Lecture 5

Potpourri

Important trait: `Drop`

```rust
struct MyType {
    s: String
}

impl Drop for MyType {
    fn drop(&mut self) {
        println("Dropping...");
    }
}
```

Destructor behavior

- Closing files
- Logging data

Never need to worry about freeing `struct`/`enum` members: always automatic

PLQ review

```
trait Mul<Rhs> {
    type Output;
    fn mul(self, rhs: Rhs) -> Self::Output;
}
```

Why is `Output` an associated type, but `Rhs` is a type parameter?

Standard library takes the opinion that given two input types, the output type is fixed

```
trait Mul<Rhs, Output> { ... }
impl Mul<u32, u32> for u32 { ... }
impl Mul<u32, f32> for u32 { ... }
```

Tow `impl`s with different output types don't collide (does compile)

```
trait Mul<Rhs> { type Output; }
impl Mul<u32> for u32 { type Output = u32 }
impl Mul<u32> for u32 { type Output = f32 }
```

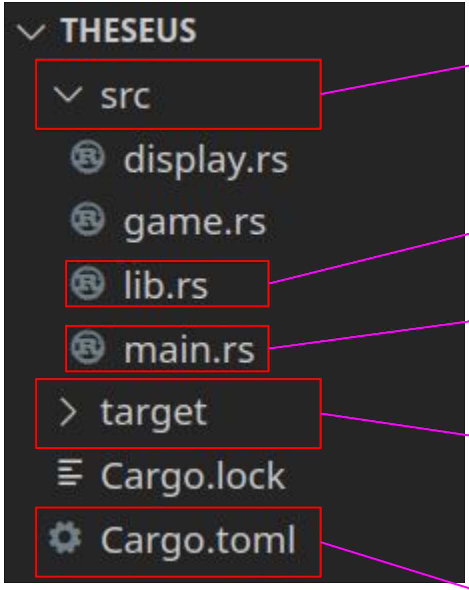
Tow `impl`s with different output types do collide (doesn't compile)

Project 1 retrospective

What worked well? What didn't? Alternate design decisions?

Modules

Anatomy of a crate



All code goes here

Crate root

Program entrypoint

Build artifacts

Metadata and dependencies

- Project name
- Version
- Compilation options

Modules

At the top level is a “crate” (package)

Crates contain modules

Modules contain modules and items

Items are

- types (struct/enum)
- traits
- functions
- impl blocks
- constants

`main.rs` and `lib.rs` are optional, but must have at least one

- No main.rs -> can’t `cargo run`
- No `lib.rs` -> can’t be depended on by another crate

Using exported items

In `main.rs` we can write

```
let game = theseus::game::Game::new();
```

From package `theseus`

and module `game`

the struct `Game`

the function `new`

Using exported items

```
use theseus::game::Game;
let game =                    Game::new();
```

The prelude

```
use std::marker::Copy
use std::ops::{Drop, Fn}
use std::mem::drop
use std::boxed::Box
use std::clone::Clone
use std::cmp::{PartialEq, PartialOrd, Eq, Ord}
use std::convert::{Into, From}
use std::default::Default
use std::iter::Iterator
use std::option::Option::{self, Some, None}
use std::result::Result::{self, Ok, Err}
use std::string::{String, ToString}
use std::vec::Vec
// <5 items omitted>
```

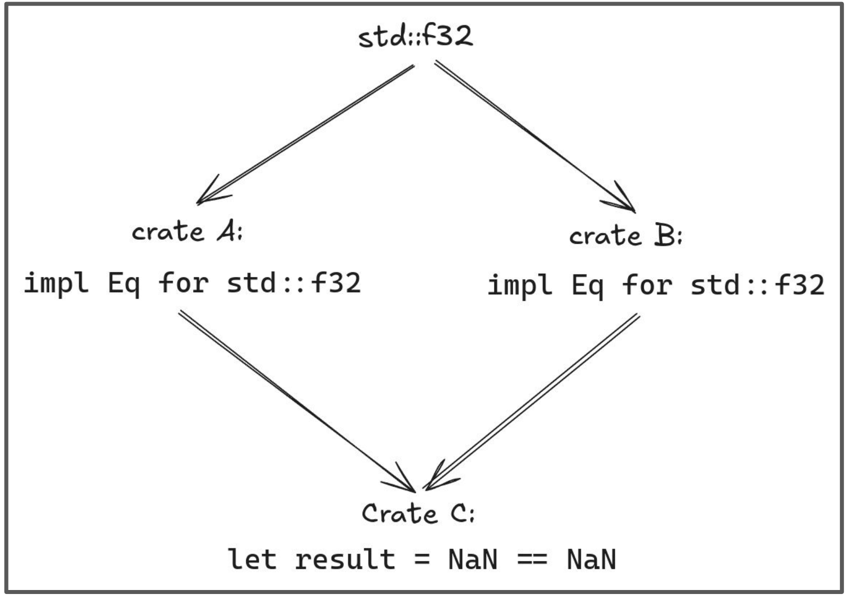
Orphan trait rule

```
impl SomeTrait for SomeType {
    ...
}
```

One must be true

- `SomeTrait` is part of this module
- `SomeType` is part of this module
- Both are part of this module

Orphan trait rule: why?



Which `impl` to use?

In summary

Code comes from three places

- Your own package
- An external package
- The standard library

These are all treated the same by the language

At the top level is a “crate” (package)

Crates contain modules

Modules contain modules and items

Items are

- types (struct/enum)
- traits
- functions
- impl blocks
- constants

Containers

Two bread-and-butter containers

Vec<T> ordered collection of values

- ArrayList<T> or std::vector or list

HashMap<K,V> unordered collection of key/value pairs

- HashMap<K,V> or std::unordered_map<K,V> or dict

What might be hard about these containers?

- Ownership?
- Mutable/immutable references?
- Trait implementations?

What traits does `Vec` satisfy

"Passes through" many traits

- `PartialEq`/`Eq`
- `PartialOrd`/`Ord`
- `Clone`
- `Debug`

Implements some new traits

- `Default`

Vec operations

Getting mutable/immutable references: easy!

```
let mut v = vec![1, 2, 3, 4, 5];
v.push(6);
v.push(7);
v.pop();
println!("{:?}", v);

let x: &u32 = v.get(0).unwrap();

let x: &mut u32 =
v.get_mut(0).unwrap();
*x = 10;
```

Why a macro?

Remove last element

Returns an `Option`

Vec operations

Moving is a “zero-cost” operation

```
let s1: String = String::from("hello");
let s2: String = s1;
// access to `s1` is invalidated by compiler
```

How do we move out of an index?

```
let v: Vec<String> = ...;

let first: String = vec.????(0);
// access to vec[0] is invalidate by compiler??
```

Vec operations

Moving is a "zero-cost" operation

```
let s1: String = String::from("hello");
let s2: String = s1;
// access to `s1` is invalidated by compiler
```

How do we move out of an index?

```
let v: Vec<String> = ...;

let first: String = vec.????(0);
// access to vec[0] is invalidate by compiler??
```

Options:

1. `v.get(0).unwrap().clone();`
2. `v.remove(0);`

Bottom line: can't move out of vector without modifying data structure

HashMap

```rust
let mut map = HashMap::new();
map.insert(1905, String::from("Rust Programming"));
map.insert(3200, String::from("Introduction to Algorithms"));

let course_name = map.get(&1905).unwrap();
course_name.push('🦀');
```

Insertion and removal: by *move* (takes ownership)

Lookup: by *reference* (does not take ownership)

HashMap

```
let mut map = HashMap::new();
map.insert(1905, String::from("Rust Programming"));
map.insert(3200, String::from("Introduction to Algorithms"));

let course_name = map.get(&1905).unwrap();
course_name.push('🦀');
```

Insertion and removal: by *move* (takes ownership)

Lookup: by *reference* (does not take ownership)

Don't be fooled!
`Vec` and `HashMap` don't have type annotations but can only hold values of a single type

HashMap

```
let mut map = HashMap::new();
map.insert(1905, String::from("Rust Programming"));
map.insert(3200, String::from("Introduction to Algorithms"));

let course_name = map.get_mut(&1905).unwrap();
course_name.push('🦀');
```

Insertion and removal: by *move* (takes ownership)

Lookup: by *reference* (does not take ownership)

Why doesn't `course_name` need to be `mut`?

Histograms

```
let grades = vec!['A', 'B', /* grades... */];
let mut histogram = HashMap::new();
```

Histograms

```
let grades = vec!['A', 'B', /* grades... */];
let mut histogram = HashMap::new();
for grade in grades {
    if histogram.contains_key(&grade) {
        *histogram.get_mut(&grade).unwrap() +=
1;
    } else {
        histogram.insert(grade, 1);
    }
}
```

Boo! What do you not like?

Entry API

```
let grades = vec!['A', 'B', /* grades...
*/];
let mut histogram = HashMap::new();
for grade in grades {
  *histogram.entry(grade).or_insert(0) += 1;
}
```

Entry API easily handles missing values

Does this program compile?

```
fn main() {
 let mut h = HashMap::new();
 h.insert("k1", 0);
 let v1 = &h["k1"];
 h.insert("k2", 1);
 let v2 = &h["k2"];
 println!("{} {}", v1, v2);
}
```

Recall

- Ownership
- Mutable/Immutable reference rules

Does this program compile?

```
fn main() {
 let mut h = HashMap::new();
 h.insert("k1", 0);
 let v1 = &h["k1"];
 h.insert("k2", 1);
 let v2 = &h["k2"];
 println!("{} {}", v1, v2);
}
```

Recall

- Ownership
- Mutable/Immutable reference rules

Mutate h while immutable reference exists

Strings

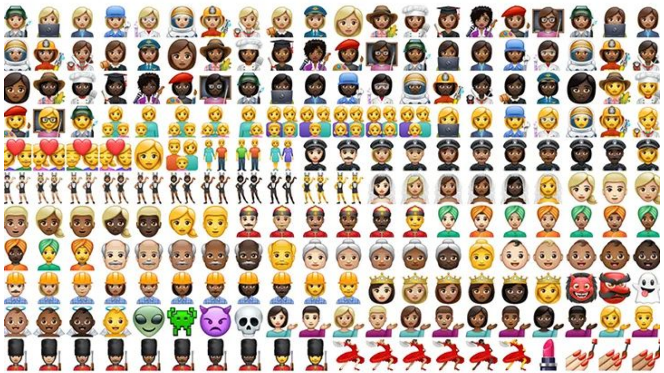
What’s a character?

`char` is 1 byte (8 bits), right?

- 2^8=256 possible values

So a `String` is just `Vec<char>`?

Where do all these characters come from?



C character encoding: ASCII

This is where “a character is 1 byte” comes from

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

- Control characters (newline, tab)
- punctuation
- alphanumeric

Moving on from ASCII

How to support all languages and characters?

- Just make `char` 32 bits?
  - All text is 4x bigger in memory
  - Not backwards compatible

ASCII only uses 7 bits

- Most significant bit is always `0` in ASCII chars

`A`	0100 0001
`[`	0101 1011
`\n`	0000 1010
`6`	0011 0110

Variable length encoding

A `char` is 1 to 4 bytes

When starting to read a string, how long is the first character?

Look at the first byte

`0xxxxxxx`	0 bytes follow
`110xxxxx`	1 byte follows
`1110xxxx`	2 bytes follow
`11110xxx`	3 bytes follow

Example four-byte character

😊

`11110000 10011111 10011000 10001010`

four-byte character

continuation sequence

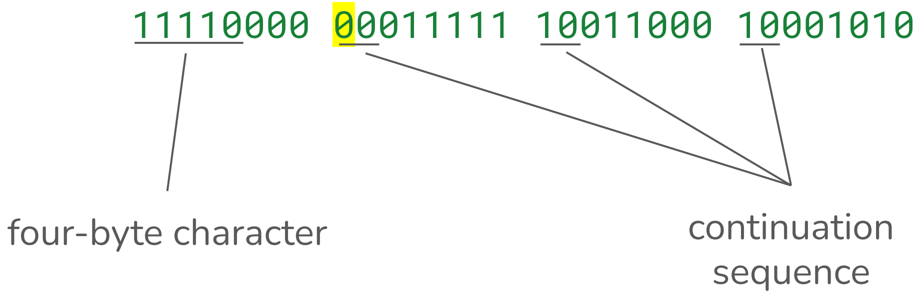
“UTF-8 encoding”

Some byte sequences are not valid UTF-8

Invalid! Multi-byte character with trailing bytes that lack the continuation sequence

Example four-byte character

😊



`Vec<u8>` vs. `String`?

`Vec<u8>`

- 0 or more bytes of data contiguous in memory
- growable

`String`

- 0 or more bytes of data contiguous in memory
- growable
- Always valid UTF-8

How is this maintained?

Assume-guarantee invariants

Creation methods:

Impossible to create `String` that doesn't satisfy invariant

- `String::new()`
- `String::from(&str)`

Modifying methods:

assume invariant holds at entry -> **guarantee** it still holds at return

- `String::push(&mut self, char)`
- `String::remove(&mut self, usize)`

Two kinds of invariants

Type invariants

Example: a value of type `Option<u32>` will always be either `None` or a `u32`

Enforced by: compiler

Assume-guarantee invariants

Example: a value of type `String` always represents valid UTF-8

Enforced by: library developer

Finally: `String`

Supports most operations you would expect

Creation

- `String::new()`
- `String::from(&str)`

Modification

- `String::push(&mut self, char)`
- `String::push_str(&mut self, &str)`
- `String::remove(&mut self, usize)`

Reading

- `String::len(&self)`

Does not support indexing—why?

Instead, use

- `.chars().nth(i)`
- `.bytes().nth(i)`

Iterators

Looping over a Vec

As we've seen:

```
let v = vec![1, 2, 3];

for element in v {
    println!("{}", element);
}
```

But wait! Looping is complicated

Looping over a Vec

```
for element in v.iter() {
    // element: &T
}
```

Iterate by reference

- Read only element access

```
for element in v.iter_mut() {
    // element: &mut T
}
```

Iterate by mutable reference

- Read/write element access

```
for element in v.into_iter() {
    // element: T
}
```

Iterate by consumption

- Can’t use v afterwards

Iterator adaptors

```
for (i, element) in v.iter().enumerate()
{
    println!("{}: {}", i, element);
}
```

Iterate over values *and indices*

```
for (x, y) in v1.iter().zip(v2.iter()) {
    println!("({}, {})", x, y);
}
```

Iterate over *two vectors simultaneously*

```
for x in v1.iter().chain(v2.iter()) {
    println!("{}", x);
}
```

Iterate over *two vectors sequentially*

Iterators are a trait

```rust
let s = String::new();
for char in s.chars() {
    // ...
}

let map = HashMap::new();
for (k, v) in map {
    // ...
}

for v in map.values() {
    // ...
}
```

Can loop over anything that implements `Iterator`

```rust
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```