Intro to Rust Lang

Smart Pointers and Trait Objects

Today: Smart Pointers and Trait Objects

- `Box<T>`
- The `Deref` trait
- The `Drop` trait
- Trait Objects
- Smart Pointers

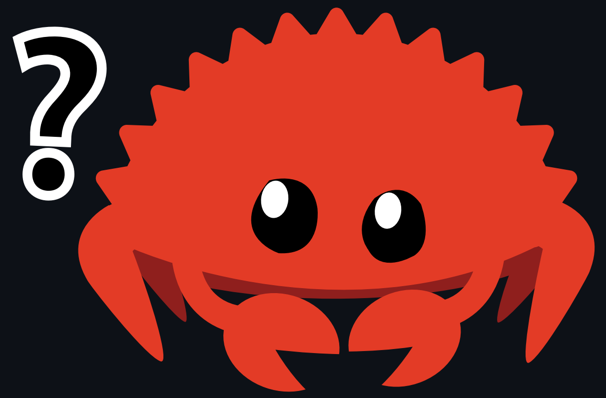
Motivation for `Box<T>`

Let's Make a List

Let's say we wanted to make a recursive-style list:

```rust
enum List {
  Cons(i32, List),
  Nil,
}

fn main() {
  // List of [1, 2, 3]
  let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```



The Compiler's Suggestion

```
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^
2 |     Cons(i32, List),
  |               ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
  |
2 |     Cons(i32, Box<List>),
  |               ++++    +
```

- The compiler is complaining because we've defined a type with *infinite size*
- The suggestion is to use a `Box<List>`

Indirection with `Box<T>`

```
let singleton = Cons(1, Box::new(Nil));
let list = Cons(1, Box::new(Cons(2,
                     Box::new(Cons(3,
                       Box::new(Nil))))));
```

- In the suggestion, "indirection" means we store a *pointer* to a `List` rather than an entire `List`
  - Pointers have fixed size, so our enum is no longer of infinite size!
- We create a `Box<List>` with the `Box::new` associated function

More about `Box<T>`

- `Box<T>` is a simple "smart" pointer to memory allocated on the heap*
  - It is "smart" because it frees the memory when dropped
- Other than the cost of allocation and pointer indirection, `Box` has no performance overhead
- `Box<T>` fully owns the data it points to (just like `Vec<T>`)

When to use `Box<T>`

- When you have a type of unknown size **at compile time** (like `List`)
- When you have a large amount of data and want to transfer ownership
    - Transferring ownership of a pointer is faster than a large chunk of data
- Trait Objects
    - We'll get to this soon...

Using Values in the `Box`

```
let x = 5;
let y = Box::new(x);

assert_eq!(5, x);
assert_eq!(5, *y);
```

- Just like a reference we can dereference a `Box<T>` to get `T`
- `Box<T>` implements the `Deref` trait which customizes the behavior of `*`

`Deref` Trait

The deref trait is defined as follows:

```
pub trait Deref {
    type Target: ?Sized;

    // Required method
    fn deref(&self) -> &Self::Target;
}
```

- Behind the scenes `*y` is actually `*(y.deref())`
- Note this does not recurse infinitely
- We can treat anything that implements `Deref` like a pointer!

Deref Coercion

Recall that we were able to coerce a `&String` into a `&str`. We can also coerce a `&Box<String>` into a `&str`!

```
fn hello(name: &str) {
    println!("Hello, {name}!");
}

fn main() {
    let m = Box::new(String::from("Rust"));
    hello(&m);
}
```

- Deref coercion converts a `&T` into `&U` if `Deref::Target = U`
- Example: Deref coercion can convert a `&String` into `&str`
  - `String` implements the `Deref` trait such that `Deref::Target = &str`

Deref Coercion Rules

Rust is able to coerce mutable to immutable but not the reverse.

- From `&T` to `&U` when `T: Deref<Target=U>`
- From `&mut T` to `&mut U` when `T: DerefMut<Target=U>`
- From `&mut T` to `&U` when `T: Deref<Target=U>`
- For more information, consult the Rustonomicon

`&T` to `&U` Example

```
fn foo(s: &[i32]) {
    print(s[0])
}

// Vec<T> implements Deref<Target=[T]>.
let owned = vec![1, 2, 3];

// Here we coerce &Vec<T> to &[T]
foo(&owned);

println!("{:?}", owned);
```

```
[1]
[1, 2, 3]
```

`&mut T` to `&mut U` Example

```rust
fn foo(s: &mut [i32]) {
    s[0] += 1;
}

// Vec<T> implements DerefMut<Target=[T]>.
let mut owned = vec![1, 2, 3];

// Here we coerce &mut Vec<T> to &mut [T]
foo(&mut owned);

println!("{:?}", owned);
```

```
[2, 2, 3]
```

- `DerefMut` also allows coercing to &[T]

The `Drop` Trait

```rust
pub trait Drop {
    fn drop(&mut self);
}
```

- Values are dropped when they go out of scope
- Dropping a value will recursively drop all its fields by default
  - This mechanism allows automatically freeing memory
- You can also provide a custom implementation of `Drop` on your types
  - Allows us to run user code when values are dropped

`Drop` Trait Example

```rust
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping `CustomSmartPointer` with data `{}`!", self.data);
    }
}
```

- This is a custom implementation that simply prints the data on drop
- The data will still be freed automatically
  - This method does not include automatic memory freeing logic

`Drop` Trait Example

```rust
let c = CustomSmartPointer {
    data: String::from("my stuff"),
};
let d = CustomSmartPointer {
    data: String::from("other stuff"),
};
println!("CustomSmartPointers created.");
```

```
CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
Dropping CustomSmartPointer with data `my stuff`!
```

- Notice how values are dropped in reverse order of creation

`Drop` Trait Usage

`Drop` trait implementations are typically not needed unless:

- You are manually managing memory
  - This likely involves using `unsafe` under the hood
- You need to do something special before a value is dropped
  - Might involve managing OS resources
  - Might involve signalling other parts of your codebase

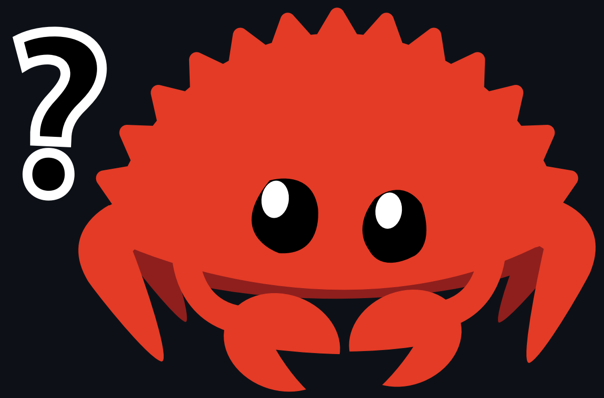
Manual Drop

What if we want to manually drop a value before the end of the scope?

```rust
let csm = CustomSmartPointer {
    data: String::from("some data"),
};
println!("CSM created.");

csm.drop();

println!("CSM dropped before the end of the scope");
```



Manual Drop

```
error[E0040]: explicit use of destructor method
  --> src/main.rs:16:7
   |
16 |     c.drop();
   |     --^^^^--
   |     | |
   |     | explicit destructor calls not allowed
   |     help: consider using `drop` function: `drop(c)`
```

- Rust won't let you explicitly call the drop trait method to avoid double drops

Manual Drop

```
let csm = CustomSmartPointer {
    data: String::from("some data"),
};
println!("CSM created.");

std::mem::drop(csm);

println!("CSM dropped before the end of the scope");
```

- This code works since we use `std::mem::drop` instead
- What's the difference?

`std::mem::drop`

Here is the actual source code of `std::mem::drop` in the standard library:

```rust
pub fn drop<T>(_x: T) {}
```

- It takes ownership of `_x`, and then `_x` reaches the end of the scope and is dropped
    - Hence, calling this function drops it, on demand!

Object Oriented Programming

- Well...
  - Not quite...

What We Know So Far...

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}

impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    <-- snip -->
}
```

- Encapsulation within `impl` blocks and crates
- Public and private functions and methods with `pub`

Inheritence?

Rust structs cannot "inherit" the implementations of methods or data fields from another struct...

- If we want to wrap another struct's functionality, we can use composition
- If we want to define interfaces, we can use traits
- If we want polymorphism...
    - Rust has something called "trait objects"

Polymorphism

- Polymorphism != Inheritance
- Polymorphism == "Code that can work with multiple data types"
- In object oriented languages, polymorphism is usually expressed with classes
- Rust expresses polymorphism with generics and traits:
  - Generics are abstract over different possible types
  - Traits impose constraints on what behaviors types must have

Trait Objects

Trait objects allow us to store objects that implement a trait.

```
pub trait Window {
    fn draw(&self);
}

pub struct LaptopScreen {
    pub windows: Vec<Box<dyn Window>>,
}
```

- In this example, `LaptopScreen` holds a vector of `Window` objects
- We use the `dyn` keyword to describe any type that implements `Window`
  - In a `Box`, since objects implementing `Window` could be of any size at runtime

Trait Objects and Closures

Since closures implement the `Fn` traits, they can be represented as trait objects!

```rust
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}

fn main() {
    let closure = returns_closure();
    print!("{}", closure(5)); // prints 6
}
```

- We can use trait objects to return dynamic types
- Deref coercion happening in the background to keep ergonomics clean!

Working With Trait Objects

```rust
struct ChromeWindow {
    width: u32,
    height: u32,
    evil_tracking: bool
}

struct FirefoxWindow {
    width: u32,
    height: u32,
}

impl Window for ChromeWindow { fn draw(&self) { ... } }
impl Window for FirefoxWindow { fn draw(&self) { .. } }
```

- Say we have **multiple** types that implement `Window`

Working With Trait Objects: Dynamic Dispatch

```rust
impl LaptopScreen {
    pub fn run(&self) {
        // `windows` is of type Vec<Box<dyn Window>>
        for window in self.windows.iter() {
            window.draw();
        }
    }
}
```

- This is different than if `windows` was `Vec<ChromeWindow>`
    - The generic parameter (in `Vec`) is known at compile time.
- Trait objects allow for types to fill in for the trait object **at runtime**

Generic Version

What about a version implemented with generics?

```rust
pub struct LaptopScreen<T: Window> {
    pub windows: Vec<T>,
}

impl<T> LaptopScreen<T>
where
    T: Window,
{
    pub fn run(&self) {
        for window in self.windows.iter() {
            window.draw();
        }
    }
}
```

- What is wrong with this version?

Trait Objects: Mixing Objects

```rust
let screen = LaptopScreen {
    windows: vec![
        Box::new(ChromeWindow {
            width: 1280,
            width: 720,
            evil_tracking: true,
        }),
        Box::new(FirefoxWindow {
            <-- snip -->
        }),
    ],
};
screen.run();
```

- This is not possible with the version using generics

Aside: Dynamically Sized Types

- `dyn Window` is an example of a dynamically sized type (DST)
- DSTs have to be in a `Box`, because we don't know the size at compile time
  - A `dyn Window` could be a `ChromeWindow` or `FirefoxWindow` object
  - How much space should we make on the stack?
- Pointers to DSTs are double the size (wide pointers)
  - If you're interested in more information, ask us after lecture!

Smart Pointers

Smart Pointers

- `Rc<T>`
- `RefCell<T>`
- Interior Mutability
- Memory Leaks

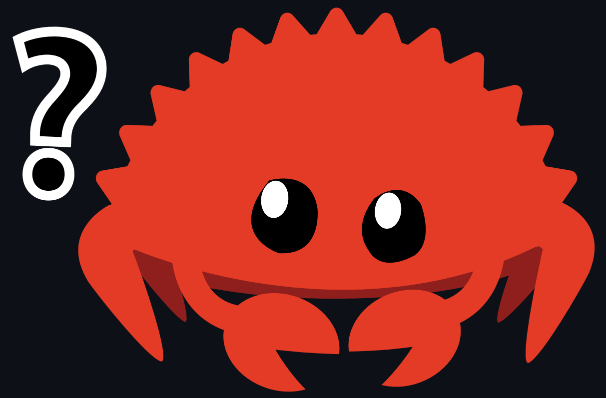
Motivation for `Rc<T>`

Let's Make a List (again)

Let's continue making the recursive-style list from the beginning of lecture with Box<T> .

```rust
enum List {
  Cons(i32, Box<List>),
  Nil,
}

fn main() {
  let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
  let b = Cons(3, Box::new(a));
  let c = Cons(4, Box::new(a));
}
```



Cargo's Suggestion

```
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0382]: use of moved value: `a`
  --> src/main.rs:11:30
   |
9  |     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
   |         - move occurs because `a` has type `List`, which does not implement the `Copy` trait
10 |     let b = Cons(3, Box::new(a));
   |                              - value moved here
11 |     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move
```

- `Cons` needs to **own** the data it holds
- We want both `b` and `c` to point to the same instance `a`
  - `a` was already moved into `b` when we create `c`

References?

```
enum List<'a> {
    Cons(i32, &'a List<'a>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
  let nil = Nil;
  let a = Cons(10, &nil);
  let b = Cons(5, &a);
  let c = Cons(3, &a);
  let d = Cons(4, &a);
}
```

- While it can be done, it's a little messy

Introducing `Rc<T>`

```
enum List {
  Cons(i32, Rc<List>),
  Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
  let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
  let b = Cons(3, Rc::clone(&a));
  let c = Cons(4, Rc::clone(&a));
}
```

- Short for reference counted
- Keeps track of the number of references to a value

`Rc<T>`: Reference Counted

- Use `Rc::new(T)` to create a new `Rc<T>`
  - `Rc::clone()` isn't a deep clone, it increments the ref counter
- When an `Rc` is cloned, increment reference count
- When an `Rc` is dropped, decrement reference count
- When the reference count reaches zero, free the memory

When to use `Rc<T>`

- Share one instance of allocated memory throughout the program
  - We can only access the data as read-only
  - We don't need to know what part of the program is going to use it last
- Only used for single-threaded scenarios
  - `Arc<T>` for multi-threaded (more on that soon)

`Rc<T>` Reference Counting Demonstrated

```rust
fn main() {
    let a = Rc::new(String::new("TODO: Steal Connor's identity"));
    // Ref count after creating a: 1

    let b = Rc::clone(&a);
    // Ref count after creating b: 2

    {
        let c = Rc::clone(&a);
        // Ref count after creating c: 3
    }
    // Ref count after dropping c: 2
}
// Ref count after dropping b and c: 0
```

- The ref count is incremented on each clone
- The ref count is decremented on each drop

`Rc<T>` Recap

- Allows sharing ***immutable*** references without lifetimes
- Should be used when the last user of the data is unknown
- Very low overhead for providing this capability
  - O(1) increment/decrement of counter
  - Potential allocation/de-allocation on heap
- Implemented using the `Drop` trait and `unsafe`!

`RefCell<T>` and Interior Mutability

First, `Cell<T>`

```rust
use std::cell::Cell;

let c1 = Cell::new(5i32);
c1.set(15i32);

let c2 = Cell::new(10i32);
c1.swap(&c2);

assert_eq!(10, c1.into_inner()); // consumes cell
assert_eq!(15, c2.get()); // returns copy of value
```

- Shareable, mutable container
- Values can be moved in and out of a cell
- Used for `Copy` types
  - (where copying or moving values isn't too resource intensive)

`RefCell<T>`

- Hold's sole ownership like `Box<T>`
- Allows borrow checker rules to be enforced at **runtime**
  - Interface with `.borrow()` or `borrow_mut()`
  - If borrowing rules are violated, `panic!`
- Typically used when Rust's conservative compile-time checking "gets in the way"
- It is **not** thread safe!
  - Use `Mutex<T>` instead

Interior Mutability

```rust
fn main() {
  let x = 5;
  let y = &mut x; // cannot borrow immutable x as mutable
}
```

- It would be useful for a value to mutate itself in its methods but appear immutable to other code
- Code outside the value's methods wouldn't be able to mutate it
- This can be achieved with `RefCell<T>`

Interior Mutability with Mock Objects

```rust
pub trait Messenger {
    // Note this takes &self and not &mut self
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}
```

- `LimitTracker` tracks a value against a maximum value and sends messages based on how close to the maximum value the current value is
- We want to mock a messenger for our limit tracker to keep track of messages for testing

Limit Tracker

```rust
impl<'a, T> LimitTracker<'a, T>
where
    T: Messenger,
{
    // --- snip ---
    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 1.0 {
            self.messenger.send("Error: You are over your quota!");
        } else if percentage_of_max >= 0.9 {
            self.messenger
                .send("Urgent warning: You've used up over 90% of your quota!");
        } else if percentage_of_max >= 0.75 {
            self.messenger
                .send("Warning: You've used up over 75% of your quota!");
        }
    }
}
```

Our Mock Messenger

```
struct MockMessenger {
  sent_messages: Vec<String>,
}

impl MockMessenger {
  fn new() -> MockMessenger {
    MockMessenger { sent_messages: vec![] }
  }
}

impl Messenger for MockMessenger {
  fn send(&self, message: &str) {
    self.sent_messages.push(String::from(message));
  }
}
```

- This code won't compile! `self.sent_messages.push` requires `&mut self`

Let's Use Interior Mutability

```
use std::cell::RefCell;

struct MockMessenger {
  sent_messages: RefCell<Vec<String>>,
}

impl MockMessenger {
  fn new() -> MockMessenger {
    MockMessenger {
      sent_messages: RefCell::new(vec![]),
    }
  }
}

impl Messenger for MockMessenger {
  fn send(&self, message: &str) {
    self.sent_messages.borrow_mut().push(String::from(message));
  }
}
```

Managing Borrows

```
impl Messenger for MockMessenger {
  fn send(&self, message: &str) {
    let mut one_borrow = self.sent_messages.borrow_mut();
    let mut two_borrow = self.sent_messages.borrow_mut();

    one_borrow.push(String::from(message));
    two_borrow.push(String::from(message));
  }
}
```

- We still use the `&` and `mut` syntax for `RefCell`
- `borrow` returns either a `Ref` or `RefMut` which implement `Deref` / `DerefMut`
  - Deref coercion applies: Can be treated as standard references

What Makes Each Smart Pointer Unique

- `Rc<T>` - Enables multiple read-only owners of the same data
- `Box<T>` - Allows immutable or mutable borrows that are checked at compile time
- `RefCell<T>` - Allows immutable/mutable borrows that are checked at ***runtime***

Combining Smart Pointers: `Rc<RefCell<T>>`

```rust
#[derive(Debug)]
enum List {
  Cons(Rc<RefCell<i32>>, Rc<List>),
  Nil,
}
```

- Common type seen in Rust
- Enables multiple owners of mutable data (with runtime checks)
- Extremely powerful, but comes with some overhead

`Rc<RefCell<T>>` List

```
let value = Rc::new(RefCell::new(5));

let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

*value.borrow_mut() += 10;

println!("a after = {:?}", a);
println!("b after = {:?}", b);
println!("c after = {:?}", c);
```

```
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 4 }, Cons(RefCell { value: 15 }, Nil))
```

Let's Try Another List

```
enum List {
  Cons(i32, RefCell<Rc<List>>),
  Nil,
}

impl List {
  fn tail(&self) -> Option<&RefCell<Rc<List>>> {
    match self {
      Cons(_, item) => Some(item),
      Nil => None,
    }
  }
}
```

- This implementation allows modifying the list structure instead of list values
- Now we have a function `tail` that gets the rest of our list

What Happens?

```
let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

println!("a initial rc count = {}", Rc::strong_count(&a));
println!("a next item = {:?}", a.tail());

let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

println!("a rc count after b creation = {}", Rc::strong_count(&a));
println!("b initial rc count = {}", Rc::strong_count(&b));
println!("b next item = {:?}", b.tail());

if let Some(link) = a.tail() {
  *link.borrow_mut() = Rc::clone(&b);
}

println!("b rc count after changing a = {}", Rc::strong_count(&b));
println!("a rc count after changing a = {}", Rc::strong_count(&a));

println!("a next item = {:?}", a.tail());
```

Answer

```
Exited with signal 6 (SIGABRT): abort program

a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b rc count after changing a = 2
a rc count after changing a = 2
a next item = Some(RefCell { value: Cons(10, RefCell { value: Cons(5, RefCell...
```

- We see that at the end we have a reference cycle!

Let's Look Closer

```
let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));
// a is Cons(5, Nil)

let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));
// b is Cons(10, a) = Cons(10, Cons(5, Nil))

if let Some(link) = a.tail() {
    // link is Nil (pointed to by a)
    *link.borrow_mut() = Rc::clone(&b);
    // link is now b = Cons(10, a)
}
// a = Cons(5, link) = Cons(5, b) = Cons(5, Cons(10, a))
// ^^^ reference cycle of a made!
```

- This can cause a memory leak!
  - `Rc` only frees when the `strong_count` is 0

Avoiding Reference Cycles

- We know `Rc::clone` increases the `strong_count`
- You can create a `Weak<T>` reference to a value with `Rc::downgrade`
  - This increases the `weak_count` and can be nonzero when the `Rc` is freed
- To ensure valid references, `Weak<T>` must be upgraded before any use
  - Returns an `Option<Rc<T>>`

`Weak<T>` Trees

```rust
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
  value: i32,
  parent: RefCell<Weak<Node>>,
  children: RefCell<Vec<Rc<Node>>>,
}
```

Weak<T> Trees In Action

```
fn main() {
  let leaf = Rc::new(Node {
    value: 3,
    parent: RefCell::new(Weak::new()),
    children: RefCell::new(vec![]),
  });

  println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

  let branch = Rc::new(Node {
    value: 5,
    parent: RefCell::new(Weak::new()),
    children: RefCell::new(vec![Rc::clone(&leaf)]),
  });

  *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

  println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
} // Tree is effectively dropped even with parent references!
```

Recap

- `Box<T>`
- The `Deref` trait
- The `Drop` trait
- Trait Objects
- Smart Pointers

Next Lecture: Unsafe

Thanks for coming!

Slides created by:

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