# 1 Traits

## 1.1 Comparing

```
use std::cmp::*;
trait PartialEq<Rhs=Self> {
  fn eq(&self, other: &Rhs) -> bool;
  ..
}
trait Eq: PartialEq {} //marker trait
trait PartialOrd<Rhs=Self>: ParitalEq<Rhs> {
  fn partial_cmp(&self, other: &Rhs)
    -> Option<Ordering>;
  ..
}
trait Ord: Eq + PartialOrd {
  fn cmp(&self, other: &Self) -> Ordering;
  ..
}
enum Ordering {
  Less,
  Equal,
  Greater,
}
```

## 1.2 Hashing

```
use std::hash::*;
trait Hasher {
  fn write(& mut self, bytes: &[u8]) -> ();
  fn finish(&self) -> u64;
  ..
}
```

Sample concrete Hasher:

`std::collections::hash_map::DefaultHasher`

```
trait Hash {
  fn hash<H>(&self, state: & mut H) ->()
    where H: Hasher;
}
```

`#[derive(Hash)]` possible on a struct if
all fields implement Hash trait.

## 1.3 Function Pointer

Coercible from normal function or closure that does not capture the environment.

```
fn normal_function(i: usize) -> usize {..}
let ptr: fn(usize) -> usize = normal_function;
let clos: fn(usize) -> usize = |x| x + 5;
```

## 1.4 Call Operator Traits

```
trait FnMut<Args>: FnOnce<Args>;
trait Fn<Args>: FnMut<Args>;
```

# 2 Collections

```
Vec<T>
BTreeSet<T> where T: Ord //ascending order
BTreeMap<K, V> where K: Ord //ascending order
HashSet<T> where T: Eq + Hash
HashMap<K, V> where K: Eq + Hash
impl BinaryHeap<T> where T: Ord { //default: max-heap
  fn push(item: T);
  fn pop() -> Option<T>;
  fn peek() -> Option<&T>;
  ..
}
impl VecDeque<T> {
  fn pop_back(&mut self) -> Option<T>;
  fn partition_point(&mut self, f: P) -> usize
    where P: FnMut(&T) -> bool;
    //return index of 1st elem of 2nd partition
    //requires elements to be in order
    //all elems in 1st partition satisfy f
  ..
}
```

For changing order:
Use wrapper `std::cmp::Reverse` NewType, or
Use Custom Ord impl

## 2.1 Entry API

```
*container.entry(key).or_insert(val) = ..;
*container.entry(key).or_default() = ..;
*container.entry(key)
  .or_insert_with(|| {..; val} ) = ..;
*container.entry(key)
  .or_insert_with_key(|key| {..; val} ) = ..;
```

# 3 Pattern Match / Destructuring

## 3.1 Matching ergonomics via default binding modes

Summary:
https://rust-lang.github.io/rfcs/2005-match-ergonomics.html

Related RFC:
https://github.com/rust-lang/rfcs/pull/1944

```
let x: &Option<_> = &Some(0);
match x {
  &Some(ref y) => {..}
  &Nonde => {..}
}
//or:
match *x {
  Some(ref y) => {..}
  Nonde => {..}
}
//or:
match x {
  Some(y) => {..} //where y is inferred to be a reference
  Nonde => {..}
}
```

## 3.2 Examples

```
let item = Some(Structure::new());
match item {
  Some(Structure { x, y: 0, z: 1 }) => { f() }
  Some(Structure { z: 2, .. }) => { g() }
  _ => {}
}
match (4, 5, 6) {
  (4, _, v @ 6) => { f(v) }
  w @ (5, _, 2) => { g(w) }
  _ => {}
}
let items = (0, 1, 2, 3, 4, 5);
match items {
  (first, .., last) => { f() }
  _ => {}
}
match item {
  Some(x) if x >= 10 && pred(x) => { f() }
  _ => {}
}
match item2 {
   mybinding @ Structure { x: 5..=10, .. }
     => { f(mybinding) }
  _ => {}
}
match item2 {
   Structure { x: mybinding, y: 5..=10, .. }
     => { f(mybinding, &y) }
  _ => {}
}
match x {
  Some(val @ 0..=10) => { f(val) }
  Some(val @ 11..20) => { g(val) }
  _ => {}
}
match x {
  val @ 0..=10 | val @ 50..=55 => { f(val) }
  _ => {}
}
match Some(x) {
  Some(4) | Some(5) => { f() }
  _ => {}
}
if let Some(MyStruct {x: 5, y, ..}) = item {
  ..
}
match Some(x) {
  //borrow instead of consume by a match
  Some(ref inner) => { f_borrow_only(inner) }
  _ => {}
}
```

## 4 Threading

See reference for more threading: Rust Atomics and Locks [1].

```
let t = std::thread::spawn(||{..})
..
t.join().unwrap();
```

### 4.1 Mutex and Guards

```
use std::sync::{Arc, Mutex};
let m = Arc::new(Mutex::new(..));
..
let m2 = m.clone();
{
  let lock_result = m2.lock();
  let mtx_guard = lock_result.unwrap();
  //std::ops::DerefMut trait for compile-
  //time deref coercion rule
  *mtx_guard = new_value;
}
match m.try_lock(){
  Ok(mut mtx_guard) => {
    *mtx_guard = new_value;
  },
  Err(_) => {}
}
```

### 4.2 Scoped Threads

```
let mut a = vec![1, 2, 3];
let mut x = 0;
std::thread::scope(|s| {
    s.spawn(|| {
      f_borrow(&a);
    });
    s.spawn(|| {
      f_borrow_mut(& mut x);
    });
    println!("hello from the main thread");
    //threads spawn in scope joined
});
f_modify_some_more(& mut x);
```

## 5 Interior Mutability

```
std:;sync::Mutex<T> / RwLock<T>: Send + Sync
std::sync::atomic::AtomicI32 / ..: Send + Sync
std::cell::Cell<T>: !Sync
std::cell::RefCell<T>: !Sync
```

Threadsafe:

```
std:;sync::Mutex<T> / RwLock<T>}
std::sync::atomic::AtomicI32 / ..
```

Value:

```
std::sync::atomic::AtomicI32 / ..
std::cell::Cell<T>
```

Reference:

```
std:;sync::Mutex<T> / RwLock<T>}
std::cell::RefCell<T>
```

## 6 Managed Memory

```
std::rc::Rc<T>: !Send
std::sync::Arc<T>: Send + Sync
  where T: Send + Sync
```

### 6.1 Managed Memory with Interior Mutability

Single thread:

```
//infallible value replacement
std::rc::Rc<std::cell::Cell<T>>

//reference replacement; runtime checking
std::rc::Rc<std::cell::RefCell<T>>
```

Threadsafe:

```
//infallible value replacement
std::sync::Arc<std::sync::atomic::AtomicType>

//reference replacement; runtime checking
std::sync::Arc<std::sync::Mutex<T>> or
  std::sync::Arc<std::sync::RwLock<T>>
```

## 7 Borrow and Reference

```
impl Option<T> {
  fn as_mut(&mut self) -> Option<& mut T>;
  fn as_ref(&self) -> Option<& T>;
  ..
}
```

TODO: expand this section with:

`std::borrow::Borrow` and
`std::convert::AsRef`

## 8 PhantomData

```
std::marker::PhantomData<T> where T: ?Sized;
```

Zero-sized type (compile-time type used by the compiler to reason about safety properties) that owns a `T`.

Use case: place inside struct to make it conceptually own a `T`.

## 9 Pinning

for address-sensitive type

goals:

- reduced surface area of interface: not expose interface that can invalidate data once data is pinned
- make sure pinned value remain valid before drop and drop happens before invalidation

`Pin<T>` $\implies$ `T: Deref / DerefMut` where `Deref::Target` or `DerefMut::Target` cannot be moved/copied

`Pin<T>` can be moved

Pin projections: give out fields of a pinned type

`Unpin` is auto trait by default

need to opt-out of `Unpin` for a type that needs to be pinned, eg: use `std::marker::PhantomPinned` to the structure to opt-out

once pinned, the structure is in an address-sensitive state

pinning an `Unpin` structure negates effects of pinning

## 9.1 Drop for Pinned Data

to guarantee validity of pinned data from the point of pinning the data to right before invalidation happens, drop needs to happen before invalidation

drop notifys and performs ops related to dependent values before content is overwritten

eg: `ManuallyDrop<T>` inhibits destructor of type `T`, therefore it voilates drop guarantees of `Pin<& mut T>` if we create pinned structure from mutable reference to `T` within `ManuallyDrop<T>`

note: if storage is never re-used/invalidated from intentionally leaking memory, such as from `mem::forget(..)`, then drop does not require to be run (drop guarantee is trivially satisfied since invalidation never happens)

`fn drop(&mut self){..}`:

implicitly needs to be treated as

`fn drop( self: Pin<& mut Self>) {..}`

utility:

`Pin::new_unchecked(& mut T) -> Pin<& mut T>`

inner drop function to manipulate it without conflicting pinning invariants

### 9.1.1 Drop for pointer types that will be used as pinning pointers

- ensure not to invalidate or move pointed data during `Drop` implementation, likewise for `Deref/DerefMut` implementations
- careful specialized implementation of assign to pinned value is possible, eg: update all uses of pinnin gaddresses to ensure pinning invariant is held
- note: `Pin::set()` to assign thru `Pin<Ptr\` possible as well: drop will run before assigning new value

## 9.2 Structural Pinning

Definition:
structure is pinned $\implies$ some inner member/field of structure is pinned

exposing non-structurally pinned field of a `Pin<& mut T>`:

```
Pin<& mut T>::map_unchecked_mut(|x| &mut x.field)
   -> & mut Field
```

or

```
Pin<& mut T>::get_unchecked_mut() -> & mut T
...
```

deduction for `Unpin` trait for a structure:
all if the its structurally pinned fields are `Unpin` $\implies$ structure can have `Unpin`

make sure that interface does not move data out of pinned fields when structure has already been pinned

in general standard library pointer types do not have structural pinning:

eg: `Box<T>: Unpin`, even if `T: Pin`, then:

`Pin<Box<T>>: Unpin` since pointers can be moved while pointed data remains valid. Content is pinned or unpinned is independent of whether pointer is pinned $\iff$ pinning is not structural

### 9.2.1 Pinning Destruction

- we assume structure and its fields may be pinned, so need to take `Pin<& mut Self>` during `drop`
- struct cannot be `#[repr(packed)]`
- pinned fields must be dropped before invalidating/reassigning data

**References**

[1] Mara Bos. Rust atomics and locks, 2015.