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

```
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`.

References

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