# Introducing Rust

- What is Rust?
  - Basic operations and types
  - **Abstraction: Traits, Enumerated types and pattern matching**
  - Memory allocation and boxes
- Why is it interesting?

# Traits (1/5)

```
trait Area {
    fn area(self) -> u32;
}

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

impl Area for Rectangle {
    fn area(self) -> u32 {
        self.width * self.height
    }
}
```

Define a trait with a single method that must be implemented

Define a struct type, **`Rectangle`**

Implement the **`Area`** trait on the **`Rectangle`** type

https://doc.rust-lang.org/book/ch10-02-traits.html

- Traits describe **functionality** that types can implement
  - Methods that must be provided, and associated types that must be specified, by types that implement the trait – but no instance variables or data
  - Similar to type classes in Haskell or interfaces in Java

# Traits (2/5)

```
trait Area {
    fn area(self) -> u32;
}

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

impl Area for Rectangle {
    fn area(self) -> u32 {
        self.width * self.height
    }
}
```

A trait can be implemented by multiple types – here we also implement the **`Area`** trait for the **`Circle`** type

```
struct Circle {
    radius: u32
}


impl Area for Circle {
    fn area(self) -> u32 {
        PI * self.radius * self.radius
    }
}
```

- Traits are an important tool for abstraction – similar role to sub-types in many languages

# Traits (3/5): Generic Functions

- Rust uses traits instead of classes and inheritance to define generic functions or methods that work with any type that implements a particular trait
- Define a trait:

```
trait Summary {
    fn summarise(self) -> String;
}
```

- Write functions that work on types that implement that trait:

```
fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarise());
}
```

Type parameter in angle brackets: **`T`** is any type that implement the **`Summary`** trait

# Traits (4/5): Deriving Common Traits

- The `derive` attribute makes compiler automatically generate implementations of some common traits:

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

The `#[derive(T)]` annotation makes compiler generate `impl` block with standard implementation of methods for derived trait `T`

- Compiler implements this for traits in the standard library that are always implemented the same way:
  - https://doc.rust-lang.org/book/appendix-03-derivable-traits.html
- Can also be implemented for user-defined traits:
  - Only useful if every implementation of the trait will follow the exact same structure
  - https://doc.rust-lang.org/book/ch19-06-macros.html#how-to-write-a-custom-derive-macro

# Traits (5/5): Associated Types

- Traits can also specify associated types – types that must be specified when a trait is implemented
- Example: **`for`** loops operate on iterators

```
fn main() {
    let a = [42, 43, 44, 45, 46];

    for x in a.iter() {
        println!("x={}", x);
    }
}
```

The **`a.iter()`** function call returns an iterator over the array

- An iterator is something that implements the **`Iterator`** trait:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    // more...
}
```

An **`impl`** of **`Iterator`** must define the type of **`item`**, as well as implementing the methods

# Enumerated Types (1/3)

```
enum TimeUnit {
    Years, Months, Days, Hours, Minutes, Seconds
}
```

Basic enums work just like in C

https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}

let when = RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);
```

Enums also generalise to store tuple-like variants:

```
enum Shape {
    Sphere {centre: Point3d, radius: f32},
    Cuboid {corner1: Point3d, corner2: Point3d}
}

let unit_sphere = Shape::Sphere{centre: ORIGIN, radius: 1.0};
```

...and struct-like variants:

# Enumerated Types (2/3)

- An **`enum`** is used when a variable, parameter, or result can have on of several possible types
  - An **`enum`** defines alternative types for a type
  - An **`enum`** can have type parameters that must be defined when the enum is instantiated:

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

  - An **`enum`** can have methods and can implement traits
- Use **`enum`** to model data that can take one of a set of related values

# Enumerated Types (3/3)

- Standard library has two extremely useful standard **`enum`** types
- The **`Option`** type represents optional values
  - In C, one might write a function to lookup a key in a database:

```
value *lookup(struct db* self, key *k) {
  // ...
}
```

    this returns a pointer to the value, or **`null`** if the key doesn't exist

  - In Rust, the equivalent function would return **`Option<Value>`**:

```
fn lookup(self, key : Key) -> Option<Value> {
  // ...
}
```

- The **`result`** type similarly encodes success or failure:

```
fn recv(self) -> Result<Message, NetworkError> {
  // ...
}
```

Definitions in the standard library:

```
enum Option<T> {
    Some(T),
    None
}
```

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

# Pattern Matching (1/4)

- Rust `match` expressions generalise the C `switch` statement
- Match against constant expressions and wildcards:

```
match meadow.count_rabbits() {
    0 => {} // nothing to say
    1 => println!("A rabbit is nosing around in the clover."),
    n => println!("There are {} rabbits hopping about in the meadow", n)
}
```

https://doc.rust-lang.org/book/ch06-02-match.html

- The value of **`meadow.count_rabbits()`** is matched against the alternatives
- If matches the constants `0` or `1`, the corresponding branch executes
- If none match, the value is stored in the variable `n` and that branch executes
  - Matching against _ gives a wildcard without assigning to a variable

# Pattern Matching (2/4)

- Patterns can be any type, not just integers

```
let calendar = match settings.get_string("calendar") {
    "gregorian" => Calendar::Gregorian,
    "chinese"   => Calendar::Chinese,
    "ethiopian" => Calendar::Ethiopian,
    _           => return parse_error("calendar", other)
};
```

- The `match` expression evaluates to the value of the chosen branch
  - Allows, e.g., use in **`let`** bindings, as shown

# Pattern Matching (3/4)

- Patterns can match against enum values:

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}

let when = RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);
```

```
match rt {
    RoughTime::InThePast(units, count)   => format!("{} {} ago", count, units.plural()),
    RoughTime::JustNow                   => format!("just now"),
    RoughTime::InTheFuture(units, count) => format!("{} {} from now", count, units.plural())
}
```

- Selects from different types of data, expressed as **`enum`** variants
- Must match against all possible variants, or include a wildcard – else compile error

# Patterns Matching (4/4)

- C functions often return pointer to value, or **`null`** if the value doesn't exist
  - Easy to forget the **`null`** check when using the value:

```
customer *get_user(struct db *db, char *username) {
  // ...
}

customer *c = get_user(db, customer_name);
book_ticket(c, event);
```

  - Program crashes with null pointer dereference at run-time if user is not found
- Rust function return an **`Option`** and pattern match on result:

```
fn get_user(self, username : String) -> Option<Customer> {
  // ...
}

match db.get_user(customer_name) {
  Some(customer) => book_ticket(customer, event),
  None           => handle_error()
}
```

https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html#the-option-enum-and-its-advantages-over-null-values

Why is this better? All enum variants must be handled, so won't compile if you forget to check the error case

# Introducing Rust

- What is Rust?
  - Basic operations and types
  - **Abstraction: Traits, Enumerated types and pattern matching**
  - Memory allocation and boxes
- Why is it interesting?