Алгебраические типы данных и обработка ошибок

Владимир Латыпов

29-09-2023

Спецкурс ФТШ

План занятия

- Структуры, перечисления (enum) — классика ООП
- Представление в памяти (`struct`, `enum`, `Vec`, `String`)
- Специфичные для Rust возможности (pattern matching, derive)
- Немного про обработку ошибок
- Сеанс лайв-кодинга
- [Не успеем] „Глубокое“ осмысления с точки зрения теории типов и комбинаторики

Малая мотивация

> Двухступенчатое погружение (с какой стороны зайти?)

Малая мотивация

```
fn user_should_be_banned(???) → bool { // ← ???
  true // TODO: add ChatGPT
}
```

Что принимать в качестве аргументов?

Малая мотивация

```
1 fn user_should_be_banned(first_name: &String) → bool {
2   // We need more info for such a serious decision!
3   ???
4 }
```

Малая мотивация

```rust
fn user_should_be_banned(first_name: &String, last_name: &String,
school_number: u32, marks: &Vec<f64>, tg_nickname: &String…) →
bool {
  true // TODO: add ChatGPT
}
```

Можно перечислить *все* данные, которые могут нам понадобиться…

- Много писать
- Изменение всех функций при добавлении
- …
- *Абстракции в коде не соответствуют абстракциям в голове*

Малая мотивация

Если бы можно было написать вот так...

```
fn user_should_be_banned(user: &User) → bool {
  true // TODO: add ChatGPT
}
```

Есть такая ~~партия~~ концепция!

Синтаксис структур
Создание (объявление) структуры

```
1 struct User {
2   name: String,
3   school_number: u32,
4   online: bool, // Для композируемости запятую можно
  написать и в конце
5 }
```

Название писать сюда (в `PascalCase`’е)

Синтаксис структур
Создание (объявление) структуры

```
1 struct User {
2   name: String,
3   school_number: u32,
4   online: bool, // Для композируемости запятую можно
  написать и в конце
5 }
```

Поля структуры. Теперь все объекты типа `User` будут *хранить*

- имя
- номер школы
- наличие в сети

Использование структуры

```
1 fn user_should_be_banned(user: &User) → bool {
2   user.school_number == 239 // TODO: add ChatGPT
3 }
```

При объявлении функции указать, что тип аргумента: `&User`.

Использование структуры

```
1 fn user_should_be_banned(user: &User) → bool {
2   user.school_number == 239 // TODO: add ChatGPT
3 }
```

Доступ к полям структуры

Полноценный тип

Создали полноценный тип, как и `u32`, `String`, `Vec`, ...

```
struct SocialNetwork {
  users: Vec<User>,
  online_user_count: usize,
  …
}
```

Добавление ассоциированной функциональности (aka методы)

То, чего так ждали любители ООП

```
impl User {
  fn introduce_self(&self) → String {
    format!("{} from {}", self.name, self.school_number)
  }

  fn … // Some methods
}
```

Разные варианты принятия

- `&self` — сокращение для `self: &Self` (по ссылке)

Разные варианты принятия

- `&self` — сокращение для `self: &Self` (по ссылке)
- `&mut self` — по мутабельной ссылке

```
1 impl User {
2   fn use(&mut self) {
3     // There are only two industries that refer to customers as
  „users“
4     self.name = "Tequila Sunset".into()
5   }
6 }
```

Разные варианты принятия

- `&self` — сокращение для `self: &Self` (по ссылке)
- `&mut self` — по мутабельной ссылке
- `self` — по значению (consuming)
  - Гарантии системы типов:
    ```
    fn accept_university_payments(
          enrolled: Schoolboy, payment: Money
        ) → Undergraduate
    ```
  - Выпотрашивание ресурсов:
    ```
    fn use_jew(jew: Person) → (Wallet, Shoes, Watches, …)
    ```

Вызов методов (`dot` operator)

- Можно вызывать вот так:
  `let vasya_introduction = User::introduce_self(&vasya);` ← как namespace C++ или получение ссылки на метод в Java.
- Удобнее: `vasya.introduce_self()`
- Автоматически берёт столько ссылок (~~и Deref-ов~~), сколько требуется «чтобы работало» (см. проробнее по ссылке).

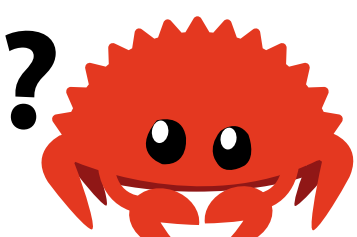
Ассоциированные константы

- Определяются в `impl` блоках

```
impl AntiSpamBot {
  // Use SCREAMING_SNAKE_CASE
  const FILTERING_SCHOOL: usize = 239;
}
```

- Иногда стоит *параметризовать* бота значением, если это не *константа*
- Использовать извне как `AntiSpamBot::FILTERING_SCHOOL`.

Вопрос на заСЫПку



Чего же не хватает?

Создание структуры

(проверить, что вы не заснули)

Есть ровно один способ создать значение структуры

```
let kms = User {
  name: "Константин".into(),
  school_number: 566,
  online: true
};
```

Сахар для инстанциирования

- Сокращение (*сахар*), чтобы взять в новую структуру все поля, кроме указанных из другого объекта:

```
let aa = User {
  name: "Анна".into(),
   ..kms
};
```

Сахар для инстанциирования

- Сокращение (*сахар*), чтобы взять в новую структуру все поля, кроме указанных из другого объекта:

```
let aa = User {
  name: "Анна".into(),
   ..kms
};
```

- Если название переменной — как у поля

```
fn generate_user() → User {
  let name = random_name();
  let school_numbner = random_school();
  User { name, school_numbner, online: true }
}
```

enum-ы

- А что, если «МИЛОРД ЛЮДИ ЖАЛУЮТСЯ»?...

```
1 fn user_should_be_banned(&self) → ??? {
2   if user.school_number == 239 {
3     // the user is banned because of incorrect school
  number
4   } else if self.name.len() >= 256 { // Should be
  windows-friendly
5     // Banned because name is too long
6   }
7 }
```

`enum`-ы

- А что, если «МИЛОРД ЛЮДИ ЖАЛУЮТСЯ»?...

```
1 fn user_should_be_banned(&self) → ??? {
2   if user.school_number == 239 {
3     // the user is banned because of incorrect school
  number
4   } else if self.name.len() >= 256 { // Should be
  windows-friendly
5     // Banned because name is too long
6   }
7 }
```

Объявление `enum`-ов

- Может иметь один вид значения из списка (*enum*'eration — перечисление)

```
enum BanReason {
  NameTooLong,
  BadGateway,
  BadSchoolNumber(u32),
  AdminDisapproval {
    comment: String,
    disapproval_date: Date
  }
}
```

Объявление `enum`-ов

- Может иметь один вид значения из списка (*enum*'eration — перечисление)
- Каждый вид значений помечен меткой (*enum discriminant*)

```
1 enum BanReason {
2   NameTooLong,
3   BadGateway,
4   BadSchoolNumber(u32),
5   AdminDisapproval {
6     comment: String,
7     disapproval_date: Date
8   }
9 }
```

Объявление `enum`-ов

- Может иметь один вид значения из списка (*enum*'eration — перечисление)
- Каждый вид значений помечен меткой (*enum discriminant*)
- Значения какого-то вида может быть просто меткой: `BanReason::NameTooLong`

```
1 enum BanReason {
2   NameTooLong,
3   BadGateway,
4   BadSchoolNumber(u32),
5   AdminDisapproval {
6     comment: String,
7     disapproval_date: Date
8   }
9 }
```

Объявление `enum`-ов

- Может иметь один вид значения из списка (*enum*'eration — перечисление)
- Каждый вид значений помечен меткой (*enum discriminant*)
- Значения какого-то вида может быть просто меткой: `BanReason::NameTooLong`
- Кроме метки ещё могут быть свои данные для каждого вида: `BanReason::BadSchoolNumber(239)`

```
1 enum BanReason {
2   NameTooLong,
3   BadGateway,
4   BadSchoolNumber(u32),
5   AdminDisapproval {
6     comment: String,
7     disapproval_date: Date
8   }
9 }
```

Возврат `enum`

```
fn user_should_be_banned(&self) → BanReason {
  if user.school_number == 239 {
    BanReason::BadSchoolNumber(239)
  } else if self.name.len() >= 256 { // Should be windows-
friendly
    BanReason::NameTooLong
  }
}
```

Pattern-matching

pattern — шаблон

Как работать с элементами `enum`-eration'ов?

Компилятор проверяет, exhaustive ли

```
impl ChatBot {
  fn handle_user(&self, user: &User) {
    match user.user_should_be_banned() {
      BanReason::NameTooLong ⇒ self.send_message("Shorten you name!"),
      BanReason::BadSchoolNumber(school) ⇒ {
        self.send_message(format!("You study in school {school}…"));
        self.send_message("Fix this and retry!");
```

```
8           },
9           _ ⇒ (), // can use trailing commas
10      }
11  }
12 }
```

Мощь pattern-matching'а

match expression (arms have same types)

deep nesting (pattern, not value!)

```
impl Expression {
  fn simplify(&self) {
    match self {
      Expression::Add(
        Expression::Const(l),
        Expression::Const(r)
      ) ⇒ Expression::Const(l + r),
      complex ⇒ complex // leave unchanged
    }
  }
}
```

Паттерны на все случаи жизни

Деструктурирование *irrefutable* паттернов

- `let (a, b) = (b, a);`
- `let (a, b) = get_a_and_b();`
- `fn takes_struct(User { name: user_name }: User)`

Advanced match arms

```
1 match reason {
2   SchoolNumber(239 | 30) ⇒ "hmm", // «ИЛИ» на паттернах
3   SchoolNumber(n) if n < 10 ⇒ { // guards (логическое
  выражение)
4     println!("{n}"); // statement
5     "strange school"
6   },
7   AdminDisapproval{
8     comment: "Rude",
9     date: when_banned @ Date { month: September, .. }
10  } ⇒ format!("Understandable: {when_banned}"), // binding
  (присвоение части паттерна)
11  _ ⇒ "БСК" // No comments/Бред сивой кобылы
12 }
```

`ref` в паттернах

- пишется перед новой переменной в паттерне
- `borrow`-ит (заимствует), а *не* `consume`-ит (поглощает, берёт по значению)

```
match reason {
  AdminDisapproval{ref comment, ..} ⇒ println!("{}", comment),
  BadSchoolNumber(ref mut num @ (1 | 0)) ⇒ *num += 1,
  _ ⇒ {} // same as ()
}
// Still can use reason!
```

Представление в памяти

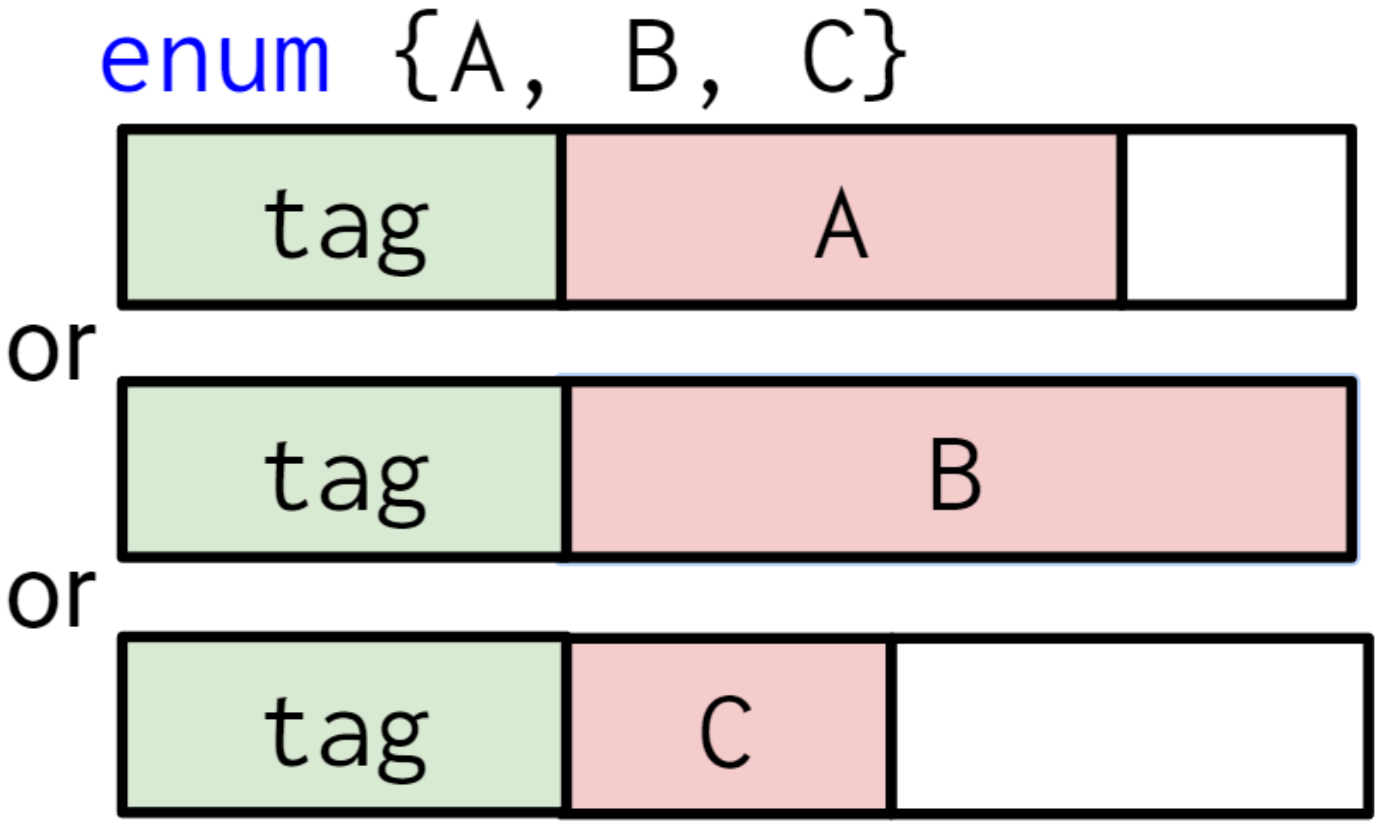
```
1 struct CompositeStruct {
2   a: A,
3   b: B,
4   c: C
5 }
```

- Сразу все варианты
- 

[1]

- Занимает $\text{sizeof}(A) + \text{sizeof}(B) + \text{sizeof}(C)$

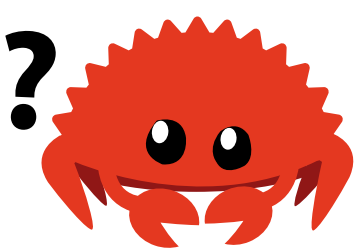
[1] С поправкой на оптимизации



- Только один из вариантов
- Занимает $\max(\text{sizeof}(A), \text{sizeof}(B), \text{sizeof}(\text{C})) + \frac{\log_2(3)}{8}$

Ликбез по реализации Vec

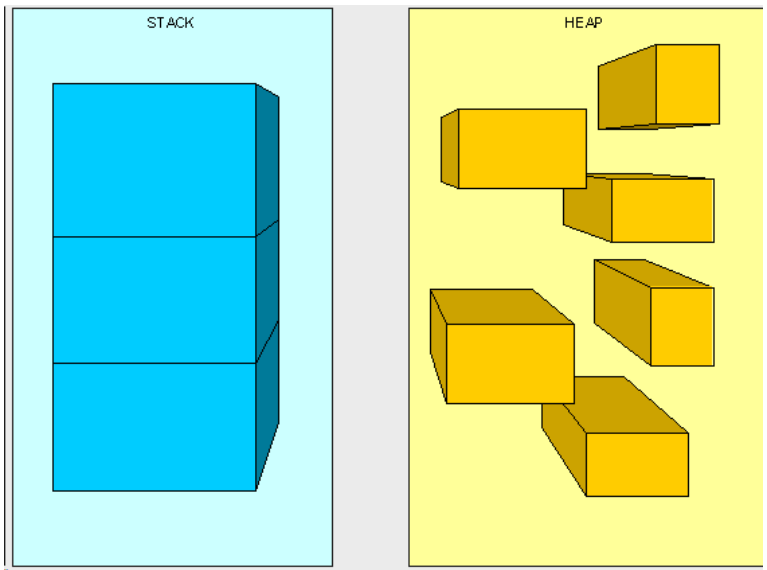
- Массивы в Rust
  - `let a: [u32] = [1, 2, 3]` — просто массив
  - `let b: &[u32] = &a // slice`
  - `let b: Vec<u32> = vec![1, 2, 3]; // vec! is a macro for Vec`



В чём принципиальное отличие Vec от массивов и slice-ов?

Стек и куча

- Стек и куча (подробнее — в 9-й лекции про `unsafe`)
- У объектов есть *заголовок*
- …фиксированного размера (на этапе *компиляции*)
- В случае `struct`/`enum` — из заголовков его полей
- `let a: u32 = 1;` содаёт объект на стеке
- Для меняющегося размера — куча
- Заголовок может *указывать* туда [схема на доске]



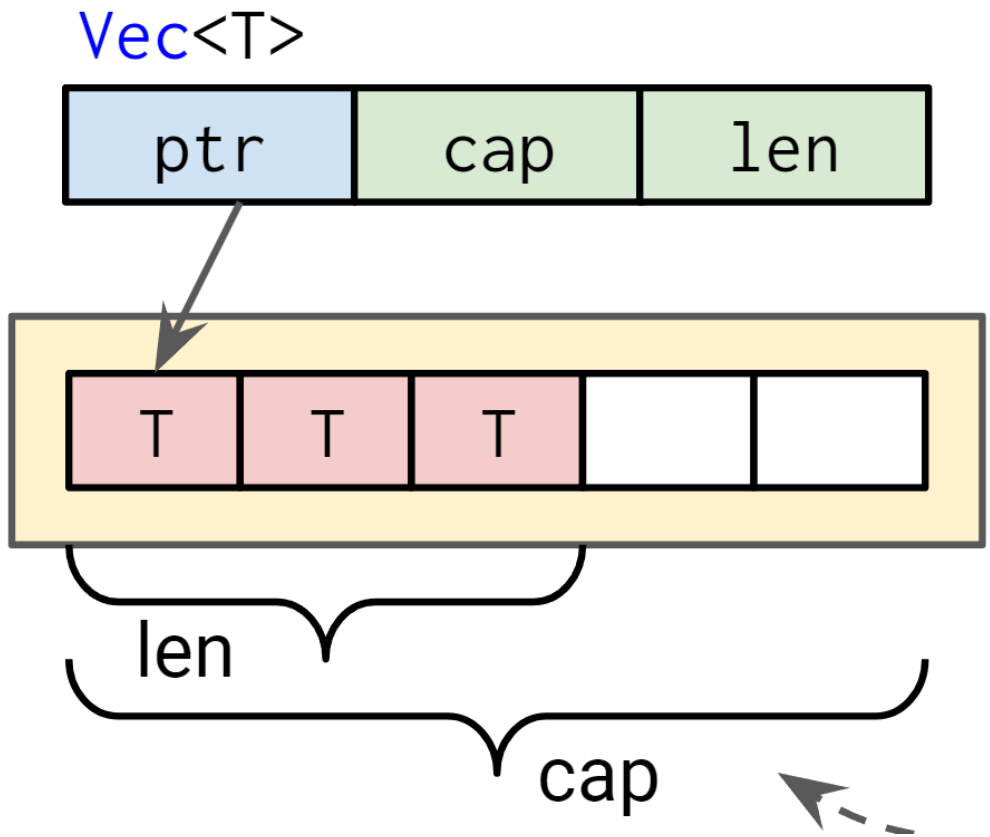
Устройство `Vec`

Заголовок хранит:

- Указатель на область памяти
- Длину (количество объектов)
- Вместимость (размер выделенной памяти)

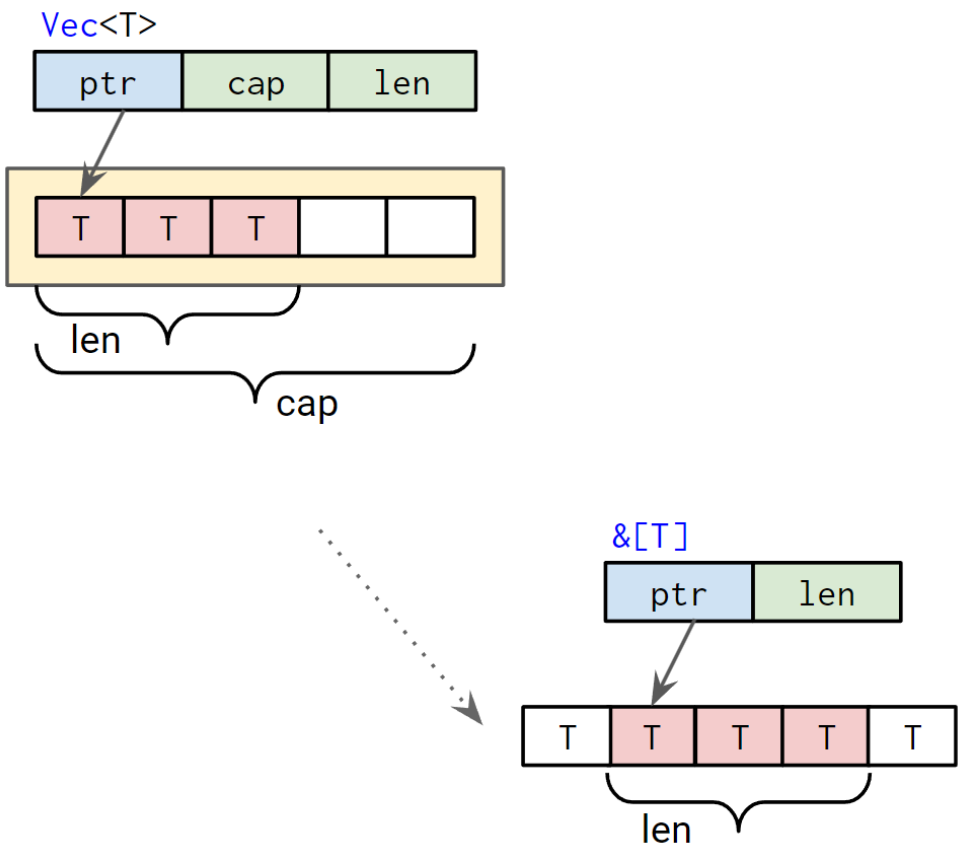
Если не хватает памяти

- выделяет новую (размера в 2 раза больше)
- переносит туда все предыдущие элементы



Паттерн: принимать диапазоны памяти как slice-ы

- `Vec<T>` умеет «автоматически» превращаться в `&[T]` (но умеет и другое!)
- Другой контейнер — тоже
- ⟶ если функция не использует над аргументом методы, специфичные для Vec (связанные с переаллокацией), стоит принимать вектор как `&[T]` или `&mut [T]`.

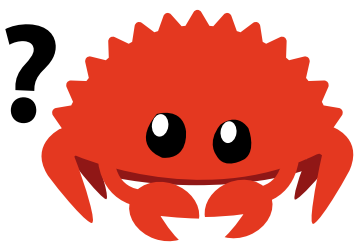


Пример

```rust
fn evenize_array(array: &mut [u32]) {
  for e in array {
    if *e % 2 == 1 {
      *e += 1;
    }
  }
}

fn main {
  let mut array: Vec<i32> = vec![1, 2, 3];
  evenize_array(array); // Auto referencve + deref
}
```

Shallow и deep copy



Чем они отличаются?

[2] С поправкой на какие-нибудь `Rc<RefCell<>>` (interior mutability)

Shallow и deep copy

- *Shallow copy* (поверхностное копирование)
  - копирует только заголовок
  - может привести к неожиданным последствиям [см. на доске]
  - в Rust есть только *shallow move*

[3] С поправкой на какие-нибудь `Rc<RefCell<>>` (interior mutability)

Shallow и deep copy

- *Shallow copy* (поверхностное копирование)
  - копирует только заголовок
  - может привести к неожиданным последствиям [см. на доске]
  - в Rust есть только *shallow move*
- *Deep copy* (глубокое копирование)
  - копирует сам объект, тех, на кого он ссылается и т.д.
  - ⟶ создаёт новый полноценный объект, независимый[4] от старого.
  - в Rust для этого есть метод `.clone()` (можно за-`derive`-ить)

[4] С поправкой на какие-нибудь `Rc<RefCell<>>` (interior mutability)

Clone

```
1 let a = vec![1, 2, 3];
2 use_a_by_value(a);
3 use_a_by_value_again(a);
```

Если вы попробуете написать такой код, то он не скомпилируется, borrow checker ее пропустит. Действительно вы передали `a` по значению в функцию, поэтому больше не можете его использовать.

Можно вторую строчку заменить на `use_a_by_value(a.clone());`, тогда функция потребит клона `a`, а не его самого. Но это может работать долго для больших векторов (например, если их так миллион), поэтому хорошо, что вам потребуется явно написать «`a.clone()`» — вы не произведёте дорогую операцию *случайно*.

Copy

Однако, если вы работаете с типом `i32`, который целиком состоит из заголовка, довольно удобно было бы, чтобы он копировался автоматически, когда надо, и borrow checker не заставлял бы нас писать `.clone()` каждый раз.

И действительно, для этого нужно в списке `derive` (см. ниже) перечислить не только `Clone`, но ещё и `Copy`. Это подскажет компилатору, что для выполнения `Clone` *достаточно скопировать заголовок* типа ⟶ копирование будет быстрым, и его можно делать автоматически где ни попадя.

Если вы попробуете указать этот аттрибут на типе с не-Clone полями, произойдёт ошибка компиляции.

Строки

- Несколько типов строк
- `String`: как `Vec`, но
  - unicode
  - методы строк
- `&str`
  - аналог `&[u8]` с обоими свойствами
  - строковые *литералы* имеют тип `&str`
- Паттерн про приём `&str` сохраняется

For this type in C...	...use this type in Rust
`char *`	`&str`
`char *`	`String`
`char *`	`&[u8]`
`char *`	`&[u8; N]`
`char *`	`Vec<u8>`
`char *`	`&u8`
`char *`	`OsStr`
`char *`	`OsString`
`char *`	`Path`
`char *`	`PathBuf`
`char *`	`CStr`
`char *`	`CString`
`char *`	`&'static str`

`#derive`-макросы

- Average Java[5]

```
class Point {
  private int x;
  private int y;

  Point (int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
```

[5] Конечно, на Java тоже можно автоматизировать подобное через Reflection или Records, но это не так круто))

```
public String toString() {
  return "Point { x=" + x + ", y=" + x + " }";
}
public Point add(Point other) {
  return new Point(this.x + other.x, this.y + other.y);
}

@Override
public bool equals(Object other) {
  // TODO
}

@Override
public int hashCode(Object other) {
```

```
    // TODO
  }



  // a lot of stuff to make it usable
}
```

Видите, где опечатка?

`#derive`-макросы

- Программисты автоматизируют, что возможно

`#derive`-макросы

- Программисты автоматизируют, что возможно
- Процедурные макросы: запускаем некий код во время компиляции для изменение программы

`#derive`-макросы

- Программисты автоматизируют, что возможно
- Процедурные макросы: запускаем некий код во время компиляции для изменение программы

```
#[derive(Eq, PartialEq, Hash, Debug, Clone, Copy)] // Copy shows that shallow copy is enough for clone (there are no references to heap)
struct Point {
  x: i32,
  y: i32
}
```

`#derive`-макросы

- Программисты автоматизируют, что возможно
- Процедурные макросы: запускаем некий код во время компиляции для изменение программы

```
#[derive(Eq, PartialEq, Hash, Debug, Clone, Copy)] // Copy shows that shallow copy is enough for clone (there are no references to heap)
struct Point {
  x: i32,
  y: i32
}
```

- Превращается в ≈такой же код, как на Java

`#derive`-макросы

- Программисты автоматизируют, что возможно
- Процедурные макросы: запускаем некий код во время компиляции для изменение программы

```
#[derive(Eq, PartialEq, Hash, Debug, Clone, Copy)] // Copy shows that shallow copy is enough for clone (there are no references to heap)
struct Point {
  x: i32,
  y: i32
}
```

- Превращается в ≈такой же код, как на Java
- `derive_more` — одна из библиотек для `derive`-ов: Add, New, ...

`#derive`-макросы

- Программисты автоматизируют, что возможно
- Процедурные макросы: запускаем некий код во время компиляции для изменение программы

```
#[derive(Eq, PartialEq, Hash, Debug, Clone, Copy)] // Copy shows that shallow copy is enough for clone (there are no references to heap)
struct Point {
  x: i32,
  y: i32
}
```

- Превращается в ≈такой же код, как на Java
- `derive_more` — одна из библиотек для `derive`-ов: Add, New, ...

- Работает и для enum-ов, когда уместно

`#derive`-макросы

- Программисты автоматизируют, что возможно
- Процедурные макросы: запускаем некий код во время компиляции для изменение программы

```
#[derive(Eq, PartialEq, Hash, Debug, Clone, Copy)] // Copy shows that shallow copy is enough for clone (there are no references to heap)
struct Point {
  x: i32,
  y: i32
}
```

- Превращается в ≈такой же код, как на Java
- `derive_more` — одна из библиотек для `derive`-ов: Add, New, ...

- Работает и для enum-ов, когда уместно
- Можете написать свои!

Misc

Tuple structs

- `struct SomeStruct(i32, User, Vec<User>)`
- `let s = SomeStruct(1, vasya, vec![ivan, vova])`
- `s.0`, `s.1`, `s.2`
- Что значат эти поля?
- Обычно лучше использовать структуры
- Часто используют для паттерна `newtype`

Алиасы

- Позволяет задавать псевдонимы типам

Алиасы

- Позволяет задавать псевдонимы типам

```
type TheUsers = Vec<NaughtyUsersThatBehaveBadly>
```

Алиасы

- Позволяет задавать псевдонимы типам

```
type TheUsers = Vec<NaughtyUsersThatBehaveBadly>
```

- Это всё тот же тип (в отличие от tuple structs)

Алиасы

- Позволяет задавать псевдонимы типам

```
type TheUsers = Vec<NaughtyUsersThatBehaveBadly>
```

- Это всё тот же тип (в отличие от tuple structs)
- Особо полезно для продвинутых фич, чтобы меньше печатать

Алиасы

- Позволяет задавать псевдонимы типам

```
type TheUsers = Vec<NaughtyUsersThatBehaveBadly>
```

- Это всё тот же тип (в отличие от tuple structs)
- Особо полезно для продвинутых фич, чтобы меньше печатать

```
type Users = Arc<Mutex<Vec<Cell<HashMap<usize, User>>>>>
```

Обработка ошибок

- Если всё пошло совсем не так и ничего не сделать: `panic!("Buy!")` ← ваша программа завершится[6], напечатав *stacktrace*.
- Но обычно, если что-то не получилось, это не так страшно! Например, вы не смогли распарсить число во вводе пользователя. Для этого используют другой тип ошибок...

[6] На самом деле, «панику» можно «поймать», но обычно так не делают

`Option` и `Result`

В `std` есть

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

Либо успех, либо `None`

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

Либо успех, либо информация об ошибке

Обработка ошибок через `enum`-ы

- Полезно, когда известен список ошибок, которые могут произойти в фукнции
- Каждый раз — какая-то одна (*один из вариантов*)
- ⟶ enum

```
fn register_user(&self, name: &str)
  → Result<User, BanReason> { … }
```

Как обрабатывать ошибки

- `.unwrap()` — получить значение или программа упадёт
- `match` — как и с любыми `enum`-ами
- Комбинаторы - расскажем про них позже
- Вопросики

Вопросики

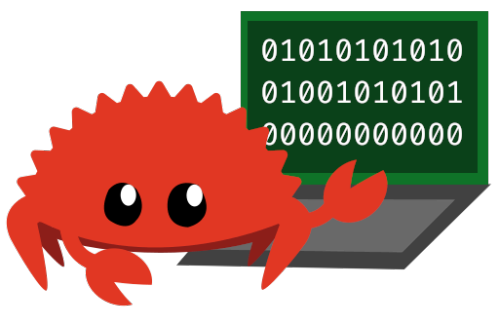
- Поддержка обработки ошибок на уровне языка
- В функциях, возвращающих `Option` или `Result`
- Применяется к `Option` или `Result`, чтобы получить содержание или вернуть ошибку

```
1 fn load_textures(filename: &str) → Result<Texture, EngineError>
  {
2   let bytes = read_file(filename)?;
3   let texture = parse_texture(bytes)?;
4
5   Ok(texture)
6 }
```

Обработка (ошибок?)

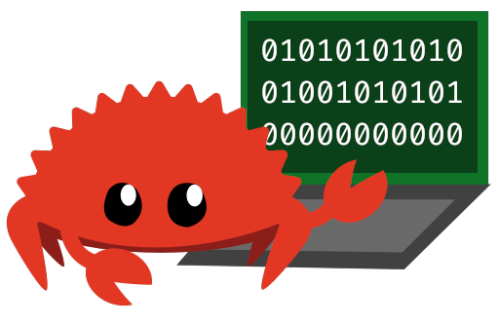
- Что вообще такое ошибка
- Что вообще такое основной сценарий исполнения кода? Пример с валидатором и мартышкой, пишущей войну и мир.

Сеанс кодирования по-живому



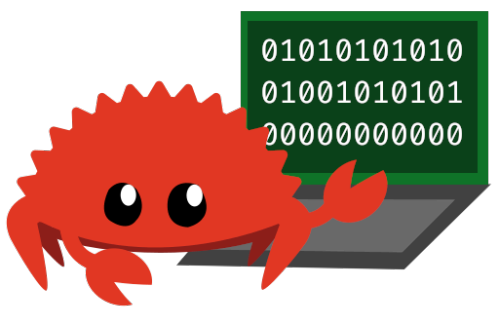
Убейте в себе C++! C++ — это как грязный, ржавый спидозный гвозь, который застрял в ступне!

Сеанс кодирования по-живому



~~Убейте в себе C++! C++ — это как грязный, ржавый спидозный гвозь, который застрял в ступне!~~

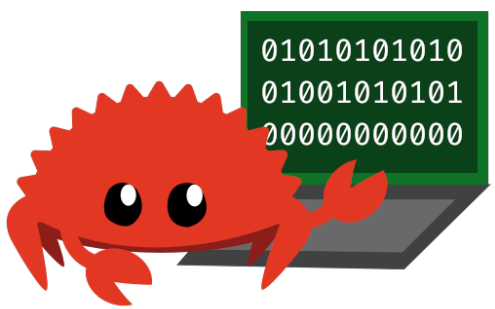
Сеанс кодирования по-живому



...не то кодирование

Сеанс кодирования по-живому

Напишем абстракцию для динамического вычисления сумм на отрезках с обработкой ошибок



Доктрина: Максимум активности от аудитории

Правила

- Лектор превращается в пишущую машинку, марионетку в руках аудитории
- Нет чёткого ТЗ, чтобы сами думали (ТЗ будет в ДЗ)
- Если будет хорошо получаться, буду троллить, иначе - давать глубокомысленные советы