    }
}

int main() {
    const size_t len{ 3 };  
    int base[] { 1, 2, 3 }, a[len], b[len], c[len];
    transform([](int x) { return 1; }, base, a, len);
    transform([](int x) { return x; }, base, b, len);
    transform([](int x) { return 10 * x + 5; }, base, c, len);
    for(size_t i{}; i < len; i++) {
        printf("Element %zd: %d %d %d\n", i, a[i], b[i], c[i]);
    }
}
```

**Output:**

```
Element 0: 1 1 15
Element 1: 1 2 25
Element 2: 1 3 35
```
#include <cstdint>
#include <cstdio>

template <typename Fn>
void transform(Fn fn, const int* in, int* out, size_t length) {
    for(size_t i{}; i < length; i++) {
        out[i] = fn(in[i]);
    }
}

int main() {
    constexpr size_t len{ 3 };
    int base[]{ 1, 2, 3 }, a[len], b[len], c[len];
    transform([](int x) { return 1; }, base, a, len);
    transform([](int x) { return x; }, base, b, len);
    transform([](int x) { return 10 * x + 5; }, base, c, len);
    for(size_t i{}; i < len; i++) {
        printf("Element %zd: %d %d %d\n", i, a[i], b[i], c[i]);
    }
}

**Output:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>35</td>
</tr>
</tbody>
</table>

Note that by declaring `transform` as a template function, you can reuse it with any function object.
Generic Lambdas

- **Generic lambdas** are lambda expression templates
  - For one or more parameter one specifies `auto` rather than a concrete type
  - The `auto` types becomes template parameters
    - Compiler will build a custom instantiation of the lambda
#include <cstddef>
#include <cstdio>

template<typename Fn, typename T>
void transform(Fn fn, const T* in, T* out, size_t len) {
    for(size_t i{0}; i < len; i++) {
        out[i] = fn(in[i]);
    }
}

int main() {
    constexpr size_t l{3};
    int base_int[] = {1, 2, 3}, a[l];
    float base_float[] = {10.f, 20.f, 30.f}, b[l];
    auto translate = [] (auto x) { return 10 * x + 5; };
    transform(translate, base_int, a, l);
    transform(translate, base_float, b, l);

    for(size_t i{0}; i < l; i++) {
        printf("Element %zd: %d %f\n", i, a[i], b[i]);
    }
}
#include <cstddef>
#include <cstdio>

template <typename Fn, typename T>
void transform(Fn fn, const T* in, T* out, size_t len) {
    for(size_t i{0}; i < len; i++) {
        out[i] = fn(in[i]);
    }
}

int main() {
    constexpr size_t l{ 3 };
    int base_int[]{ 1, 2, 3 }, a[l];
    float base_float[]{ 10.f, 20.f, 30.f }, b[l];
    auto translate = [](auto x) { return 10 * x + 5; };
    transform(translate, base_int, a, l);
    transform(translate, base_float, b, l);

    for(size_t i{0}; i < l; i++) {
        printf("Element %zd: %d %f\n", i, a[i], b[i]);
    }
}
#include <cstdint>
#include <cstdio>

template <typename Fn, typename T>
void transform(Fn fn, const T* in, T* out, size_t len) {
    for(size_t i{}; i < len; i++) {
        out[i] = fn(in[i]);
    }
}

int main() {
    constexpr size_t l{3};
    constexpr int base_int[]{1, 2, 3}, a[l];
    float base_float[]{10.f, 20.f, 30.f}, b[l];
    auto translate = [](auto x) { return 10 * x + 5; };
    transform(translate, base_int, a, l);
    transform(translate, base_float, b, l);

    for(size_t i{}; i < l; i++) {
        printf("Element %zd: %d %f
", i, a[i], b[i]);
    }
}

Output:
Element 0: 15 105.000000
Element 1: 25 205.000000
Element 2: 35 305.000000
Lambda Captures

- Lambda captures inject objects into the lambda
  - This can be used to modify behavior of the lambda
  - Declared within brackets \[ \]
  - Capture list before parameter list
  - Can contain any number of comma separated values
    - Which can then be used within lambda’s body
  - Can capture by reference or value
Lambda Captures

#include <cstdint>
#include <cstdio>

int main() {
    char to_count{ 's' };
    auto s_counter = [to_count](const char* str) {
        size_t index{}, result{0};
        while(str[index]) {
            if(str[index] == to_count)
                result++;
            index++;
        }
        return result;
    };
    auto sally = s_counter("Sally sells seashells by the seashore.");
    printf("Sally: %zd\n", sally);
    auto sailor = s_counter("Sailor went to sea to see what he could see.");
    printf("Sailor: %zd\n", sailor);
}

lambda version of `CountIf`

to_count captured and can now be used within lambda’s body
#include <cstddef>
#include <cstdio>

int main() {
    char to_count{ 's' };  
    auto s_counter = [to_count](const char* str) {
        size_t index{}, result{};
        while(str[index]) {
            if(str[index] == to_count) 
                result++;
            index++;
        }
        return result;
    };
    auto sally = s_counter("Sally sells seashells by the seashore.");
    printf("Sally: %zd\n", sally);
    auto sailor = s_counter("Sailor went to sea to see what he could see.");
    printf("Sailor: %zd\n", sailor);
}

Output:

Sally: 7
Sailor: 3
Lambda Captures

```c
#include <cstdint>
#include <stdio>

int main() {
    char to_count{ 's' };
    size_t tally{};
    auto s_counter = [to_count, &tally](const char* str) {
        size_t index{}, result{};
        while (str[index]) {
            if (str[index] == to_count)
                result++;
            index++;
        }
        tally += result;
        return result;
    };
    printf("Tally: %zd\n", tally);
    auto sally = s_counter("Sally sells seashells by the seashore.");
    printf("Sally: %zd\n", sally);
    printf("Tally: %zd\n", tally);
    auto sailor = s_counter("Sailor went to sea to see what he could see.");
    printf("Sailor: %zd\n", sailor);
    printf("Tally: %zd\n", tally);
}
#include <cstdint>
#include <stdio.h>

int main() {
    char to_count{ 's' }; // Logically incorrect, should be 'S'
    size_t tally{}
    auto s_counter = [to_count, &tally](const char* str) {
        size_t index{}, result{}
        while(str[index]) {
            if(str[index] == to_count)
                result++;
            index++;
        }
        tally += result;
        return result;
    };
    printf("Tally: %zd\n", tally);
    auto sally = s_counter("Sally sells seashells by the seashore.");
    printf("Sally: %zd\n", sally);
    printf("Tally: %zd\n", tally);
    auto sailor = s_counter("Sailor went to sea to see what he could see.");
    printf("Sailor: %zd\n", sailor);
    printf("Tally: %zd\n", tally);
    return 0;
}
The setting: programming for autonomous vehicle

- Code is complex and large (hundreds of thousands of lines of code)
- Entire solution consists of several binaries
- Deployment requires uploading to car, a time-consuming process
  - Changing code, compiling, uploading, and executing takes several hours per iteration
Working Example

• Entire software development broken into teams
  ◆ Each team responsible for a *service*
    ▪ E.g., steering wheel control, audio/video, vehicle detection
  ◆ Services interact via a service bus
    ▪ Each service publishes to bus
    ▪ Services *subscribe* to other services as needed
    ▪ *Service bus architecture*
• Your team: autonomous braking service
  - Service must determine whether collision is about to happen, and if so, tell car to brake
  - Service subscribes to two events:
    - SpeedUpdate class: informs that the car speed has changed
    - CarDetected class: informs that another car detected in front of you
• Your team: autonomous braking service
  - Service must determine whether collision is about to happen, and if so, tell car to brake
  - Service publishes BrakeCommand to service bus when imminent collision detected
struct SpeedUpdate {
    double velocity_mps;
};

struct CarDetected {
    double distance_m;
    double velocity_mps;
};

struct BrakeCommand {
    double time_to_collision_s;
};
• You publish BrakeCommand using ServiceBus object that has a publish method

```c
struct ServiceBus {
    void publish(const BrakeCommand&);
    --snip--
};
```
The Plan:

- Expose observe method that subscribes to SpeedUpdate and CarDetected events on the service bus

- Build AutoBrake class that keeps a reference to publish method of the service bus
  - Say what? Reference to a method?
The Plan:

```cpp
template <typename T>
struct AutoBrake {
    AutoBrake(const T& publish);
    void observe (const SpeedUpdate&);
    void observe (const CarDetected&);
private:
    const T& publish
    --snip--
};
```
Working Example

Note that the arrow from Auto Braking Service to the Service bus does not mean that SpeedUpdate information is being sent to the Service bus, but rather that Automatic braking service relies on SpeedUpdate information from the Service bus.

Figure 10-1: A high-level depiction of the interaction between services and the service bus
• Service integrates into car software like:

```cpp
--snip--
int main() {
    ServiceBus bus;
    AutoBrake auto_brake{ [&bus](const auto& cmd) {
        bus.publish(cmd);
    }
    }
    while (true) { // Service bus's event loop
        auto_brake.observe(SpeedUpdate{ 10L });
        auto_brake.observe(CarDetected{ 250L, 25L });
    }
}
```
Working Example

```cpp
int main() {
    ServiceBus bus;
    AutoBrake auto_brake{ [&bus](const auto& cmd) {
        bus.publish(cmd);
    };
    while (true) { // Service bus's event loop
        auto_brake.observe(SpeedUpdate{ 10L });
        auto_brake.observe(CarDetected{ 250L, 25L });
    }
}
```

Note `auto_brake` constructed with a lambda that captures a reference to a `ServiceBus`. Details of how `auto_brake` decides to brake is hidden from other teams.
Working Example

- Service bus mediates all interservice communication
- Code passes any commands from the AutoBrake **directly to the** ServiceBus
  - **Within event loop, a** ServiceBus **can pass** SpeedUpdate **and** CarDetected **objects to** the observe method on your **auto_brake**
Implementing AutoBrake

- Conceptually simple: iterate among writing code, compiling production binary, uploading to car, and testing functionality manually
- Problems: will likely cause a lot of code (and potentially car) crashes
Implementing AutoBrake

- Better: write code, compile a unit-test binary, run on desktop environment
- This way, can iterate among steps quickly
  - Once reasonably confident code works, do manual test with live car
- Unit-test binary will be console app targeting desktop OS
  - In binary, run suite of tests that pass specific inputs to an AutoBrake and assert it produces expected results
Requirements

- **AutoBrake** will consider car’s initial speed zero
- **AutoBrake** should have configurable sensitivity threshold based on number of seconds to impact
  - Must not be less than 1 second. Default is 5 secs
- **AutoBrake** must save car’s speed in between SpeedUpdate observations
- Each time **AutoBrake** observes CarDetected event, it must publish BrakeCommand if impact forecasted to occur in less that sens. thresh
Test-Driven Development (TDD)

- We’ll try implementing auto braking service using TDD
- So, the idea: if you’re going to be coding unit tests anyway, why not code them first?
- TDD or Not TDD: Something of a religious war
  - Like vim vs emacs, where prens go, big endian vs little endian
TDD Advantages

- Key notion: write the code that tests a requirement *before* implementing solution
- Proponents claim:
  - Code is more modular, robust, clean, and well designed
- Good tests are excellent documentation
- Good test suite is a working set of examples that prevents regression
TDD Advantages

- Key notion: write the code that tests a requirement *before* implementing solution
- Great way to submit bug reports
  - Found by failed unit test
  - Once fixed, stays fixed, because test and code that fixes bug becomes part of the test suite
TDD: Red–Green–Refactor

• Red: First implement a failing test
  Why? Make sure you’re actually testing something!

• Green: Implement code that makes the test pass (no more, no less)

• Refactor: restructure existing code without changing functionality
  E.g., replace code with library, rewrite for performance, elegance
  If it breaks, test suite will tell you
• Need a *skeleton class* that implements interface without functionality
  
  - Useful in TDD: you can’t write a test without a shell of class you’re testing
struct SpeedUpdate {
    double velocity_mps;
};

struct CarDetected {
    double distance_m;
    double velocity_mps;
};

struct BrakeCommand {
    double time_to_collision_s;
};

template <typename T>
struct AutoBrake {
    AutoBrake(const T& publish) : publish{ publish } { }
    void observe (const SpeedUpdate& cd) { }
    void observe (const CarDetected& cd) { }
    void set_collision_threshold_s(double x) {
        collision_threshold_s = x;
    }
    double get_collision_threshold_s() const {
        return collision_threshold_s;
    }
    double get_speed_mps() const {
        return speed_mps;
    }
private:
    double collision_threshold_s;
    double speed_mps;
    const T& publish;
};
Template type allows for programming generically against any type that supports invocation with a BrakeCommand.

Skeleton class has no instructions in the body of methods. Because return type here is `void`, don’t even need return statements.
Note setter and getter methods for collision threshold.
Necessary to enforce the class invariant for collision threshold.
Note only a getter for speed.
Assertions

- Essential element of a unit test
- An assertion tests that some condition is met
  - If not met, test fails

```cpp
#include <stdexcept>
constexpr void assert_that(bool statement, const char* message) {
    if(!statement) throw std::runtime_error{ message };
}

int main() {
    assert_that(1 + 2 > 2, "Something is profoundly wrong with the universe.");
    assert_that(24 == 42, "This assertion will generate an exception!");
}
```

What does constexpr mean?
Assertions

• Essential element of a unit test
• An assertion tests that some condition is met
  ♦ If not met, test fails

```cpp
#include <stdexcept>
constexpr void assert_that(bool statement, const char* message) {
    if(!statement) throw std::runtime_error{ message };}

int main() {
    assert_that(1 + 2 > 2, "Something is profoundly wrong with the universe.");
    assert_that(24 == 42, "This assertion will generate an exception!");
}
```
What does constexpr mean? It instructs the compiler to evaluate the expression at compile time, if possible,
constexpr

What does constexpr mean? It instructs the compiler to evaluate the expression at compile time, if possible.
#include <cstdio>

constexpr int isqrt(int n) {
    int i=1;
    while (i*i<n) ++i;
    return i-(i*i!=n);
}

int main() {
    constexpr int x = isqrt(1764);  //
    printf("%d", x);
}

Advantage: Significant impact on readability. Also potential significant improvement in runtime performance.
Assertions

- Essential element of a unit test
- An assertion tests that some condition is met
  - If not met, test fails

```cpp
#include <stdexcept>
constexpr void assert_that(bool statement, const char* message) {
    if (!statement) throw std::runtime_error{ message }; 
}

int main() {
    assert_that(1 + 2 > 2, "Something is profoundly wrong with the universe.");
    assert_that(24 == 42, "This assertion will generate an exception!");
}
```

```
libc++abi.dylib: terminating with uncaught exception of type std::runtime_error: This assertion will generate an exception!
Abort trap: 6
```
Requirement: Initial Speed 0

- Want to test that initial speed is 0
- Write a function that creates an `AutoBrake`, “exercises” the class, and makes an assertion about result

```c++
void initial_speed_is_zero() {
    AutoBrake auto_brake{ [](const BrakeCommand&) {} };
    assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0");
}
```
• **Construct** AutoBrake with empty BrakeCommand publish function
  
  - This unit test not concerned with publishing, so give it simplest argument that will compile
  - When you don’t care about a dependence, can just implement a *stub*: empty implementation that performs some innocuous task

```c
void initial_speed_is_zero() {
    AutoBrake auto_brake{ [](const BrakeCommand&) {} };
    assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0");
}
```
Test Harness

- Test harness: code that executes unit tests
- Idea: create code that invokes unit tests, but handles failed assertions gracefully
  - E.g., doesn’t crash on failed test(s)
Test Harness

- Test harness: code that executes unit tests

```cpp
#include <stdio>
#include <exception>

---snip---

void run_test(void (*unit_test)(), const char* name) {
    try {
        unit_test();
        printf("[+] Test %s successful.\n", name);
    } catch (const std::exception& e) {
        printf("[-] Test failure in %s. %s.\n", name, e.what());
    }
}
```
Test Harness

- What is with the strange declaration of `run_test`?

```c
#include <cstdio>
#include <exception>

--snip--

void run_test(void (*unit_test)(), const char* name) {
    try {
        unit_test();
        printf("[+] Test %s successful.\n", name);
    } catch (const std::exception& e) {
        printf("[-] Test failure in %s. %s.\n", name, e.what());
    }
}
```
Recall: Function Pointers

• Declaring a function pointer is similar to declaring a function

```c
#include <stdio.h>
void my_int_func(int x) {
    printf("%d\n", x);
}

int main() {
    void (*foo)(int);
    /* the ampersand is actually optional */
    foo = &my_int_func;

    /* call my_int_func (note that you do not need to write (*foo)(2)) */
    foo(2);
    /* but if you want to, you may */
    (*foo)(2);

    return 0;
}
```

Thanks Alex Allain:
https://www.cprogramming.com/tutorial/function-pointers.html
Recall: Function Pointers

• Declaring a function pointer is similar to declaring a function

```c
#include <stdio.h>

double my_other_example(int a, int b, char* c) {
    return 0;
}

int main() {
    double (*my_func_ptr)(int, int, char*);
    my_func_ptr = my_other_example;

    return 0;
}
```
• Test harness: code that executes unit tests

```c
#include <stdio>
#include <exception>

--snip--

void run_test(void (*unit_test)(), const char* name) {
    try {
        unit_test();
        printf("[+] Test %s successful.\n", name);
    }
    catch (const std::exception& e) {
        printf("[-] Test failure in %s. %s.\n", name, e.what());
    }
}
```
Test Harness

• To make a *unit-test program* that will run all of the unit tests, place `run_test` inside the main function of a new program...
template <typename T>
struct AutoBrake {
    AutoBrake(const T& publish) : publish(publish) {} {
        void observe(const SpeedUpdate& cd) {}{
        void observe(const CarDetected& cd) {}{
        void set_collision_threshold_s(double x) {
            collision_threshold_s = x;
        }{
        double get_collision_threshold_s() const {
            return collision_threshold_s;
        }{
        double get_speed_mps() const {
            return speed_mps;
        }{
    private:
        double collision_threshold_s;
        double speed_mps = 1;
        const T& publish;
    };

    constexpr void assert_that(bool statement, const char* message) { {
        if(!statement) throw std::runtime_error{ message };
    }{
    void initial_speed_is_zero() {
        AutoBrake auto_brake{ [](const BrakeCommand& e) {} };
        assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0");
    }{
    void run_test(void (*unit_test)(), const char* name) { {
        try {
            unit_test();
            printf("[+] Test %s successful.\n", name);
        } catch (const std::exception& e) { {
            printf("[-] Test failure in %s. %s.\n", name, e.what());
        }
    }{
    int main() {
        run_test(initial_speed_is_zero, "initial speed is 0");
    }
template <typename T>
struct AutoBrake {
    AutoBrake(const T& publish) : publish(publish) {} 
    void observe (const SpeedUpdate& cd) {} 
    void observe (const CarDetected& cd) {} 
    void set_collision_threshold_s(double x) {
        collision_threshold_s = x;
    }
    double get_collision_threshold_s() const {
        return collision_threshold_s;
    }
    double get_speed_mps() const {
        return speed_mps;
    }
private:
    double collision_threshold_s;
    double speed_mps = 1;
    const T& publish;
};

constexpr void assert_that(bool statement, const char* message) {
    if(!statement) throw std::runtime_error{ message };
}

void initial_speed_is_zero() {
    AutoBrake auto_brake{ [](const BrakeCommand&) {} };
    assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0");
}

void run_test(void (*unit_test)(), const char* name) {
    try {
        unit_test();
        printf("[+] Test %s successful.\n", name);
    } catch (const std::exception& e) {
        printf("[-] Test failure in %s. %s.\n", name, e.what());
    }
}

int main() {
    run_test(initial_speed_is_zero, "initial speed is 0");
    printf("[-] Test failure in initial speed is 0. speed not equal 0.\n");
template <typename T>
struct AutoBrake {
    AutoBrake(const T& publish) : publish(publish) {} 
    void observe(const SpeedUpdate& cd) {} 
    void observe(const CarDetected& cd) {} 
    void set_collision_threshold_s(double x) {
        collision_threshold_s = x;
    }
    double get_collision_threshold_s() const {
        return collision_threshold_s;
    }
    double get_speed_mps() const {
        return speed_mps;
    }

private:
    double collision_threshold_s;
    double speed_mps;
    const T& publish;
};

const expr void assert_that(bool statement, const char* message) {
    if (!statement) throw std::runtime_error{ message };
}

void initial_speed_is_zero() {
    AutoBrake auto_brake{ [] (const BrakeCommand&) {} };
    assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0");
}

void run_test(void (*unit_test)(), const char* name) {
    try {
        unit_test();
        printf("[+] Test %s successful.\n", name);
    } catch (const std::exception& e) {
        printf("[-] Test failure in %s. %s.\n", name, e.what());
    }
}

int main() {
    run_test(initial_speed_is_zero, "initial speed is 0");
}
template<typename T>
struct AutoBrake {
    AutoBrake(const T& publish) : publish(publish) {}  
    void observe(const SpeedUpdate& cd) {}  
    void observe(const CarDetected& cd) {}  
    void set_collision_threshold_s(double x) {
        collision_threshold_s = x;
    }  
    double get_collision_threshold_s() const {
        return collision_threshold_s;
    }  
    double get_speed_mps() const {
        return speed_mps;
    }  
private:
    double collision_threshold_s;
    double speed_mps;
    const T& publish;
};

constexpr void assert_that(bool statement, const char* message) {
    if(!statement) throw std::runtime_error{ message };  
}

void initial_speed_is_zero() {
    AutoBrake auto_brake{ [](const BrakeCommand& cd) {} };  
    assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0");
}

void run_test(void (*unit_test)(), const char* name) {
    try {
    unit_test();
    printf("[+ Test %s successful.\n", name);
    } catch (const std::exception& e) {
    printf("[- Test failure in %s. %s.\n", name, e.what());
    }  
}

int main() {
    run_test(initial_speed_is_zero, "initial speed is 0");
}

[+] Test initial speed is 0 successful.
void initial_sensitivity_is_five() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} };
  assert_that(auto_brake.get_collision_threshold_s() == 5L,
              "sensitivity is not 5");
}

void run_test(void (*unit_test)(), const char* name) {
  try {
    unit_test();
    printf("[+] Test %s successful.\n", name);
  } catch (const std::exception& e) {
    printf("[-] Test failure in %s. %s.\n", name, e.what());
  }
}

int main() {
  run_test(initial_speed_is_zero, "initial speed is 0");
  run_test(initial_sensitivity_is_five, "initial sensitivity is 5");
}
Requirement: Sensitivity Must Always be Greater Than 1

```c++
void sensitivity_greater_than_1() {
    AutoBrake auto_brake{ [] (const BrakeCommand&){} };
    try {
        auto_brake.set_collision_threshold_s(0.5L);
    } catch (const std::exception&) {
        return;
    }
    assert_that(false, "no exception thrown");
}

void run_test(void (*unit_test)(), const char* name) {
    try {
        unit_test();
        printf("[+] Test %s successful.\n", name);
    } catch (const std::exception& e) {
        printf("[-] Test failure in %s. %s.\n", name, e.what());
    }
}

int main() {
    run_test(initial_speed_is_zero, "initial speed is 0");
    run_test(initial_sensitivity_is_five, "initial sensitivity is 5");
    run_test(sensitivity_greater_than_1, "sensitivity greater than 1");
}
void sensitivity_greater_than_1() {
    AutoBrake auto_brake{ [](const BrakeCommand&){} }; 
    try {
        auto_brake.set_collision_threshold_s(0.5L);
    } catch (const std::exception&) {
        return;
    }
    assert_that(false, "no exception thrown");
}

void run_test(void (*unit_test)(), const char* name) {
    try {
        unit_test();
        printf("[+] Test %s successful.\n", name);
    } catch (const std::exception& e) {
        printf("[+] Test failure in %s. %s.\n", name, e.what());
    }
}

int main() {
    run_test(initial_speed_is_zero, "initial speed is 0");
    run_test(initial_sensitivity_is_five, "initial sensitivity is 5");
    run_test(sensitivity_greater_than_1, "sensitivity greater than 1");
}

[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[-] Test failure in sensitivity greater than 1. no exception thrown.
Requirement: Sensitivity Must Always be Greater Than 1

template<typename T>
struct AutoBrake {
    AutoBrake(const T& publish) : collision_threshold_s {5},
        speed_mps{},
        publish{ publish } {}

    void observe (const SpeedUpdate& cd) {}
    void observe (const CarDetected& cd) {}
    void set_collision_threshold_s(double x) {
        if (x < 1) throw std::invalid_argument{ "Collision less than 1." };
        collision_threshold_s = x;
    }
    double get_collision_threshold_s() const {
        return collision_threshold_s;
    }
    double get_speed_mps() const {
        return speed_mps;
    }
private:
    double collision_threshold_s;
    double speed_mps;
    const T& publish;
};
Requirement: Sensitivity Must Always be Greater Than 1

```cpp
template <typename T>
struct AutoBrake {
    AutoBrake(const T& publish) :
        collision_threshold_s {5},
        speed_mps{},
        publish{ publish } {}

    void observe (const SpeedUpdate& cd) {} 
    void observe (const CarDetected& cd) {} 
    void set_collision_threshold_s(double x) {
        if (x < 1) throw std::invalid_argument{ "Collision less than 1." }; 
        collision_threshold_s = x;
    }
    double get_collision_threshold_s() const {
        return collision_threshold_s;
    }
    double get_speed_mps() const {
        return speed_mps;
    }
}
private:
    double collision_threshold_s;
    double speed_mps;
    const T& publish;
};
```

[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
Mocking Dependencies

• Mock class (think "mock up"): a special implementation that you generate for the purpose of testing a class that depends on the mock
  - That is, your class depends, say, on class $\text{foo}$. But you may not have the full $\text{foo}$ implementation (perhaps it isn’t even coded yet)
  - Use the mock to test interactions with your class
Aside: std::function

- `std::function` from `<functional>` header is a polymorphic container for callable objects.
- In other words, a generic function pointer.
  - You can store a static function, a function object, or a lambda into a `std::function`.
Declaring a function

- To declare a function you must provide a single template parameter containing the function prototype of the callable object

```
std::function<return-type(arg-type-1, arg-type-2, etc.>>
```

- `std::function` class template has many constructors
  - Default constructor constructs a `std::function` in empty mode – it contains no callable object
Empty Functions

- If you declare a `std::function` with no contained object, “calling it” will throw a `std::bad_function_call` exception.

```cpp
#include <cstdio>
#include <functional>

int main() {
    std::function<void()> func;
    try {
        func();
    } catch(const std::bad_function_call& e) {
        printf("Exception: %s", e.what());
    }
}
```

Exception: std::exception
Assigning a Callable Object to a Function

• Two ways: use the constructor or use the assignment operator of `function`

```c++
#include <cstdio>
#include <functional>

void static_func() {
    printf("A static function.\n");
}

int main() {
    std::function<void()> func{ [] { printf("A lambda.\n"); } };
    func();
    func = static_func;
    func();
}
```

```plaintext
A lambda.
A static function.
```
Example

• You can construct a function with any callable object that supports the function semantics implied by the template parameter of the function
```
#include <cstdint>
#include <cstdlib>
#include <functional>

struct CountIf {
    --snip--
};

size_t count_spaces(const char* str) {
    size_t index{}, result{};
    while(str[index]) {
        if(str[index] == ' ')
            result++;
        index++;
    }
    return result;
}

std::function<size_t(const char*)> funcs[]{
    count_spaces,
    CountIf{ 'e' },
    [](const char* str) {
        size_t index{};
        while(str[index])
            index++;
        return index;
    }
};

auto text = "Sailor went to sea to see what he could see.";

int main() {
    size_t index{};
    for(const auto& func : funcs) {
        printf("func %zd: %zd\n", index++, func(text));
    }
}```
#include <cstdint>
#include <stdio.h>
#include <functional>

struct CountIf {
  --snip--
};

size_t count_spaces(const char* str) {
  size_t index{}, result{}
  while(str[index]) {
    if(str[index] == ' ')
      result++;
    index++;
  }
  return result;
}

std::function<size_t(const char*)> funcs[] = {
  count_spaces,
  CountIf('e'),
  [](const char* str) {
    size_t index{}
    while(str[index])
      index++;
    return index;
  }
};

auto text = "Sailor went to sea to see what he could see."

int main() {
  size_t index{}
  for(const auto& func : funcs) {
    printf("func #%zd: %zd\n", index++, func(text));
  }
}

  func #0: 9
  func #1: 7
  func #2: 44
Runtime Overhead

- Using a function comes with a runtime overhead cost
  - function might need to make a dynamic allocation to store callable object
  - Compiler has difficulty optimizing away function invocations, so often incur an indirect function call
    - Requires additional pointer dereferences
Indirect Function Call?

- Direct function call: function call is made with a fixed address in instruction
  - For those in CS 301, `jal` to fixed address that has been placed in the executable by the linker
- Indirect function call: function call is made with address of callee in a register
  - Register is previously loaded either with fixed address of function being called, or with a value fetched from somewhere else (e.g., memory or another register) where the function address has been stored
Indirect Function Call?

• Direct function call: will always call the same function

• Indirect function call: can call different functions, depending on what was loaded in register before call is made
  - The indirection requires extra effort
We’re back: Mocking Dependencies

- **AutoBrake** has dependencies:
  - CarDetected
  - SpeedUpdated
  - Generic dependence on a publish object callable with a single `BrakeCommand` parameter

- Suppose you want to refactor the service bus
  - Want to accept a `std::function` to subscribe to each service
Mocking Dependencies

• Suppose you want to refactor the service bus
  ◦ Want to accept a `std::function` to subscribe to each service

```cpp
#include <functional>

using SpeedUpdateCallback = function<void(const SpeedUpdate&)>
using CarDetectedCallback = function<void(const CarDetected&)>

struct IServiceBus {
    virtual ~IServiceBus() = default;
    virtual void publish(const BrakeCommand&) = 0;
    virtual void subscribe(SpeedUpdateCallback) = 0;
    virtual void subscribe(CarDetectedCallback) = 0;
};
```
Mocking Dependencies

- **IServiceBus** is an interface
  - So no need to know implementation details
  - And you can do your own wiring into the service bus

```cpp
#include <functional>

using SpeedUpdateCallback = function<void(const SpeedUpdate&)>;
using CarDetectedCallback = function<void(const CarDetected&)>;

struct IServiceBus {
    virtual ~IServiceBus() = default;
    virtual void publish(const BrakeCommand&) = 0;
    virtual void subscribe(SpeedUpdateCallback) = 0;
    virtual void subscribe(CarDetectedCallback) = 0;
};
```

Recall that **using** keyword is like the C language `typedef`
Mocking Dependencies

• Suppose you want to refactor the service bus
  ✷ Want to accept a `std::function` to subscribe to each service

• A problem: How do you test `AutoBrake` in isolation?
  ✷ Using the real production bus is not testing anymore, but integration
    ▪ And definitely not an easily configurable, isolated unit test
Mocking Dependencies

• But, you don’t depend on implementation: you depend on the interface!

• So create a mock class that implements the IServiceBus interface, and use that within AutoBrake
  • AutoBrake interacts with the mock, not the production service bus
Mocking Dependencies

• AND because you have complete control over the mock and it’s a unit-test-specific class, you can do just about anything you want with it
  ♦ Can record arbitrarily detailed info about how the mock gets called by AutoBrake
    ▪ E.g., number of times the mock is called and with which parameters
  ♦ Can perform arbitrary computation in the mock
Mocking Dependencies

- You have **complete** control over the inputs and the outputs of the dependencies of AutoBrake. E.g., ...
  - How does AutoBrake handle the case where the service bus throws an out-of-memory exception inside of a publish invocation? (You can test that!)
  - How many times did AutoBrake register a callback for SpeedUpdates? (Can test that too!)
#include <functional>

using SpeedUpdateCallback = function<void(const SpeedUpdate&)>;
using CarDetectedCallback = function<void(const CarDetected&)>;

struct IServiceBus {
    virtual ~IServiceBus() = default;
    virtual void publish(const BrakeCommand&) = 0;
    virtual void subscribe(SpeedUpdateCallback callback) = 0;
    virtual void subscribe(CarDetectedCallback callback) = 0;
};

struct MockServiceBus : IServiceBus {
    void publish(const BrakeCommand& cmd) override {
        commands_published++;
        last_command = cmd;
    }
    void subscribe(SpeedUpdateCallback callback) override {
        speed_update_callback = callback;
    }
    void subscribe(CarDetectedCallback callback) override {
        car_detected_callback = callback;
    }
    BrakeCommand last_command{};
    int commands_published{};
    SpeedUpdateCallback speed_update_callback{};
    CarDetectedCallback car_detected_callback{};
};
```cpp
#include <functional>

using SpeedUpdateCallback = function<void(const SpeedUpdate&)>;
using CarDetectedCallback = function<void(const CarDetected&)>;

struct IServiceBus {
    virtual ~IServiceBus() = default;
    virtual void publish(const BrakeCommand&) = 0;
    virtual void subscribe(SpeedUpdateCallback callback) = 0;
    virtual void subscribe(CarDetectedCallback callback) = 0;
};

struct MockServiceBus : IServiceBus {
    void publish(const BrakeCommand& cmd) override {
        commands_published++;
        last_command = cmd;
    }
    void subscribe(SpeedUpdateCallback callback) override {
        speed_update_callback = callback;
    }
    void subscribe(CarDetectedCallback callback) override {
        car_detected_callback = callback;
    }
    BrakeCommand last_command{}
    int commands_published{}
    SpeedUpdateCallback speed_update_callback{}
    CarDetectedCallback car_detected_callback{}
};
```

This publish records the number of time publish was called and the last command that was published.
One Note:

- Mocks are very useful, but for this example, if you refactor the service bus, you’ll have to refactor your unit tests as well
  - No way around that, unless the interface to the service bus doesn’t change
Unit Testing and Mocking Frameworks

- Unit-testing frameworks make unit testing easier, just as IDEs can help make coding easier
  - Provide commonly used functions and the scaffolding necessary to tie tests into a user-friendly program
  - Functionality to help create concise, expressive tests
The Catch Unit-Testing Framework

- Catch Unit Testing Framework: One of three described in your text
- Very straightforward
- Written by Phil Nash
- Available at https://github.com/catchorg/Catch2/
- Header only library
  ▶ So you can download the single-header version and #include in each unit-testing translation unit
Catch

- Easiest way to use this
  - Download single `catch.hpp` header file
    - [https://raw.githubusercontent.com/catchorg/Catch2/v2.x/single_include/catch2/catch.hpp](https://raw.githubusercontent.com/catchorg/Catch2/v2.x/single_include/catch2/catch.hpp)
  - Put it in your project directory
  - Be sure to `#include` it in unit test code
• **Defining an entry point**
  - Provide your test binary’s entry point with `#define CATCH_CONFIG_MAIN`
  - That’s it: Within the `catch.hpp` header file, it looks for `CATCH_CONFIG_MAIN` preprocessor definition
  - When found, Catch will add a `main` function (so you don’t have to)
  - Automatically grabs all unit tests you have defined and wraps them in a test harness
Catch

• Building: just build the executable as usual

  ✤ E.g., if the code below is listing_10_30.cpp, then just make listing_10_30

    #define CATCH_CONFIG_MAIN
    #include "catch.hpp"

    TEST_CASE("AutoBrake") {
      // unit test here
    }

  ✤ Note it has no main method
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("AutoBrake") {
    // unit test here
}

• Running listing_10_30 gives

test cases: 1 | 1 passed
assertions: - none -
Recall...

- Earlier, we defined separate functions for each unit test
- Passed a pointer to each function as the first parameter to `run_test`
- Passed name of the test as the second parameter
  - Which is redundant if you named unit test function well
- Implemented an `assert` function for each unit test
Catch

- Catch does all of that implicitly
- For each unit test, use `TEST_CASE` macro and Catch does all of the integration for you

```cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("AutoBrake") {
  // unit test here
}
```
The Catch entry point here detects that one unit test called “AutoBrake” has been declared.

It also provides a warning that we have not made any assertions

```cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("AutoBrake") {
    // unit test here
}
```

---

test cases: 1 | 1 passed
assertions: - none -
Catch: Making Assertions

• Catch comes with a built-in assertion, with two distinct families of macros
  - **REQUIRE**: will fail a test immediately
  - **CHECK**: will allow test to run to completion, but still cause a failure
    - Useful if a group of related assertions can help lead the programmer toward a bug
  - Also, macros for assertions that should be false
    - **REQUIRE_FALSE**
    - **CHECK_FALSE**
Catch: Making Assertions

- **Usage:** wrap a Boolean expression with **REQUIRE** macro
  - If expression evaluates to false, assertion fails
  - You provide *assertion expression* that evaluates to true if assertion passes, false if it doesn’t
- **Syntax:** `REQUIRE(assertion-expression);`
Example: initial_speed_is_zero

```cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <functional>

struct IServiceBus {
    //snip--
};

struct MockServiceBus : IServiceBus {
    //snip--
};

struct AutoBrake {
    //snip--
};

TEST_CASE("initial car speed is zero") {
    MockServiceBus bus{};
    AutoBrake auto_break{ bus };
    REQUIRE(auto_break.get_speed_mps() == 0);
}
```
TEST_CASE("AutoBrake") {
    MockServiceBus bus{};
    AutoBrake auto_break{ bus }; 

    SECTION("initializes speed to zero") {
        REQUIRE(auto_break.get_speed_mps() == Approx(0));
    }

    SECTION("initializes sensitivity to five") {
        REQUIRE(auto_break.get_collision_threshold_s() == Approx(5));
    }

    SECTION("throws when sensitivity less than one") {
        REQUIRE_THROWS(auto_break.set_collision_threshold_s(0.5L));
    }

    SECTION("saves speed after update") {
        bus.speed_update_callback(SpeedUpdate{ 100L });
        REQUIRE(100L == auto_break.get_speed_mps());
        bus.speed_update_callback(SpeedUpdate{ 50L });
        REQUIRE(50L == auto_break.get_speed_mps());
        bus.speed_update_callback(SpeedUpdate{ 0L });
        REQUIRE(0L == auto_break.get_speed_mps());
    }

    SECTION("no alert when not imminent") {
        auto_break.set_collision_threshold_s(2L);
        bus.speed_update_callback(SpeedUpdate{ 100L });
        bus.car_detected_callback(CarDetected{ 1000L, 50L });
        REQUIRE(bus.commands_published == 0);
    }

    SECTION("alert when imminent") {
        auto_break.set_collision_threshold_s(10L);
        bus.speed_update_callback(SpeedUpdate{ 100L });
        bus.car_detected_callback(CarDetected{ 100L, 0L });
        REQUIRE(bus.commands_published == 1);
        REQUIRE(bus.last_command.time_to_collision_s == Approx(1));
    }
}
TEST_CASE("AutoBrake")
{
  MockServiceBus bus{};
  AutoBrake auto_break{ bus };

  SECTION("initializes speed to zero")
  {
    REQUIRE(auto_break.get_speed_mps() == Approx(0));
  }

  SECTION("initializes sensitivity to five")
  {
    REQUIRE(auto_break.get_collision_threshold_s() == Approx(5));
  }

  SECTION("throws when sensitivity less than one")
  {
    REQUIRE_THROWS(auto_break.set_collision_threshold_s(0.5L));
  }

  SECTION("saves speed after update")
  {
    bus.speed_update_callback(SpeedUpdate{ 100L });
    REQUIRE(100L == auto_break.get_speed_mps());
    bus.speed_update_callback(SpeedUpdate{ 50L });
    REQUIRE(50L == auto_break.get_speed_mps());
    bus.speed_update_callback(SpeedUpdate{ 0L });
    REQUIRE(0L == auto_break.get_speed_mps());
  }

  SECTION("no alert when not imminent")
  {
    auto_break.set_collision_threshold_s(2L);
    bus.speed_update_callback(SpeedUpdate{ 100L });
    bus.car_detected_callback(CarDetected{ 1000L, 50L });
    REQUIRE(bus.commands_published == 0);
  }

  SECTION("alert when imminent")
  {
    auto_break.set_collision_threshold_s(10L);
    bus.speed_update_callback(SpeedUpdate{ 100L });
    bus.car_detected_callback(CarDetected{ 100L, 0L });
    REQUIRE(bus.commands_published == 1);
    REQUIRE(bus.last_command.time_to_collision_s == Approx(1));
  }
}


```c
TEST_CASE("AutoBrake") {  
    MockServiceBus bus{};
    AutoBrake auto_break{ bus };

    SECTION("initializes speed to zero") {  
        REQUIRE(auto_break.get_speed_mps() == Approx(0));
    }

    SECTION("initializes sensitivity to five") {  
        REQUIRE(auto_break.get_collision_threshold_s() == Approx(5));
    }

    SECTION("throws when sensitivity less than one") {  
        REQUIRE_THROWS(auto_break.set_collision_threshold_s(0.5L));
    }

    SECTION("saves speed after update") {  
        bus.speed_update_callback(SpeedUpdate{ 100L });
        REQUIRE(100L == auto_break.get_speed_mps());
        bus.speed_update_callback(SpeedUpdate{ 50L });
        REQUIRE(50L == auto_break.get_speed_mps());
        bus.speed_update_callback(SpeedUpdate{ 0L });
        REQUIRE(0L == auto_break.get_speed_mps());
    }

    SECTION("no alert when not imminent") {  
        auto_break.set_collision_threshold_s(2L);
        bus.speed_update_callback(SpeedUpdate{ 100L });
        bus.car_detected_callback(CarDetected{ 1000L, 50L });
        REQUIRE(bus.commands_published == 0);
    }

    SECTION("alert when imminent") {  
        auto_break.set_collision_threshold_s(10L);
        bus.speed_update_callback(SpeedUpdate{ 100L });
        bus.car_detected_callback(CarDetected{ 100L, 0L });
        REQUIRE(bus.commands_published == 1);
        REQUIRE(bus.last_command.time_to_collision_s == Approx(1));
    }
}
```

How you implement floating point assertions

The code snippet above demonstrates how to implement floating point assertions in C++. The `Approx` function from the `leakage` library is used to check if a floating point value is close to zero or a specific value.
Sample Output

tests is a Catch v2.13.3 host application.
Run with -? for options

AutoBrake
  initializes speed to zero

tests.cpp:82

-----------------------------
tests.cpp:83: FAILED:
  REQUIRE( auto_break.get_speed_mps() == Approx(0) )
with expansion:
    0.0 == Approx( 0.0 )
-----------------------------

test cases: 1 | 1 failed
assertions: 9 | 8 passed | 1 failed
Sample Output

All tests passed (9 assertions in 1 test case)
Testing: Summary

- Unit tests
-Mocks
- Test-driven development
- Assertions
-Mocks
- Unit-testing frameworks