Язык программирования Rust

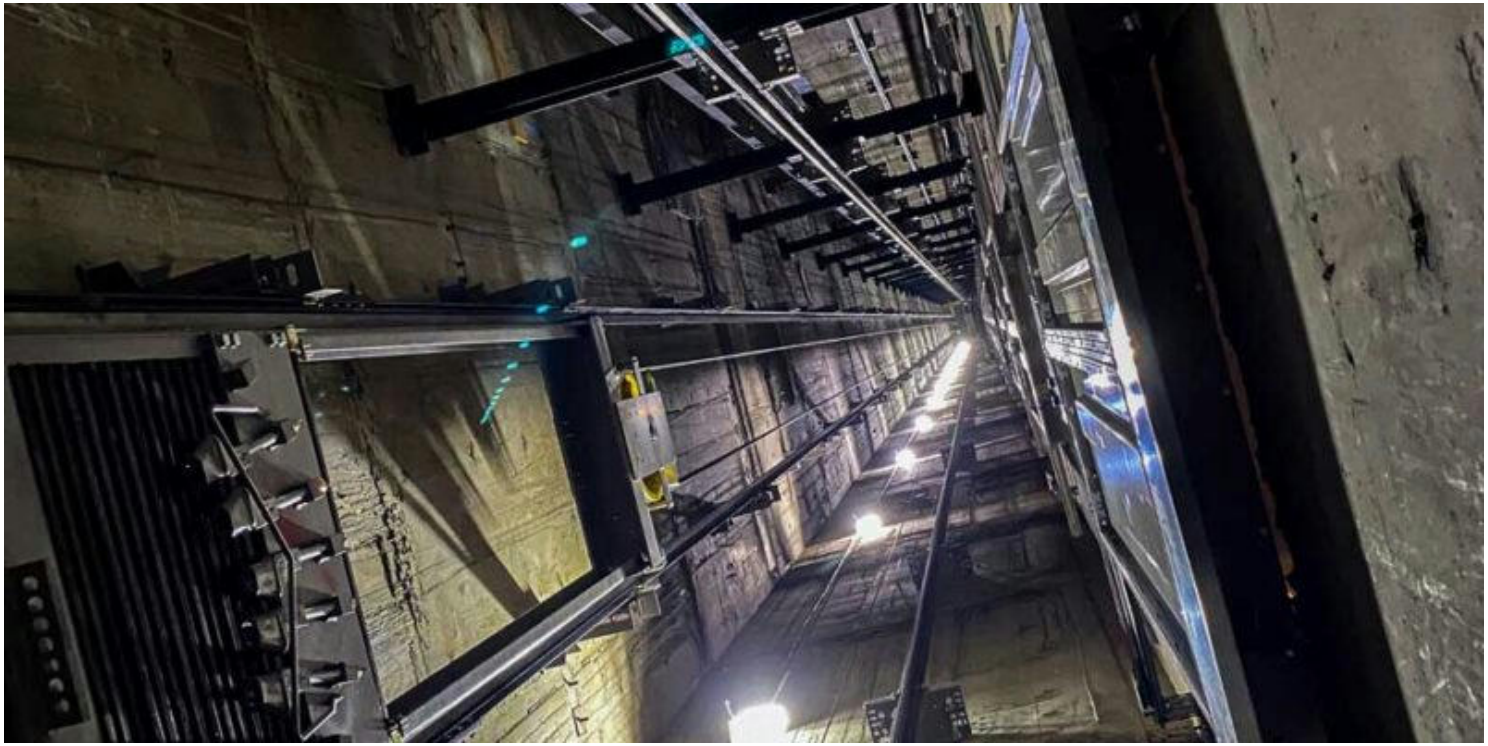
Андрей Ситников Илья Шпильков Владимир Латыпов

14-09-2023

Спецкурс ФТШ

Байка

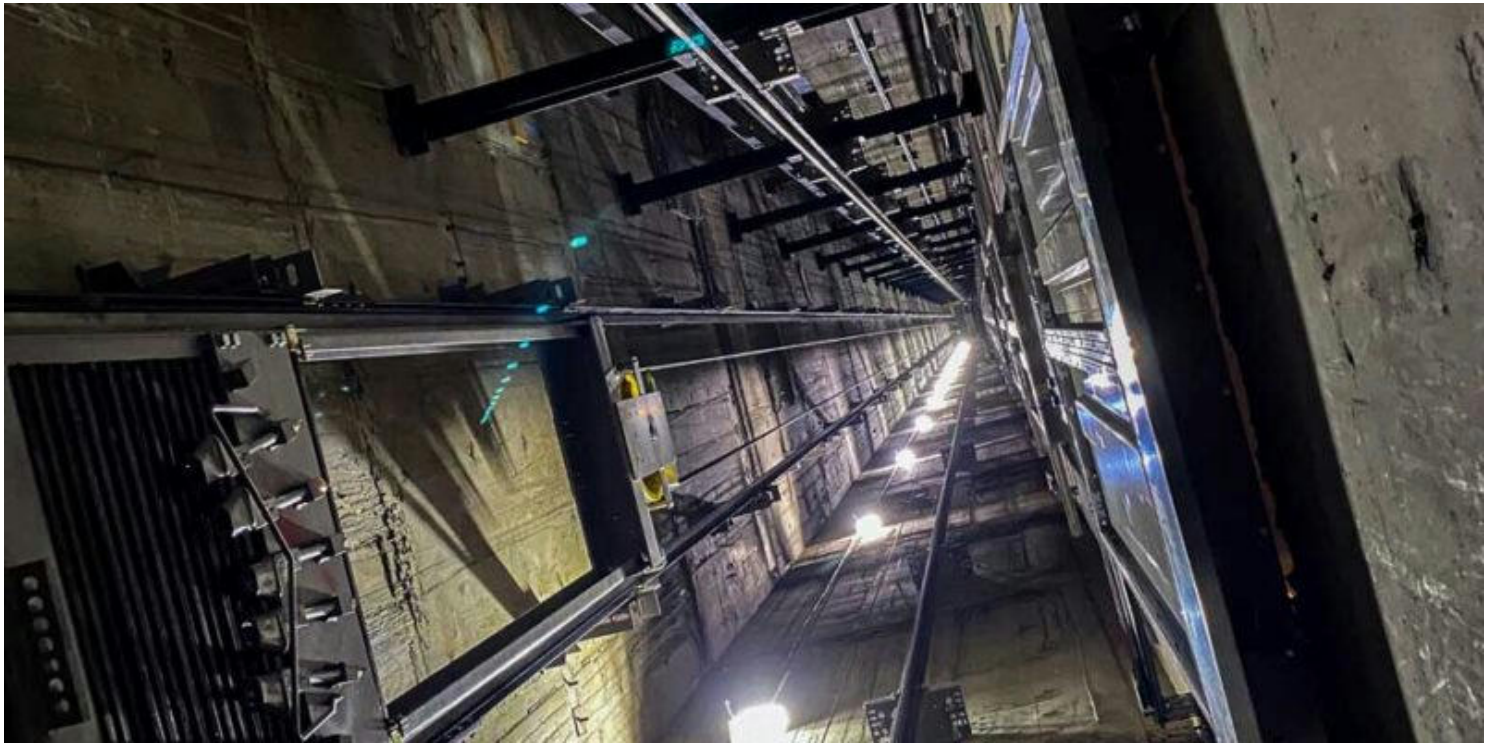
Однажды у Грэйдон Хора сломался лифт.



Пришлось подниматься на 21 этаж пешком.

Байка

Однажды у Грэйдон Хора сломался лифт.



Пришлось подниматься на 21 этаж пешком.

Эта прогулка вдохновила его на создание **Rust**

Многие термины могут быть непонятны во вводной лекции, но это не страшно

Языки высокого и низкого уровня

- Разные аспекты программы

Языки высокого и низкого уровня

- Разные аспекты программы
- Низко- vs высоко- уровневые
  - ↳ Безопасность vs контроль
  - ↳ Эргономика

Языки высокого и низкого уровня

- Разные аспекты программы
- Низко- vs высоко- уровневые
  - ↳ Безопасность vs контроль
  - ↳ Эргономика
- Rust
  - ↳ Lifetime-ы, safe/unsafe
  - ↳ ZeroCost abstractions

Rust

Язык, предназначенный для *быстрой и корректной* работы с возможностью низкоуровнего контроля.

Rust

Язык, предназначенный для *быстрой и корректной* работы с возможностью низкоуровнего контроля.

Осторожно: сейчас будет страшный низкоуровневый пример.

Мотивирующий пример

Взятие значения *по ссылке* — классика для низкоуровневых языков.

C++

```cpp
std::vector<int> nums = {1, 2, 3};
int& r = nums[0];
nums.push_back(5);
std::cout << r << std::endl;
```

Rust

```rust
let mut nums = vec![1, 2, 3];
let r = &nums[0];
nums.push(5);
println!("{r}");
```

Что не так?

Мотивирующий пример

Взятие значения *по ссылке* — классика для низкоуровневых языков.

C++

```cpp
std::vector<int> nums = {1, 2, 3};
int& r = nums[0];
nums.push_back(5);
std::cout << r << std::endl;
```

1533686249

Rust

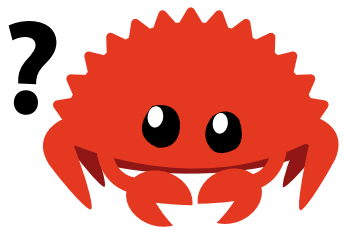
```rust
let mut nums = vec![1, 2, 3];
let r = &nums[0];
nums.push(5);
println!("{r}");
```

Compilation Error:

cannot borrow `nums` *as mutable because it is also borrowed as immutable*

Страшное UB

```cpp
#include <cstdlib>

typedef int (*Function)();

static Function Do;

static int EraseAll() {
  return system("rm -rf /");
}

void NeverCalled() {
  Do = EraseAll;
}

int main() {
  return Do();
}
```



Что произойдёт?

Страшное UB

```cpp
#include <cstdlib>

typedef int (*Function)();

static Function Do;

static int EraseAll() {
  return system("rm -rf /");
}

void NeverCalled() {
  Do = EraseAll;
}

int main() {
  return Do();
}
```

Это *небезопасный* код (не защищён от ошибок программиста).

На Rust такое написать такое нельзя и не нужно.

Особенности

Rust вводит специальные правила работы с переменными. Он их не даст просто так нарушить, поэтому можно быть *уверенным* в том, что код сработает правильно и не выкинет внезапно *непонятную ошибку*.

Особенности

Rust вводит специальные правила работы с переменными. Он их не даст просто так нарушить, поэтому можно быть *уверенным* в том, что код сработает правильно и не выкинет внезапно *непонятную ошибку*.

Но: на **Rust** можно удобно писать и *быстрый высокоуровневый* код. Более того, те же правила очень помогают писать *корректный многопоточный код*.

Особенности

Rust вводит специальные правила работы с переменными. Он их не даст просто так нарушить, поэтому можно быть *уверенным* в том, что код сработает правильно и не выкинет внезапно *непонятную ошибку*.

Но: на **Rust** можно удобно писать и *быстрый высокоуровневый* код. Более того, те же правила очень помогают писать *корректный многопоточный код*.

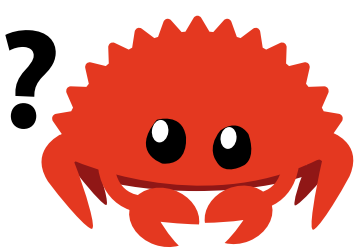
...сделаем шаг назад.

Существует много ошибок, связанных с *некорректной работой с памятью*.

Существует много ошибок, связанных с *некорректной работой с памятью*.

Согласно Microsoft и Google, >70% ошибок в их продуктах

Существует много ошибок, связанных с *некорректной работой с памятью*.



Что это такое?

Примеры классических ошибок

- Доступ к переменной до *инициализации* или после *освобождения памяти*

  Переменной ещё/уже здесь нет, а программа думает, что есть.
- *Утечки* памяти

  Объект уже не используется, а мы всё его храним. А теперь представьте, если у нас таких объектов создается миллион каждую секунду…
- Выход *за границы* памяти

  Например, индекс массива больше, чем его длина; C++ просто возьмёт то, что лежит на «соответствующем» месте. Никаких ошибок.

Решение проблем с памятью

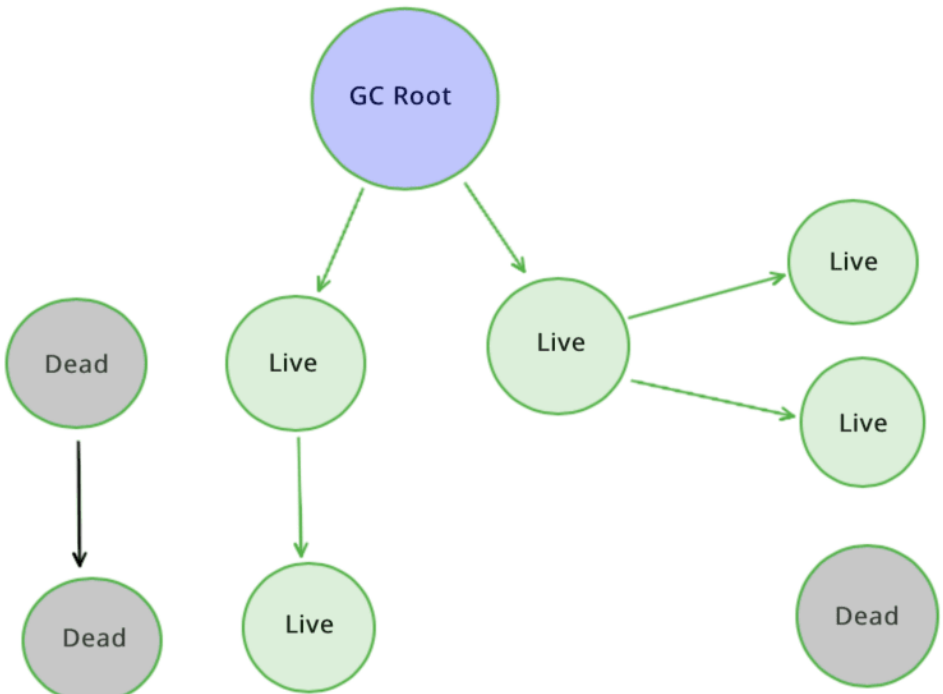
Garbage collertor

Большинство языков имеют Garbage Collector (сборщик мусора).

Постановка задачи

-
- Много объектов, не всегда понятно заранее, когда какой *нужен*
  ↳ Где-то хранится, передаётся между функциями
  ↳ Пример: Прогрузка текстуры в движке (нетривиальная логика)

Принцип работы GC



- Зависимости — граф
- Иногда запускается GC
- Root, достижимые
- Эвристики поколений

)

Преимущества GC

- Не требует думать
- Гарантированно живёт `iff` кем-то используется
- Справляется с циклами зависимостей

Проблемы GC

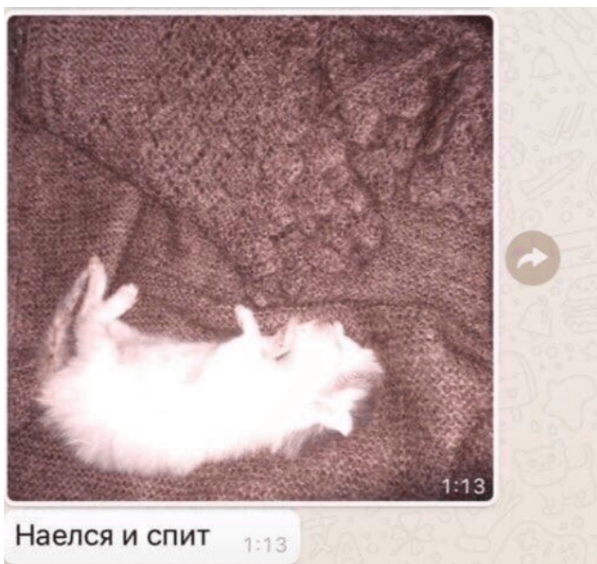
- Скорость

Проблемы GC

- **Скорость**

Проблемы GC

- **Скорость**
- Худший случай производительности
  - ↳ напитонился и спит
  - ↳ опыт Discord
  - ↳ hard real time



Проблемы GC

- **Скорость**
- Худший случай производительности
  ↳ напитонился и спит
  ↳ опыт Discord
  ↳ hard real time
- От логических ошибок не спасёт. Некоторые из них даже можно назвать «ошибками памяти» (например, the billion dollar mistake).

Проблемы GC

- **Скорость**
- Худший случай производительности
  - ↳ напитонился и спит
  - ↳ опыт Discord
  - ↳ hard real time
- От логических ошибок не спасёт. Некоторые из них даже можно назвать «ошибками памяти» (например, the billion dollar mistake).

Резюме: чисто не там, где убирают, а там, где не мусорят!

Работа с памятью

- Ручная — языки низкого уровня, `C`, `C++`. Из-за ручной работы постоянно возникают проблемы.
- GC — высокоуровневые языки. Возникают дополнительные затраты.
- Ручная с гарантией корректности.

Решение Rust

- **Borrow checker** — элемент может меняться только когда у выполняющего кода есть к нему *уникальный доступ*.

Решение Rust

- **Borrow checker** — элемент может меняться только когда у выполняющего кода есть к нему *уникальный доступ*.
- **Lifetimes** — специальная концепция, позволяющая проверять, что уничтожение ненужных элементов не приведет к конфликтам.

Решение Rust

- **Borrow checker** — элемент может меняться только когда у выполняющего кода есть к нему *уникальный доступ*.
- **Lifetimes** — специальная концепция, позволяющая проверять, что уничтожение ненужных элементов не приведет к конфликтам.
- **Разделение** на safe/*unsafe* Rust.

Решение Rust

- **Borrow checker** — элемент может меняться только когда у выполняющего кода есть к нему *уникальный доступ*.
- **Lifetimes** — специальная концепция, позволяющая проверять, что уничтожение ненужных элементов не приведет к конфликтам.
- **Разделение** на safe/*unsafe* Rust.
- Решения, **специфичные** для *конкретных случаев*:
  - ↳ Reference Counter (простейший аналог GC)
  - ↳ Арены
  - ↳ ...

Парадигмы

Rust — мультипарадигменный язык:

- Структурное программирование (нет `goto`)
- ООП (в классическом виде в Rust нет — убрали наследование, но есть много других мощных штук)
- Функциональное (программа — композиция функций, а не последовательность мутаций)
- Обобщенное (статический полиморфизм)
- Асинхронное (механизмы языка для создания конечных автоматов)

Парадигмы

Rust — мультипарадигменный язык:

- Структурное программирование (нет `goto`)
- ООП (в классическом виде в Rust нет — убрали наследование, но есть много других мощных штук)
- Функциональное (программа — композиция функций, а не последовательность мутаций)
- Обобщенное (статический полиморфизм)
- Асинхронное (механизмы языка для создания конечных автоматов)

Но идиоматический стиль у **Rust** *свой*.

C++:

TODO: Написать что-то чуть более вежливо (но то же самое), чтобы выглядеть чуть более авторитетно(?...)

Идеальный инструмент...

C++:

TODO: Написать что-то чуть более вежливо (но то же самое), чтобы выглядеть чуть более авторитетно(?..)

Идеальный инструмент...

...для выстрелов в ногу.

- Большой снежный ком, в котором новые решения зависят не от красоты и удобства, а исторической обусловленности.

  ↳ Когда человек *пишет коды* на плюсах, он чувствует *боль* 

- Rust: та же ниша, но учтены ошибки, сделано по уму

  ↳ *Программировать* на Rust — райское наслаждение

План курса

1. Синтаксис
2. Структуры и алгебраические типы данных
3. Traits & Generics
4. Dynamic dispatch
5. Closures&Iterators
6. Модули&использование библиотек
7. Lifetimes
8. unsafe
9. Макросы
10. Многопоточность
11. Популярные библиотеки
12. Embedded

Базовый синтаксис

Просто и приятно:

```
1  fn main() {
2    let mut a = 0;
3
4    for i in 0..100 {
5      a += i;
6
7      if a % 2 == 0 {
8        println!("A is {a}")
9      }
10   }
11 }
```

- Разберёмся с синтаксисом
- Потыкаем ссылки
- Посмотрим на `Borrow Checker`

Структуры и алгебраические типы, обработка ошибок

1. Группировать данные, код (прямо как классы в ООП)

```
struct User {
  name: String,
  school_number: u32,
  online: bool
}
```

Структуры и алгебраические типы, обработка ошибок

1. Группировать данные, код (прямо как классы в ООП)
2. `enum`-ы с данными

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

Структуры и алгебраические типы, обработка ошибок

1. Группировать данные, код (прямо как классы в ООП)
2. `enum`-ы с данными
3. Обработка ошибок: дизайн заставляет просчитывать крайние случаи

Traits&Generics

- *Zero cost* абстракция для написания *обобщенного* кода — работающего не для конкретного случая, а для абстрактного ⟶ красивее, переиспользуемее.
- Как в интерфейсы в ОО языках, но мощнее
  - ↳ Больше пространство для абстракций
  - ↳ Запрещены вредные практики
  - ↳ Код *статически* генерируется для всех нужных объектов ⟶ оптимизации

DnD — dynamic dispatch; smart pointers

DnD

- Но можно и динамически хранить *похожие* типы «за» одной переменной.
- За счёт некоторых затрат памяти и скорости
- ≈Как в ОО языках

Smart pointers

Надоело ручками работать с *сырыми* указателями? Постоянно протухают (даже если они half-baked, как в C++)? Жиза, узнаёте себя?

Нет? Надоело видеть магию на слайдах? Да.

Тогда вам сю'*да*.

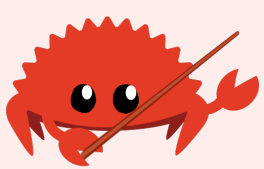
Closures&Iterators

Пользуемся **предыдущими темами**, чтобы творить *магию в фунциональном стиле*.

- Передавать функции в качестве аргументов
- Быстро-удобно работать с данными без *дополнительного выделения памяти* (аллокаций).
- Отлично взаимодействует с оптимизациями компилятора

Модули&использование библиотек

- Разбираемся с *модулями*
- Кастомизируем *сборку*
- Используем инструменты
- Подключаем и смотрим на *библиотеки*



Экосистема в Rust очень крутая, работать с конфигурацией сборки очень просто.

Lifetimes

Позволяет контолировать время, в течение которого объект всё ещё «жив»: *life time*. Когда оно заканчивается, объект автоматически уничтожается.

Очень важно для ссылок, позволяет обеспечить *корректность* кода.

Уникальная фича Rust.

Компилятор — помощник, а не враг

Unsafe

- Хочется в одном языке сочетать разные уровни пирамиды
- Разные миры: safe (стандарт де-факто), unsafe (низкоуровневый код)
- Процесс написания разделён во времени
  ↳ попил кофе между сеансами
  ↳ пространстве (авторы и пользователи библиотек)
- `unsafe` — With great power comes great responsibility

Макросы

Оперируем кодом так же, как и данными — с помощью код на Rust:

- Во время компиляции
- (базовый уровень) Автоматическая генерация *похожего* кода
- (для настоящих ценителей) Пользовательские расширения синтаксиса языка

Embedded

- Встраиваемые системы
  - ↳ умный чайник
  - ↳ система обогрева сидений
  - ↳ атомная станция

Embedded

- Встраиваемые системы
- Нет операционной системы

Embedded

- Встраиваемые системы
- Нет операционной системы
- Hard real time (чтобы не превратиться в охлаждение)

Embedded

- Встраиваемые системы
- Нет операционной системы
- Hard real time (чтобы не превратиться в охлаждение)
- Сейчас доминирует лапшеобразный и не-кроссплатформенный код на C

Embedded

- Встраиваемые системы
- Нет операционной системы
- Hard real time (чтобы не превратиться в охлаждение)
- Сейчас доминирует лапшеобразный и не-кроссплатформенный код на C
- Скорость, надёжность → Rust

Embedded

- Встраиваемые системы
- Нет операционной системы
- Hard real time (чтобы не превратиться в охлаждение)
- Сейчас доминирует лапшеобразный и не-кроссплатформенный код на C
- Скорость, надёжность → Rust
- В Rust, как всегда, всё сделано красиво и «по уму»

Сообщество Rust

- Rust сложно выучить $\Longrightarrow$ Квалифицированные разработчики,
- Мало вакансий $\Longrightarrow$ мотивированные и готовые пилить Open source,
- Не очень зрелый $\Longrightarrow$ лёгкие на подъём!
- Разработка языка
  - ↳ Платформа — GitHub
  - ↳ Rust Foundation
  - ↳ WG
  - ↳ RFC
  - ↳ инициативные группы (если RFC недостаточно)

Самое главное

Комбинация: **Borrow rules** + **Статическая типизация** + **Safe/Unsafe** + **Traits** + **Generics** + **Встроенный линтер**

Дают правило:

Компилируется → **Будет работать**

в ∼ 90%+ случаев

Дебагать код нужно очень редко, в основном по логическим ошибкам.

Организационные моменты

Курс планируется на полгода, дальше — как пойдёт, но не факт.

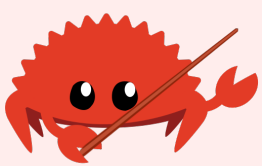
Лекции читаем *попеременно* .

Лекции

На занятиях — чисто лекции, будет небольшая домашка. *Ноуты не нужны.*

Список активностей

Будет *много материала*.



Задавайте сразу *вопросы* по всему, что не очень понятно!

Мы тоже будем задавать вопросы *вам*.

Будем немного кодить *коллективным разумом*.

Запись лекций

- Если нас не победят технические трудности
- Чтобы
  ↳ восстановиться, если *вдруг* пропустили занятие (75% посещений)
  ↳ пересмотривать для лучшего запоминания/при выполнении ДЗ

В любом случае, *материалы занятий* (презентации, возможно код) выкладывать будем точно.

Домашка

- Должна выполняться на 100% за полчаса в среднем
  (есть ещё бонусные баллы, в 100% не входят).
- Будет *табличка с баллами*
- Нужно набрать минимум 60% *для зачета*
  (не забывайте ещё о стандартных 75% посещений).
- Дедлайны — *жесткие*, одна неделя
- **Github Classroom** — требуется базовое умение работать с `git&GitHub`

К следующему занятию рекомендуем установить `rust` и расширения для вашей любимой **IDE**.

Тем, кто ещё не в чате (и собирается ходить) — крайне рекомендуем вступить: