



***Intro to Rust Lang***  
***Modules and***  
***Testing***

# Today: Modules and Testing

- Modules, Packages, and Crates
  - The `use` keyword
  - Module Paths and the File System
- Testing
  - Unit Testing
  - Integration Testing
- Code Review

# Large Programs

As your programs get larger, the organization of the code becomes increasingly important. It is generally good practice to:

- Split code into multiple folders and files
- Group related functionality
- Separate code with distinct features
- *Encapsulate* implementation details
- *Modularize* your program

# Module System

Rust implements a number of organizational features, collectively referred to as the *module system*.

- **Packages:** A Cargo feature that lets you build, test, and share crates
- **Crates:** A tree of modules that produces a library or executable
- **Modules:** Lets you control the organization, scope, and privacy of paths
- **Paths:** A way of naming an item, such as a struct, function, or module

# Packages and Crates

# Crate

A *crate* is the smallest amount of code that the Rust compiler considers at a time.

- The equivalent in C/C++ is a *compilation unit*
- Running `rustc` on a single file also builds a crate
- Crates contain modules
  - Modules can be defined in other files
  - Paths allow modules to refer to other modules

# Crate

There are two types of crates: **binary** crates and **library** crates.

- A binary crate can be compiled to an executable
  - Contains a `main` function
  - Examples include command-line utilities or servers
- A library crate has no `main` function, and does not compile to an executable
  - Defines functionality intended to be shared with multiple projects
- Each crate also has a file referred to as the *crate root*
  - *The Rust compiler looks at this file first, and it is also the root module of the crate (more on modules later!)*

# Package

A package is a bundle of one or more crates.

- A package is defined by a `Cargo.toml` file at the root of your directory
  - `Cargo.toml` describes how to build all of the crates
- A package can contain **any number of** binary crates, but **at most one** library crate



# cargo new

Let's walk through what happens when we create a package with `cargo new`.

```
$ cargo new my-project
    Created binary (application) `my-project` package

$ ls my-project
Cargo.toml
src

$ ls my-project/src
main.rs
```

- Creates a new package called `my-project`
- Creates a `src/main.rs` file that prints `"Hello, world!"`
- Creates a `Cargo.toml` in the root directory

# Cargo.toml

Let's take a look inside the `Cargo.toml`.

```
[package]
name = "my-project"
version = "0.1.0"
edition = "2021"
```

## [dependencies]

- File written in `toml`, a file format for configuration files
- Notice how there is no explicit mention of `src/main.rs`
- Cargo follows the convention that `src/main.rs` is the root of a *binary* crate
- Similarly, a `src/lib.rs` file is the root of a *library* crate

## Example: `cargo`

Cargo is itself a Rust package that ships with installations of Rust!

- Contains the binary crate that compiles to the executable `cargo`
- Contains a library crate that the `cargo` binary depends on

## Aside: Package vs Project vs Program

- "Package" is the only term of these three with a formal definition in Rust
- "Project" is a very overloaded term
  - More meaningful in the context of an *IDE*
- "Program"
  - Ask the mathematicians  $\overline{\setminus( )/}$

# Modules

# Modules

*Modules* let us organize code within a crate for readability and easy reuse.

- Modules are collections of *items*
  - Items are functions, structs, traits, etc.
- Allows us to control the privacy of items
- Mitigates namespace collisions
- Here is a [cheat sheet](#) from the Rust Book!

# Root Module

The root module is in our `main.rs` (for a binary crate) or `lib.rs` (for a library crate).

```
$ cargo new restaurant
```

`src/main.rs`

```
fn main() {  
    println!("Hello, world!");  
}
```

# Declaring Modules

We can declare a new module with the keyword `mod`.

src/main.rs

```
fn main() {  
    println!("Hello, World!");  
}  
  
mod kitchen {  
    // `cook` is defined in the module `kitchen`  
    fn cook() {  
        println!("I'm cooking");  
    }  
}
```



# Using Modules

To use items outside of a module, we must declare them as `pub`.

src/main.rs

```
fn main() {
    kitchen::cook();
}

mod kitchen {
    pub fn cook() { println!("I'm cooking"); }

    // Only items internal to the `kitchen` should be able to access this
    fn examine_ingredients() {}
}
```

- By default, all module items are private in Rust

# Declaring Submodules

We can declare submodules inside of other modules.

src/main.rs

```
fn main() {
    kitchen::stove::cook();
}

mod kitchen {
    pub mod stove {
        pub fn cook() { println!("I'm cooking"); }
    }
}
```

- Submodules also have to be declared as `pub mod` to be accessible
- The module system is a tree, just like a file system

# Modules as Files

In addition to declaring modules *within* files, creating a file named `module_name.rs` declares a corresponding module named `module_name`.

```
src
├── module_name.rs
└── main.rs
```

- Allows us to represent our module structure in the file system
- Let's try moving the `kitchen` module to its own file!

# Modules as Files

src/main.rs

```
mod kitchen; // The compiler will look for `kitchen.rs`  
  
fn main() {  
    kitchen::stove::cook();  
}
```

src/kitchen.rs

```
pub mod stove {  
    pub fn cook() { println!("I'm cooking"); }  
}  
  
fn examine_ingredients() {}
```

- What about moving the `stove` submodule to its own file as well?

# Submodules as Files

We can move the `stove` submodule into a file `src/kitchen/stove.rs` to indicate that `stove` is a submodule of `kitchen`.

`src/kitchen.rs`

```
pub mod stove; // note this still has to be `pub`  
  
fn examine_ingredients() {}
```

`src/kitchen/stove.rs`

```
pub fn cook() {  
    println!("I'm cooking");  
}
```

- `main.rs` is unchanged (omitted for slide real estate)

# Alternate Submodule File Naming

We could also replace `src/kitchen.rs` with `src/kitchen/mod.rs`.

`src/kitchen/mod.rs`

```
pub mod stove;  
  
fn examine_ingredients() {}
```

`src/kitchen/stove.rs`

```
pub fn cook() {  
    println!("I'm cooking");  
}
```

- The only difference is in which file the `kitchen` module is defined

# Alternate Submodule File Naming

In terms of Rust's module system, these two file trees are (essentially) identical.

```
src
├── kitchen
│   └── stove.rs
├── kitchen.rs
└── main.rs
```

```
src
├── kitchen
│   ├── mod.rs
│   └── stove.rs
└── main.rs
```

- This is a stylistic choice that each instructor has a very strong opinion on
  - Ask at your own peril...

# File Structure Comparison: Choice 1

```
src
├── bathroom
│   ├── mod.rs
│   ├── sink.rs
│   └── toilet.rs
├── dining_room
│   ├── guests.rs
│   ├── mod.rs
│   ├── seats.rs
│   └── tables.rs
├── garden
│   ├── dirt.rs
│   ├── mod.rs
│   ├── plants.rs
│   └── water.rs
├── kitchen
│   ├── dish_washer.rs
│   ├── mod.rs
│   ├── oven.rs
│   └── stove.rs
└── lib.rs
```



# File Structure Comparison: Choice 2

```
src
├── bathroom
│   ├── sink.rs
│   └── toilet.rs
├── dining_room
│   ├── guests.rs
│   ├── seats.rs
│   └── tables.rs
├── garden
│   ├── dirt.rs
│   ├── plants.rs
│   └── water.rs
├── kitchen
│   ├── dish_washer.rs
│   ├── oven.rs
│   └── stove.rs
├── bathroom.rs
├── dining_room.rs
├── kitchen.rs
├── garden.rs
└── lib.rs
```

# File Structure Comparison

Consistency with surrounding codebase is *always* most important!

See discussions:

- [Rust Users Forum](#)
- [Rust Internals Forum](#)
- [Reddit](#)

# The Module Tree, Visualized

Even with our file system changes, the module tree stays the same!

```
crate restaurant
├── mod kitchen: pub(crate)
│   ├── fn examine_ingredients: pub(self)
│   └── mod stove: pub
│       └── fn cook: pub
└── fn main: pub(crate)
```

- We can customize our file structure without changing any behavior!

# Module Paths

To use any item in a module, we need to know its *path*, just like a filesystem.

There are two types of paths:

- An *absolute path* is the full path starting from the crate root
- A *relative path* starts from the current module and use `self`, `super`, or an identifier in the current module
- Components of paths are separated by double colons ( `::` )

# Paths for Referring to Modules

You may have noticed a path from the previous sequence:

```
kitchen::stove::cook();
```

This is saying:

- In the module `kitchen`
  - In the submodule `stove`
    - Call the function `cook`
- This is a path relative to the current module (in this case, the root)

# Using Paths

src/main.rs

```
mod kitchen;  
  
fn main() {  
    kitchen::stove::cook();  
}
```

- Not too hard to write...

# Using Verbose Paths

What if we had a deeper module tree?

src/main.rs

```
fn main() {  
    kitchen::stove::stovetop::burner::gas::gasknob::pot::cook();  
    kitchen::stove::stovetop::burner::gas::gasknob::pot::cook();  
    kitchen::stove::stovetop::burner::gas::gasknob::pot::cook();  
}
```

- A lot more verbose...
  - Especially if we need to write this multiple times

# The `use` Keyword

We can bring paths into scope with the `use` keyword.

`src/main.rs`

```
mod kitchen;

use kitchen::stove::stovetop::burner::gas::gasknob::pot;

fn main() {
    pot::cook();
    pot::cook();
    pot::cook();
}
```

- It is idiomatic to `use` up to the *parent* of a function, rather than the function item itself



## More `use` Syntax

We can also import items from the Rust standard library (`std`).

```
use std::collections::HashMap;  
use std::io::Bytes;  
use std::io::Write;
```

- `HashMap` and `Bytes` are structs, and `Write` is a trait
- It is idiomatic to import structs, enums, traits, etc. directly

## More `use` Syntax

We can combine those 2 `std::io` imports into one statement:

```
use std::collections::HashMap;  
use std::io::{Bytes, Write};  
  
use std::io::*; // Also possible!
```

- You could also write `use std::io::*` to bring in everything from the `std::io` module (including `Bytes` and `Write`)
  - Called the "glob operator"
  - Generally not recommended since it clutters the namespace

## Aside: Binary and Library Crate Paths

In the past examples, we were using a binary crate ( `src/main.rs` ). All the same principles apply to using a library crate.

However, if you use *both* a binary *and* a library crate, things are slightly different.

```
src
├── kitchen
│   ├── mod.rs
│   └── stove.rs
├── lib.rs <- What happens when we add this?
└── main.rs
```

## Aside: Binary and Library Crate Paths

```
src
├── kitchen
│   ├── mod.rs
│   └── stove.rs
├── lib.rs
└── main.rs (wants to call functions from lib.rs)
```

Typically when you have both a binary and library crate in the same package, you want to use functions and types defined in `lib.rs` from `main.rs`.

- If you have both a `main.rs` file and a `lib.rs` file, *both* are crate roots
- So how can we get items from a separate module tree?

# Accessing Library from Binary

Let's try to refactor our previous example:

restaurant/src/lib.rs

```
pub mod kitchen; // Now marked `pub`!
```

restaurant/src/main.rs

```
fn main() {  
    ???::kitchen::stove::cook();  
}
```

- All files in `src/kitchen` remain unchanged
- What do we put in `???` ?

# Accessing Library from Binary

We treat our library crate as an *external* crate, with the same name as our package.

restaurant/src/main.rs

```
fn main() {  
    restaurant::kitchen::stove::cook();  
}
```

- Similar to how you would treat `std` as an external crate
- We'll talk about external crates more next week!

# The `super` Keyword

We can also construct relative paths that begin in the parent module with `super`.

```
crate restaurant
├── mod kitchen: pub(crate)
│   ├── fn examine_ingredients: pub(self)
│   └── mod stove: pub
│       └── fn cook: pub
└── fn main: pub(crate)
```

src/kitchen/stove.rs

```
pub fn cook() {
    super::examine_ingredients(); // Make sure you do this before cooking!
    println!("I'm cooking");
}
```

# Privacy

```
mod kitchen: pub(crate)
└─ fn examine_ingredients: pub(self)
└─ mod stove: pub
    └─ fn cook: pub
```

src/kitchen/stove.rs

```
pub fn cook() {
    super::examine_ingredients(); // Make sure you do this before cooking!
    println!("I'm cooking");
}
```

- `examine_ingredients` does not need to be public in this case
- `stove` can access anything in its parent module `kitchen`
  - Note that privacy only applies upwards, not downwards



# Privacy of Types

We can also use `pub` to designate structs and enums as public.

```
pub struct Breakfast {  
    pub toast: String,  
    seasonal_fruit: String,  
}  
  
pub enum Appetizer {  
    Soup,  
    Salad,  
}
```

- We can mark specific fields of structs public, allowing direct access
- If an enum is public, so are its variants!

## Recap: Modules

- You can split a package into crates
  - Crates into modules
    - Modules into items
- You can refer to items defined in other modules with paths
- All module components are private by default, unless you mark them as `pub`

# Testing

# Testing

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

- Edsger W. Dijkstra, *The Humble Programmer*

# Testing

Correctness of a program is complex and not easy to prove.

- Rust's type system helps with this, but it certainly cannot catch everything
- Rust includes a testing framework for this reason!

# What is a Test?

Generally we want to perform at least 3 actions when running a test:

1. Set up needed data or state
2. Run the evaluated code
3. Determine if the results are as expected

# Writing Tests

In Rust, a test is a function annotated with the `#[test]` attribute.

src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

- After running `cargo new adder --lib`, this code will be in `src/lib.rs`

# Writing Tests

Let's break this down.

```
#[test]
fn it_works() {
    let result = 2 + 2;
    assert_eq!(result, 4);
}
```

- The `#[test]` attribute indicates that this is a test function
- We set up the value `result` by adding `2 + 2`
- We use the `assert_eq!` macro to assert that `result` is correct
- We don't need to return anything, since not panicking *is* the test!



# Running Tests

We run tests with `cargo test`.

```
$ cargo test
  Compiling adder v0.1.0 (file:///projects/adder)
  Finished test [unoptimized + debuginfo] target(s) in 0.57s
  Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

  Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

# Running Tests

Let's break down the output of `cargo test`.

```
running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

- We see `test result: ok`, meaning we have passed all the tests
- In this case, only 1 test has run, and it has passed

# Documentation Tests

You may have seen something similar to this in your homework:

```
Doc-tests adder
running 0 tests
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

- All of the code examples in documentation comments are treated as tests!
- This is useful for keeping your docs and code in sync

# `#[cfg(test)]`

You may have also noticed this `#[cfg(test)]` attribute in your homework:

```
#[cfg(test)]
mod tests {
    // <-- snip -->
}
```

- This tells the compiler that this entire module should *only* be used for testing
- The compiler ignores this module when compiling with `cargo build`

# Writing Better Tests

Let's try and be more creative with our tests.

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

# Failing Tests

Let's see what we get:

```
$ cargo test

running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:

---- tests::another stdout ----
thread 'tests::another' panicked at 'Make this test fail', src/lib.rs:10:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

error: test failed, to rerun pass `--lib`
```

# Failing Tests

```
failures:

---- tests::another stdout ----
thread 'tests::another' panicked at 'Make this test fail', src/lib.rs:10:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
  tests::another

test result: FAILED. 1 passed; 1 failed; <-- snip -->
error: test failed, to rerun pass `--lib`
```

- Instead of `ok`, we get that the result of `tests::another` is `FAILED`

# Checking Results

We can use the `assert!` macro to ensure that something is `true`.

```
#[test]
fn larger_can_hold_smaller() {
    let larger = Rectangle {
        width: 8,
        height: 7,
    };
    let smaller = Rectangle {
        width: 5,
        height: 1,
    };

    assert!(larger.can_hold(&smaller));
}
```



# Testing Equality

Rust also provides a way to check equality between two values with `assert_eq!`.

```
#[test]
fn it_adds_two() {
    assert_eq!(4, add_two(2));
}
```

# Testing Equality

If `add_two(2)` somehow evaluated to `5`, we would get this output:

```
---- tests::it_adds_two stdout ----  
thread 'tests::it_adds_two' panicked at 'assertion failed: `(left == right)`  
  left: `4`,  
 right: `5`', src/lib.rs:11:9  
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

- You get a nicer error message from `assert_eq!` versus using `assert!(left == right)`

# Custom Error Messages

We can also write our own custom error messages in `assert!`

```
#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{}`",
        result
    );
}
```

## `#[should_panic]`

If you want test that your code (correctly) panics, you can use `#[should_panic]`:

```
#[test]
#[should_panic(expected = "not less than 100")]
fn greater_than_100() {
    this_better_be_less_than_100(200);
}
```

- The `#[should_panic]` attribute says that this test expects a panic!
- Adding the `expected = "..."` means we want a specific panic message

# Controlling Test Behavior

`cargo test` compiles your code in test mode and runs the resulting test binary.

- By default, it will run all tests in parallel and will not print any test output
- Other testing configurations are available
- *Note that you can run `cargo test --help`, and `cargo test -- --help` for help*

# Running Tests in Parallel

- Suppose each of your tests all write to some shared file on disk
  - All tests write to a file `output.txt`
- They later assert that the file still contains that data they wrote
- You probably don't want all of them to run at the same time!

# Test Threads

By default, Rust will run all tests in parallel, on different threads.

You can use `--test-threads` flag to control the number of threads.

```
$ cargo test -- --test-threads=1
```

- Only use this when you actually need to, otherwise the benefits of running tests in parallel are lost

# Showing Output

If you want to prevent the capturing of output, you can use `--no-capture`.

```
$ cargo test -- --no-capture  
$ cargo test -- --show-output
```

- `--no-capture` will print the full output of every test that is run
- Using `--show-output` will only show the output of passed tests
- With 1000 tests, this might become verbose!
- If only we could only run a subset of the tests...



# Running Tests by Name

Let's say we have 1000 tests, but only one is named `one_hundred`. We can run `cargo test one_hundred` to only run the `one_hundred` test.

```
$ cargo test one_hundred

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 999 filtered out; finished in 0.00s
```

- Notice how there are now `999 filtered out` tests, these were the tests that didn't match the name `one_hundred`

# Multiple Tests by Name

`cargo` will actually find *any* test that matches the name you passed in.

```
$ cargo test add

running 2 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 998 filtered out; finished in 0.00s
```

- If you want an exact name, use `cargo test {name} -- --exact`

# Ignoring Tests

We can ignore some tests by using the `#[ignore]` attribute.

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

- If we want to run only ignored tests, we can run `cargo test -- --ignored`
- If we want to run all tests, we can run `cargo test -- --include-ignored`

# Test Organization

The Rust community thinks about tests in terms of two main categories: unit tests and integration tests.

- Unit tests test each unit of code in isolation
- Integration tests are external to your library, testing the entire system

# Unit Tests

Unit tests are almost always contained within the `src` directory.

- The convention is to create submodules named `tests` for every module you want to test
  - Make sure to add the attribute `#[cfg(test)]` !
- Prevents deploying extra code in production that is only used for testing

# Testing Private Functions

You can unit test private functions as long as the module the test lives in has access to it.

```
pub fn add_two(a: i32) -> i32 { internal_adder(a, 2) }
fn internal_adder(a: i32, b: i32) -> i32 { a + b }

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

# Integration Tests

Integration Tests use your library in the same way any other code would.

- They can only call functions that are part of your library's public API
- Useful for testing if many parts of your library work together correctly

# Integration Tests

To create integration tests, we need a `tests` directory.

```
adder
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── integration_test.rs
```

- Notice how `tests` is *outside* of `src`



# Integration Tests

Since we are now external to our own library, we must import everything as if it were a 3rd-party crate.

adder/tests/integration\_test.rs

```
use adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

- Note that we don't need to annotate anything with `#[cfg(test)]`
- We run test files using the name of the integration test file, like

```
cargo test --test integration_test
```

# Integration Tests for Binary Crates

We cannot create integration tests for a binary crate.

- Binary crates do not expose their functions
- This is why most binary crates will be paired with a library crate, even if they don't *need* to expose any functions

## Recap: Testing

- Unit tests examine parts of a library in isolation and can test private implementation details
- Integration tests check that many parts of the library work together correctly
- Even though Rust can prevent some kinds of bugs, tests are still extremely important to reduce logical bugs!

# Homework 6

Homework 6 is going to be very different from the previous 5 homeworks!

- You will be following a tutorial from the [official Rust Book](#) (our textbook)
- Your task is to implement a miniature version of the popular CLI tool `grep`
- We are not giving you *any* starter code
- Your assignment will be manually graded on both correctness and robustness

# Code Review

# Code?

- What is code?
- Why do we write code?

# Typical Assignment Workflow

You are probably very used to [this workflow](#):

- Get assignment handout and code outline
- "Fill in the blanks"
- Debug a few things
- Graded by autograder
- **All done!**
  - Never use this code again
  - Delete it at the end of the semester

# Real World Workflow

In the real world, the workflow is almost the complete opposite:

- Jump into a *massive* codebase
- Figure out where the "blanks" are to add new features
- Read and debug someone else's code
- Run tests that may not be exhaustive
- **Not done!**
  - Your users and your team are constantly "grading" your work



# Homework 6 Grading

We will be manually grading for both correctness and *robustness*.

- By following the tutorial, you will easily get 100% on this assignment
- We will give you up to 100 extra credit points for style, quality, documentation, and robustness
- If you simply copy and paste everything from the tutorial, *you may get around 15/100 extra credit points*
  - We are going to be super strict!
    - Adopting the 15-410 (CMU Operating Systems) mindset

# Style

You can follow the [Rust style guide](#) for advice.

Key things:

- Format your code correctly with `cargo fmt`
- Run the `cargo clippy` linter
- Make sure comments are styled correctly
- Use descriptive naming

# Documentation

Most of the extra credit grade will come from your documentation.

- Documentation should be **descriptive and succinct**
  - For this assignment, explain the features of your executable!
- For fellow developers, explain:
  - Design
  - Architecture / structure of your code
  - *Why* does this function need to exist?

# Errors

There are 3 types of errors:

- Hmm...
  - Try to *resolve*
- That's not right...
  - Try to *report*
- Uh-oh
  - Try to help the developer find the *fatal* problem faster

# Testing

Write good tests!

- 1000 tests testing the same thing?
- 5 tests testing edge cases?

# Next Lecture: Crates, Closures, and Iterators

Thanks for coming!

*Slides created by:*

Connor Tsui, Benjamin Owad, David Rudo,  
Jessica Ruan, Fiona Fisher, Terrance Chen

