

Полное руководство по Rust

Мартин Гайслер

Содержание

Добро пожаловать в «Полное руководство по Rust»	11
1 Запуск курса	13
1.1 Структура курса	14.
1.2 Горячие клавиши	17.
1.3 Переводы	17.
2 Использование Cargo	19
2.1 Экосистема Rust	19.
2.2 Примеры кода в данном обучении	20.
2.3 Запуск кода локально с помощью Cargo	21.
День 1: утро	23
3 Добро пожаловать в первый день	24
4 Hello, World	26
4.1 Что такое Rust?	26.
4.2 Преимущества Rust	27.
4.3 Playground	27.
5 Типы и значения	29
5.1 Hello, World	29.
5.2 Переменные	30.
5.3 Значения	30.
5.4 Арифметика	31.
5.5 Вывод типа	31.
5.6 Упражнение: Фибоначчи	32.
5.6.1 Решение	32.
6 Основы управления потоком	34
6.1 Блоки и области видимости	34.
6.2 if выражения	35.
6.3 match выражения	35.
6.4 Циклы	37.
6.4.1 for	37.
6.4.2 loop	37.
6.5 break и continue	38.

6.5.1 Метки	38
6.6 Функции	39
6.7 Макросы	39
6.8 Упражнение: Последовательность Коллатца	40
6.8.1 Решение	41
II День 1: После полудня	42
7 Добро пожаловать обратно	43
8 Кортежи и массивы	44
8.1 Массивы	44
8.2 Кортежи	45
8.3 Итерация по массиву	45
8.4 Шаблоны и деструктуризация	46
8.5 Упражнение: Вложенные массивы	46
8.5.1 Решение	47
9 Ссылки	48
9.1 Разделяемые ссылки	48
9.2 Исключительные ссылки	49
9.3 Срезы	50
9.4 Строки	50
9.5 Корректность ссылок	51
9.6 Упражнение: Геометрия	52
9.6.1 Решение	53
10 Пользовательские типы	54
10.1 Именованные структуры	54
10.2 Кортежные структуры	55
10.3 Перечисления	56
10.4 Псевдонимы типов	58
10.5 const	59
10.6 static	59
10.7 Упражнение: События лифта	60
10.7.1 Решение	61
III День 2: Утро	64
11 Добро пожаловать во второй день	65
12 Сопоставление с образцом	66
12.1 Неотразимые шаблоны	66
12.2 Сопоставление значений	67
12.3 Структуры	69
12.4 Перечисления	69
12.5 Управление потоком с помощью let	70
12.5.1 if let Выражения	70
12.5.2 while let Операторы	71
12.5.3 let else Операторы	71

12.6 Упражнение: Вычисление выражений	72
12.6.1 Решение	76
13 Методы и трейты	79
13.1 Методы	79
13.2 Трейты	81
13.2.1 Реализация трейтов	81
13.2.2 Супертрейты	82
13.2.3 Ассоциированные типы	83
13.3 Производные реализации	83
13.4 Упражнение: Трейт логгера	84
13.4.1 Решение	85
14 Джениерики	86
14.1 Джениерик-функции	86
14.2 Ограничения трейтов	87
14.3 Джениерик-типы данных	88
14.4 Джениерик-трейты	89
14.5 <code>impl Trait</code>	90
14.6 <code>dyn Trait</code>	90
14.7 Упражнение: Generic <code>min</code>	92
14.7.1 Решение	92
IV День 2: Послеобеденное время	94
15 Добро пожаловать обратно	95
16 Типы стандартной библиотеки	96
16.1 Стандартная библиотека	96
16.2 Документация	96
16.3 <code>Option</code>	97
16.4 <code>Result</code>	98
16.5 <code>String</code>	98
16.6 <code>Vec</code>	99
16.7 <code>HashMap</code>	100
16.8 Упражнение: Счётчик	101
16.8.1 Решение	103
17 Замыкания	104
17.1 Синтаксис замыканий	104
17.2 Захват	105
17.3 Трейты замыканий	105
17.4 Упражнение: Фильтр логов	107
17.4.1 Решение	107
18 Трейты стандартной библиотеки	109
18.1 Сравнения	109
18.2 Операторы	111
18.3 <code>From</code> и <code>Into</code>	111
18.4 Приведение типов	112
18.5 Чтение и запись	113

18.6 Трейт Default	113
18.7 Упражнение: ROT13	114
18.7.1 Решение	115
V День 3: Утро	117
19 Добро пожаловать в День 3	118
20 Управление памятью	119
20.1 Обзор памяти программы	119
20.2 Подходы к управлению памятью	120
20.3 Владение	121
20.4 Семантика перемещения	122
20.5 Clone	124
20.6 Копируемые типы	125
20.7 Трейт Drop	126
20.8 Упражнение: тип Builder	127
20.8.1 Решение	129
21 Умные указатели	131
21.1 Box<T>	131
21.2 Rc	133
21.3 Владеющие объекты трейтов	133
21.4 Упражнение: бинарное дерево	135
21.4.1 Решение	137
VI День 3: после обеда	141
22 Добро пожаловать обратно	142
23 Заимствования	143
23.1 Заимствование значения	143
23.2 Проверка заимствований	144
23.3 Ошибки заимствований	146
23.4 Внутренняя изменяемость	146
23.4.1 Cell	146
23.4.2 RefCell	147
23.5 Упражнение: Статистика здоровья	148
23.5.1 Решение	149
24 Времена жизни	151
24.1 Аннотации времени жизни	151
24.2 Времена жизни в вызовах функций	152
24.3 Времена жизни в структурах данных	153
24.4 Упражнение: Разбор Protobuf	154
24.4.1 Решение	159

VII День 4: Утро	165
25 Добро пожаловать в День 4	166
26 Итераторы	167
26.1 Мотивация итераторов	167
26.2 Iterator Трейт	168
26.3 Iterator Вспомогательные методы	169
26.4 collect	170
26.5 IntoIterator	170
26.6 Упражнение: Цепочки методов итератора	172
26.6.1 Решение	173
27 Модули	174
27.1 Модули	174
27.2 Иерархия файловой системы	175
27.3 Видимость	176
27.4 Видимость и инкапсуляция	177
27.5 use, super, self	178
27.6 Упражнение: Модули для GUI-библиотеки	179
27.6.1 Решение	181
28 Тестирование	185
28.1 Модульные тесты	185
28.2 Другие виды тестов	186
28.3 Линтеры компилятора и Clippy	187
28.4 Упражнение: Алгоритм Луна	187
28.4.1 Решение	188
VIII День 4: Вторая половина дня	191
29 Добро пожаловать обратно	192
30 Обработка ошибок	193
30.1 Паники	193
30.2 Result	194
30.3 Оператор try	195
30.4 Преобразования try	196
30.5 Динамические типы ошибок	198
30.6 thiserror	198
30.7 anyhow	199
30.8 Упражнение: переписывание с Result	200
30.8.1 Решение	202
31 Небезопасный Rust	204
31.1 Небезопасный Rust	204
31.2 Разыменование сырых указателей	205
31.3 Изменяемые статические переменные	206
31.4 Объединения	207
31.5 Небезопасные функции	207
31.5.1 Небезопасные функции Rust	207

31.5.2 Небезопасные внешние функции	208
31.5.3 Вызов небезопасных функций	209
31.6 Реализация небезопасных трейтов	210
31.7 Безопасная оболочка FFI	210
31.7.1 Решение	213
IX Android	217
32 Добро пожаловать в Rust на Android	218
33 Настройка	219
34 Правила сборки	220
34.1 Rust бинарные файлы	220
34.2 Rust библиотеки	221
35 AIDL	223
35.1 Учебник по сервису дня рождения	223
35.1.1 Интерфейсы AIDL	223
35.1.2 Сгенерированный API сервиса	224
35.1.3 Реализация сервиса	224
35.1.4 Сервер AIDL	225
35.1.5 Развёртывание	226
35.1.6 Клиент AIDL	227
35.1.7 Изменение API	228
35.1.8 Обновление клиента и сервиса	228
35.2 Работа с типами AIDL	229
35.2.1 Примитивные типы	229
35.2.2 Типы массивов	230
35.2.3 Отправка объектов	230
35.2.4 Parcelable	231
35.2.5 Отправка файлов	232
36 Тестирование в Android	234
36.1 GoogleTest	235
36.2 Мокирование	237
37 Логирование	239
38 Взаимодействие	241
38.1 Взаимодействие с C	241
38.1.1 Простая библиотека на C	242
38.1.2 Использование Bindgen	242
38.1.3 Запуск нашего бинарного файла	243
38.1.4 Простая библиотека на Rust	244
38.1.5 Вызов Rust	244
38.2 C C++	245
38.2.1 Модуль-мост	245
38.2.2 Объявления Rust Bridge	246
38.2.3 Сгенерированный C++	247
38.2.4 Объявления моста C++	247

38.2.5 Общие типы	248
38.2.6 Общие перечисления	249
38.2.7 Обработка ошибок в Rust	250
38.2.8 Обработка ошибок в C++	250
38.2.9 Дополнительные типы	250
38.2.10 Сборка для Android	251
38.2.11 Сборка для Android	252
38.2.12 Сборка для Android	252
38.3 Взаимодействие с Java	252
X Chromium	255
39 Добро пожаловать в Rust в Chromium	256
40 Настройка	257
41 Сравнение экосистем Chromium и Cargo	259
42 Политика Rust в Chromium	261
43 Правила сборки	263
43.1 Включение unsafe-кода на Rust	263
43.2 Зависимость от Rust-кода из C++ в Chromium	264
43.3 Visual Studio Code	264
43.4 Практическое задание по правилам сборки	265
44 Тестирование	267
44.1 Библиотека trust_gtest_interop	268
44.2 Правила GN для тестов на Rust	268
44.3 Макрос chromium::import!	269
44.4 Практическое задание по тестированию	269
45 Взаимодействие с C++	270
45.1 Пример биндингов	271
45.2 Ограничения CXX	271
45.3 Обработка ошибок в CXX	272
45.3.1 Обработка ошибок в CXX: пример QR	272
45.3.2 Обработка ошибок CXX: пример с PNG	273
45.4 Использование cxx в Chromium	274
45.5 Упражнение: взаимодействие с C++	274
46 Добавление сторонних Crates	276
46.1 Настройка файла Cargo.toml для добавления Crates	276
46.2 Настройка gnrt_config.toml	277
46.3 Загрузка Crates	277
46.4 Генерация правил сборки gn	278
46.5 Решение проблем	278
46.5.1 Скрипты сборки, генерирующие код	278
46.5.2 Скрипты сборки, компилирующие C++ или выполняющие произвольные действия	279
46.6 Зависимость от Crate	279
46.7 Аудит сторонних Crates	279

46.8 Проверка Crates в исходном коде Chromium	280
46.9 Обновление Crates	280
46.10 Упражнение	280
47 Итоговое упражнение	282
48 Решения упражнений	284
XI Bare Metal: Утро	285
49 Добро пожаловать в Bare Metal Rust	286
50 no_std	288
50.1 Минимальная no_std программа	289
50.2 alloc	289
51 Микроконтроллеры	291
51.1 Raw MMIO	291
51.2 Peripheral Access Crates	293
51.3 HAL crates	294
51.4 Board support crates	295
51.5 Паттерн состояния типа	295
51.6 embedded-hal	296
51.7 probe-rs и cargo-embed	296
51.7.1 Отладка	297
51.8 Другие проекты	297
52 Упражнения	299
52.1 Компас	299
52.2 Упражнение Bare Metal Rust Morning	301
XII Bare Metal: После полудня	305
53 Процессоры приложений	306
53.1 Подготовка к Rust	306
53.2 Встроенная ассемблерная вставка	309
53.3 Нестабильный доступ к памяти для MMIO	310
53.4 Напишем драйвер UART	310
53.4.1 Дополнительные трейты	311
53.4.2 Использование	312
53.5 Улучшенный драйвер UART	313
53.5.1 Bitfllags	313
53.5.2 Несколько регистров	314
53.5.3 Драйвер	315
53.6 safle-mmio	316
53.6.1 Драйвер	317
53.6.2 Использование	318
53.7 Логирование	319
53.7.1 Использование	320
53.8 Исключения	321

53.9 aarch64-rt	323
53.10 Другие проекты	324
54 Полезные crates	325
54.1 zerocopy	325
54.2 aarch64-paging	326
54.3 buddy_system_allocator	326
54.4 tinyvec	327
54.5 spin	327
55 Bare-Metal на Android	329
55.1 vmbase	330
56 Упражнения	331
56.1 Драйвер RTC	331
56.2 Bare Metal Rust: послеобеденное занятие	338
XIII Конкурентность: утро	343
57 Добро пожаловать в раздел Конкурентность в Rust	344
58 Потоки	345
58.1 Обычные потоки	345
58.2 Scoped-потоки	346
59 Каналы	348
59.1 Отправители и Получатели	348
59.2 Неограниченные каналы	349
59.3 Ограниченные каналы	349
60 Send и Sync	351
60.1 Маркерные трейты	351
60.2 Send	351
60.3 Sync	352
60.4 Примеры	352
61 Общий состояния	354
61.1 Arc	354
61.2 Mutex	355
61.3 Пример	356
62 Упражнения	358
62.1 Обед философов	358
62.2 Многопоточный проверщик ссылок	359
62.3 Решения	362
XIV Конкурентность: вторая половина дня	367
63 Добро пожаловать	368

64 Основы Async	369
64.1 async/await	369
64.2 Futures	370
64.3 Машина состояний	371
64.4 Среды выполнения	373
64.4.1 Tokio	373
64.5 Задачи	374
65 Каналы и управление потоком	376
65.1 Async-каналы	376
65.2 Join	377
65.3 Select	378
66 Подводные камни	379
66.1 Блокировка исполнителя	379
66.2 Pin	380
66.3 Async-трейты	382
66.4 Отмена	384
67 Упражнения	387
67.1 Обед философов — Async	387
67.2 Приложение для группового чата	388
67.3 Решения	391
XV Идиоматичный Rust	396
68 Добро пожаловать в Идиоматичный Rust	397
69 Использование системы типов	400
69.1 Паттерн Newtype	401
69.1.1 Семантическая путаница	401
69.1.2 Парсить, а не валидировать	402
69.1.3 Действительно ли это инкапсулировано?	403
XVI Заключительные слова	404
70 Спасибо!	405
71 Глоссарий	406
72 Другие ресурсы по Rust	410
73 Благодарности	412

Добро пожаловать в «Полное руководство по Rust»



сборка успешно участники 317 звёзды 30k

Это бесплатный курс по Rust, разработанный командой Android в Google. Курс охватывает полный спектр Rust — от базового синтаксиса до продвинутых тем, таких как обобщения и обработка ошибок.

Последнюю версию курса можно найти по адресу <https://google.github.io/comprehensive-rust/>. Если вы читаете курс в другом месте, пожалуйста, проверяйте там наличие обновлений.

Курс доступен на других языках. Выберите предпочтаемый язык в правом верхнем углу страницы или ознакомьтесь со страницей переводов для списка всех доступных версий.

Курс также доступен в формате PDF.

Цель курса — обучить вас Rust. Мы предполагаем, что вы ничего не знаете о Rust и надеемся:

- Дать вам всестороннее понимание синтаксиса и языка Rust.
- Позволить вам изменять существующие программы и писать новые на Rust.
- Показать вам распространённые идиомы Rust.

Первые четыре дня курса мы называем Основы Rust.

Опираясь на это, вам предлагается углубиться в одну или несколько специализированных тем:

- [Android: полудневный курс по использованию Rust для разработки на платформе Android \(AOSP\)](#). Включает взаимодействие с C, C++ и Java.
- [Chromium: полудневный курс по использованию Rust в браузерах на базе Chromium](#). Включает взаимодействие с C++ и способы подключения сторонних crate в Chromium.
- [Bare-metal: полный день занятий по использованию Rust для разработки bare-metal \(встраиваемых систем\)](#). Рассматриваются как микроконтроллеры, так и процессоры приложений.
- [Конкурентность: полный день занятий по конкурентности в Rust](#). Рассматриваются как классическая конкурентность (прерванное планирование с использованием потоков и мьютексов), так и конкурентность async/await (кооперативная многозадачность с использованием futures).

Не цели

Rust — большой язык, и мы не сможем охватить его полностью за несколько дней.
Некоторые не цели этого курса:

- Изучение разработки макросов: пожалуйста, обратитесь к Книге по Rust и Rust by Example.

Предположения

Курс предполагает, что вы уже умеете программировать. Rust — это язык со статической типизацией, и мы иногда будем проводить сравнения с C и C++, чтобы лучше объяснить или противопоставить подход Rust.

Если вы умеете программировать на языке с динамической типизацией, таком как Python или JavaScript, то вы также сможете без труда следовать материалу.

Это приметы заметки докладчика . Мы будем использовать их для добавления дополнительной информации на слайды. Это могут быть ключевые моменты, которые преподаватель должен осветить, а также ответы на типичные вопросы, возникающие на занятиях.

Глава 1

Запуск курса

Эта страница предназначена для преподавателя курса.

Здесь приведена некоторая справочная информация о том, как мы проводим курс внутри Google.

Обычно занятия проходят с 9:00 до 16:00 с перерывом на обед продолжительностью 1 час.

Это оставляет по 3 часа на утреннюю и дневную сессии. Обе сессии включают несколько перерывов и время для выполнения упражнений студентами.

Перед проведением курса вам следует:

1. Ознакомьтесь с материалами курса. Мы включили заметки для докладчика, чтобы выделить ключевые моменты (пожалуйста, помогите нам, добавляя дополнительные заметки для докладчика!). При проведении презентации убедитесь, что заметки для докладчика открыты во всплывающем окне (нажмите на ссылку с маленькой стрелкой рядом с «Speaker Notes»). Так у вас будет чистый экран для презентации перед аудиторией.
2. Определитесь с датами. Поскольку курс длится четыре дня, мы рекомендуем распределить занятия на две недели. Участники курса отмечают, что перерыв между занятиями помогает им лучше усвоить всю предоставленную информацию.
3. Найдите помещение, достаточно большое для ваших очных участников. Мы рекомендуем размер группы от 15 до 25 человек. Это достаточно мало, чтобы участники чувствовали себя комфортно, задавая вопросы, — а также достаточно мало, чтобы один инструктор успевал отвечать на них. Убедитесь, что в помещении есть письменные столы для вас и для студентов: всем потребуется возможность сидеть и работать с ноутбуками. В частности, вам предстоит много заниматься live-coding в качестве инструктора, поэтому кафедра будет для вас мало полезна.
4. В день проведения курса приходите в аудиторию немного заранее, чтобы подготовить всё необходимое. Мы рекомендуем проводить презентацию непосредственно с помощью `mdbook serve`, запущенного на вашем ноутбуке ([см. инструкции по установке](#)). Это обеспечивает оптимальную производительность без задержек при переключении страниц. Использование ноутбука также позволит вам исправлять опечатки по мере их обнаружения вами или участниками курса.
5. Позвольте участникам самостоятельно или в небольших группах решать упражнения. Обычно мы уделяем упражнениям 30–45 минут утром и столько же после обеда (включая время на разбор решений). Обязательно спрашивайте участников, если у них возникают трудности или нужна ваша помощь. Если вы заметите, что у нескольких человек одна и та же проблема, озвучьте её перед всем классом

и предложите решение, например, показав, где найти соответствующую информацию в стандартной библиотеке.

На этом всё, желаем успехов в проведении курса! Надеемся, что он будет для вас таким же увлекательным, как и для нас!

Пожалуйста, оставьте отзыв после прохождения, чтобы мы могли продолжать совершенствовать курс. Нам будет очень интересно узнать, что вам понравилось и что можно улучшить. Ваши студенты также будут очень рады отправить нам свои отзывы!

1.1 Структура курса

Эта страница предназначена для преподавателя курса.

Основы Rust

Первые четыре дня посвящены Основам Rust. Темп занятий интенсивный, и мы охватываем большой объём материала!

Расписание курса:

- День 1, утро (2 часа 10 минут, включая перерывы)

Сегмент	Продолжительность
Приветствие	5 минут
Hello, World	15 минут
Типы и значения	40 минут
Основы управления потоком	— 45 минут

- День 1, после обеда (2 часа 45 минут, включая перерывы)

Сегмент	Продолжительность
Кортежи и массивы	35 минут
Ссылки	55 минут
Пользовательские типы	1 час

- День 2, утро (2 часа 45 минут, включая перерывы)

Сегмент	Продолжительность
Приветствие	3 минуты
Сопоставление с образцом	50 минут
Методы и трейты	45 минут
Обобщения	45 минут

- День 2, после обеда (2 часа 50 минут, включая перерывы)

Сегмент	Продолжительность
Типы стандартной библиотеки	1 час
Замыкания	30 минут
Трейты стандартной библиотеки	1 час

- День 3, утро (2 часа 20 минут, включая перерывы)

Сегмент	Продолжительность
Приветствие	3 минуты
Управление памятью	1 час
Умные указатели	55 минут

- День 3, после обеда (1 час 55 минут, включая перерывы)

Сегмент	Продолжительность
Заемствования	55 минут
Времена жизни	50 минут

- День 4, утро (2 часа 50 минут, включая перерывы)

Длительность сегмента
Добро пожаловать, 3 минуты
Итераторы, 55 минут
Модули, 45 минут
Тестирование, 45 минут

- День 4, вторая половина дня (2 часа 20 минут, включая перерывы)

Сегмент	Продолжительность
Обработка ошибок, 55 минут	
Unsafe Rust	1 час 15 минут

Глубокое погружение

В дополнение к 4-дневному курсу по Основам Rust рассматриваются некоторые более специализированные темы:

Rust в Android

Курс «Rust в Android» — полудневное обучение по использованию Rust для разработки под платформу Android. Включает взаимодействие с C, C++ и Java.

Для работы потребуется исходный код AOSP. Сделайте checkout репозитория курса на той же машине и переместите каталог `src/android/` в корень вашего исходного кода AOSP. Это обеспечит, что система сборки Android обнаружит файлы `Android.bp` в каталоге `src/android/`.

Убедитесь, что команда `adb sync` работает с вашим эмулятором или реальным устройством, и предварительно соберите все примеры Android с помощью `src/android/build_all.sh`. Изучите скрипт, чтобы понять, какие команды он выполняет, и убедитесь, что они работают при запуске вручную.

Rust в Chromium

Курс «Rust в Chromium» — полудневное обучение по использованию Rust в составе браузера Chromium. Он включает использование Rust в системе сборки Chromium `gn`, подключение сторонних библиотек («crates») и взаимодействие с C++.

Вам потребуется уметь собирать Chromium — рекомендуется отладочная компонентная сборка для ускорения, но подойдет любая сборка. Убедитесь, что вы можете запустить собранный вами браузер Chromium.

Bare-Metal Rust

Курс «Bare-Metal Rust» — это полный день обучения по использованию Rust для bare-metal (встраиваемой) разработки. Рассматриваются как микроконтроллеры, так и процессоры приложений.

Для части, связанной с микроконтроллерами, необходимо заранее приобрести плату разработки BBC micro:bit v2. Всем потребуется установить ряд пакетов, как описано на странице приветствия.

Конкурентность в Rust

Курс «Конкурентность в Rust» — это полный день обучения классической, а также `async / await` конкурентности.

Вам потребуется свежий crate с загруженными и готовыми к использованию зависимостями. Затем вы сможете копировать и вставлять примеры в файл `src/main.rs` для экспериментов:

```
cargo init concurrency
cd concurrency
cargo add tokio --features full
cargo run
```

Расписание курса:

- Утро (3 часа 20 минут, включая перерывы)

Сегмент	Продолжительность
Потоки	30 минут
Каналы	20 минут
Send и Sync —	15 минут
Общее состояние	30 минут
Упражнения	1 час 10 минут

- День (3 часа 30 минут, включая перерывы)

Сегмент	Продолжительность
Основы Async	40 минут
Каналы и управление потоком —	20 минут
Подводные камни	55 минут
Упражнения	1 час 10 минут

Идиоматичный Rust

Идиоматичный Rust — это двухдневный курс, посвящённый идиомам и паттернам Rust.

Перед началом этого курса рекомендуется ознакомиться с материалом из Основы Rust.

Расписание курса:

- Утро (25 минут, включая перерывы)

Сегмент	Продолжительность
Использование системы типов	— 25 минут

Формат

Курс предполагает высокую интерактивность, и мы рекомендуем позволять вопросам направлять изучение Rust!

1.2 Горячие клавиши

В mdBook есть несколько полезных горячих клавиш:

- Стрелка влево: перейти на предыдущую страницу.
- Стрелка вправо: перейти на следующую страницу.
- Ctrl + Enter: выполнить выделенный код.
- s: активировать строку поиска.

1.3 Переводы

Курс был переведён на другие языки замечательными волонтёрами:

- Бразильский португальский: @rastringer, @hugojacob, @joaovicmendes и @henrifl75.
- Китайский (упрощённый): @suetflei, @wnghl, @anlunx, @kongy, @noahdragon, @super-whd, @SketchK и @nodmp.
- Китайский (традиционный) от @hueich, @victorhsieh, @mingyc, @kuanhungchen и @johnathan79717.
- Персидский от @DannyRavi, @javad-jaflari, @Alix1383, @moaminsharifi, @hamidrezakr и @mehrad77.
- Японский от @CoinEZ-JPN, @momotaro1105, @HidenoriKobayashi и @kantasv.
- Корейский от @keinspace, @jiyongp, @jooyunghan и @namhyung.
- Испанский от @deavid.
- Украинский от @git-user-cpp, @yaremam и @reta.

Используйте переключатель языков в правом верхнем углу для смены языка.

Незавершённые переводы

Существует большое количество переводов, находящихся в процессе выполнения. Мы предоставляем ссылки на недавно обновлённые переводы:

- Арабский от @younies
- Бенгальский от @raselmandol.
- Французский перевод выполнен @KookaS, @vcaen и @AdrienBaudemont.
- Немецкий перевод выполнен @Throvn и @ronaldflw.
- Итальянский перевод выполнен @henrythebuilder и @detro.

Полный список переводов с их текущим статусом доступен либо по состоянию на последнее обновление, либо синхронизирован с последней версией курса.

Если вы хотите помочь в этом проекте, пожалуйста, ознакомьтесь с нашими инструкциями по началу работы. Координация переводов осуществляется через систему отслеживания задач.

Глава 2

Использование Cargo

Когда вы начнёте изучать Rust, вы вскоре встретите Cargo — стандартный инструмент в экосистеме Rust для сборки и запуска приложений на Rust. Здесь мы кратко расскажем, что такое Cargo, как он вписывается в более широкую экосистему и как используется в этом обучающем курсе.

Установка

Пожалуйста, следуйте инструкциям на <https://rustup.rs/>.

Это обеспечит вас инструментом сборки Cargo (`cargo`) и компилятором Rust (`rustc`). Вы также получите `rustup` — утилиту командной строки, которую можно использовать для установки различных версий компилятора.

После установки Rust следует настроить ваш редактор или IDE для работы с Rust. Большинство редакторов делают это, взаимодействуя с `rust-analyzer`, который обеспечивает автодополнение и переход к определению для VS Code, Emacs, Vim/Neovim и многих других. Также доступна другая IDE под названием RustRover.

- В Debian/Ubuntu вы можете установить `rustup` через `apt`:
`sudo apt install rustup`
- На macOS можно использовать Homebrew для установки Rust, однако это может привести к установке устаревшей версии. Поэтому рекомендуется устанавливать Rust с официального сайта.

2.1 Экосистема Rust

Экосистема Rust состоит из множества инструментов, основными из которых являются:

- `rustc` : компилятор Rust, который преобразует `.rs` файлы в бинарные и другие промежуточные форматы.
- `cargo` : менеджер зависимостей и инструмент сборки Rust. Cargo умеет загружать зависимости, обычно размещённые на <https://crates.io>, и передавать их `rustc` при сборке вашего проекта. Cargo также оснащён встроенным запускательем тестов, который используется для выполнения модульных тестов.

- `rustup` : установщик и обновлятор инструментов Rust. Этот инструмент используется для установки и обновления `rustc` и `cargo` при выходе новых версий Rust. Кроме того, `rustup` может загружать документацию для стандартной библиотеки. Вы можете иметь несколько версий Rust, установленных одновременно, и `rustup` позволит переключаться между ними по мере необходимости.

Ключевые моменты:

- Rust имеет быстрый цикл выпуска с новой версией каждые шесть недель. Новые версии сохраняют обратную совместимость со старыми версиями и при этом добавляют новую функциональность.
- Существует три канала выпуска: «stable», «beta» и «nightly».
- Новые функции тестируются в «nightly», а «beta» становится «stable» каждые шесть недель.
- Зависимости также могут разрешаться из альтернативных реестров, git, паков и других источников.
- В Rust также существуют издания: текущее издание — Rust 2024. Предыдущими изданиями были Rust 2015, Rust 2018 и Rust 2021.
 - Издания позволяют вносить изменения в язык, несовместимые с предыдущими версиями.
 - Чтобы избежать нарушения работоспособности кода, издания являются опциональными: вы выбираете издание для вашего crate через файл `Cargo.toml`.
 - Чтобы не разделять экосистему, компиляторы Rust могут смешивать код, написанный для разных изданий.
 - Следует отметить, что прямое использование компилятора без посредничества `cargo` встречается довольно редко (большинство пользователей этого не делают).
 - Стоит упомянуть, что сам Cargo является чрезвычайно мощным и комплексным инструментом. Он поддерживает множество продвинутых функций, включая, но не ограничиваясь:
 - * Структура проекта/пакета
 - * **рабочие пространства**
 - * Управление и кэширование зависимостей для разработки и выполнения
 - * **скрипты сборки**
 - * **глобальная установка**
 - * Он также расширяем с помощью плагинов для подкоманд (например, `cargo clippy`).
 - Читайте больше в официальной книге Cargo

2.2 Примеры кода в этом обучающем курсе

В этом курсе мы преимущественно будем изучать язык Rust через примеры, которые можно выполнять прямо в браузере. Это значительно упрощает настройку и обеспечивает единообразный опыт для всех участников.

Тем не менее, рекомендуется установить Cargo: это облегчит выполнение упражнений. В последний день мы проведём более масштабное упражнение, демонстрирующее работу с зависимостями, для которого необходим Cargo.

Кодовые блоки в этом курсе полностью интерактивны:

```
fn main() {
    println!("Edit me!");
}
```

Вы можете использовать Ctrl + Enter для выполнения кода, когда фокус находится в текстовом поле.

Большинство примеров кода можно редактировать, как показано выше. Некоторые примеры кода не подлежат редактированию по разным причинам:

- Встроенные playground не поддерживают выполнение модульных тестов. Скопируйте код и откройте его в полноценном Playground для демонстрации модульных тестов.
- Встроенные playground теряют своё состояние сразу после перехода на другую страницу! Поэтому студентам рекомендуется решать упражнения с помощью локальной установки Rust или через Playground.

2.3 Запуск кода локально с помощью Cargo

Если вы хотите поэкспериментировать с кодом на своей системе, сначала необходимо установить Rust. Сделайте это, следуя инструкциям в Книге по Rust. Это обеспечит вам рабочие `rustc` и `cargo`. На момент написания последняя стабильная стабильная версия Rust имеет следующие номера версий:

```
% rustc --version
rustc 1.69.0 (84c898d65 2023-04-16)
% cargo --version
cargo 1.69.0 (6e9a83356 2023-04-12)
```

Вы также можете использовать любую более позднюю версию, так как Rust поддерживает обратную совместимость.

После этого выполните следующие шаги, чтобы собрать бинарный файл Rust из одного из примеров этого курса:

1. Нажмите кнопку «Копировать в буфер обмена» на примере, который хотите скопировать.
2. Используйте `cargo new exercise` для создания новой директории `exercise`/для вашего кода:

```
$ cargo new exercise
Создан бинарный (приложение) пакет `exercise`
```

3. Перейдите в каталог `exercise/` и используйте `cargo run` для сборки и запуска вашего бинарного файла:

```
$ cd exercise
$ cargo run
Компиляция exercise v0.1.0 (/home/mgeisler/tmp/exercise)
Завершена сборка dev [без оптимизации + отладочная информация] цели(ей) за 0.75 с
Запуск `target/debug/exercise`
Hello, world!
```

4. Замените шаблонный код в `src/main.rs` на собственный. Например, используя пример на предыдущей странице, сделайте `src/main.rs` выглядящим следующим образом:

```
fn main() {
    println!("Edit me!");
}
```

5. Используйте `cargo build` для сборки и запуска обновлённого бинарного файла:

```
$ cargo run
    Компиляция exercise v0.1.0 (/home/mgeisler/tmp/exercise)
    Завершена сборка dev [без оптимизации + отладочная информация] цели(ей) за 0.24 с
        Запуск `target/debug/exercise`
Отредактируйте меня!
```

6. Используйте `cargo check` для быстрой проверки проекта на ошибки, используйте `cargo build` для компиляции без запуска. Результат будет находиться в `target/debug/` для обычной отладочной сборки. Используйте `cargo build --release` для создания оптимизированной релизной сборки в `target/release/`.
7. Вы можете добавить зависимости для вашего проекта, отредактировав файл `Cargo.toml`. При выполнении команды `cargo` он автоматически загрузит и скомпилирует отсутствующие зависимости.

Постарайтесь побудить участников курса установить Cargo и использовать локальный редактор. Это облегчит их работу, поскольку у них будет полноценная среда разработки.

Часть I

День 1: Утро

Глава 3

Добро пожаловать в первый день

Это первый день изучения Основ Rust. Сегодня мы рассмотрим множество тем:

- Базовый синтаксис Rust: переменные, скалярные и составные типы, перечисления, структуры, ссылки, функции и методы.
- Типы и вывод типов.
- Конструкции управления потоком: циклы, условные операторы и прочее.
- Пользовательские типы: структуры и перечисления.

Расписание

С учётом 10-минутных перерывов эта сессия займет около 2 часов 10 минут. В неё входит:

Сегмент	Продолжительность
Приветствие	5 минут
Hello, World	15 минут
Типы и значения	40 минут
Основы управления потоком	— 45 минут

На этот слайд отводится примерно 5 минут.

Пожалуйста, напомните студентам, что:

- Они должны задавать вопросы по мере их возникновения, не откладывая до конца.
- Занятие предназначено для интерактивного взаимодействия, и обсуждения настоятельно приветствуются!
 - В качестве преподавателя вы должны стараться поддерживать релевантность обсуждений, то есть направлять их на сравнение того, как Rust реализует различные аспекты по сравнению с другими языками. Найти правильный баланс может быть сложно, но лучше склоняться к разрешению обсуждений, поскольку они вовлекают участников гораздо больше, чем односторонняя коммуникация.
- Вопросы, вероятно, приведут к обсуждению тем, которые будут рассмотрены позже на слайдах.
 - Это совершенно нормально! Повторение — важная часть процесса обучения. Помните, что слайды служат лишь вспомогательным материалом, и вы можете пропускать их по своему усмотрению.

Идея первого дня — показать «базовые» концепции Rust, которые должны иметь непосредственные аналоги в других языках. Более продвинутые темы Rust рассматриваются в последующие дни.

Если вы преподаёте этот курс в классе, это хорошее место для обсуждения расписания. Обратите внимание, что в конце каждого раздела предусмотрено упражнение, за которым следует перерыв. Планируется рассмотреть решение упражнения после перерыва. Указанное здесь время является рекомендацией для соблюдения расписания курса. Не стесняйтесь проявлять гибкость и при необходимости вносить изменения!

Глава 4

Hello, World

Этот раздел займет около 15 минут. В него входит:

Слайд	Продолжительность
Что такое Rust?	10 минут
Приемущества Rust	— 3 минуты
Playground	2 минуты

4.1 Что такое Rust?

Rust — новый язык программирования, первая версия которого 1.0 вышла в 2015 году:

- Rust — статически компилируемый язык, выполняющий роль, аналогичную C++.
 - rustc использует LLVM в качестве бэкенда.
- Rust поддерживает множество платформ и архитектур:
 - x86, ARM, WebAssembly, ...
 - Linux, Mac, Windows, ...
- Rust применяется для широкого спектра устройств:
 - прошивки и загрузчики,
 - умные дисплеи,
 - мобильные телефоны,
 - настольные компьютеры,
 - серверы.

На этот слайд должно уйти около 10 минут.

Rust занимает ту же область, что и C++:

- Высокая гибкость.
- Высокий уровень контроля.
- Может масштабироваться до очень ограниченных устройств, таких как микроконтроллеры.
- Не имеет времени выполнения или сборщика мусора.
- Сосредоточен на надежности и безопасности без ущерба для производительности.

4.2 Преимущества Rust

Некоторые уникальные преимущества Rust:

- *Безопасность памяти во время компиляции* — целые классы ошибок памяти предотвращаются на этапе компиляции
 - Отсутствие неинициализированных переменных.
 - Отсутствие двойного освобождения памяти.
 - Отсутствие использования после освобождения.
 - Отсутствие NULLуказателей .
 - Отсутствие забытых блокированных мьютексов.
 - Отсутствие гонок данных между потоками.
 - Отсутствие инвалидирования итераторов.
- *Отсутствие неопределённого поведения во время выполнения*— действия Rust никогда не остаются неуточнёнными.
 - Доступ к массиву проверяется на выход за границы.
 - Переполнение целочисленных значений определено (panic или wrap-around).
- *Современные возможности языка*— выразительные и эргономичные, как в языках высокого уровня.
 - Перечисления и сопоставление с образцом.
 - Обобщения.
 - Отсутствие накладных расходов при FFI.
 - Абстракции с нулевой стоимостью.
 - Отличные сообщения об ошибках компилятора.
 - Встроенный менеджер зависимостей.
 - Встроенная поддержка тестирования.
 - Отличная поддержка Language Server Protocol.

Этот слайд должен занять около 3 минут.

Не тратьте здесь много времени. Все эти пункты будут рассмотрены более подробно позже.

Обязательно спросите у группы, с какими языками программирования у них есть опыт. В зависимости от ответа вы можете выделить различные особенности Rust:

- Опыт работы с C или C++: Rust устраняет целый класс ошибок времени выполнения с помощью borrow checker. Вы получаете производительность, как в C и C++, но без проблем с безопасностью памяти. Кроме того, вы получаете современный язык с такими конструкциями, как сопоставление с образцом и встроенное управление зависимостями.
- Опыт работы с Java, Go, Python, JavaScript...: вы получаете ту же безопасность памяти, что и в этих языках, а также похожее ощущение языка высокого уровня. Кроме того, вы получаете быструю и предсказуемую производительность, как в C и C++ (без сборщика мусора), а также доступ к низкоуровневому оборудованию (если это потребуется).

4.3 Playground

Rust Playground предоставляет простой способ запускать короткие программы на Rust и служит основой для примеров и упражнений в этом курсе. Попробуйте запустить программу «hello-world», с которой он начинается. Она оснащена несколькими полезными функциями:

- В разделе "Tools" используйте опцию `rustfmt` для форматирования кода в "стандартном" стиле.

- В Rust существуют два основных "профиля" для генерации кода: `Debug` (дополнительные проверки во время выполнения, меньше оптимизаций) и `Release` (меньше проверок во время выполнения, больше оптимизаций). Они доступны в разделе "`Debug`" в верхней части интерфейса.
- Если вам интересно, используйте "ASM" в разделе "..." для просмотра сгенерированного ассемблерного кода.

Этот слайд рассчитан примерно на 2 минуты.

Когда студенты отправляются на перерыв, поощряйте их открыть playground и немного поэкспериментировать. Поощряйте их держать вкладку открытой и пробовать разные вещи в течение всего курса. Это особенно полезно для продвинутых студентов, желающих узнать больше об оптимизациях Rust или сгенерированном ассемблерном коде.

Глава 5

Типы и значения

Этот сегмент рассчитан примерно на 40 минут. Он включает:

Слайд	Продолжительность
Hello, World	5 минут
Переменные	5 минут
Значения	5 минут
Арифметика	3 минуты
Вывод типа	3 минуты
Упражнение: Фибоначчи	— 15 минут

5.1 Привет, мир

Давайте перейдём к самой простой программе на Rust — классической программе Hello World:

```
fn main() {
    println!("Привет 🌎!");
}
```

Что вы видите:

- Функции объявляются с помощью `fn`.
- Функция `main` является точкой входа в программу.
- Блоки ограничиваются фигурными скобками, как в C и C++.
- Операторы заканчиваются `;`.
- В Rust используются гигиеничные макросы, `println!` — пример такого макроса.
- Строки в Rust кодируются в UTF-8 и могут содержать любые символы Юникода.

На этот слайд отводится примерно 5 минут.

Этот слайд предназначен для того, чтобы студенты почувствовали себя уверенно при работе с кодом на Rust. Они увидят огромное количество кода на Rust в течение следующих четырёх дней, поэтому мы начинаем с чего-то знакомого.

Ключевые моменты:

- Rust во многом похож на другие языки из семейства C/C++/Java. Он императивен и не пытается изобретать велосипед без крайней необходимости.

- Rust — современный язык с полной поддержкой таких возможностей, как Юникод.
- Rust использует макросы в ситуациях, когда требуется переменное число аргументов (перегрузка функций отсутствует).
- Гигиеничность макросов означает, что они не захватывают случайно идентификаторы из области видимости, в которой используются. Макросы Rust на самом деле лишь частично гигиеничны.
- Rust является мультипарадигменным языком. Например, он обладает мощными возможностями объектно-ориентированного программирования и, хотя не является функциональным языком, включает ряд функциональных концепций.

5.2 Переменные

Rust обеспечивает безопасность типов за счёт статической типизации. Привязки переменных выполняются с помощью `let`:

```
fn main() {
    let x: i32 = 10;
    println!("x: {}", x);
    // x = 20;
    // println!("x: {}", x);
}
```

На этот слайд отводится примерно 5 минут.

- Раскомментируйте `x = 20`, чтобы продемонстрировать, что переменные по умолчанию являются неизменяемыми. Добавьте ключевое слово `mut`, чтобы разрешить изменения.
- Для этого слайда включены предупреждения, например, о неиспользуемых переменных или избыточном `mut`. В большинстве слайдов они опущены, чтобы не отвлекать внимание предупреждениями. Попробуйте убрать изменение, но оставить ключевое слово `mut`.
- Здесь `i32` — это тип переменной. Он должен быть известен во время компиляции, но вывод типа (рассмотренный далее) позволяет программисту часто опускать его.

5.3 Значения

Ниже приведены некоторые базовые встроенные типы и синтаксис литеральных значений для каждого из них.

Типы	Литералы
Знаковые целые числа	<code>i8, i16, i32, i64, i128, isize</code>
Беззнаковые целые числа	<code>u8, u16, u32, u64, u128, usize</code>
Числа с плавающей запятой	<code>f32, f64</code>
Скалярные значения Юникода	<code>char</code>
Булевые значения	<code>bool</code>
	<code>-10, 0, 1_000, 123_i64</code>
	<code>0, 123, 10_u16</code>
	<code>3.14, -10.0e20, 2_f32</code>
	<code>'a', 'ä', '∞'</code>
	<code>true, false</code>

Ширина типов следующая:

- `iN`, `uN` и `fN` имеют ширину N бит,
- `isize` и `usize` соответствуют ширине указателя,

- `char` занимает 32 бита,
- `bool` занимает 8 бит.

На этот слайд отводится примерно 5 минут.

Существуют несколько синтаксисов, не показанных выше:

- Все подчёркивания в числах можно опускать, они служат только для удобочитаемости. Таким образом, `1_000` можно записать как `1000` (или `10_00`), а `123_i64` можно записать как `123i64`.

5.4 Арифметика

```
fn interproduct(a: i32, b: i32, c: i32) -> i32 {
    return a * b + b * c + c * a;
}

fn main() {
    println!("результат: {}", interproduct(120, 100, 248));
}
```

Этот слайд должен занять около 3 минут.

Это первый случай, когда мы видим функцию, отличную от `main`, но смысл должен быть понятен: Она принимает три целых числа и возвращает целое число. Функции будут рассмотрены более подробно позже.

Арифметика очень похожа на другие языки с аналогичным приоритетом операций.

Что насчёт переполнения целых чисел? В C и C++ переполнение знаковых целых чисел фактически неопределено и может приводить к непредсказуемому поведению во время выполнения. В Rust это определено.

Измените типы `i32` на `i16`, чтобы увидеть переполнение целого числа, которое вызывает панику (проверяется) в режиме отладки и оборачивается в режиме релиза. Существуют и другие варианты, такие как переполнение, насыщение и перенос. К ним обращаются с помощью синтаксиса методов, например, `(a * b).saturating_add(b * c).saturating_add(c * a)`.

На самом деле компилятор обнаружит переполнение константных выражений, поэтому в примере требуется отдельная функция.

5.5 Вывод типа

Rust анализирует, как переменная используется, чтобы определить её тип:

```
fn takes_u32(x: u32) {
    println!("u32: {}", x);
}

fn takes_i8(y: i8) { println!(
    "i8: {}", y
);}

fn main() {
    let x = 10;
    let y = 20;
```

```

    takes_u32(x);
    takes_i8(y);
    // takes_u32(y);
}

```

Этот слайд должен занять около 3 минут.

Этот слайд демонстрирует, как компилятор Rust выводит типы на основе ограничений, заданных объявлением и использованием переменных.

Очень важно подчеркнуть, что переменные, объявленные таким образом, не являются динамическим «любым типом», способным хранить любые данные. Машинный код, сгенерированный таким объявлением, идентичен коду при явном указании типа. Компилятор выполняет эту работу за нас и помогает писать более лаконичный код.

Если ничто не ограничивает тип целочисленной литералы, Rust по умолчанию использует `i32`. Это иногда отображается как `{integer}` в сообщениях об ошибках. Аналогично, литералы с плавающей точкой по умолчанию имеют тип `f64`.

```

fn main() {
    let x = 3.14;
    let y = 20;
    assert_eq!(x, y);
    // ОШИБКА: отсутствует реализация для `float == {integer}`
}

```

5.6 Упражнение: Фибоначчи

Последовательность Фибоначчи начинается с $[0, 1]$. Для $n > 1$ n -й элемент последовательности Фибоначчи вычисляется рекурсивно как сумма $(n-1)$ -го и $(n-2)$ -го элементов.

Напишите функцию `fib(n)`, которая вычисляет n -й элемент последовательности Фибоначчи. Когда эта функция вызовет панику?

```

fn fib(n: u32) -> u32 {
    if n < 2 {
        // Базовый случай.
        return todo!("Реализовать это");
    } else {
        // Рекурсивный случай.
        return todo!("Реализовать это");
    }
}

fn main() {
    let n = 20;
    println!("fib({}) = {}", n, fib(n));
}

```

5.6.1 Решение

```

fn fib(n: u32) -> u32 {
    if n < 2 {
        return n;
    }
}

```

```
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

fn main() {
    let n = 20;
    println!("fib({n}) = {}", fib(n));
}
```

Глава 6

Основы управления потоком

Этот раздел займет примерно 45 минут. В нем содержится:

Слайд	Продолжительность
Блоки и области видимости	5 минут
ifl-выражения	4 минуты
match-выражения	5 минут
Циклы	5 минут
break и continue	4 минуты
Функции	3 минуты
Макросы	2 минуты
Упражнение: последовательность Коллатца — 15 минут	

- Теперь мы рассмотрим различные виды управления потоком в Rust.
- Большая часть этого будет вам хорошо знакома из других языков программирования.

6.1 Блоки и области видимости

Блок в Rust содержит последовательность выражений, заключённых в фигурные скобки `{ }`. Каждый блок имеет значение и тип, которые соответствуют последнему выражению блока:

```
fn main() {
    let z = 13;
    let x = {
        let y = 10;
        dbg!(y);
        z - y
    };
    dbg!(x);
    // dbg!(y);
}
```

Если последнее выражение заканчивается на `;`, тогда результирующее значение и тип будут `()`.

Область видимости переменной ограничена окружающим блоком.

На этот слайд отводится примерно 5 минут.

- Вы можете объяснить, что `dbg!` — это макрос Rust, который выводит и возвращает значение заданного выражения для быстрой и простой отладки.
- Вы можете показать, как изменяется значение блока, изменения последнюю строку в блоке. Например, добавляя или удаляя точку с запятой либо используя `return`.
- Продемонстрируйте, что попытка доступа к увне его области видимости не скомпилируется.
- Значения фактически «освобождаются» при выходе из их области видимости, даже если их данные на стеке всё ещё присутствуют.

6.2 if expressions

Вы используете `if expressions` точно так же, как `if` операторы в других языках:

```
fn main() {
    let x = 10;
    if x == 0 {
        println!("zero!");
    } else if x < 100 {
        println!("biggish");
    } else {
        println!("huge");
    }
}
```

Кроме того, вы можете использовать `if` как выражение. Последнее выражение каждого блока становится значением `if expression`:

```
fn main() {
    let x = 10;
    let size = if x < 20 { "small" } else { "large" };
    println!("number size: {}", size);
}
```

Этот слайд должен занять примерно 4 минуты.

Поскольку `if` является выражением и должен иметь определённый тип, оба его ветвящихся блока должны иметь одинаковый тип. Покажите, что произойдёт, если добавить `;` после `"small"` во втором примере.

Выражение `if` следует использовать так же, как и другие выражения. Например, когда оно используется в операторе `let`, оператор также должен завершаться `;`. Удалите `;` перед `println!`, чтобы увидеть ошибку компилятора.

6.3 match выражения

`match` может использоваться для проверки значения на соответствие одному или нескольким вариантам:

```
fn main() {
    let val = 1;
    match val {
        1 => println!("one"),
        2 => println!("two"),
        _ => println!("other")
    }
}
```

```

    10 => println!("ten"),
    100 => println!("one hundred"),
    _ => {
        println!("something else");
    }
}
}

```

Как и if выражения, match также может возвращать значение;

```

fn main() {
    let flag = true;
    let val = match flag {
        true => 1,
        false => 0,
    };
    println!("Значение {flag} равно {val}");
}

```

На этот слайд отводится примерно 5 минут.

- Ветви match оцениваются сверху вниз, и выполняется тело первой подходящей ветви.
- Отсутствует переход между случаями, как это реализовано в switch в других языках.
- Тело ветви match может быть одиночным выражением или блоком. Технически это одно и то же, поскольку блоки также являются выражениями, но студенты могут пока не полностью осознавать эту симметрию.
- Выражения match должны быть исчерпывающими, то есть покрывать все возможные значения или содержать случай по умолчанию, например `_`. Исчерпывающую полноту проще всего продемонстрировать на перечислениях, но они ещё не были введены. Вместо этого мы демонстрируем сопоставление с `bool` — самым простым примитивным типом.
- Этот слайд вводит конструкцию match без обсуждения сопоставления с образцом, предоставляя студентам возможность ознакомиться с синтаксисом без излишней предварительной информации. Завтра мы подробно рассмотрим сопоставление с образцом, поэтому здесь старайтесь не углубляться в детали.

Дополнительные материалы для изучения

- Для дополнительной мотивации использования match вы можете сравнить примеры с их эквивалентами, написанными с помощью if. Во втором случае сопоставление с `bool` через блок `if {} else {}` довольно похоже. Однако в первом примере, который проверяет несколько случаев, выражение match может быть более лаконичным, чем цепочка `if {} else if {} else if {} else`.
- Конструкция match также поддерживает match guards, позволяющие добавить произвольное логическое условие, которое оценивается для определения, следует ли использовать соответствующую ветвь. Однако обсуждение match guards требует объяснения сопоставления с образцом, чего мы стараемся избежать на этом слайде.

6.4 Циклы

В Rust существуют три ключевых слова для циклов: `while`, `loop` и `for`:

`while`

Ключевое слово `while` работает аналогично другим языкам, выполняя тело цикла, пока условие истинно.

```
fn main() {
    let mut x = 200;
    while x >= 10 {
        x = x / 2;
    }
    dbg!(x);
}
```

6.4.1 `for`

Цикл `for` итерирует по диапазонам значений или элементам коллекции:

```
fn main() {
    for x in 1..5 {
        dbg!(x);
    }

    for elem in [2, 4, 8, 16, 32] {
        dbg!(elem);
    }
}
```

- В основе циклов `for` лежит концепция, называемая «итераторами», которая позволяет обрабатывать итерацию по различным типам диапазонов и коллекций. Итераторы будут рассмотрены более подробно позже.
- Обратите внимание, что первый цикл `for` итерирует только до 4. Покажите синтаксис `1..=5` для включающего диапазона.

6.4.2 `loop`

Оператор `loop` выполняется бесконечно, пока не встретит `break`.

```
fn main() {
    let mut i = 0;
    loop {
        i += 1;
        dbg!(i);
        if i > 100 {
            break;
        }
    }
}
```

- Оператор `loop` работает как цикл `while true`. Используйте его для таких задач, как серверы, которые будут обслуживать подключения бесконечно.

6.5 break и continue

Если вы хотите немедленно перейти к следующей итерации, используйте `continue`.

Если необходимо досрочно выйти из любого цикла, используйте `break`. В `loop` можно передать необязательное выражение, которое станет значением выражения `loop`.

```
fn main() {
    let mut i = 0;
    loop {
        i += 1;
        if i > 5 {
            break;
        }
        if i % 2 == 0 {
            continue;
        }
        dbg!(i);
    }
}
```

Этот слайд и его подразделы займут около 4 минут.

Обратите внимание, что `loop` — единственная конструкция цикла, которая может возвращать нетривиальное значение. Это связано с тем, что он гарантированно возвращает значение только при операторе `break` (в отличие от циклов `while` и `for`, которые также могут завершаться при ложном условии).

6.5.1 Метки

Операторы `continue` и `break` могут принимать необязательный аргумент-метку, используемую для выхода из вложенных циклов:

```
fn main() {
    let s = [[5, 6, 7], [8, 9, 10], [21, 15,      32]];
    let mut elements_searched = 0;
    let target_value = 10;
    'outer: for i in 0..=2 {
        for j in 0..=2 {
            elements_searched += 1;
            if s[i][j] == target_value {
                break 'outer;
            }
        }
    }
    dbg!(elements_searched);
}
```

- Метка `break` также работает с произвольными блоками, например, `'label: {`
 `break 'label;`
 `println!("Эта строка пропускается");`
}

6.6 Функции

```
fn gcd(a: u32, b: u32) -> u32 {
    if b > 0 { gcd(b, a % b) } else { a }
}

fn main() {
    dbg!(gcd(143, 52));
}
```

Этот слайд должен занять около 3 минут.

- Параметры объявления следуют за типом (обратный порядок по сравнению с некоторыми языками программирования), затем указывается тип возвращаемого значения.
- Последнее выражение в теле функции (или любом блоке) становится возвращаемым значением. Просто опустите ; в конце выражения. Ключевое слово return может использоваться для раннего возврата, но форма «чистого значения» является идиоматичной в конце функции (рефакторинг gcd с использованием return).
- Некоторые функции не возвращают значение и возвращают 'unit type', () . Компилятор выведет это, если тип возвращаемого значения опущен.
- Перегрузка не поддерживается — каждая функция имеет единственную реализацию.
 - Всегда принимает фиксированное количество параметров. Аргументы по умолчанию не поддерживаются. Макросы могут использоваться для поддержки вариативных функций.
 - Всегда принимает один набор типов параметров. Эти типы могут быть обобщёнными, что будет рассмотрено позже.

6.7 Макросы

Макросы разворачиваются в код Rust во время компиляции и могут принимать переменное число аргументов. Они отличаются тем, что в конце стоит ! . Стандартная библиотека Rust содержит набор полезных макросов.

- println!(format, ...) выводит строку в стандартный вывод, применяя форматирование, описанное в std::fmt .
- format!(format, ...) работает так же, как println!, но возвращает результат в виде строки.
- dbg!(выражение) логирует значение выражения и возвращает его.
- todo!() обозначает участок кода как ещё не реализованный. Если будет выполнен, вызовет панику.

```
fn factorial(n: u32) -> u32 {
    let mut product = 1;
    for i in 1..=n {
        product *= dbg!(i);
    }
    product
}

fn fizzbuzz(n: u32) -> u32 {
    todo!()
}

fn main() {
    let n = 4;
```

```

    println!("{}! = {}", factorial(n));
}

```

Этот слайд рассчитан примерно на 2 минуты.

Основной вывод из этого раздела заключается в том, что эти распространённые удобства существуют и как их использовать. Почему они определены как макросы и к чему они разворачиваются, не является особенно критичным.

Курс не охватывает определение макросов, но в одном из последующих разделов будет описано использование derive-макросов.

Дополнительно для изучения

В стандартной библиотеке предоставлено множество других полезных макросов. Некоторые другие примеры, которыми вы можете поделиться со студентами, если они захотят узнать больше:

- **assert!** и связанные с ним макросы можно использовать для добавления утверждений в ваш код. Они широко применяются при написании тестов.
- **unreachable!** используется для обозначения ветви управления, которая никогда не должна быть достигнута.
- **eprintln!** позволяет выводить данные в stderr.

6.8 Упражнение: Последовательность Коллатца

Последовательность Коллатца определяется следующим образом для произвольного n_1 большего нуля:

- Если n_i равно 1, то последовательность завершается на n_i .
- Если n_i чётно, то $n_{i+1} = n_i / 2$.
- Если n_i нечётно, то $n_{i+1} = 3 * n_i + 1$.

Например, начиная с $n_1 = 3$:

- 3 нечётно, поэтому $n_2 = 3 * 3 + 1 = 10$;
- 10 чётно, поэтому $n_3 = 10 / 2 = 5$;
- 5 нечётно, поэтому $n_4 = 3 * 5 + 1 = 16$;
- 16 чётно, поэтому $n_5 = 16 / 2 = 8$;
- 8 чётно, поэтому $n_6 = 8 / 2 = 4$;
- 4 чётно, поэтому $n_7 = 4 / 2 = 2$;
- 2 чётно, поэтому $n_8 = 1$; и
- последовательность завершается.

Напишите функцию для вычисления длины последовательности Коллатца для заданного начального значения n .

```

// Определяет длину последовательности Коллатца, начинающейся с `n`.
fn collatz_length(mut n: i32) -> u32 {
    todo! ("Реализовать это ")
}

fn main() {
    println!("Длина: {}", collatz_length(11)); // должно быть 15
}

```

6.8.1 Решение

```
// Определяет длину последовательности Коллатца, начинающейся с `n`.  
fn collatz_length(mut n: i32) -> u32 {  
    let mut len = 1;  
    while n > 1 {  
        n = if n % 2 == 0 { n / 2 } else { 3 * n + 1 };  
        len += 1;  
    }  
    len  
}  
  
fn main() {  
    println!("Длина: {}", collatz_length(11)); // должно быть 15  
}
```

Часть II

День 1: Вторая половина дня

Глава 7

С возвращением

С учётом 10-минутных перерывов эта сессия займет около 2 часов 45 минут. В ней содержится:

Сегмент	Продолжительность
Кортежи и массивы	35 минут
Ссылки	55 минут
Пользовательские типы	1 час

Глава 8

Кортежи и массивы

Этот раздел займет около 35 минут. В нём содержится:

Слайд	Продолжительность
Массивы	5 минут
Кортежи	5 минут
Итерация по массиву	3 минуты
Шаблоны и деструктуризация —	5 минут
Упражнение: вложенные массивы	15 минут

- Мы рассмотрели, как работают примитивные типы в Rust. Теперь пришло время начать создавать новые составные типы.

8.1 Массивы

```
fn main() {
    let mut a: [i8; 5] = [5,     4, 3, 2, 1];
    a[2] = 0;
    println!("a: {a:?}");
}
```

На этот слайд отводится примерно 5 минут.

- Массивы также можно инициализировать с помощью сокращённого синтаксиса, например, `[0; 1024]`. Это может быть полезно, когда необходимо инициализировать все элементы одним значением или если у вас большой массив, который сложно инициализировать вручную.
- Значение типа массива `[T; N]` содержит `N` (константу времени компиляции) элементов одного типа `T`. Обратите внимание, что длина массива является частью его типа, что означает, что `[u8; 3]` и `[u8; 4]` считаются двумя разными типами. Срезы, размер которых определяется во время выполнения, рассматриваются далее.
- Попробуйте обратиться к элементу массива за пределами допустимого диапазона. Компилятор способен определить, что индекс небезопасен, и не скомпилирует код:

```
fn main() {
    let mut a: [i8; 5] = [5,     4, 3, 2, 1];
    a[6] = 0;
    println!("a: {a:?}");
}
```

- Обращения к массивам проверяются во время выполнения. Rust обычно может оптимизировать эти проверки; то есть, если компилятор может доказать безопасность доступа, он удаляет проверку во время выполнения для повышения производительности. Их можно избежать, используя unsafe Rust. Оптимизация настолько эффективна, что трудно привести пример, когда проверки во время выполнения не срабатывают. Следующий код скомпилируется, но вызовет панику во время выполнения:

```
fn get_index() -> usize {
    6
}
```

```
fn main() {
    let mut a: [i8; 5] = [5,     4, 3, 2, 1];
    a[get_index()] = 0;
    println!("a: {a:?}");
}
```

- Мы можем использовать литералы для присвоения значений массивам.
- Макрос `println!` требует реализации `Debug` с параметром формата `:?`:
`{}` выводит значение по умолчанию, `{ :? }` выводит отладочную информацию. Типы, такие как целые числа и строки, реализуют вывод по умолчанию, а массивы — только отладочный вывод. Это означает, что здесь необходимо использовать отладочный вывод.
- Добавление `#`, например `{a:#?}`, вызывает формат «красивой печати», который может быть легче для восприятия.

8.2 Кортежи

```
fn main() {
    let t: (i8, bool) = (7, true);
    dbg!(t.0);
    dbg!(t.1);
}
```

На этот слайд отводится примерно 5 минут.

- Как и массивы, кортежи имеют фиксированную длину.
- Кортежи объединяют значения разных типов в составной тип.
- К полям кортежа можно обращаться через точку и индекс значения, например `t.0`, `t.1`.
- Пустой кортеж `()` называется «типов единицы» и обозначает отсутствие возвращаемого значения, аналогично `void` в других языках.

8.3 Итерация по массивам

Оператор `for` поддерживает итерацию по массивам (но не по кортежам).

```

fn main() {
    let primes = [2, 3, 5, 7, 11, 13, 17, 19];
    for prime in primes {
        for i in 2..prime {
            assert_ne!(prime % i, 0);
        }
    }
}

```

Этот слайд должен занять около 3 минут.

Эта функциональность использует трейдIntoIterator, но мы ещё не рассматривали его.

Макрос assert_ne! используется здесь впервые. Также существуют макросы assert_eq! и assert!. Они всегда проверяются, тогда как варианты только для отладки, такие как debug_assert!, в релизных сборках компилируются в пустой код.

8.4 Сопоставление с образцом и деструктуризация

Rust поддерживает использование сопоставления с образцом для деструктуризации сложного значения, например кортежа, на его составные части:

```

fn check_order(tuple: (i32, i32, i32)) -> bool {
    let (left, middle, right) = tuple;
    left < middle && middle < right
}

fn main() {
    let tuple = (1, 5, 3);
    println!(
        "{tuple:?}: {}",
        if check_order(tuple) { "ordered" } else { "unordered" }
    );
}

```

На этот слайд отводится примерно 5 минут.

- Используемые здесь шаблоны являются «неотразимыми», что означает, что компилятор может статически проверить, что значение справа от = имеет ту же структуру, что и шаблон.
- Имя переменной — это неотразимый шаблон, который всегда соответствует любому значению, поэтому мы также можем использовать let для объявления одной переменной.
- Rust также поддерживает использование шаблонов в условных операторах, позволяя одновременно выполнять сравнение на равенство и деструктуризацию. Этот вид сопоставления с образцом будет рассмотрен более подробно позже.
- Отредактируйте приведённые выше примеры, чтобы показать ошибку компилятора, когда шаблон не соответствует значению, с которым происходит сопоставление.

8.5 Упражнение: Вложенные массивы

Массивы могут содержать другие массивы:

```
let array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
```

Каков тип этой переменной?

Используйте массив, подобный приведённому выше, чтобы написать функцию transpose, которая транспонирует матрицу (преобразует строки в столбцы):

```
"transpose" [[1 2 3]]    [1 4 7]
             [[4 5 6]]    ==  [[2 5 8]
             [[7 8 9]]           [3 6 9]
```

Скопируйте приведённый ниже код на <https://play.rust-lang.org/> и реализуйте функцию. Эта функция работает только с матрицами размером 3x3.

```
fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    todo!()
}

fn main() {
    let matrix = [
        [101, 102, 103], // <-- комментарий заставляет rustfmt добавить новую строку
        [201, 202, 203],
        [301, 302, 303],
    ];

    dbg!(matrix);
    let transposed = transpose(matrix);
    dbg!(transposed);
}
```

8.5.1 Решение

```
fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    let mut result = [[0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            result[j][i] = matrix[i][j];
        }
    }
    result
}

fn main() {
    let matrix = [
        [101, 102, 103], // <-- комментарий заставляет rustfmt добавить новую строку
        [201, 202, 203],
        [301, 302, 303],
    ];

    dbg!(matrix);
    let transposed = transpose(matrix);
    dbg!(transposed);
}
```

Глава 9

Ссылки

Этот раздел займет примерно 55 минут. В нем содержится:

Слайд	Продолжительность
Общие ссылки	10 минут
Исключительные ссылки	5 минут
Срезы	10 минут
Строки	10 минут
Срок действия ссылки	3 минуты
Упражнение: Геометрия	20 минут

9.1 Общие ссылки

Ссылка предоставляет способ доступа к другому значению без передачи владения этим значением и также называется «заимствованием». Общие ссылки являются только для чтения, и данные, на которые они ссылаются, не могут изменяться.

```
fn main() {
    let a = 'A';
    let b = 'B';

    let mut r: &char = &a;
    dbg!(r);

    r = &b;
    dbg!(r);
}
```

Общий ссылочный тип `T` имеет тип `&T`. Значение ссылки создаётся с помощью оператора `&`. Оператор `*` «разыменовывает» ссылку, возвращая её значение.

Этот слайд должен занять около 7 минут.

- В Rust ссылки никогда не бывают `null`, поэтому проверка на `null` не требуется.

- Говорят, что ссылка «заимствует» значение, на которое она указывает, и это хорошая модель для студентов, не знакомых с указателями: код может использовать ссылку для доступа к значению, но оно по-прежнему «принадлежит» исходной переменной. Курс подробно рассмотрит владение на третьем дне.
- Ссылки реализованы как указатели, и ключевое преимущество в том, что они могут быть значительно меньше объекта, на который указывают. Студенты, знакомые с C или C++, узнают ссылки как указатели. В последующих частях курса будет рассмотрено, как Rust предотвращает ошибки безопасности памяти, возникающие при использовании сырьих указателей.
- Явное взятие ссылки с помощью `&` обычно требуется. Однако Rust выполняет автоматическое взятие ссылки и разыменование при вызове методов.
- Rust автоматически разыменовывает в некоторых случаях, в частности при вызове методов (попробуйте `r.is_ascii()`). Нет необходимости в операторе `->` как в C++.
- В этом примере `x` является изменяемой ссылкой, чтобы её можно было переназначить (`x = &b`). Обратите внимание, что это повторное связывание `x`, поэтому оно ссылается на другой объект. Это отличается от C++, где присваивание ссылке изменяет значение, на которое она ссылается.
- Общая ссылка не позволяет изменять значение, на которое она ссылается, даже если это значение было изменяемым. Попробуйте `*x = 'X'`.
- Rust отслеживает время жизни всех ссылок, чтобы гарантировать, что они живут достаточно долго. Подвешенные ссылки не могут возникнуть в безопасном Rust.
- Мы подробнее поговорим о заимствовании и предотвращении подвешенных ссылок, когда дойдём до владения.

9.2 Исключительные ссылки

Исключительные ссылки, также известные как изменяемые ссылки, позволяют изменять значение, на которое они ссылаются. Они имеют тип `&mut T`.

```
fn main() {
    let mut point = (1, 2);
    let x_coord = &mut point.0;
    *x_coord = 20;
    println!("point: {point:?}");
}
```

На этот слайд отводится примерно 5 минут.

Ключевые моменты:

- «Эксклюзивный» означает, что только эта ссылка может использоваться для доступа к значению. Никакие другие ссылки (разделяемые или эксклюзивные) не могут существовать одновременно, и к значению, на которое ссылаются, нельзя получить доступ, пока существует эксклюзивная ссылка. Попробуйте создать `&point.0` или изменить `point.0` пока `x_coord` активна.
- Обязательно обратите внимание на разницу между `let mut x_coord: &i32` и `let x_coord: &mut i32`. Первое представляет собой разделяемую ссылку, которая может быть связана с разными значениями, в то время как второе — эксклюзивную ссылку на изменяемое значение.

9.3 Срезы

Срез предоставляет представление на более крупную коллекцию:

```
fn main() {
    let a: [i32; 6] = [10, 20, 30, 40, 50, 60];
    println!("a: {a:?}");

    let s: &[i32] = &a[2..4];

    println!("s: {s:?}");
}
```

- Срезы заимствуют данные из исходного типа.

Этот слайд должен занять около 7 минут.

- Мы создаём срез, заимствуя а и указывая начальный и конечный индексы в квадратных скобках.
- Если срез начинается с индекса 0, синтаксис диапазонов Rust позволяет опустить начальный индекс, то есть &a[0..a.len()] и &a[..a.len()] эквивалентны.
- То же самое справедливо для последнего индекса, поэтому &a[2..a.len()] и &a[2..] эквивалентны.
- Чтобы легко создать срез всего массива, можно использовать &a[..].
- s — это ссылка на срез i32 s. Обратите внимание, что тип s (&[i32]) больше не содержит длину массива. Это позволяет выполнять вычисления над срезами разного размера.
- Срезы всегда заимствуют данные из другого объекта. В этом примере а должен оставаться «живым» (в области видимости) как минимум столько же, сколько и наш срез.
- Нельзя «увеличить» срез после его создания:
 - Нельзя добавлять элементы в срез, так как он не владеет базовым буфером.
 - Нельзя увеличить срез, чтобы он указывал на большую часть базового буфера. Срез теряет информацию о базовом буфере, поэтому нельзя определить, насколько его можно увеличить.
 - Чтобы получить больший срез, необходимо обратиться к исходному буферу и создать оттуда более крупный срез.

9.4 Строки

Теперь мы можем понять два типа строк в Rust:

- &str — это срез байтов в кодировке UTF-8, аналогичный &[u8].
- String — это владеющий буфер байтов в кодировке UTF-8, аналогичный Vec<T>.

```
fn main() {
    let s1: &str = "World";
    println!("s1: {s1}");

    let mut s2: String = String::from("Hello ");
    println!("s2: {s2}");
```

```

s2.push_str(s1);
println!("s2: {s2}");

let s3: &str = &s2[2..9];
println!("s3: {s3}");
}

```

На этот слайд должно уйти около 10 минут.

- `&str` представляет срез строки — неизменяемую ссылку на данные строки в кодировке UTF-8, хранящиеся в блоке памяти. Строковые литералы ("Привет") хранятся в бинарном файле программы.
- Тип `String` в Rust является оболочкой вокруг вектора байтов. Как и в случае с `Vec<T>`, он владеет своими данными.
- Как и многие другие типы, `String::from()` создаёт строку из строкового литерала; `String::new()` создаёт новую пустую строку, в которую можно добавлять данные с помощью методов `push()` и `push_str()`.
- Макрос `format!()` представляет удобный способ создания владимой строки из динамических значений. Он принимает ту же спецификацию формата, что и `println!()`.
- Вы можете заимствовать срезы `&str` из `String` через `&` и при необходимости с помощью выбора диапазона. Если вы выберете диапазон байтов, который не выровнен по границам символов, выражение вызовет панику. Итератор `chars` перебирает символы и предпочтительнее попыток правильно определить границы символов.
- Для программистов C++: рассматривайте `&str` как `std::string_view` из C++, но такой, который всегда указывает на корректную строку в памяти. Rust `String` является приближительным эквивалентом `std::string` из C++ (главное отличие: он может содержать только байты в кодировке UTF-8 и никогда не использует оптимизацию для коротких строк).
- Литералы байтовых строк позволяют создавать значение `&[u8]` напрямую:

```

fn main() {
    println!("{:?}", b"abc");
    println!("{:?}", &[97, 98, 99]);
}

```

- Необработанные строки позволяют создавать значение `&str` с отключёнными escape-последовательностями: `\n` == `\\\n`. Вы можете вставлять двойные кавычки, используя одинаковое количество `#` с обеих сторон кавычек:

```

fn main() {
    println!(r#"<a href="link.html">link</a>"#);
    println!("<a href=\"link.html\">link</a>");
}

```

9.5 Корректность ссылок

Rust применяет ряд правил к ссылкам, которые делают их всегда безопасными для использования. Одно из правил заключается в том, что ссылки никогда не могут быть `null`, что обеспечивает их безопасное использование без проверок на `null`. Другое правило, которое мы рассмотрим сейчас, заключается в том, что ссылки не могут превышать время жизни данных, на которые они указывают.

```

fn main() {
    let x_ref = {
        let x = 10;
}

```

```

    &x
};

dbg!(x_ref);
}

```

Этот слайд должен занять около 3 минут.

- Этот слайд побуждает студентов рассматривать ссылки не просто как указатели, поскольку в Rust действуют иные правила для ссылок по сравнению с другими языками.
- Остальные правила заимствования Rust мы рассмотрим на третьем дне, когда будем говорить о системе владения Rust.

Дополнительные материалы для изучения

- Эквивалентом nullability в Rust является тип `Option`, который можно использовать для придания любому типу свойства «nullable» (не только ссылкам или указателям). Однако мы ещё не вводили перечисления или сопоставление с образцом, поэтому постарайтесь не углубляться в детали на данном этапе.

9.6 Упражнение: Геометрия

Мы создадим несколько вспомогательных функций для трёхмерной геометрии, представляя точку как `[f64; 3]`. Определение сигнатур функций остаётся на ваше усмотрение.

```
// Вычислить величину вектора, суммируя квадраты его координат
// и извлекая квадратный корень. Используйте метод `sqrt()` для вычисления квадратного
// корня, например, `v.sqrt()`.
```

```

fn magnitude(...) -> f64 {
    todo!()
}

// Нормализовать вектор, вычислив его величину и разделив все его // координаты на
// эту величину.

fn normalize(...) {
    todo!()
}

// Используйте следующий `main` для проверки вашей работы.

fn main() {
    println!("Величина единичного вектора: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("Величина of {v:?}: {}", magnitude(&v));
    normalize(&mut v);
    println!("Величина of {v:?} после нормализации: {}", magnitude(&v));
}

```

9.6.1 Решение

```
// Вычисляет величину заданного вектора.  
fn magnitude(vector: &[f64; 3]) -> f64 {  
    let mut mag_squared = 0.0;  
    for coord in vector {  
        mag_squared += coord * coord;  
    }  
    mag_squared.sqrt()  
}  
  
// Изменяет величину вектора на 1.0 без изменения его направления.  
fn normalize(vector: &mut [f64; 3]) {  
    let mag = magnitude(vector);  
    for item in vector {  
        *item /= mag;  
    }  
}  
  
fn main() {  
    println!("Величина единичного вектора: {}", magnitude(&[0.0, 1.0, 0.0]));  
  
    let mut v = [1.0, 2.0, 9.0];  
    println!("Величина of {v:?}: {}", magnitude(&v));  
    normalize(&mut v);  
    println!("Величина of {v:?} после нормализации: {}", magnitude(&v));  
}
```

- Обратите внимание, что в функции `normalize` мы смогли выполнить операцию `*item /= mag` для изменения каждого элемента. Это возможно, потому что мы итерируемся с использованием изменяемой ссылки на массив, что заставляет цикл `for` предоставлять изменяемые ссылки на каждый элемент.

Глава 10

Пользовательские типы

Этот раздел займет примерно 1 час. В нем рассматривается:

Слайд	Продолжительность
Именованные структуры	10 минут
Кортежные структуры	10 минут
Перечисления	5 минут
Псевдонимы типов	2 минуты
Константы	10 минут
Статические переменные	5 минут
Упражнение: События лифта	— 15 минут

10.1 Именованные структуры

Как и в C и C++, Rust поддерживает пользовательские структуры:

```
struct Person {
    name: String,
    age: u8,
}

fn describe(person: &Person) {
    println!("{} is {} years old", person.name, person.age);
}

fn main() {
    let mut peter = Person {
        name: String::from("Peter"),
        age: 27,
    };
    describe(&peter);

    peter.age = 28;
    describe(&peter);
```

```

let name = String::from("Avery");
let age = 39;
let avery = Person { name, age };
describe(&avery);
}

```

На этот слайд должно уйти около 10 минут.

Ключевые моменты:

- Структуры работают так же, как в C или C++.
 - Как и в C++, в отличие от C, для определения типа не требуется `typedef`.
 - В отличие от C++, наследование между структурами отсутствует.
- Сейчас может быть подходящее время, чтобы сообщить, что существуют разные типы структур.
 - Структуры нулевого размера (например, `struct Foo ;`) могут использоваться при реализации трейта для некоторого типа, но не содержат данных, которые нужно хранить в самом значении.
 - Следующий слайд познакомит с кортежными структурами, которые применяются, когда имена полей не важны.
- Если у вас уже есть переменные с нужными именами, то можно создать структуру, используя сокращённый синтаксис.
- Поля структур не поддерживают значения по умолчанию. Значения по умолчанию задаются через реализацию трейта `Default`, который мы рассмотрим позже.

Дополнительные материалы для изучения

- Здесь также можно продемонстрировать синтаксис обновления структуры:


```
let jackie = Person { name: String::from("Jackie"), ..avery };
```
- Это позволяет скопировать большинство полей из старой структуры без необходимости явно указывать их все. Такой элемент всегда должен быть последним.
- Это преимущественно используется в сочетании с трейтом `Default`. Мы подробно рассмотрим синтаксис обновления структур на слайде, посвящённом трейту `Default`, поэтому здесь обсуждать его не нужно, если только студенты не зададут вопросы.

10.2 Кортежные структуры

Если имена полей не важны, можно использовать кортежную структуру:

```
struct Point(i32, i32);
```

```
fn main() {
    let p = Point(17, 23);
    println!("({}, {})", p.0, p.1);
}
```

Это часто используется для обёрток с одним полем (называемых newtypes):

```
struct PoundsOfForce(f64);
struct Newtons(f64);
```

```
fn compute_thruster_force() -> PoundsOfForce {
```

```

    todo! ( "Спросить ракетного учёного из NASA" )
}

fn set_thruster_force(force: Newtons) {
    // ...
}

fn main() {
    let force = compute_thruster_force();
    set_thruster_force(force);
}

```

На этот слайд должно уйти около 10 минут.

- Newtypes — отличный способ закодировать дополнительную информацию о значении в примитивном типе, например:
 - Число измеряется в определённых единицах: Newtons в приведённом выше примере.
 - Значение прошло проверку при создании, поэтому вам больше не нужно проверять его при каждом использовании: PhoneNumber(String) или OddNumber(u32).
- Паттерн newtype подробно рассматривается в модуле «Идиоматичный Rust».
- Показать, как добавить значение типа f64 к типу Newtons, обратившись к единственному полю newtype.
 - Rust, как правило, не любит неявные вещи, такие как автоматическое распаковывание или, например, использование булевых значений в качестве целочисленных.
 - Перегрузка операторов обсуждается на третьем дне (дженерики).
- Если у кортежной структуры нет полей, то () можно опустить. Результат представляет собой тип с нулевым размером (ZST), для которого существует только одно значение — имя типа.
 - Это характерно для типов, которые реализуют определённое поведение, но не содержат данных (представьте себе NullReader, который реализует поведение чтения, всегда возвращая EOF).
- Пример является тонкой отсылкой к аварии Mars Climate Orbiter.

10.3 Перечисления

Ключевое слово enum позволяет создать тип, имеющий несколько различных вариантов:

```

#[derive(Debug)]
enum Direction {
    Left,
    Right,
}

#[derive(Debug)]
enum PlayerMove {
    Pass,                                // Простой вариант
    Run(Direction),                        // Вариант кортежа
    Teleport { x: u32, y: u32 },          // Вариант структуры
}

fn main() {
    let dir = Direction::Left;
    let player_move: PlayerMove = PlayerMove::Run(dir);
}

```

```
    println!("На этом ходе: {player_move:?}");
}
```

На этот слайд отводится примерно 5 минут.

Ключевые моменты:

- Перечисления позволяют объединить набор значений под одним типом.
- Direction — это тип с вариантами. Существует два значения типа Direction: Direction::Left и Direction::Right.
- PlayerMove — это тип с тремя вариантами. Помимо полезных данных, Rust хранит дискриминант, чтобы во время выполнения знать, какой вариант содержится в значении PlayerMove.
- Сейчас может быть подходящее время для сравнения структур и перечислений:
 - В обоих случаях можно иметь простую версию без полей (unit struct) или версию с разными типами полей (полезные данные вариантов).
 - Вы даже можете реализовать разные варианты перечисления отдельными структурами, но тогда они не будут одним типом, как если бы все они были определены в одном перечислении.
- Rust использует минимальное пространство для хранения дискриминанта.
 - При необходимости он хранит целое число наименьшего требуемого размера.
 - Если допустимые значения вариантов не покрывают все битовые паттерны, Rust использует недопустимые битовые паттерны для кодирования дискриминанта (оптимизация «ниши»). Например, Option<&u8> хранит либо указатель на целое число, либо NULL для варианта None.
 - Вы можете управлять дискриминантом при необходимости (например, для совместимости с C):

```
#[repr(u32)]
enum Bar {
    A, // 0
    B = 10000,
    C, // 10001
}

fn main() {
    println!("A: {}", Bar::A as u32);
    println!("B: {}", Bar::B as u32);
    println!("C: {}", Bar::C as u32);
}
```

Без использования ярлыка тип дискриминанта занимает 2 байта, поскольку значение 10001 помещается в 2 байта.

Дополнительные материалы для изучения

В Rust реализованы несколько оптимизаций, позволяющих перечислениям занимать меньше памяти.

- Оптимизация нулевого указателя: для некоторых типов Rust гарантирует, что size_of::<T>() равно size_of::<Option<T>>().

Пример кода, демонстрирующий, как битовое представление может выглядеть на практике. Важно отметить, что компилятор не гарантирует данное представление, поэтому оно является полностью небезопасным.

```
use std::mem::transmute;

macro_rules! dbg_bits {
    ($e:expr, $bit_type:ty) => {
        println!("- {}: {:?}", stringify!($e), transmute::<_, $bit_type>($e));
    }
}
```

```

    } ;

}

fn main() {
    unsafe {
        println!("bool:");
        dbg_bits!(false, u8);
        dbg_bits!(true, u8);

        println!("Option<bool>:");
        dbg_bits!(None::<bool>, u8);
        dbg_bits!(Some(false), u8);
        dbg_bits!(Some(true), u8);

        println!("Option<Option<bool>>:");
        dbg_bits!(Some(Some(false)), u8);
        dbg_bits!(Some(Some(true)), u8);
        dbg_bits!(Some(None::<bool>), u8);
        dbg_bits!(None::<Option<bool>>, u8);

        println!("Option<&i32>:");
        dbg_bits!(None::<&i32>, usize);
        dbg_bits!(Some(&0i32), usize);
    }
}

```

10.4 Псевдонимы типов

Псевдоним типа создаёт имя для другого типа. Оба типа могут использоваться взаимозаменяюще.

```

enum CarryableConcreteItem {
    Left,
    Right,
}

type Item = CarryableConcreteItem;

// Псевдонимы особенно полезны для длинных и сложных типов:
use std::cell::RefCell;
use std::sync::{Arc, RwLock};
type PlayerInventory = RwLock<Vec<Arc<RefCell<Item>>>;

```

Этот слайд рассчитан примерно на 2 минуты.

- Newtype часто является лучшей альтернативой, поскольку создаёт отдельный тип. Предпочитайте struct InventoryCount(usize) вместо type InventoryCount = usize .
- Программисты на С узнают в этом аналог typedef.

10.5 const

Константы вычисляются во время компиляции, и их значения вставляются в код в местах использования:

```
const DIGEST_SIZE: usize = 3;
const FILL_VALUE: u8 = calculate_fill_value();

const fn calculate_fill_value() -> u8 {
    if DIGEST_SIZE < 10 { 42 } else { 13 }
}

fn compute_digest(text: &str) -> [u8; DIGEST_SIZE] {
    let mut digest = [FILL_VALUE; DIGEST_SIZE];
    for (idx, &b) in text.as_bytes().iter().enumerate() {
        digest[idx % DIGEST_SIZE] = digest[idx % DIGEST_SIZE].wrapping_add(b);
    }
    digest
}

fn main() {
    let digest = compute_digest("Hello");
    println!("digest: {digest:?}");
}
```

Согласно Rust RFC Book, эти функции инлайнятся при использовании.

Только функции, помеченные как `const`, могут вызываться во время компиляции для генерации `const` значений. Однако `const`-функции могут вызываться и во время выполнения.

На этот слайд должно уйти около 10 минут.

- Упомянуть, что `const` семантически аналогично `constexpr` в C++.

10.6 static

Статические переменные существуют на протяжении всего времени выполнения программы и, следовательно, не перемещаются:

```
static BANNER: &str = "Welcome to RustOS 3.14";

fn main() {
    println!("{} BANNER");
}
```

Как отмечается в Rust RFC Book, они не вставляются онлайн при использовании и имеют фактическое связанное расположение в памяти. Это полезно для небезопасного и встроенного кода, и переменная существует на протяжении всего времени выполнения программы. Когда глобальное значение не требует идентичности объекта, обычно предпочтительнее использовать `const`.

На этот слайд отводится примерно 5 минут.

- `static` похоже на изменяемые глобальные переменные в C++.
- `static` обеспечивает идентичность объекта: адрес в памяти и состояние, необходимое для типов с внутренней изменяемостью, таких как `Mutex<T>`.

Дополнительные материалы для изучения

Поскольку `static` переменные доступны из любого потока, они должны быть `Sync`. Внутренняя изменяемость возможна через `Mutex`, атомарные типы или аналогичные.

Часто используют `OnceLock` для статических переменных для поддержки инициализации при первом использовании. `OnceCell` не является `Sync` поэтому не может использоваться в данном контексте.

Потоково-локальные данные могут быть созданы с помощью макроязыка `std::thread_local`.

10.7 Упражнение: События лифта

Мы создадим структуру данных для представления события в системе управления лифтом. Ваша задача — определить типы и функции для создания различных событий. Используйте `#[derive(Debug)]` для того, чтобы типы можно было форматировать с помощью `{:?}`.

В этом упражнении требуется только создать и заполнить структуры данных так, чтобы `main` запускался без ошибок. Следующая часть курса будет посвящена извлечению данных из этих структур.

```
#!/allow(dead_code)

#[derive(Debug)]
/// Событие в системе лифта, на которое должен реагировать контроллер.
enum Event {
    // TODO: добавить необходимые варианты
}

/// Направление движения.
#[derive(Debug)]
enum Direction {
    Up,
    Down,
}

/// Кабина прибыла на указанный этаж.
fn car_arrived(floor: i32) -> Event {
    todo!()
}

/// Двери автомобиля открылись.
fn car_door_opened() -> Event {
    todo!()
}

/// Двери автомобиля закрылись.
fn car_door_closed() -> Event {
    todo!()
}

/// В лобби лифта на указанном этаже была нажата кнопка направления.
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    todo!()
}
```

```

}

/// В кабине лифта была нажата кнопка этажа .
fn car_floor_button_pressed(floor: i32) -> Event {
    todo!()
}

fn main() {
    println!(
        "Пассажир на первом этаже нажал кнопку вверх: {:?}", lobby_call_
        button_pressed(0, Direction::Up)
    );
    println!("Кабина прибыла на первый этаж: {:?}", car_arrived(0));
    println!("Дверь кабины открылась: {:?}", car_door_opened());
    println!(
        "Пассажир нажал кнопку третьего этажа: {:?}", car_floor_
        button_pressed(3)
    );
    println!("Дверь машины закрыта: {:?}", car_door_closed());
    println!("Машинка прибыла на 3-й этаж: {:?}", car_arrived(3));
}

```

На этот слайд и его под-слайды следует выделить примерно 15 минут.

- Если студенты спрашивают о `#![allow(dead_code)]` в начале упражнения, это необходимо, поскольку единственное, что мы делаем с типом `Event` — выводим его на печать. Из-за особенностей проверки неиспользуемого кода компилятором он считает, что этот код не используется. Для целей данного упражнения это можно игнорировать.

10.7.1 Решение

```

#![allow(dead_code)]

#[derive(Debug)]
/// Событие в системе лифта, на которое должен реагировать контроллер.
enum Event {
    /// Кнопка была нажата .
    ButtonPressed(Button),

    /// Машинка прибыла на указанный этаж .
    CarArrived(Floor),

    /// Двери машины открылись .
    CarDoorOpened,

    /// Двери машины закрылись .
    CarDoorClosed,
}

/// Этаж представлен целым числом .
type Floor = i32;

```

```

/// Направление движения.
#[derive(Debug)]
enum Direction {
    Up,
    Down,
}

/// Кнопка, доступная пользователю.
#[derive(Debug)]
enum Button {
    /// Кнопка в вестибюле лифта на указанном этаже.
    LobbyCall(Direction, Floor),

    /// Кнопка этажа внутри кабины.
    CarFloor(Floor),
}

/// Кабина прибыла на указанный этаж.
fn car_arrived(floor: i32) -> Event {
    Event::CarArrived(floor)
}

/// Двери автомобиля открылись.
fn car_door_opened() -> Event {
    Event::CarDoorOpened
}

/// Двери автомобиля закрылись.
fn car_door_closed() -> Event {
    Event::CarDoorClosed
}

/// В лобби лифта на указанном этаже была нажата кнопка направления.
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    Event::ButtonPressed(Button::LobbyCall(dir, floor))
}

/// В кабине лифта была нажата кнопка этажа.
fn car_floor_button_pressed(floor: i32) -> Event {
    Event::ButtonPressed(Button::CarFloor(floor))
}

fn main() {
    println!(
        "Пассажир на первом этаже нажал кнопку вверх: {:?}" , lobby_call_
        button_pressed(0, Direction::Up)
    );
    println!("Кабина прибыла на первый этаж: {:?}", car_arrived(0));
    println!("Дверь кабины открылась: {:?}", car_door_opened());
    println!(
        "Пассажир нажал кнопку 3-го этажа: {:?}" ,

```

```
    car_floor_button_pressed(3)
);
println!("Дверь машины закрыта: {:?}", car_door_closed());
println!("Машина прибыла на 3-й этаж: {:?}", car_arrived(3));
}
```

Часть III

День 2: Утро

Глава 11

Добро пожаловать во второй день

Теперь, когда мы достаточно познакомились с Rust, сегодня мы сосредоточимся на системе типов Rust:

- Сопоставление с образцом: извлечение данных из структур.
- Методы: связывание функций с типами.
- Трейты: поведение, общее для нескольких типов.
- Джениерики: параметризация типов другими типами.
- Типы и трейты стандартной библиотеки: обзор богатой стандартной библиотеки Rust.
- Замыкания: указатели на функции с данными.

Расписание

С учётом 10-минутных перерывов эта сессия займет около 2 часов 45 минут. В ней содержится:

Сегмент	Продолжительность
Приветствие	3 минуты
Сопоставление с образцом	50 минут
Методы и трейты	45 минут
Обобщения	45 минут

Глава 12

Сопоставление с образцом

Этот раздел займет около 50 минут. В нем рассматривается:

Слайд	Продолжительность
Неотразимые шаблоны	5 минут
Сопоставление значений	10 минут
Деструктуризация структур	4 минуты
Деструктуризация перечислений	4 минуты
Управление потоком с помощью let	10 минут
Упражнение: оценка выражений	— 15 минут

12.1 Неотразимые шаблоны

В первый день мы кратко рассмотрели, как шаблоны могут использоваться для деструктуризации составных значений. Давайте повторим это и обсудим несколько других возможностей, которые могут выражать шаблоны:

```
fn takes_tuple(tuple: (char, i32, bool)) {
    let a = tuple.0;
    let b = tuple.1;
    let c = tuple.2;

    // Это делает то же самое, что и выше.
    let (a, b, c) = tuple;

    // Игнорируется первый элемент, связываются только второй и третий.
    let (_, b, c) = tuple;

    // Игнорируется всё, кроме последнего элемента.
    let (.., c) = tuple;
}

fn main() {
    takes_tuple(('a', 777, true));
}
```

На этот слайд отводится примерно 5 минут.

- Все продемонстрированные шаблоны являются *неотразимыми*, то есть они всегда совпадают со значением справа.
- Шаблоны специфичны для типа, включая неотразимые шаблоны. Попробуйте добавить или удалить элемент из кортежа и посмотрите на возникающие ошибки компилятора.
- Имена переменных — это шаблоны, которые всегда совпадают и связывают совпадшее значение с новой переменной с этим именем.
- `_` — это шаблон, который всегда совпадает с любым значением, отбрасывая совпадшее значение.
- `...` Позволяет игнорировать несколько значений одновременно.

Дополнительные материалы для изучения

- Вы также можете продемонстрировать более продвинутые применения `...`, например, игнорирование средних элементов кортежа.

```
fn принимает_кортеж(tuple: (char, i32, bool, u8))
{ let (first, ..., last) = tuple;
}
```

- Все эти шаблоны также применимы к массивам:

```
fn принимает_массив(array: [u8; 5])
{ let [first, ..., last] = array;
}
```

12.2 Сопоставление значений

Ключевое слово `match` позволяет сопоставлять значение с одним или несколькими шаблонами. Шаблоны могут быть простыми значениями, аналогично `switch` в C и C++, но также могут использоваться для выражения более сложных условий:

```
#[rustfmt::skip]
fn main() {
    let input = 'x';
    match input {
        'q'                      => println!("Выход"),
        'a' | 's' | 'w' | 'd'    => println!("Движение"),
        '0'..='9'                 => println!("Ввод числа"), key if
            key.is_lowercase()   => println!("Строчная буква: {key}"),
            _                     => println!("Something else"),
    }
}
```

Переменная в шаблоне (`key` в этом примере) создаёт связывание, которое можно использовать внутри ветви `match`. Мы подробнее рассмотрим это на следующем слайде.

Ограничение `match guard` заставляет ветвь совпадать только если условие истинно. Если условие ложно, `match` продолжит проверку последующих случаев.

На этот слайд должно уйти около 10 минут.

Ключевые моменты:

- Можно указать, как некоторые конкретные символы используются в шаблоне
 - | какили
 - .. может расширяться столько, сколько необходимо
 - 1..=5 представляет включающий диапазон
 - _ является подстановочным символом
- Match guards как отдельная синтаксическая конструкция важны и необходимы, когда нужно лаконично выразить более сложные идеи, чем это позволяют одни только шаблоны.
- Они не равнозначны отдельному if выражению внутри ветви match. If-выражение внутри блока ветви (после =>) выполняется после выбора ветви match. Неудача условия if внутри этого блока не приведёт к рассмотрению других ветвей исходного выражения match.
- Условие, определённое в guard, применяется к каждому выражению в шаблоне с |.
- Обратите внимание, что нельзя использовать существующую переменную в качестве условия в ветке match, так как это будет интерпретировано как шаблон имени переменной, создающий новую переменную, которая затеняет существующую. Например:

```
let expected = 5;
match 123 {
    expected => println!("Ожидаемое значение 5, фактическое {expected}"),
    => println!("Значение было другим"),
}
```

Здесь мы пытаемся сопоставить число 123, где в первом случае хотим проверить, равно ли значение 5. Наивно ожидается, что первый случай не совпадёт, поскольку значение не равно 5, но вместо этого это интерпретируется как шаблон переменной , который всегда совпадает, то есть первая ветка всегда будет выполнена. Если вместо этого использовать константу, то всё будет работать как ожидается.

Дополнительно для изучения

- Другой элемент синтаксиса шаблонов, который можно показать студентам, — это синтаксис @ , связывающий часть шаблона с переменной. Например:

```
let opt = Some(123);
match opt {
    outer @ Some(inner) => {
        println!("outer: {outer:?}", inner: {inner});
    }
    None => {}
}
```

В этом примере inner имеет значение 123, которое оно извлекло из Option посредством деструктуризации, а outer захватывает всё выражение Some(inner), поэтому содержит полный Option::Some(123) . Это используется редко, но может быть полезно в более сложных шаблонах

12.3 Структуры

Как и кортежи, структуры также можно деструктурировать с помощью сопоставления:

```
struct Foo {
    x: (u32, u32),
    y: u32,
}

#[rustfmt::skip]
fn main() {
    let foo = Foo { x: (1, 2), y: 3 };
    match foo {
        Foo { y: 2, x: i } => println!("y = 2, x = {i:?}"),
        Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
        Foo { y, .. }         => println!("y = {y}, остальные поля были проигнорированы"),
    }
}
```

Этот слайд должен занять примерно 4 минуты.

- Измените литеральные значения в `foo`, чтобы они соответствовали другим шаблонам.
- Добавьте новое поле в `Foo` и внесите необходимые изменения в шаблон.

Дополнительные материалы для изучения

- Попробуйте `match &foo` и проверьте тип захватов. Синтаксис шаблона остаётся прежним, но захваты становятся разделяемыми ссылками. Это называется эргономикой `match` и часто полезно при использовании `match self` в методах перечисления.
 - Аналогичный эффект возникает с `match &mut foo : захваты становятся эксплюзивными ссылками.`
- Различие между захватом и константным выражением может быть трудно заметить. Попробуйте заменить `2` во втором варианте на переменную и убедитесь, что это тонко не работает.
Измените её на `const` и убедитесь, что снова работает.

12.4 Перечисления

Как и кортежи, перечисления также можно деструктурировать с помощью сопоставления:

Шаблоны также можно использовать для связывания переменных с частями значений.
. Так вы исследуете структуру ваших типов. Начнём с простого `enum` типа:

```
enum Result {
    Ok(i32),
    Err(String),
}

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("cannot divide {} into two equal parts"))
    }
}
```

```

    }
}

fn main() {
    let n = 100;
    match divide_in_two(n) {
        Result::Ok(half) => println!("{} divided in two is {}", half),
        Result::Err(msg) => println!("sorry, an error happened: {}", msg),
    }
}

```

Здесь мы использовали ветви для деструктуризации значения `Result`. В первой ветви переменная `half` привязывается к значению внутри варианта `Ok`. Во второй ветви переменная `msg` привязывается к сообщению об ошибке.

Этот слайд должен занять примерно 4 минуты.

- Выражение `if / else` возвращает enum, который затем распаковывается с помощью `match`.
- Вы можете попробовать добавить третий вариант в определение enum и отображать ошибки при выполнении кода. Укажите места, где ваш код в настоящее время является неисчерпывающим, и как компилятор пытается предоставить вам подсказки.
- Значения в вариантах enum могут быть доступны только после сопоставления с образцом.
- Продемонстрируйте, что происходит, когда сопоставление является неисчерпывающим. Обратите внимание на преимущество, которое предоставляет компилятор Rust, подтверждая, что все случаи обработаны.
- Продемонстрируйте синтаксис варианта в стиле `struct`, добавив его в определение enum и в `match`. Укажите, как это синтаксически похоже на сопоставление со `struct`.

12.5 Управление потоком с помощью `let`

В Rust существуют несколько конструкций управления потоком, которые отличаются от других языков. Они используются для сопоставления с образцом:

- выражения `if let`
- выражения `while let`
- выражения `let else`

12.5.1 Выражения `if let`

Выражение `if let` позволяет выполнять разный код в зависимости от того, соответствует ли значение образцу:

используйте `std::time::Duration`:

```

fn sleep_for(secs: f32) {
    let result = Duration::try_from_secs_f32(secs);

    if let Ok(duration) = result {
        std::thread::sleep(duration);
        println!("спал в течение {}", duration);
    }
}

fn main() {
}

```

```

    sleep_for(-10.0);
    sleep_for(0.8);
}

```

- В отличие от `match`, `if let` не требует покрытия всех ветвей. Это может сделать его более лаконичным, чем `match`.
- Распространённое применение — обработка значений `Some` при работе с `Option`.
- В отличие от `match`, `if let` поддерживает охранные выражения для сопоставления с образцом.
- С `else` этот оператор может использоваться как выражение.

12.5.2 `while let` операторы

Как и с `if let`, существует вариант `while let`, который многократно проверяет значение на соответствие образцу:

```

fn main() {
    let mut name = String::from("Полное руководство по Rust");
    while let Some(c) = name.pop() {
        dbg!(c);
    }
    // (Существуют более эффективные способы обращения строки!)
}

```

Здесь `String::pop` возвращает `Some(c)` до тех пор, пока строка не станет пустой, после чего возвращается `None`. `while let` позволяет нам продолжать итерацию по всем элементам.

- Отметьте, что цикл `while let` будет продолжаться, пока значение соответствует шаблону.
- Вы можете переписать цикл `while let` как бесконечный цикл с оператором `if let`, который прерывается, когда для `name.pop()` нет значения для извлечения. `while let` предоставляет синтаксический сахар для описанного выше сценария.
- Эта форма не может использоваться как выражение, поскольку она может не иметь значения, если условие ложно.

12.5.3 `let else` операторы

Для распространённого случая сопоставления с шаблоном и возврата из функции используйте `let else`. Случай “`else`” должен приводить к выходу из блока (`return`, `break` или `panic` — всё, кроме простого завершения блока).

```

fn hex_or_die_trying(maybe_string: Option<String>) -> Result<u32, String> {
    let s = if let Some(s) = maybe_string {
        s
    } else {
        return Err(String::from("получено None"));
    };

    let first_byte_char = if let Some(first) = s.chars().next() {
        first
    } else {
        return Err(String::from("получена пустая строка"));
    };
}

```

```

let digit = if let Some(digit) = first_byte_char.to_digit(16) {
    digit
} else {
    return Err(String::from("не является шестнадцатеричным символом"));
};

Ok(digit)
}

fn main() {
    println!("результат: {:?}", hex_or_die_trying(Some(String::from("foo"))));
}

```

Переписанная версия:

```

fn hex_or_die_trying(maybe_string: Option<String>) -> Result<u32, String> { let
    Some(s) = maybe_string else { return Err(String::from(
        "получено None")); }

    let Some(first_byte_char) = s.chars().next() else {
        return Err(String::from("получена пустая строка"));
    };

    let Some(digit) = first_byte_char.to_digit(16) else {
        return Err(String::from("not a hex digit"));
    };

    Ok(digit)
}

```

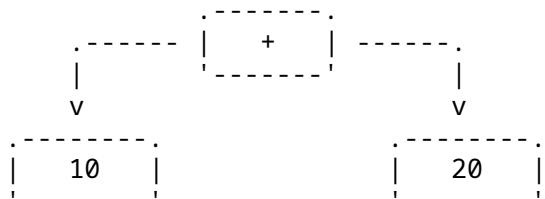
Дополнительные материалы для изучения

- Такой ранний возврат в управляемом потоке часто встречается в коде обработки ошибок Rust, где вы пытаетесь получить значение из `Result`, возвращая ошибку, если `Result` является `Err`.
- Если студенты спросят, вы также можете продемонстрировать, как реальный код обработки ошибок пишется с использованием `?.`

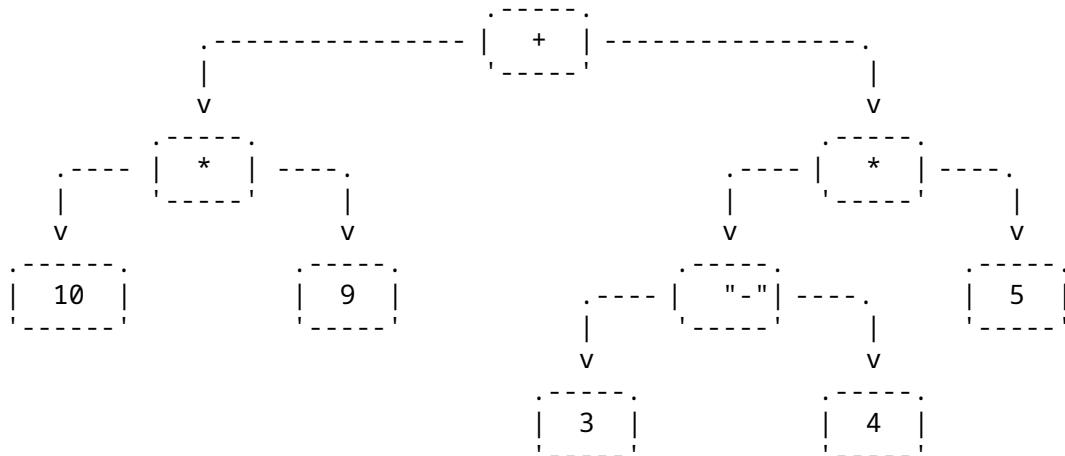
12.6 Упражнение: Вычисление выражений

Давайте напишем простой рекурсивный вычислитель арифметических выражений.

Примером небольшого арифметического выражения может быть `10 + 20`, которое вычисляется в `30`. Мы можем представить выражение в виде дерева:



Большее и более сложное выражение — это $(10 * 9) + ((3 - 4) * 5)$, которое вычисляется в 85. Мы представляем это в виде гораздо более крупного дерева:



В коде мы будем представлять дерево с помощью двух типов:

```
/// Операция, выполняемая над двумя подвыражениями.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// Выражение в виде дерева.
#[derive(Debug)]
enum Expression {
    /// Операция над двумя подвыражениями.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },
    /// Литеральное значение
    Value(i64),
}
```

Тип `Box` является умным указателем и будет подробно рассмотрен позже в курсе. Выражение можно «упаковать» с помощью `Box::new`, как показано в тестах. Для вычисления упакованного выражения используйте оператор разыменования (`*`), чтобы «распаковать» его: `eval(*boxed_expr)`.

Скопируйте и вставьте код в Rust playground и начните реализовывать функцию `eval`. Итоговый продукт должен успешно проходить тесты. Полезно использовать `todo!()` и добиваться прохождения тестов по одному. Вы также можете временно пропустить тест с помощью `#[ignore]`:

```
#[test]
#[ignore]
fn test_value() { ... }

/// Операция, выполняемая над двумя подвыражениями.
#[derive(Debug)]
```

```

enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// Выражение в виде дерева.
#[derive(Debug)]
enum Expression {
    /// Операция над двумя подвыражениями.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// Литеральное значение
    Value(i64),
}

fn eval(e: Expression) -> i64 {
    todo!()
}

#[test]
fn test_value() {
    assert_eq!(eval(Expression::Value(19)), 19);
}

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        30
    );
}

#[test]
fn test_recursion() {
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        })
    };
}

```

```

        }),
        right: Box::new(Expression::Value(5)),
    };
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        85
    );
}

#[test]
fn test_zeros() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Mul,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
}
}

#[test]
fn test_div() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(2)),
        }),
        5
    )
}

```

```
}
```

12.6.1 Решение

```
/// Операция, выполняемая над двумя подвыражениями.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// Выражение в виде дерева.
#[derive(Debug)]
enum Expression {
    /// Операция над двумя подвыражениями.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// Литеральное значение
    Value(i64),
}

fn eval(e: Expression) -> i64 {
    match e {
        Expression::Op { op, left, right } => {
            let left = eval(*left);
            let right = eval(*right);
            match op {
                Operation::Add => left + right,
                Operation::Sub => left - right,
                Operation::Mul => left * right,
                Operation::Div => left / right,
            }
        }
        Expression::Value(v) => v,
    }
}

#[test]
fn test_value() {
    assert_eq!(eval(Expression::Value(19)), 19);
}

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        })
    );
}
```

```

        } ),
        30
    );
}

#[test]
fn test_recursion() {
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        }),
        right: Box::new(Expression::Value(5)),
    };
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        85
    );
}

#[test]
fn test_zeros() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Mul,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {

```

```
        op: Operation::Sub,
        left: Box::new(Expression::Value(0)),
        right: Box::new(Expression::Value(0))
    } ) ,
    0
);
}

#[test]
fn test_div() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(2)),
        } ) ,
        5
    )
}
```

Глава 13

Методы и трейты

Этот раздел займет примерно 45 минут. В нем содержится:

Слайд	Продолжительность
Методы	10 минут
трейты	15 минут
Автоматическая реализация	3 минуты
Упражнение: Универсальный логгер	15 минут

13.1 Методы

Rust позволяет ассоциировать функции с вашими новыми типами. Вы делаете это с помощью `impl`блока :

```
#[derive(Debug)]
struct CarRace {
    name: String,
    laps: Vec<i32>,
}

impl CarRace {
    // Нет получателя, статический метод
    fn new(name: &str) -> Self {
        Self { name: String::from(name), laps: Vec::new() }
    }

    // Исключительный заимствованный доступ на чтение и запись к self
    fn add_lap(&mut self, lap: i32) {
        self.laps.push(lap);
    }

    // Совместный и только для чтения заимствованный доступ к self
    fn print_laps(&self) {
        println!("Записано {} кругов для {}: ", self.laps.len(), self.name);
        for (idx, lap) in self.laps.iter().enumerate() {
```

```

        println!( "Круг {idx}: {lap} сек");
    }

// Исключительное владение self (рассмотрено позже)
fn finish(self) {
    let total: i32 = self.laps.iter().sum();
    println!("Гонка {} завершена, общее время кругов: {}", self.name, total);
}

fn main() {
    let mut race = CarRace::new( "Гран-при Монако" );
    race.add_lap(70);
    race.add_lap(68);
    race.print_laps();
    race.add_lap(71);
    race.print_laps();
    race.finish();
    // race.add_lap(42);
}

```

Аргументы `self` определяют «приёмник» — объект, над которым выполняется метод.
Существует несколько распространённых приёмников для метода:

- `&self`: заимствует объект у вызывающего с помощью разделяемой и неизменяемой ссылки. Объект может быть использован повторно после этого.
- `&mut self`: заимствует объект у вызывающего с помощью уникальной и изменяемой ссылки. Объект может быть использован повторно после этого.
- `self`: принимает владение объектом и перемещает его от вызывающего. Метод становится владельцем объекта. Объект будет удалён (деаллоцирован) при возврате из метода, если его владение явно не передано. Полное владение не означает автоматически изменяемость.
- `mut self`: то же самое, но метод может изменять объект.
- Отсутствие приёмника: это становится статическим методом структуры. Обычно используется для создания конструкторов, которые по соглашению называются `new`.

На этот слайд следует отвести около 8 минут.

Ключевые моменты:

- Полезно вводить методы, сравнивая их с функциями.
 - Методы вызываются на экземпляре типа (например, структуры или перечисления), первый параметр представляет экземпляр как `self`.
 - Разработчики могут использовать методы, чтобы воспользоваться синтаксисом приёмника метода и лучше организовать код. Используя методы, мы можем хранить весь код реализации в одном предсказуемом месте.
 - Обратите внимание, что методы также можно вызывать как ассоциированные функции, явно передавая приёмник, например, `CarRace::add_lap(&mut race, 20)`.
- Укажите использование ключевого слова `self`, являющегося приёмником метода.
 - Поясните, что это сокращённое обозначение для `self: Self` и, возможно, продемонстрируйте, как можно использовать имя структуры.
 - Объясните, что `Self` является псевдонимом типа для типа, в котором находится `impl`блок, и может применяться

в других частях блока.

- Отметьте, что `self` используется аналогично другим структурам, и для обращения к отдельным полям можно применять точечную нотацию.
- Это подходящий момент, чтобы продемонстрировать различия между `&self` и `self`, попробовав вызвать `finish` дважды.
- Помимо вариантов с `self`, существуют специальные обёрточные типы, разрешённые в качестве типов-приёмников, например `Box<Self>`.

13.2 Трейты

Rust позволяет абстрагироваться от типов с помощью трейтов. Они аналогичны интерфейсам:

```
trait Pet {  
    /// Возвращает предложение от этого питомца.  
    fn talk(&self) -> String;  
  
    /// Выводит строку в терминал с приветствием этого питомца.  
    fn greet(&self);  
}
```

На этот слайд и его под-слайды следует выделить примерно 15 минут.

- Трейт определяет набор методов, которые должны быть реализованы типами для реализации данного трейта.
- В разделе «Обобщения» далее мы рассмотрим, как создавать функциональность, обобщённую для всех типов, реализующих трейт.

13.2.1 Реализация трейтов

```
trait Pet {  
    fn talk(&self) -> String;  
  
    fn greet(&self) {  
        println!("О, ты такой милый! Как тебя зовут? {}", self.talk());  
    }  
}  
  
struct Dog {  
    name: String,  
    age: i8,  
}  
  
impl Pet for Dog {  
    fn talk(&self) -> String {  
        format!("Гав, меня зовут {}!", self.name)  
    }  
}  
  
fn main() {  
    let fido = Dog { name: String::from("Fido"), age: 5 };  
    dbg!(fido.talk());  
}
```

```
fido.greet();  
}
```

- Для реализации Trait для Type используется блок `impl Trait for Type { . . . }`.
- В отличие от интерфейсов Go, простого совпадения методов недостаточно: тип `Cat` с методом `talk()` не будет автоматически удовлетворять `Pet`, если он не находится в блоке `impl Pet`.
- Трейты могут предоставлять реализации по умолчанию для некоторых методов. Реализации по умолчанию могут опираться на все методы трейта. В этом случае `greet` предоставлен и опирается на `alk`.
- Для данного типа допускается несколько `impl` блоков. Это включает как собственные `impl` блоки, так и `impl` блоки трейтов. Аналогично, для данного типа может быть реализовано несколько трейтов (и часто типы реализуют множество трейтов!). `impl` блоки могут быть распределены по нескольким модулям или файлам.

13.2.2 Супертрейты

Трейт может требовать, чтобы типы, его реализующие, также реализовывали другие трейты, называемые *supertraits*. Здесь любой тип, реализующий `Pet`, должен реализовывать `Animal`.

```
trait Animal {  
    fn leg_count(&self) -> u32;  
}  
  
trait Pet: Animal {  
    fn name(&self) -> String;  
}  
  
struct Dog(String);  
  
impl Animal for Dog {  
    fn leg_count(&self) -> u32 {  
        4  
    }  
}  
  
impl Pet for Dog {  
    fn name(&self) -> String {  
        self.0.clone()  
    }  
}  
  
fn main() {  
    let puppy = Dog(String::from("Rex"));  
    println!("{} has {} legs", puppy.name(), puppy.leg_count());  
}
```

Это иногда называют «наследованием трейтов», но студентам не следует ожидать, что это будет работать как наследование в объектно-ориентированном программировании. Это просто задаёт дополнительное требование к реализациям трейта.

13.2.3 Ассоциированные типы

Ассоциированные типы — это типы-заполнители, которые определяются реализацией трейта.

```
#[derive(Debug)]
struct Meters(i32);
#[derive(Debug)]
struct MetersSquared(i32);

trait Multiply {
    type Output;
    fn multiply(&self, other: &Self) -> Self::Output;
}

impl Multiply for Meters {
    type Output = MetersSquared;
    fn multiply(&self, other: &Self) -> Self::Output {
        MetersSquared(self.0 * other.0)
    }
}

fn main() {
    println!("{}:{}?", Meters(10).multiply(&Meters(20)));
}
```

- Ассоциированные типы иногда также называют «выходными типами». Ключевое наблюдение заключается в том, что реализатор, а не вызывающий, выбирает этот тип.
- Многие трейты стандартной библиотеки имеют ассоциированные типы, включая арифметические операторы и Iterator.

13.3 Производные реализации

Поддерживаемые трейты могут быть автоматически реализованы для ваших пользовательских типов следующим образом:

```
#[derive(Debug, Clone, Default)]
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
    let p1 = Player::default(); // Трейт Default добавляет конструктор `default`.
    let mut p2 = p1.clone(); // Трейт Clone добавляет метод `clone`.
    p2.name = String::from("EldurScrollz");
    // Трейт Debug добавляет поддержку вывода с помощью `{:?}`.
    println!("{} vs. {}", p1, p2);
}
```

Этот слайд должен занять около 3 минут.

- Производные реализации выполняются с помощью макросов, и многие crates предоставляют полезные derive-макросы.

для добавления полезной функциональности. Например, `serde` может автоматически реализовать поддержку сериализации для структуры с помощью `#[derive(Serialize)]`.

- Автоматическая реализация обычно предоставляется для трейтов, которые имеют стандартную шаблонную реализацию, корректную в большинстве случаев. Например, продемонстрируем, как ручная реализация `Clone` может быть повторяющейся по сравнению с использованием `derive` для трейта:

```
impl Clone for Player {
    fn clone(&self) -> Self {
        Player {
            name: self.name.clone(),
            strength: self.strength.clone(),
            hit_points: self.hit_points.clone(),
        }
    }
}
```

Не все вызовы `.clone()` в приведённом примере необходимы, но это демонстрирует общий шаблон, которому следуют ручные реализации, что помогает студентам понять преимущества использования `derive`.

13.4 Упражнение: Трейт логгера

Давайте разработаем простой инструмент логирования, используя трейт `Logger` с методом `log`. Код, который может логировать свой прогресс, тогда может принимать параметр типа `&impl Logger`. В процессе тестирования это может записывать сообщения в тестовый лог-файл, тогда как в релизной сборке сообщения будут отправляться на сервер логирования.

Однако приведённый ниже `StderrLogger` регистрирует все сообщения независимо от уровня подробности. Ваша задача — написать `VerbosityFilter` тип, который будет игнорировать сообщения с уровнем подробности выше максимального.

Это распространённый шаблон: структура, обрабатывающая реализацию трейта и реализующая тот же трейт, добавляя при этом дополнительное поведение. В разделе «Обобщения» мы рассмотрим, как сделать обёртку обобщённой по типу, который она обрабатывает.

```
trait Logger {
    // Записать сообщение с указанным уровнем подробности.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Записывать только сообщения с уровнем подробности не выше указанного.
struct VerbosityFilter {
    max_verbosity: u8,
    inner: StderrLogger,
}
```

```
// TODO: Реализовать трейд `Logger` для `VerbosityFilter`.

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

13.4.1 Решение

```
trait Logger {
    // Записать сообщение с указанным уровнем подробности.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Записывать только сообщения с уровнем подробности не выше указанного.
struct VerbosityFilter {
    max_verbosity: u8,
    inner: StderrLogger,
}

impl Logger for VerbosityFilter {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

Глава 14

Обобщения

Этот раздел займет примерно 45 минут. В нем содержится:

Слайд	Продолжительность
Обобщённые функции	5 минут
Ограничения трейтов	10 минут
Обобщённые типы данных	10 минут
impl Trait	5 минут
dyn Trait	5 минут
Упражнение: обобщённая функция min	10 минут

14.1 Джениерик-функции

Rust поддерживает обобщения, которые позволяют абстрагировать алгоритмы или структуры данных (например, сортировку или бинарное дерево) по используемым или хранимым типам.

```
fn pick<T>(cond: bool, left: T, right: T) -> T {
    if cond { left } else { right }
}

fn main() {
    println!("picked a number: {:?}", pick(true, 222, 333));
    println!("picked a string: {:?}", pick(false, 'L', 'R'));
}
```

На этот слайд отводится примерно 5 минут.

- Полезно показать мономорфизированные версии функции `pick` — либо до обсуждения обобщённой функции `pick`, чтобы продемонстрировать, как обобщения уменьшают дублирование кода, либо после обсуждения обобщений, чтобы показать, как работает мономорфизация.

```
fn pick_i32(cond: bool, left: i32, right: i32) -> i32 {
    if cond { left } else { right }
}
```

```
fn pick_char(cond: bool, left: char, right: char) -> char {
    if cond { left } else { right }
}
```

- Rust выводит тип для T на основе типов аргументов и возвращаемого значения.
- В этом примере мы используем только примитивные типы i32 и char для T, но здесь можно использовать любой тип, включая пользовательские.

```
struct Foo {
    val: u8,
}
```

```
pick(false, Foo { val: 7 }, Foo { val: 99 });
```

- Это похоже на шаблоны C++, но Rust частично компилирует обобщённую функцию немедленно, поэтому она должна быть корректной для всех типов, удовлетворяющих ограничениям. Например, попробуйте изменить pick так, чтобы она возвращала left + right если cond равно false. Даже если используется только инстанцирование pick с целыми числами, Rust всё равно считает это некорректным. C++ позволил бы сделать это.
- Обобщённый код преобразуется в не обобщённый на основе точек вызова. Это абстракция с нулевыми издержками: вы получаете точно такой же результат, как если бы вручную написали структуры данных без абстракции.

14.2 Ограничения трейтов

При работе с обобщениями часто требуется, чтобы типы реализовывали определённый трейд, чтобы можно было вызывать методы этого трейда.

Вы можете сделать это с помощью T: трейты:

```
fn duplicate<T: Clone>(a: T) -> (T, T) {
    (a.clone(), a.clone())
}
```

```
struct NotCloneable;
```

```
fn main() {
    let foo = String::from("foo");
    let pair = duplicate(foo);
    println!("{}{:?}{}", "pair:", pair);
}
```

На этот слайд следует отвести около 8 минут.

- Попробуйте создать NotCloneable и передать его в duplicate.
- Когда необходимо несколько трейтов, используйте + для их объединения.
- Покажите where-еклауз, с которой студенты столкнутся при чтении кода.

```
fn duplicate<T>(a: T) -> (T, T)
where
    T: Clone,
```

```
        (a.clone(), a.clone())
    }
```

- Это упрощает сигнатуру функции, если у вас много параметров.
- У неё есть дополнительные возможности, делающие её более мощной.
 - * Если кто-то спросит, дополнительная возможность в том, что тип слева от `:` может быть произвольным, например `Option<T>`.
- Обратите внимание, что Rust пока не поддерживает специализацию. Например, учитывая исходный `duplicate`, недопустимо добавлять специализированный `duplicate(a: u32)`.

14.3 Джениерик-типы данных

Вы можете использовать обобщения для абстрагирования от конкретного типа поля. Возвращаясь к упражнению из предыдущего раздела:

```
pub trait Logger {
    // Записать сообщение с указанным уровнем подробности.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Записывать только сообщения с уровнем подробности не выше указанного.
struct VerbosityFilter<L> {
    max_verbosity: u8,
    inner: L,
}

impl<L: Logger> Logger for VerbosityFilter<L> {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

На этот слайд должно уйти около 10 минут.

- Вопрос: почему Луказывается дважды `impl<L: Logger> .. VerbosityFilter<L>`? Разве это

не избыточно?

- Это связано с тем, что это секция реализации для обобщённого типа. Они являются независимыми обобщениями.
 - Это означает, что эти методы определены для любого L.
 - Возможно написать `impl VerbosityFilter<StderrLogger> { .. }.`
 - * `VerbosityFilter` по-прежнему является обобщённым, и вы можете использовать `VerbosityFilter<f64>`, но методы в этом блоке будут доступны только для `VerbosityFilter<StderrLogger>`
- Обратите внимание, что мы не накладываем ограничение трейта на сам тип `VerbosityFilter`. Вы также можете накладывать ограничения там, но в Rust обычно ограничения трейтов ставятся только на `impl`-блоки.

14.4 Джинерик-трейты

Трейты также могут быть обобщёнными, так же как типы и функции. Параметры трейта получают конкретные типы при его использовании. Например, трейд `From<T>` используется для определения преобразований типов:

```
pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}

#[derive(Debug)]
struct Foo(String);

impl From<u32> for Foo {
    fn from(from: u32) -> Foo {
        Foo(format!("Converted from integer: {}", from))
    }
}

impl From<bool> for Foo {
    fn from(from: bool) -> Foo {
        Foo(format!("Converted from bool: {}", from))
    }
}

fn main() {
    let from_int = Foo::from(123);
    let from_bool = Foo::from(true);
    dbg!(from_int);
    dbg!(from_bool);
}
```

- Трейт `From` будет рассмотрен позже в курсе, но его определение в документации `std` является простым и приведено здесь для справки.
- Реализации трейта не обязаны охватывать все возможные параметры типа. Вызов `Foo::from("hello")` не скомпилируется, поскольку для `Foo` не существует реализации `From<&str>`.
- Обобщённые трейты принимают типы в качестве «входных» параметров, тогда как ассоциированные типы представляют собой своего рода «выходной» тип. Трейт может иметь несколько реализаций для разных входных типов.
- Фактически, Rust требует, чтобы для любого типа существовала не более одной реализации трейта.

Т. В отличие от некоторых других языков, в Rust отсутствует эвристика для выбора «наиболее конкретного» совпадения. Ведётся работа по добавлению этой поддержки, называемой специализацией.

14.5 `impl Trait`

Аналогично ограничениям трейтов, синтаксис `impl Trait` может использоваться в аргументах функций и возвращаемых значениях:

```
// Синтаксический сахар для:
// fn add_42_millions<T: Into<i32>>(x: T) -> i32 {
fn add_42_millions(x: impl Into<i32>) -> i32 {
    x.into() + 42_000_000
}

fn pair_of(x: u32) -> impl std::fmt::Debug {
    (x + 1, x - 1)
}

fn main() {
    let many = add_42_millions(42_i8);
    dbg!(many);
    let many_more = add_42_millions(10_000_000);
    dbg!(many_more);
    let debuggable = pair_of(27);
    dbg!(debuggable);
}
```

На этот слайд отводится примерно 5 минут.

`impl Trait` позволяет работать с типами, которые вы не можете назвать. Значение `impl Trait` немного отличается в разных позициях.

- Для параметра `impl Trait` является анонимным универсальным параметром с ограничением по трейту.
- Для возвращаемого типа это означает, что возвращаемый тип — некоторый конкретный тип, реализующий трейт, без указания имени типа. Это может быть полезно, когда вы не хотите раскрывать конкретный тип в публичном API.

Вывод типов в позиции возвращаемого значения затруднён. Функция, возвращающая `impl Foo`, выбирает конкретный тип, который она возвращает, без явного указания в исходном коде. Функция, возвращающая универсальный тип, например `collect() -> B`, может возвращать любой тип, удовлетворяющий `B`, и вызывающий может выбрать конкретный тип, например с помощью `let x: Vec<_> = foo.collect()` или с использованием `turbofish foo.collect::<Vec<_>>()`.

Каков тип переменной `debuggable`? Попробуйте `let debuggable: () = ..` чтобы увидеть, какое сообщение об ошибке выводится.

14.6 `dyn` трейты

Помимо использования трейтов для статической диспетчеризации через дженерики, Rust также поддерживает их использование для стирания типа и динамической диспетчеризации через объекты трейтов:

```

struct Dog {
    name: String,
    age: i8,
}
struct Cat {
    жизни: i8,
}

trait Pet {
    fn talk(&self) -> String;
}

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Гав, меня зовут {}!", self.name)
    }
}

impl Pet для Cat { fn talk(& self
) -> String { String::from("Мяу!") }
}

// Использует дженерики и статическую диспетчеризацию .
fn generic(pet: & impl Pet) {
    println!("Привет, кто ты? {}", pet.talk());
}

// Использует стирание типа и динамическую диспетчеризацию .
fn dynamic(pet: & dyn Pet) {
    println!("Привет, кто ты? {}", pet.talk());
}

fn main() {
    let cat = Cat { жизни: 9 };
    let dog = Dog { имя: String::from("Fido"), возраст: 5 };

    generic(&cat);
    generic(&dog);

    dynamic(&cat);
    dynamic(&dog);
}

```

На этот слайд отводится примерно 5 минут.

- Обобщения, включая `impl Trait`, используют мономорфизацию для создания специализированного экземпляра функции для каждого конкретного типа, с которым обобщение инстанцируется. Это означает, что вызов метода трейта внутри обобщённой функции по-прежнему использует статическую диспетчеризацию, поскольку компилятор обладает полной информацией о типе и может определить, какую реализацию трейта применять.

- При использовании `dyn Trait` применяется динамическая диспетчеризация через виртуальную таблицу методов (vtable). Это означает, что существует единственная версия `fn dynamic`, которая используется независимо от типа передаваемого `Pet`.
- При использовании `dyn Trait` объект трейта должен находиться за некоторой формой косвенной ссылки. В данном случае это ссылка, хотя также могут использоваться умные указатели, такие как `Box` (это будет продемонстрировано на третьем дне).
- Во время выполнения `&dyn Pet` представляется как «толстый указатель», то есть пара из двух указателей: один указывает на конкретный объект, реализующий `Pet`, а другой — на vtable для реализации трейта для этого типа. При вызове метода `talk` на `&dyn Pet` компилятор ищет указатель на функцию `talk` в vtable, а затем вызывает эту функцию, передавая указатель на `Dog` или `Cat`. Компилятору не требуется знать конкретный тип `Pet` для выполнения этого.
- Тип `dyn Trait` считается «стираемым» (type-erased), поскольку на этапе компиляции отсутствует информация о конкретном типе.

14.7 Упражнение: универсальная функция `min`

В этом кратком упражнении вы реализуете универсальную функцию `min`, которая определяет минимум из двух значений с использованием трейта `Ord`.

```
use std::cmp::Ordering;

// TODO: реализовать функцию `min`, используемую в тестах.

#[test]
fn integers() {
    assert_eq!(min(0, 10), 0);
    assert_eq!(min(500, 123), 123);
}

#[test]
fn chars() {
    assert_eq!(min('a', 'z'), 'a');
    assert_eq!(min('7', '1'), '1');
}

#[test]
fn strings() {
    assert_eq!(min("hello", "goodbye"), "goodbye");
    assert_eq!(min("bat", "armadillo"), "armadillo");
}
```

Этот слайд и его подразделы должны занять примерно 10 минут.

- Продемонстрировать студентам трейты `Ord` и перечисление `Ordering`.

14.7.1 Решение

```
use std::cmp::Ordering;
```

```

fn min<T: Ord>(l: T, r: T)    -> T  {
    match l.cmp(&r) {
        Ordering::Less | Ordering::Equal => l,
        Ordering::Greater => r,
    }
}

#[test]
fn integers() {
    assert_eq!(min(0, 10), 0);
    assert_eq!(min(500, 123), 123);
}

#[test]
fn chars() {
    assert_eq!(min('a', 'z'), 'a');
    assert_eq!(min('7', '1'), '1');
}

#[test]
fn strings() {
    assert_eq!(min("hello", "goodbye"), "goodbye");
    assert_eq!(min("bat", "armadillo"), "armadillo");
}

```

Часть IV

День 2: вторая половина дня

Глава 15

С возвращением

С учётом 10-минутных перерывов, эта сессия должна занять примерно 2 часа 50 минут. В неё входит:

Сегмент	Продолжительность
Типы стандартной библиотеки 1 час	
Замыкания	30 минут
Трейты стандартной библиотеки 1 час	

Глава 16

Типы стандартной библиотеки

Этот раздел займет примерно 1 час. В нем рассматривается:

Слайд	Продолжительность
Стандартная библиотека	3 минуты
Документация	5 минут
Option	10 минут
Result	5 минут
String	5 минут
Vec	5 минут
HashMap	5 минут
Упражнение: Счётчик	20 минут

Для каждого слайда в этом разделе уделите время изучению страниц документации, выделяя наиболее распространённые методы.

16.1 Стандартная библиотека

Rust поставляется со стандартной библиотекой, которая устанавливает набор общих типов, используемых библиотеками и программами на Rust. Таким образом, две библиотеки могут эффективно взаимодействовать, поскольку обе используют один и тот же тип `String`.

На самом деле Rust содержит несколько уровней стандартной библиотеки: `core`, `alloc` и `std`.

- `core` включает самые базовые типы и функции, которые не зависят от `libc`, аллокатора или даже наличия операционной системы.
- `alloc` включает типы, требующие глобального аллокатора кучи, такие как `Vec`, `Box` и `Arc`.
- Встраиваемые приложения на Rust часто используют только `core`, а иногда и `alloc`.

16.2 Документация

Rust поставляется с обширной документацией. Например:

- Все подробности о циклах.
- Примитивные типы, такие как `u8`.
- Типы стандартной библиотеки, такие как `Option` или `BinaryHeap`.

Используйте `rustup doc --std` или <https://std.rs> для просмотра документации.

На самом деле, вы можете документировать собственный код:

```
/// Определяет, делится ли первый аргумент на второй без остатка.
///
/// Если второй аргумент равен нулю, результат — false.
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    if rhs == 0 {
        return false;
    }
    lhs % rhs == 0
}
```

Содержимое интерпретируется как Markdown. Все опубликованные библиотеки Rust автоматически документируются на docs.rs с помощью инструмента `rustdoc`. Принято документировать все публичные элементы API, используя данный шаблон.

Для документирования элемента изнутри (например, внутри модуля) используйте `///!` или `/*! ... */`, называемые «внутренними комментариями документации»:

```
/*!
 * Этот модуль содержит функциональность, связанную с делимостью целых чисел.
 */


```

На этот слайд отводится примерно 5 минут.

- Покажите студентам сгенерированную документацию для `rand::rand` по адресу <https://docs.rs/rand>.

16.3 Option

Мы уже рассмотрели некоторое использование `Option<T>`. Он хранит либо значение типа `T`, либо ничего. Например, `String::find` возвращает `Option<usize>`.

```
fn main() {
    let name = "Löwe 老虎 Léopard Gepardi";
    let mut position: Option<usize> = name.find('é');
    dbg!(position);
    assert_eq!(position.unwrap(), 14);
    position = name.find('Z');
    dbg!(position);
    assert_eq!(position.expect("Символ не найден"), 0);
}
```

На этот слайд должно уйти около 10 минут.

- `Option` широко используется не только в стандартной библиотеке.
- `unwrap` возвращает значение из `Option` или вызывает панику. `expect` похож на `unwrap`, но принимает сообщение об ошибке.
 - Вы можете вызвать панику при `None`, но не можете «случайно» забыть проверить значение на `None`.
 - При быстром прототипировании часто используется `unwrap / expect` повсеместно, однако в промышленном коде обычно `None` обрабатывается более корректно.

- «Нишевое оптимизирование» означает, что `Option<T>` часто занимает в памяти тот же размер, что и `T`, если существует представление, не являющееся допустимым значением `T`. Например, ссылка не может быть `NULL`, поэтому `Option<&T>` автоматически использует `NULL` для представления варианта `None`, и, следовательно, может храниться в той же области памяти, что и `&T`.

16.4 Result

`Result` похож на `Option`, но указывает на успешное или неуспешное выполнение операции, каждая из которых представлена отдельным вариантом перечисления. Он является обобщённым типом: `Result<T, E>`, где `T` используется в варианте `Ok`, а `E` — в варианте `Err`.

```
use std::fs::File;
use std::io::Read;

fn main() {
    let file: Result<File, std::io::Error> = File::open("diary.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            if let Ok(bytes) = file.read_to_string(&mut contents) {
                println!("Дорогой дневник: {contents} ({bytes} байт)");
            } else {
                println!("Не удалось прочитать содержимое файла");
            }
        }
        Err(err) => {
            println!("Дневник не удалось открыть: {err}");
        }
    }
}
```

На этот слайд отводится примерно 5 минут.

- Как и в случае с `Option`, успешное значение находится внутри `Result`, что заставляет разработчика явно извлекать его. Это способствует проверке ошибок. В случае, когда ошибка не должна происходить, можно вызвать `unwrap()` or `expect()`, что также сигнализирует о намерениях разработчика.
- Документация по `Result` рекомендуется к прочтению. Не в рамках курса, но стоит упомянуть. Она содержит множество удобных методов и функций, которые помогают при функциональном стиле программирования.
- `Result` — это стандартный тип для реализации обработки ошибок, как мы увидим в День 4.

16.5 String

`String` — это изменяемая строка, кодированная в UTF-8:

```
fn main() {
    let mut s1 = String::new();
    s1.push_str("Hello");
    println!("s1: len = {}, capacity = {}", s1.len(), s1.capacity());
```

```

let mut s2 = String::with_capacity(s1.len() + 1);
s2.push_str(&s1);
s2.push('!');
println!("s2: len = {}, capacity = {}", s2.len(), s2.capacity());

let s3 = String::from(" ");
println!("s3: len = {}, number of chars = {}", s3.len(), s3.chars().count());
}

```

`String` реализует `Deref<Target = str>`, что означает, что вы можете вызывать все методы `str` на объекте `String`.

На этот слайд отводится примерно 5 минут.

- `String::new` возвращает новую пустую строку, используйте `String::with_capacity`, когда знаете, сколько данных хотите добавить в строку.
- `String::len` возвращает размер `String` в байтах (который может отличаться от длины в символах).
- `String::chars` возвращает итератор по фактическим символам. Обратите внимание, что `char` может отличаться от того, что человек считает «символом» из-за кластеров графем.
- Когда говорят о строках, могут иметь в виду либо `&str`, либо `String`.
- Когда тип реализует `Deref<Target = T>`, компилятор позволяет прозрачно вызывать методы из `T`.
 - Мы ещё не обсуждали трейд `Deref`, поэтому на данном этапе это в основном объясняет структуру боковой панели в документации.
 - `String` реализует `Deref<Target = str>`, что прозрачно даёт доступ к методам `str`.
 - Напишите и сравните `let s3 = s1.deref();` и `let s3 = &*s1;`.
- `String` реализован как оболочка вокруг вектора байтов, многие операции, поддерживаемые для векторов, также поддерживаются для `String`, но с дополнительными гарантиями.
- Сравнение различных способов индексирования `String`:
 - До символа с помощью `s3.chars().nth(i).unwrap()`, где `i` находится в пределах или выходит за их пределы.
 - До подстроки с помощью `s3[0..4]`, при условии, что этот срез находится на границах символов или нет.
- Многие типы могут быть преобразованы в строку с помощью метода `to_string`. Этот трейд автоматически реализуется для всех типов, которые реализуют `Display`, поэтому всё, что может быть отформатировано, также может быть преобразовано в строку.

16.6 Vec

`Vec` — стандартный изменяемый буфер, выделяемый в куче:

```

fn main() {
    let mut v1 = Vec::new();
    v1.push(42);
    println!("v1: len = {}, capacity = {}", v1.len(), v1.capacity());

    let mut v2 = Vec::with_capacity(v1.len() + 1);
    v2.extend(v1.iter());
    v2.push(9999);
}

```

```

    println!("v2: len = {}, capacity = {}", v2.len(), v2.capacity());

    // Канонический макрос для инициализации вектора с элементами.
    let mut v3 = vec![0, 0, 1, 2, 3, 4];

    // Сохраняем только чётные элементы.
    v3.retain(|x| x % 2 == 0);
    println!("{}v3:{}");

    // Удаляем последовательные дубликаты.
    v3.dedup();
    println!("{}v3:{}");
}

```

Vec реализует `Deref<Target = [T]>`, что позволяет вызывать методы среза на Vec .

На этот слайд отводится примерно 5 минут.

- Vec является типом коллекции, наряду с String и HashMap . Данные, которые он содержит, хранятся в куче. Это означает, что объём данных не обязательно должен быть известен во время компиляции. Он может увеличиваться или уменьшаться во время выполнения.
- Обратите внимание, что Vec<T> также является обобщённым типом, но указывать T явно не требуется. Как и всегда при выводе типов в Rust, T устанавливается во время первого вызова push .
- vec! [...] является каноническим макросом, который используется вместо Vec::new() и поддерживает добавление начальных элементов в вектор.
- Для индексирования вектора используется [] , но при выходе за границы они вызовут panic . В качестве альтернативы использование get вернёт Option . Функция pop удаляет последний элемент.

16.7 HashMap

Стандартная хеш-таблица с защитой от атак HashDoS:

```

use std::collections::HashMap;

fn main() {
    let mut page_counts = HashMap::new();
    page_counts.insert("Приключения Гекльберри Финна", 207);
    page_counts.insert("Сказки братьев Гримм", 751);
    page_counts.insert("Гордость и предубеждение", 303);

    if !page_counts.contains_key("Отверженные") {
        println!(
            "Нам известно {} книг, но нет Отверженных.", page_
            counts.len()
        );
    }

    for book in ["Гордость и предубеждение", "Алиса в стране чудес"] { match page_
        counts.get(book) {
            Some(count) => println!("{}book}: {}count} страниц", book, count),
            None => println!("{}book} неизвестна."),
        }
    }
}

```

```

    }

    // Используйте метод .entry() для вставки значения, если ничего не найдено.
    for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {
        let page_count: &mut i32 = page_counts.entry(book).or_insert(0);
        *page_count += 1;
    }

    dbg!(page_counts);
}

```

На этот слайд отводится примерно 5 минут.

- HashMap не определён в прелюдии и должен быть импортирован в область видимости.
 - Попробуйте следующие строки кода. Первая строка проверит, есть ли книга в hashmap, и если нет — вернёт альтернативное значение. Вторая строка вставит альтернативное значение в hashmap, если книга не найдена.
- ```

let pc1 = page_counts
 .get("Harry Potter and the Sorcerer's Stone")
 .unwrap_or(&336);
let pc2 = page_counts
 .entry("The Hunger Games")
 .or_insert(374);

```
- В отличие от vec!, к сожалению, нет стандартного макроса hashmap!.

– Хотя, начиная с Rust 1.56, HashMap реализует `From<[(K, V); N]>`, что позволяет легко инициализировать хеш-таблицу из литерального массива:

```

let page_counts = HashMap::from([
 ("Гарри Поттер и философский камень".to_string(), 336),
 ("Голодные игры".to_string(), 374),
]);

```

- Альтернативно, HashMap может быть построен из любого Iterator, который возвращает кортежи ключ-значение.
- Этот тип имеет несколько «специфичных для методов» возвращаемых типов, таких как `std::collections::hash_map::Keys`. Эти типы часто встречаются при поиске в документации Rust. Покажите студентам документацию для этого типа и полезную ссылку на метод `keys`.

## 16.8 Упражнение: Счётчик

В этом упражнении вы возьмёте очень простую структуру данных и сделаете её обобщённой. Она использует `std::collections::HashMap` для отслеживания, какие значения уже встречались и сколько раз каждое из них появлялось.

Начальная версия `Counter` жёстко задана для работы только с значениями типа `i32`. Сделайте структуру и её методы обобщёнными по типу отслеживаемого значения, чтобы `Counter` мог отслеживать значения любого типа.

Если вы закончите раньше, попробуйте использовать метод `entry`, чтобы сократить количество обращений к хеш-таблице при реализации метода `count` вдвое.

```

use std::collections::HashMap;

/// Counter подсчитывает количество появлений каждого значения типа T.
struct Counter {
 values: HashMap<u32, u64>,
}

impl Counter {
 /// Создаёт новый экземпляр Counter.
 fn new() -> Self {
 Counter {
 values: HashMap::new(),
 }
 }

 /// Подсчитывает появление заданного значения.
 fn count(&mut self, value: u32) {
 if self.values.contains_key(&value) {
 *self.values.get_mut(&value).unwrap() += 1;
 } else {
 self.values.insert(value, 1);
 }
 }

 /// Возвращает количество раз, которое заданное значение было встречено.
 fn times_seen(&self, value: u32) -> u64 {
 self.values.get(&value).copied().unwrap_or_default()
 }
}

fn main() {
 let mut ctr = Counter::new();
 ctr.count(13);
 ctr.count(14);
 ctr.count(16);
 ctr.count(14);
 ctr.count(14);
 ctr.count(11);

 for i in 10..20 {
 println!("наблюдал {} значений, равных {}", ctr.times_seen(i), i);
 }

 let mut strctr = Counter::new();
 strctr.count("apple");
 strctr.count("orange");
 strctr.count("apple");
 println!("получено {} яблок ", strctr.times_seen("apple"));
}
}

```

### 16.8.1 Решение

```
use std::collections::HashMap;
use std::hash::Hash;

/// Counter подсчитывает количество появлений каждого значения типа T .
struct Counter<T> {
 values: HashMap<T, u64>,
}

impl<T: Eq + Hash> Counter<T> {
 /// Создаёт новый экземпляр Counter .
 fn new() -> Self {
 Counter { values: HashMap::new() }
 }

 /// Подсчитывает появление заданного значения .
 fn count(&mut self, value: T) {
 *self.values.entry(value).or_default() += 1;
 }

 /// Возвращает количество раз, которое заданное значение было встречено .
 fn times_seen(&self, value: T) -> u64 {
 self.values.get(&value).copied().unwrap_or_default()
 }
}

fn main() {
 let mut ctr = Counter::new();
 ctr.count(13);
 ctr.count(14);
 ctr.count(16);
 ctr.count(14);
 ctr.count(14);
 ctr.count(11);

 for i in 10..20 {
 println!("наблюдал {} значений, равных {}", ctr.times_seen(i), i);
 }

 let mut strctr = Counter::new();
 strctr.count("apple");
 strctr.count("orange");
 strctr.count("apple");
 println!("получено {} яблок ", strctr.times_seen("apple"));
}
```

# Глава 17

## Замыкания

Этот раздел займет примерно 30 минут. В нем рассматриваются:

| Слайд                               | Продолжительность |
|-------------------------------------|-------------------|
| Синтаксис замыканий                 | 3 минуты          |
| Захват переменных                   | 5 минут           |
| Трейты замыканий                    | 10 минут          |
| Упражнение: Фильтр логов — 10 минут |                   |

### 17.1 Синтаксис замыканий

Замыкания создаются с помощью вертикальных черт: | . . | . . .

```
fn main() {
 // Типы аргументов и возвращаемого значения могут быть выведены для упрощённого синтаксиса:
 let double_it = |n| n * 2;
 dbg!(double_it(50));

 // Или можно указать типы и заключить тело в фигурные скобки для полной явности:
 let add_1f32 = |x: f32| -> f32 { x + 1.0 };
 dbg!(add_1f32(50.));
}
```

Этот слайд должен занять около 3 минут.

- Аргументы располагаются между | . . | . Тело может быть окружено { . . } , но если это одно выражение, эти скобки можно опустить.
- Типы аргументов необязательны и выводятся автоматически, если не указаны. Тип возвращаемого значения также необязателен, но его можно указать только при использовании { . . } вокруг тела.
- Оба примера можно записать просто как вложенные функции — они не захватывают переменные из своей лексической области видимости. Далее мы рассмотрим захваты.

## Дополнительные материалы для изучения

- Возможность хранить функции в переменных относится не только к замыканиям: обычные функции также можно помещать в переменные и вызывать так же, как и замыкания. Пример в [playground](#).
  - Связанный пример также демонстрирует, что замыкания, которые ничего не захватывают, могут преобразовываться в обычный указатель на функцию.

## 17.2 Захват

Замыкание может захватывать переменные из окружения, в котором оно было определено.

```
fn main() {
 let max_value = 5;
 let clamp = |v| {
 if v > max_value { max_value } else { v }
 };

 dbg!(clamp(1));
 dbg!(clamp(3));
 dbg!(clamp(5));
 dbg!(clamp(7));
 dbg!(clamp(10));
}
```

На этот слайд отводится примерно 5 минут.

- По умолчанию замыкание захватывает значения по ссылке. Здесь `max_value` захватывается замыканием `clamp`, но всё ещё доступна в `main` для вывода. Попробуйте сделать `max_value` изменяемой, изменить её и снова вывести ограниченные значения. Почему это не работает?
- Если замыкание изменяет значения, оно захватывает их по изменяемой ссылке. Попробуйте добавить `max_value += 1` в `clamp`.
- Вы можете заставить замыкание перемещать значения вместо ссылки, используя ключевое слово `move`. Это может помочь с временем жизни, например, если замыкание должно жить дольше захваченных значений (подробнее о времени жизни будет позже).

Это выглядит как `move |v| ...`. Попробуйте добавить это ключевое слово и проверьте, сможет ли `main` ещё получить доступ к `max_value` после определения `clamp`.

- По умолчанию замыкания захватывают каждую переменную из внешней области видимости наиболее щадящим способом доступа (по общей ссылке, если возможно, затем по исключительной ссылке, затем путём перемещения). Ключевое слово `move` принуждает захват по значению.

## 17.3 Трейты замыканий

Замыкания или лямбда-выражения имеют типы, которые нельзя явно указать. Однако они реализуют специальные трейты `Fn`, `FnMut` и `FnOnce`:

Специальные типы `fn(..) -> T` относятся к указателям на функции — либо адресу функции, либо замыканию, которое ничего не захватывает.

```

fn apply_and_log(
 func: impl FnOnce(&'static str) -> String,
 func_name: &'static str,
 input: &'static str,
) {
 println!("Вызов {}({}):", func_name, input);
 func(input)
}

fn main() {
 let suffix = "-itis";
 let add_suffix = |x| format!("{}{}{}", x, suffix);
 apply_and_log(&add_suffix, "add_suffix", "senior");
 apply_and_log(&add_suffix, "add_suffix", "appendix");

 let mut v = Vec::new();
 let mut accumulate = |x| {
 v.push(x);
 v.join("/")
 };
 apply_and_log(&mut accumulate, "accumulate", "red");
 apply_and_log(&mut accumulate, "accumulate", "green");
 apply_and_log(&mut accumulate, "accumulate", "blue");

 let take_and_reverse = |prefix| {
 let mut acc = String::from(prefix);
 acc.push_str(&v.into_iter().rev().collect::<Vec<_>>().join("/"));
 acc
 };
 apply_and_log(take_and_reverse, "take_and_reverse", "reversed:");
}

```

На этот слайд должно уйти около 10 минут.

An Fn (например, add\_suffix) не потребляет и не изменяет захваченные значения. Его можно вызвать, имея только разделяемую ссылку на замыкание, что означает, что замыкание может выполняться многократно и даже параллельно.

An FnMut (например, accumulate) может изменять захваченные значения. Объект замыкания доступен через исключительную ссылку, поэтому его можно вызывать многократно, но не параллельно.

Если у вас есть FnOnce (например, take\_and\_reverse), вы можете вызвать его только один раз. Это приводит к потреблению замыкания и всех значений, захваченных с помощью move.

FnMut является подтипов FnOnce. Fn является подтипов FnMut и FnOnce. То есть вы можете использовать FnMut везде, где требуется FnOnce, а Fn — везде, где требуется FnMut или FnOnce.

При определении функции, принимающей замыкание, следует использовать FnOnce если возможно (то есть выываете его один раз), иначе FnMut, и в последнюю очередь Fn. Это обеспечивает максимальную гибкость для вызывающего.

В противоположность этому, если у вас есть замыкание, максимальная гибкость достигается с помощью Fn (которое может быть передано потребителю любого из трёх трейтов замыканий), затем FnMut и, наконец, FnOnce.

Компилятор также выводит Copy (например, для add\_suffix) и Clone (например, для take\_and\_reverse), в зависимости от того, что захватывает замыкание. Указатели на функции (ссылки на fn элементы) реализуют

Copy и Fn.

## 17.4 Упражнение: Фильтр логов

Основываясь на универсальном логгере из сегодняшнего утра, реализуйте `Filter`, который использует замыкание для фильтрации сообщений журнала, отправляя те, которые проходят предикат фильтрации, во внутренний логгер.

```
pub trait Logger {
 // Записать сообщение с указанным уровнем подробности.
 fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
 fn log(&self, verbosity: u8, message: &str) {
 eprintln!("verbosity={verbosity}: {message}");
 }
}

// TODO: определить и реализовать `Filter`.

fn main() {
 let logger = Filter::new(StderrLogger, |_verbosity, msg| msg.contains("yikes"));
 logger.log(5, "FYI");
 logger.log(1, "yikes, что-то пошло не так");
 logger.log(2, "uhoh");
}
```

### 17.4.1 Решение

```
pub trait Logger {
 // Записать сообщение с указанным уровнем подробности.
 fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
 fn log(&self, verbosity: u8, message: &str) {
 eprintln!("verbosity={verbosity}: {message}");
 }
}

/// Логировать только сообщения, соответствующие предикату фильтрации.
struct Filter<L, P> {
 inner: L,
 predicate: P,
}
```

```

impl<L, P> Filter<L, P>
where
 L: Logger,
 P: Fn(u8, &str) -> bool,
{
 fn new(inner: L, predicate: P) -> Self {
 Self { inner, predicate }
 }
}
impl<L, P> Logger for Filter<L, P>
where
 L: Logger,
 P: Fn(u8, &str) -> bool,
{
 fn log(&self, verbosity: u8, message: &str) {
 if (self.predicate)(verbosity, message) {
 self.inner.log(verbosity, message);
 }
 }
}

fn main() {
 let logger = Filter::new(StderrLogger, |_verbosity, msg| msg.contains("yikes"));
 logger.log(5, "FYI");
 logger.log(1, "yikes, что-то пошло не так");
 logger.log(2, "uhoh");
}

```

- Обратите внимание, что ограничение P: Fn(u8, &str) -> bool в первом блоке реализации Filter не является строго обязательным, но оно облегчает вывод типов при вызове new. Продемонстрируйте удаление этого ограничения и покажите, как теперь компилятору требуются аннотации типов для замыкания, передаваемого в new.

## Глава 18

# Трейты стандартной библиотеки

Этот раздел займет примерно 1 час. В нем рассматривается:

| Слайд                                             | Продолжительность |
|---------------------------------------------------|-------------------|
| Сравнения                                         | 5 минут           |
| Операторы                                         | 5 минут           |
| From и Into                                       | 5 минут           |
| Приведение типов                                  | 5 минут           |
| Чтение и запись                                   | 5 минут           |
| Deflault, синтаксис обновления структуры, 5 минут |                   |
| Упражнение: ROT13                                 | 30 минут          |

Как и с типами стандартной библиотеки, уделите время изучению документации для каждого трейта.

Этот раздел длинный. Сделайте перерыв примерно посередине.

### 18.1 Сравнения

Эти трейты поддерживают сравнение значений. Все трейты могут быть выведены для типов, содержащих поля, реализующие эти трейты.

#### PartialEq и Eq

PartialEq является частичным отношением эквивалентности с обязательным методом `eq` и предоставленным методом `ne`. Операторы `==` и `!=` вызывают эти методы.

```
struct Key {
 id: u32,
 metadata: Option<String>,
}
impl PartialEq для Key {
 fn eq(&self, other: &Self) -> bool {
 self.id == other.id
 }
}
```

```
 }
}
```

`Eq` является полным отношением эквивалентности (рефлексивным, симметричным и транзитивным) и подразумевает `PartialEq`. Функции, требующие полного отношения эквивалентности, используют `Eq` в качестве ограничения трейта.

## PartialOrd и Ord

`PartialOrd` определяет частичный порядок с методом `partial_cmp`. Он используется для реализации операторов `<`, `<=`, `>=` и `>`.

```
use std::cmp::Ordering;
#[derive(Eq, PartialEq)]
struct Citation {
 author: String,
 year: u32,
}
impl PartialOrd for Citation {
 fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
 match self.author.partial_cmp(&other.author) {
 Some(Ordering::Equal) => self.year.partial_cmp(&other.year),
 author_ord => author_ord,
 }
 }
}
```

`Ord` является полным порядком, при котором `cmp` возвращает `Ordering`.

На этот слайд отводится примерно 5 минут.

- `PartialEq` может быть реализован между разными типами, но `Eq` — нет, поскольку он рефлексивен:

```
struct Key {
 id: u32,
 metadata: Option<String>,
}
impl PartialEq<u32> for Key {
 fn eq(&self, other: &u32) -> bool {
 self.id == *other
 }
}
```

- На практике часто используют `derive` для этих трейтов, но реализуют их вручную редко.
- При сравнении ссылок в Rust сравниваются значения, на которые они указывают, а не сами ссылки. Это означает, что ссылки на два разных объекта могут считаться равными, если значения, на которые они указывают, совпадают:

```
fn main() {
 let a = "Hello";
 let b = String::from("Hello");
 assert_eq!(a, b);
}
```

## 18.2 Операторы

Перегрузка операторов реализуется через трейты в `std::ops`:

```
#[derive(Debug, Copy, Clone)]
struct Point {
 x: i32,
 y: i32,
}

impl std::ops::Add for Point {
 type Output = Self;

 fn add(self, other: Self) -> Self {
 Self { x: self.x + other.x, y: self.y + other.y }
 }
}

fn main() {
 let p1 = Point { x: 10, y: 20 };
 let p2 = Point { x: 100, y: 200 };
 println!("{} + {} = {}", p1, p2, p1 + p2);
}
```

На этот слайд отводится примерно 5 минут.

Пункты для обсуждения:

- Вы можете реализовать `Add` для `&Point`. В каких ситуациях это полезно?
  - Ответ: `Add::add` потребляет `self`. Если тип `T`, для которого вы перегружаете оператор `,`, не является `Copy`, следует рассмотреть возможность перегрузки оператора для `&T`. Это позволяет избежать ненужного клонирования на месте вызова.
- Почему `Output` является ассоциированным типом? Можно ли сделать его параметром типа метода?
  - Краткий ответ: параметры типа функции контролируются вызывающей стороной, а ассоциированные типы (например, `Output`) контролируются реализацией трейта.
- Вы можете реализовать `Add` для двух разных типов, например, `impl Add<(i32, i32)> for Point` будет складывать кортеж с `Point`.

Трейт `Not` (! оператор) примечателен тем, что он не приводит к преобразованию в булев тип, как аналогичный оператор в языках семейства С; вместо этого для целочисленных типов он инвертирует каждый бит числа, что арифметически эквивалентно вычитанию числа из -1: `!5 == -6`.

## 18.3 From и Into

Типы реализуют `From` и `Into` для упрощения преобразований типов. В отличие от `as`, эти трейты соответствуют безпотерянным, безошибочным преобразованиям.

```
fn main() {
 let s = String::from("hello");
 let addr = std::net::Ipv4Addr::from([127, 0, 0, 1]);
 let one = i16::from(true);
 let bigger = i32::from(123_i16);
```

```
 println!("{} , {} , {} , {}");
}
```

`Into` автоматически реализуется, когда реализован `From`:

```
fn main() {
 let s: String = "hello".into();
 let addr: std::net::Ipv4Addr = [127, 0, 0, 1].into();
 let one: i16 = true.into();
 let bigger: i32 = 123_i16.into();
 println!("{} , {} , {} , {}");
}
```

На этот слайд отводится примерно 5 минут.

- Именно поэтому обычно реализуют только `From`, поскольку для вашего типа автоматически будет реализован `Into`.
- При объявлении типа аргумента функции, например «что угодно, что может быть преобразовано в `String`», правило обратное — следует использовать `Into`. Ваша функция будет принимать типы, которые реализуют `From`, а также те, которые *only* реализуют `Into`.

## 18.4 Приведение типов

В Rust отсутствуют неявные преобразования типов, но поддерживаются явные приведения с помощью `as`. Они, как правило, следуют семантике С там, где она определена.

```
fn main() {
 let value: i64 = 1000;
 println!("as u16: {}", value as u16);
 println!("as i16: {}", value as i16);
 println!("as u8: {}", value as u8);
}
```

Результаты `as` всегда определены в Rust и согласованы между платформами. Это может не совпадать с вашей интуицией при изменении знака или приведении к меньшему типу — ознакомьтесь с документацией и добавьте комментарии для ясности.

Приведение типов с помощью `as` является относительно острым инструментом, который легко использовать неправильно и который может стать источником тонких ошибок при последующем сопровождении, когда изменяются используемые типы или диапазоны значений в типах. Приведения типов лучше использовать только тогда, когда намерение состоит в указании безусловного усечения (например, выбор младших 32 бит из `u64` с помощью `as u32`, независимо от содержимого старших бит).

Для безошибочных приведений типов (например, `u32` к `u64`) предпочтительно использовать `From` или `Into` вместо `as` для подтверждения того, что приведение действительно безошибочно. Для потенциально ошибочных приведений доступны `TryFrom` и `TryInto`, когда необходимо обработать случаи, в которых приведение возможно, и случаи, в которых оно невозможно.

На этот слайд отводится примерно 5 минут.

Рекомендуется сделать перерыв после этого слайда.

`as` похоже на статическое приведение типов в C++. Использование `as` в случаях, когда данные могут быть потеряны, как правило, не рекомендуется или, по крайней мере, требует пояснительного комментария.

Это часто встречается при приведении целых чисел к `usize` для использования в качестве индекса.

## 18.5 Чтение и запись

Используя `Read` и `BufRead`, вы можете абстрагироваться от источников типа `u8`:

```
use std::io::{BufRead, BufferedReader, Read, Result};

fn count_lines<R: Read>(reader: R) -> usize {
 let buf_reader = BufferedReader::new(reader);
 buf_reader.lines().count()
}

fn main() -> Result<()> {
 let slice: &[u8] = b"foo\nbar\nbaz\n";
 println!("строки в срезе: {}", count_lines(slice));

 let file = std::fs::File::open(std::env::current_exe()?)?;
 println!("строки в файле: {}", count_lines(file));
 Ok(())
}
```

Аналогично, `Write` позволяет абстрагироваться от приёмников типа `u8`:

```
use std::io::{Result, Write};

fn log<W: Write>(writer: &mut W, msg: &str) -> Result<()> {
 writer.write_all(msg.as_bytes())?;
 writer.write_all("\n".as_bytes())
}

fn main() -> Result<()> {
 let mut buffer = Vec::new();
 log(&mut buffer, "Привет")?;
 log(&mut buffer, "World")?;
 println!("Logged: {buffer:?}");
 Ok(())
}
```

## 18.6 Трейт Default

Трейт `Default` создаёт значение по умолчанию для типа.

```
#[derive(Debug, Default)]
struct Derived {
 x: u32,
 y: String,
 z: Implemented,
}

#[derive(Debug)]
struct Implemented(String);

impl Default for Implemented {
```

```

fn default() -> Self {
 Self("John Smith".into())
}

fn main() {
 let default_struct = Derived::default();
 dbg!(default_struct);

 let almost_default_struct =
 Derived { y: "Y is set!".into(), ..Derived::default() };
 dbg!(almost_default_struct);

 let nothing: Option<Derived> = None;
 dbg!(nothing.unwrap_or_default());
}

```

На этот слайд отводится примерно 5 минут.

- Он может быть реализован напрямую или получен с помощью#[derive(Default)].
- Реализация, полученная через derive, создаст значение, в котором все поля установлены в значения по умолчанию.
  - Это означает, что все типы в структуре также должны реализовывать Default.
- Стандартные типы Rust часто реализуют Defaultc разумными значениями (например, 0, "" и т.д.).
- Частичная инициализация структуры хорошо сочетается с использованием Default.
- Стандартная библиотека Rust учитывает, что типы могут реализовывать Default и предоставляет удобные методы, которые его используют.
- Синтаксис .. называется синтаксисом обновления структуры.

## 18.7 Упражнение: ROT13

В этом примере вы реализуете классический шифр "ROT13". Скопируйте этот код в play ground и реализуйте отсутствующие части. Поворачивайте только ASCII-символы алфавита, чтобы результат оставался корректным UTF-8.

```

use std::io::Read;

struct RotDecoder<R: Read> {
 input: R,
 rot: u8,
}

// Реализация трейта `Read` для `RotDecoder`.

#[cfg(test)]
mod test {
 use super::*;

 #[test]
 fn joke() {
 let mut rot =

```

```

 RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot: 13 };
let mut result = String::new();
rot.read_to_string(&mut result).unwrap();
assert_eq!(&result, "To get to the other side!");
}

#[test]
fn binary() {
 let input: Vec<u8> = (0..=255u8).collect();
 let mut rot = RotDecoder::<&[u8]> { input: input.as_slice(), rot: 13 };
 let mut buf = [0u8; 256];
 assert_eq!(rot.read(&mut buf).unwrap(), 256);
 for i in 0..=255 {
 if input[i] != buf[i] {
 assert!(input[i].is_ascii_alphabetic());
 assert!(buf[i].is_ascii_alphabetic());
 }
 }
}
}

```

Что произойдет, если объединить два экземпляра `RotDecoder`, каждый из которых выполняет сдвиг на 13 символов?

### 18.7.1 Решение

```

use std::io::Read;

struct RotDecoder<R: Read> {
 input: R,
 rot: u8,
}

impl<R: Read> Read для RotDecoder<R> {
 fn read(&mut self, buf: &mut [u8]) -> std::io::Result<usize> {
 let size = self.input.read(buf)?;
 for b in &mut buf[..size] {
 if b.is_ascii_alphabetic() {
 let base = if b.is_ascii_uppercase() { 'A' } else { 'a' } as u8;
 *b = (*b - base + self.rot) % 26 + base;
 }
 }
 Ok(size)
 }
}

#[cfg(test)]
mod test {
 use super::*;

 #[test]

```

```

fn joke() {
 let mut rot =
 RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot: 13 };
 let mut result = String::new();
 rot.read_to_string(&mut result).unwrap();
 assert_eq!(&result, "To get to the other side!");
}

#[test]
fn binary() {
 let input: Vec<u8> = (0..=255u8).collect();
 let mut rot = RotDecoder::<&[u8]> { input: input.as_slice(), rot: 13 };
 let mut buf = [0u8; 256];
 assert_eq!(rot.read(&mut buf).unwrap(), 256);
 for i in 0..=255 {
 if input[i] != buf[i] {
 assert!(input[i].is_ascii_alphabetic());
 assert!(buf[i].is_ascii_alphabetic());
 }
 }
}
}

```

## **Часть V**

# **День 3: Утро**

## Глава 19

# Добро пожаловать в День 3

Сегодня мы рассмотрим:

- Управление памятью, время жизни и borrow checker: как Rust обеспечивает безопасность памяти.
- Умные указатели: типы указателей стандартной библиотеки.

## Расписание

С учётом 10-минутных перерывов эта сессия займет около 2 часов 20 минут. В ней содержится:

| Сегмент            | Продолжительность |
|--------------------|-------------------|
| Приветствие        | 3 минуты          |
| Управление памятью | 1 час             |
| Умные указатели    | 55 минут          |

## Глава 20

# Управление памятью

Этот раздел займет примерно 1 час. В нем рассматривается:

| Слайд                                   | Продолжительность |
|-----------------------------------------|-------------------|
| Обзор памяти программы                  | 5 минут           |
| Подходы к управлению памятью — 10 минут |                   |
| Владение                                | 5 минут           |
| Семантика перемещения                   | 5 минут           |
| Клонирование                            | 2 минуты          |
| Типы копирования                        | 5 минут           |
| Drop                                    | 10 минут          |
| Упражнение: Тип Builder                 | 20 минут          |

### 20.1 Обзор памяти программы

Программы выделяют память двумя способами:

- Стек: Непрерывная область памяти для локальных переменных.
  - Значения имеют фиксированный размер, известный на этапе компиляции.
  - Чрезвычайно быстро: достаточно сдвинуть указатель стека.
  - Простое управление: следует за вызовами функций.
  - Отличная локальность памяти.
- Куча: Хранение значений вне вызовов функций.
  - Значения имеют динамический размер, определяемый во время выполнения.
  - Чуть медленнее стека: требуется некоторый учёт.
  - Отсутствует гарантия локальности памяти.

#### Пример

Создание `String` размещает метаданные фиксированного размера в стеке и данные динамического размера, саму строку, в куче:

```

fn main() {
 let s1 = String::from("Hello");
}

Стек
. - - - - - . . Куча
: : :
: s1 : :
+-----+-----+ : :
: | capacity | 5 | : +-----+-----+-----+-----+
: | ptr | o+---+-----+-->| H | e | l | l | o | :
: | len | 5 | : +-----+-----+-----+-----+
: +-----+-----+ : :
: : :
```

```

На этот слайд отводится примерно 5 минут.

- Упомяните, что `String` основан на `Vec`, поэтому у него есть `capacity` и `length`, и он может увеличиваться, если является изменяемым, за счёт перераспределения памяти в куче.
- Если студенты спросят об этом, вы можете отметить, что базовая память выделяется в куче с использованием системного аллокатора, и что можно реализовать собственные аллокаторы с помощью [Allocator API](#).

Дополнительные материалы для изучения

Мы можем исследовать расположение памяти с помощью `unsafe Rust`. Однако следует подчеркнуть, что это по праву считается небезопасным!

```

fn main() {
    let mut s1 = String::from("Hello");
    s1.push(' ');
    s1.push_str("world");
    // НЕ ДЕЛАЙТЕ ТАК ДОМА! Только в образовательных целях.
    // String не гарантирует своё расположение в памяти, поэтому это может привести
    // к неопределённому поведению.
    unsafe {
        let (capacity, ptr, len): (usize, usize, usize) = std::mem::transmute(s1);
        println!("capacity = {capacity}, ptr = {ptr:#x}, len = {len}");
    }
}

```

20.2 Подходы к управлению памятью

Традиционно языки программирования делятся на две основные категории:

- Полный контроль через ручное управление памятью: C, C++, Pascal, ...
 - Программист самостоятельно решает, когда выделять или освобождать память в куче.
 - Программист должен определить, указывает ли указатель ещё на действительную область памяти.
 - Исследования показывают, что программисты допускают ошибки.
- Полная безопасность через автоматическое управление памятью во время выполнения: Java, Python, Go, Haskell, ...

- Система времени выполнения гарантирует, что память не будет освобождена, пока на неё существуют ссылки.
- Обычно реализуется с помощью подсчёта ссылок или сборщика мусора.

Rust предлагает новый подход:

Полный контроль и безопасность за счёт проверки корректного управления памятью на этапе компиляции.

Это достигается с помощью явной концепции владения.

На этот слайд должно уйти около 10 минут.

Этот слайд предназначен для помощи студентам, знакомым с другими языками, чтобы понять контекст Rust.

- В C управление кучей осуществляется вручную с помощью `malloc` и `free`. Распространённые ошибки включают забывание вызова `free`, многократный вызов для одного и того же указателя или разыменование указателя после освобождения памяти, на которую он указывал.
- В C++ существуют инструменты, такие как умные указатели (`unique_ptr`, `shared_ptr`), которые используют гарантии языка относительно вызова деструкторов для обеспечения освобождения памяти при возврате из функции. Тем не менее, довольно легко неправильно использовать эти инструменты и создавать ошибки, аналогичные ошибкам в C.
- Java, Go и Python полагаются на сборщик мусора для определения недоступной памяти и её освобождения. Это гарантирует, что любой указатель можно разыменовать, исключая ошибки использования после освобождения и другие классы ошибок. Однако сборщик мусора имеет накладные расходы во время выполнения и его сложно правильно настроить.

Модель владения и заимствования Rust во многих случаях обеспечивает производительность, сравнимую с C, с операциями выделения и освобождения памяти точно там, где это необходимо — без дополнительных затрат. Кроме того, Rust предоставляет инструменты, аналогичные умным указателям C++. При необходимости доступны другие варианты, такие как подсчёт ссылок, а также существуют `crates`, поддерживающие сборку мусора во время выполнения (не рассматривается в этом курсе).

20.3 Владение

Все привязки переменных имеют область видимости, в которой они действительны, и использование переменной вне этой области видимости является ошибкой:

```
struct Point(i32, i32);

fn main() {
    let p = Point(3, 4);
    dbg!(p.0);
}
dbg!(p.1);
```

Говорят, что переменная владеет значением. Каждое значение в Rust в любой момент времени имеет ровно одного владельца.

В конце области видимости переменная уничтожается (*dropped*), и данные освобождаются. Здесь может выполняться деструктор для освобождения ресурсов.

На этот слайд отводится примерно 5 минут.

Студенты, знакомые с реализациями сборщиков мусора, знают, что сборщик начинает с набора «корней» для поиска всей достижимой памяти. Принцип «одного владельца» в Rust — аналогичная концепция.

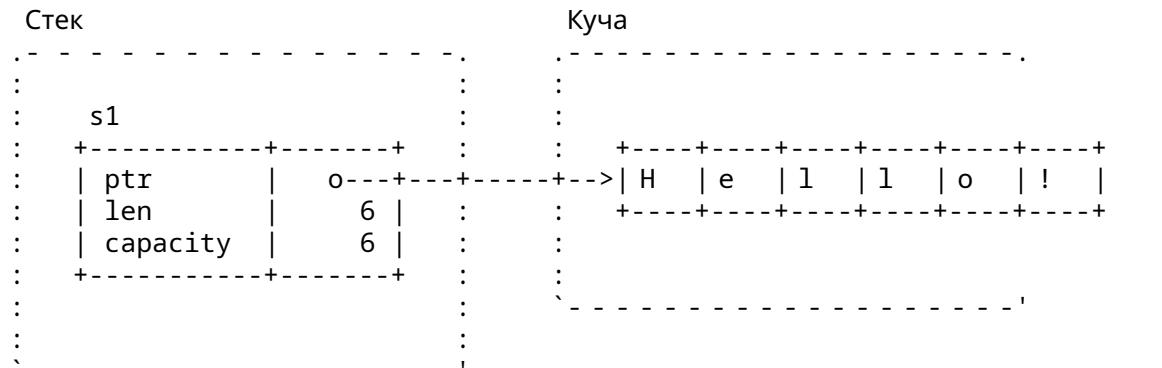
20.4 Семантика перемещения

Присваивание передаёт владение между переменными:

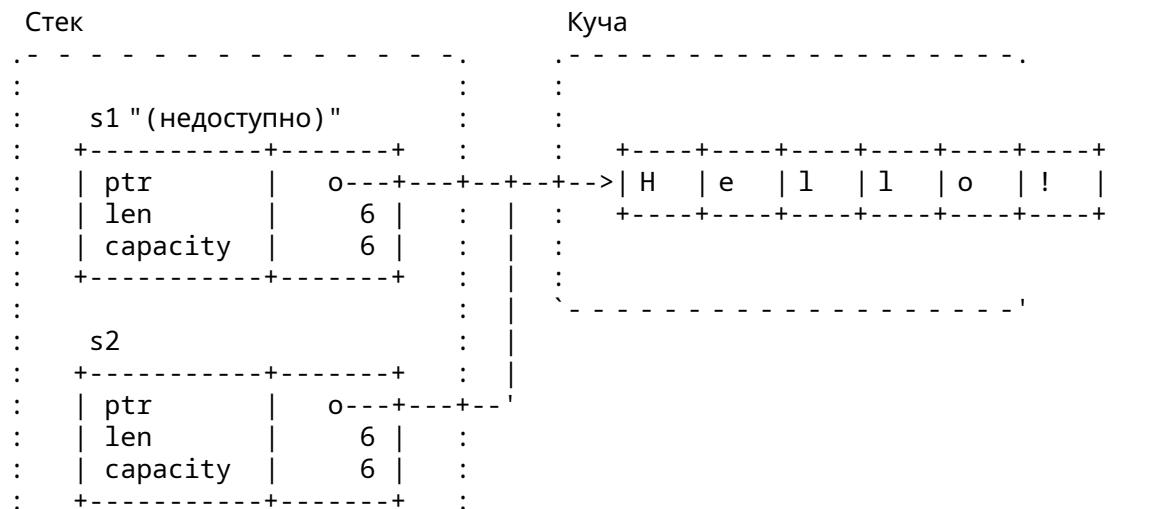
```
fn main() {
    let s1 = String::from("Hello!");
    let s2 = s1;
    dbg!(s2);
    // dbg!(s1);
}
```

- Присваивание `s1` переменной `s2` передаёт владение.
 - Когда `s1` выходит из области видимости, ничего не происходит: оно ни чем не владеет.
 - Когда `s2` выходит из области видимости, данные строки освобождаются.

Перед переносом в s2:



После переноса в s2:



```
:           :
```

Когда вы передаёте значение функции, оно присваивается параметру функции. Это передаёт владение:

```
fn say_hello(name: String) {
    println!("Hello {name}")
}

fn main() {
    let name = String::from("Alice");
    say_hello(name);
    // say_hello(name);
}
```

На этот слайд отводится примерно 5 минут.

- Отметьте, что это противоположно поведению по умолчанию в C++, где копирование происходит по значению, если не используется `std::move` (и определён конструктор перемещения!).
- Только владение перемещается. Вопрос о том, генерируется ли машинный код для непосредственной обработки данных, зависит от оптимизации, и такие копии активно устраняются оптимизатором.
- Простые значения (например, целые числа) могут быть помечены как `Copy`(см. последующие слайды).
- В Rust клонирование является явным (с помощью `clone`).

В примере `say_hello`:

- При первом вызове `say_hello` функция `main` теряет владение `name`. После этого `name` нельзя использовать внутри `main`.
- Память в куче, выделенная для `name`, будет освобождена в конце выполнения функции `say_hello`.
- Функция `main` может сохранить владение, если передаст `name` по ссылке (`&name`), и если `say_hello` принимает ссылку в качестве параметра.
- Альтернативно функция `main` может передать клонированный `name` первом вызове (`name.clone()`).
- Rust усложняет непреднамеренное создание копий по сравнению с C++, делая семантику перемещения стандартной и заставляя программистов явно указывать клонирование.

Дополнительные материалы для изучения

Защитные копии в современном C++

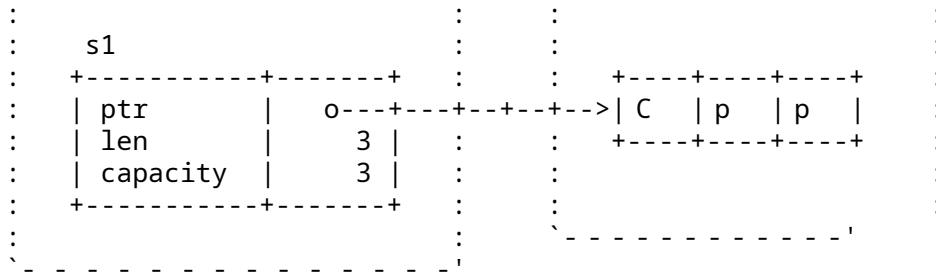
Современный C++ решает эту задачу иначе:

```
std::string s1 = "Cpp";
std::string s2 = s1; // Дублирование данных из s1.
```

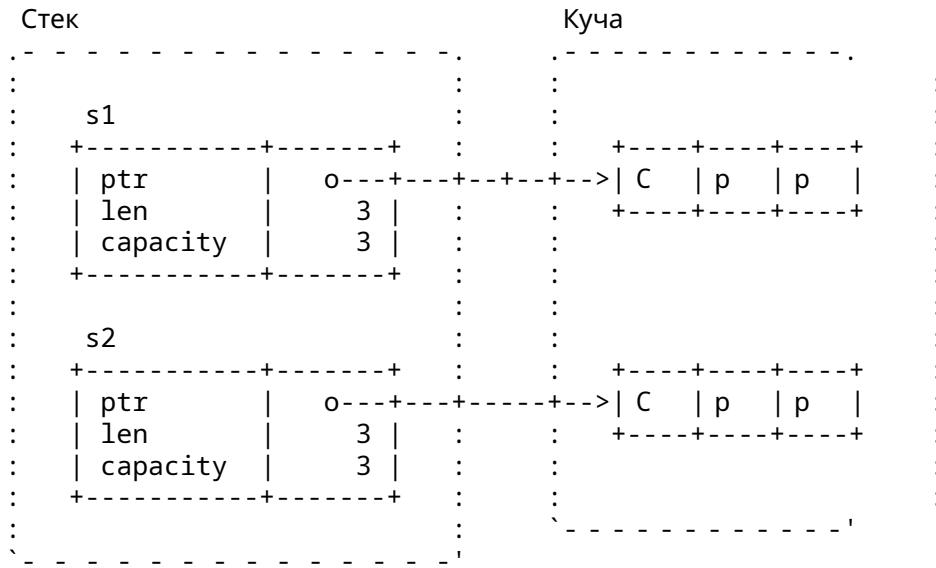
- Данные в куче из `s1` дублируются, и `s2` получает свою независимую копию.
- Когда `s1` и `s2` выходят из области видимости, каждый освобождает свою память.

До операции копирующего присваивания:

Стек	Куча
-----.	-----.



После операции копирующего присваивания:



Ключевые моменты:

- C++ сделал несколько иной выбор, чем Rust. Поскольку = копирует данные, строковые данные должны быть клонированы. В противном случае при выходе любой из строк из области видимости произошёл бы двойной вызов освобождения памяти.
- В C++ также существует `std::move`, который используется для указания, что значение может быть перемещено. Если бы пример был `s2 = std::move(s1)`, выделение памяти в куче не происходило бы. После перемещения, `s1` будет находиться в допустимом, но неопределённом состоянии. В отличие от Rust, программисту разрешается продолжать использовать `s1`.
- В отличие от Rust, = в C++ может выполнять произвольный код, определяемый типом, который копируется или перемещается.

20.5 Clone

Иногда необходимо создать копию значения. Трейт `Clone` решает эту задачу.

```
fn say_hello(name: String) {
    println!("Hello {}")
```

```

fn main() {
    let name = String::from("Alice");
    say_hello(name.clone());
    say_hello(name);
}

```

Этот слайд рассчитан примерно на 2 минуты.

- Идея Clone заключается в том, чтобы облегчить обнаружение мест, где происходят выделения в куче. Обращайте внимание на `.clone()` и несколько других, таких как `vec!` или `Box::new`.
- Часто практикуется «клонирование» для обхода проблем с проверкой заимствований, с последующим возвращением к оптимизации этих клонов.
- `clone` обычно выполняет глубокое копирование значения, что означает, что, например, при клонировании массива клонируются все его элементы.
- Поведение `clone` определяется пользователем, поэтому при необходимости может реализовываться пользовательская логика клонирования.

20.6 Копируемые типы

Хотя семантика перемещения является значением по умолчанию, некоторые типы копируются по умолчанию:

```

fn main() {
    let x = 42;
    let y = x;
    dbg!(x); // не был бы доступен, если бы не Copy
    dbg!(y);
}

```

Эти типы реализуют трейд `Copy`.

Вы можете включить использование семантики копирования для собственных типов:

```

#[derive(Copy, Clone, Debug)]
struct Point(i32, i32);

```

```

fn main() {
    let p1 = Point(3, 4);
    let p2 = p1;
    println!("p1: {p1:?}");
    println!("p2: {p2:?}");
}

```

- После присваивания `p1`, и `p2` владеют собственными данными.
- Мы также можем использовать `p1.clone()` для явного копирования данных.

На этот слайд отводится примерно 5 минут.

Копирование и клонирование — это не одно и то же:

- Копирование означает побитовое копирование областей памяти и не работает с произвольными объектами.
- Копирование не допускает пользовательскую логику (в отличие от конструкторов копирования в C++).
- Клонирование — более общая операция, которая также позволяет задавать пользовательское поведение через реализацию трейта `Clone`.

- Копирование не работает для типов, реализующих трейт Drop.

В приведённом выше примере выполните следующее:

- Добавьте поле `String` в структуру `Point`. Код не скомпилируется, так как `String` не является типом с поддержкой `Copy`.
- Удалите `Copy` из атрибута `derive`. Ошибка компиляции теперь возникает в вызове `println!` для `p1`.
- Покажите, что всё работает, если клонировать `p1` вместо этого .

Дополнительные материалы для изучения

- Разделяемые ссылки являются `Copy / Clone`, изменяемые ссылки — нет. Это связано с тем, что Rust требует эксклюзивности изменяемых ссылок, поэтому копирование разделяемой ссылки допустимо, а копирование изменяемой ссылки нарушило бы правила заимствования Rust.

20.7 Трейт Drop

Значения, реализующие `Drop`, могут задавать код, который выполняется при выходе из области видимости:

```
struct Droppable {
    name: &'static str,
}

impl Drop for Droppable {
    fn drop(&mut self) {
        println!("Dropping {}", self.name);
    }
}

fn main() {
    let a = Droppable { name: "a" };
    {
        let b = Droppable { name: "b" };
        {
            let c = Droppable { name: "c" };
            let d = Droppable { name: "d" };
            println!("Выход из внутреннего блока");
        }
        println!("Выход из следующего блока");
    }
    drop(a);
    println!("Выход из main");
}
```

На этот слайд следует отвести около 8 минут.

- Обратите внимание, что `std::mem::drop`не то же самое, что `std::ops::Drop::drop`.
- Значения автоматически удаляются при выходе из области видимости.

- Когда значение удаляется, если оно реализует `std::ops::Drop`, вызывается его реализация `Drop::drop`.
- Все его поля также будут удалены, независимо от того, реализует ли оно `Drop`.
- `std::mem::drop` — это просто пустая функция, которая принимает любое значение. Суть в том, что он принимает владение значением, поэтому по окончании своей области видимости оно будет удалено. Это делает его удобным способом явно удалять значения раньше, чем они покинули бы область видимости.
 - Это может быть полезно для объектов, которые выполняют определённые действия при `drop`: освобождение блокировок, закрытие файлов и т.д.

Пункты для обсуждения:

- Почему `Drop::drop` принимает `self`?
 - Краткий ответ: если бы принимал, то `std::mem::drop` вызывался бы в конце блока, что привело бы к повторному вызову `Drop::drop` и переполнению стека!
- Попробуйте заменить `drop(a)` на `a.drop()`.

20.8 Упражнение: тип Builder

В этом примере мы реализуем сложный тип данных, который владеет всеми своими данными. Мы будем использовать «паттерн строителя» для поэтапного создания нового значения с помощью удобных функций.

Заполните пропущенные части.

```
#[derive(Debug)]
enum Language {
    Rust,
    Java,
    Perl,
}

#[derive(Clone, Debug)]
struct Dependency {
    name: String,
    version_expression: String,
}

/// Представление программного пакета.
#[derive(Debug)]
struct Package {
    name: String,
    version: String,
    authors: Vec<String>,
    dependencies: Vec<Dependency>,
    language: Option<Language>,
}

impl Package {
    /// Возвращает представление данного пакета в виде зависимости для использования
    /// при // сборке других пакетов.
    fn as_dependency(&self) -> Dependency {
```

```

        todo!("1")
    }
}

/// Конструктор для Package. Используйте `build()` для создания самого `Package`.
struct PackageBuilder(Package);

impl PackageBuilder {
    fn new(name: impl Into<String>) -> Self {
        todo!("2")
    }

    /// Устанавливает версию пакета.
    fn version(mut self, version: impl Into<String>) -> Self {
        self.0.version = version.into();
        self
    }

    /// Устанавливает авторов пакета.
    fn authors(mut self, authors: Vec<String>) -> Self {
        todo!("3")
    }

    /// Добавить дополнительную зависимость.
    fn dependency(mut self, dependency: Dependency) -> Self {
        todo!("4")
    }

    /// Установить язык. Если не задан, язык по умолчанию — None.
    fn language(mut self, language: Language) -> Self {
        todo!("5")
    }

    fn build(self) -> Package {
        self.0
    }
}

fn main() {
    let base64 = PackageBuilder::new("base64").version("0.13").build();
    dbg!(&base64);
    let log =
        PackageBuilder::new("log").version("0.4").language(Language::Rust).build();
    dbg!(&log);
    let serde = PackageBuilder::new("serde")
        .authors(vec!["dmitche".into()])
        .version(String::from("4.0"))
        .dependency(base64.as_dependency())
        .dependency(log.as_dependency())
        .build();
    dbg!(serde);
}

```

```
}
```

20.8.1 Решение

```
#[derive(Debug)]
enum Language {
    Rust,
    Java,
    Perl,
}

#[derive(Clone, Debug)]
struct Dependency {
    name: String,
    version_expression: String,
}

/// Представление программного пакета.
#[derive(Debug)]
struct Package {
    name: String,
    version: String,
    authors: Vec<String>,
    dependencies: Vec<Dependency>,
    language: Option<Language>,
}

impl Package {
    /// Возвращает представление данного пакета в виде зависимости для использования
    /// при // сборке других пакетов.
    fn as_dependency(&self) -> Dependency {
        Dependency {
            name: self.name.clone(),
            version_expression: self.version.clone(),
        }
    }
}

/// Конструктор для Package. Используйте `build()` для создания самого `Package`.
struct PackageBuilder(Package);

impl PackageBuilder {
    fn new(name: impl Into<String>) -> Self {
        Self(Package {
            name: name.into(),
            version: "0.1".into(),
            authors: Vec::new(),
            dependencies: Vec::new(),
            language: None,
        })
    }
}
```

```

/// Устанавливает версию пакета.
fn version(&mut self, version: impl Into<String>) -> Self {
    self.0.version = version.into();
    self
}

/// Устанавливает авторов пакета.
fn authors(&mut self, authors: Vec<String>) -> Self {
    self.0.authors = authors;
    self
}

/// Добавить дополнительную зависимость.
fn dependency(&mut self, dependency: Dependency) -> Self {
    self.0.dependencies.push(dependency);
    self
}

/// Установить язык. Если не задан, язык по умолчанию — None.
fn language(&mut self, language: Language) -> Self {
    self.0.language = Some(language);
    self
}

fn build(self) -> Package {
    self.0
}
}

fn main() {
    let base64 = PackageBuilder::new("base64").version("0.13").build();
    dbg!(&base64);
    let log =
        PackageBuilder::new("log").version("0.4").language(Language::Rust).build();
    dbg!(&log);
    let serde = PackageBuilder::new("serde")
        .authors(vec!["djmitch".into()])
        .version(String::from("4.0"))
        .dependency(base64.as_dependency())
        .dependency(log.as_dependency())
        .build();
    dbg!(serde);
}

```

Глава 21

Умные указатели

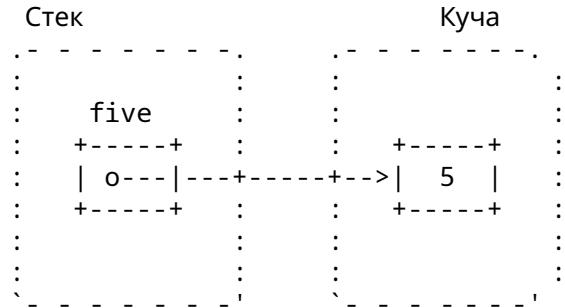
Этот раздел займет примерно 55 минут. В нем содержится:

Слайд	Продолжительность
Box	10 минут
Rc	5 минут
Владельческие объекты трейтов	10 минут
Упражнение: бинарное дерево	30 минут

21.1 Box<T>

Box является владением указателя на данные в куче:

```
fn main() {
    let five = Box::new(5);
    println!("five: {}", *five);
}
```



Box<T> реализует Deref<Target = T>, что означает, что вы можете вызывать методы типа T непосредственно на Box<T>.

Рекурсивные типы данных или типы данных с динамическим размером не могут храниться inline без косвенной адресации. Box обеспечивает такую косвенную адресацию:

```

#[derive(Debug)]
enum List<T> {
    // Непустой список: первый элемент и остальная часть списка.
    Element(T, Box<List<T>>),
    // Пустой список.
    Nil,
}

fn main() {
    let list: List<i32> =
        List::Element(1, Box::new(List::Element(2, Box::new(List::Nil))));  

    println!("{}list:{}");
}

```



На этот слайд следует отвести около 8 минут.

- Box похожа на `std::unique_ptr` в C++, за исключением того, что гарантированно не равна `null`.
- Box может быть полезна, когда вы:
 - имеете тип, размер которого нельзя определить во время компиляции, но компилятор Rust требует точного размера.
 - хотите передать владение большим объёмом данных. Чтобы избежать копирования больших объёмов данных в стеке, храните данные в куче в Box, чтобы перемещался только указатель.
- Если бы Box не использовалась и мы попытались встроить List напрямую в List, компилятор не смог бы вычислить фиксированный размер структуры в памяти (размер List был бы бесконечным).
- Box решает эту проблему, поскольку имеет тот же размер, что и обычный указатель, и просто указывает на следующий элемент List в куче.
- Удалите Box в определении List и продемонстрируйте ошибку компилятора. Мы получаем сообщение «рекурсия без индирекции», поскольку для рекурсивных данных необходимо использовать индирекцию — Box или ссылку какого-либо типа, вместо прямого хранения значения.
- Хотя Box похоже на `std::unique_ptr` в C++, оно не может быть пустым или нулевым. Это делает Box одним из типов, позволяющих компилятору оптимизировать хранение некоторых перечислений (так называемая «оптимизация ниш»).

21.2 Rc

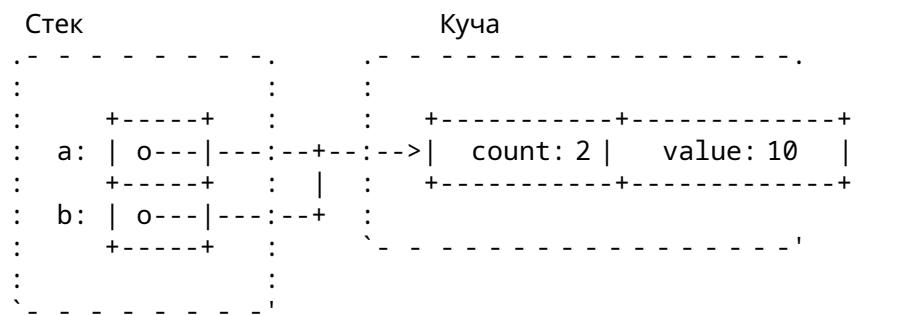
Rc — это указатель с подсчётом ссылок. Используйте его, когда необходимо ссылаться на одни и те же данные из нескольких мест:

```
use std::rc::Rc;

fn main() {
    let a = Rc::new(10);
    let b = Rc::clone(&a);

    dbg!(a);
    dbg!(b);
}
```

Каждый **Rc** указывает на одну и ту же общую структуру данных, содержащую сильные и слабые указатели, а также значение:



- См. [Arc и Mutex](#), если вы работаете в многопоточном контексте.
- Вы можете понизить `shared pointer` до **Weak** указателя для создания циклов, которые будут удалены.

На этот слайд отводится примерно 5 минут.

- Счётчик **Rc** гарантирует, что содержащиеся данные остаются валидными, пока существуют ссылки.
- **Rc** в Rust аналогичен `std::shared_ptr` в C++.
- `Rc::clone` дешёвый: он создаёт указатель на ту же область памяти и увеличивает счётчик ссылок. Не выполняет глубокое клонирование и обычно может игнорироваться при поиске проблем с производительностью в коде.
- `make_mut` фактически клонирует внутреннее значение при необходимости («clone -он-write») и возвращает изменяемую ссылку.
- Используйте `Rc::strong_count` для проверки счётчика ссылок.
- `Rc::downgrade` создаёт слабо-считаемый объект для создания циклов, которые будут корректно удалены (вероятно, в сочетании с `RefCell`).

21.3 Владеющие объекты трейтов

Ранее мы видели, как объекты трейтов могут использоваться со ссылками, например `&dyn Pet`. Однако, мы также можем использовать объекты трейтов с умными указателями, такими как `Box`, для создания владения объектом трейта: `Box<dyn Pet>`.

```
struct Dog {
    name: String,
    age: i8,
}

struct Cat {
    жизни: i8,
}

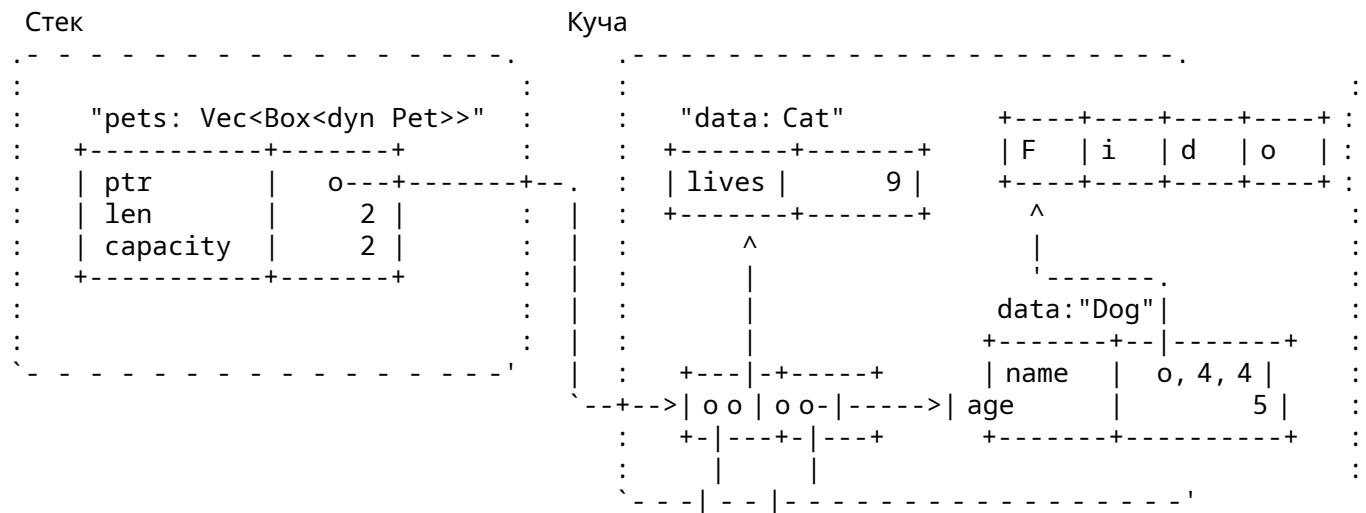
trait Pet {
    fn talk(&self) -> String;
}

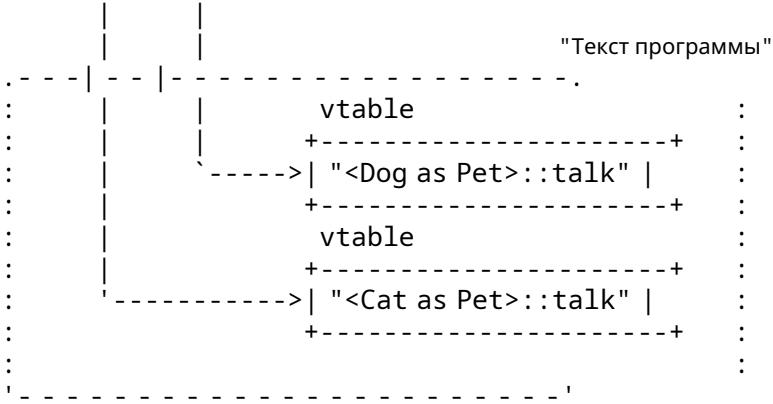
impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Гав, меня зовут {}!", self.name)
    }
}

impl Pet для Cat { fn talk(&self
) -> String { String::from("Мяу!") }
}

fn main() {
    let pets: Vec<Box<dyn Pet>> = vec![
        Box::new(Cat { lives: 9 }),
        Box::new(Dog { name: String::from("Fido"), age: 5 }),
    ];
    for pet in pets {
        println!("Привет, кто ты? {}", pet.talk());
    }
}
```

Расположение в памяти после выделения pets:





На этот слайд должно уйти около 10 минут.

- Типы, реализующие заданный трейд, могут иметь разные размеры. Это делает невозможным использование таких конструкций, как `Vec<dyn Pet>` в приведённом выше примере.
 - `dyn Pet` — это способ сообщить компилятору о типе с динамическим размером, который реализует трейд `Pet`.
 - В примере `pets` располагается в стеке, а данные вектора — в куче. Два элемента вектора являются *flat pointers*:
 - Fat pointer — это указатель двойной ширины. Он состоит из двух компонентов : указателя на фактический объект и указателя на таблицу виртуальных методов (vtable) для реализации `Pet` этого конкретного объекта.
 - Данные для `Dog` по имени `Fido` — это поля `name` и `age`. У `Cat` есть поле `lives`
 - Сравните эти выводы в приведённом выше примере:
- ```
println!("{} {}", std::mem::size_of::<Dog>(), std::mem::size_of::<Cat>());
println!("{} {}", std::mem::size_of::<&Dog>(), std::mem::size_of::<&Cat>());
println!("{} {}", std::mem::size_of::<& dyn Pet>());
println!("{} {}", std::mem::size_of::<Box< dyn Pet>>());
```

## 21.4 Упражнение: бинарное дерево

Двоичное дерево — это древовидная структура данных, в которой каждый узел имеет двух потомков: левого и правого. Мы создадим дерево, в котором каждый узел хранит значение. Для заданного узла  $N$  все узлы в левом поддереве  $N$  содержат меньшие значения, а все узлы в правом поддереве  $N$  — большие значения.

Реализуйте следующие типы так, чтобы указанные тесты проходили успешно.

```
// Узел двоичного дерева.
#[derive(Debug)]
struct Node<T: Ord> {
 value: T,
 left: Subtree<T>,
 right: Subtree<T>,
}

// Возможно пустое поддерево .
#[derive(Debug)]
struct Subtree<T: Ord>(Option<Box<Node<T>>);
```

```

/// Контейнер, хранящий набор значений с использованием двоичного дерева.
///
/// Если одно и то же значение добавляется несколько раз, оно сохраняется только один раз.
#[derive(Debug)]
pub struct BinaryTree<T: Ord> {
 root: Subtree<T>,
}

impl<T: Ord> BinaryTree<T> {
 fn new() -> Self {
 Self { root: Subtree::new() }
 }

 fn insert(&mut self, value: T) {
 self.root.insert(value);
 }

 fn has(&self, value: &T) -> bool {
 self.root.has(value)
 }

 fn len(&self) -> usize {
 self.root.len()
 }
}

// Реализуйте методы `new`, `insert`, `len` и `has` для `Subtree`.

#[cfg(test)]
mod tests {
 use super::*;

 #[test]
 fn len() {
 let mut tree = BinaryTree::new();
 assert_eq!(tree.len(), 0);
 tree.insert(2);
 assert_eq!(tree.len(), 1);
 tree.insert(1);
 assert_eq!(tree.len(), 2);
 tree.insert(2); // неуникальный элемент
 assert_eq!(tree.len(), 2);
 tree.insert(3);
 assert_eq!(tree.len(), 3);
 }

 #[test]
 fn has() {
 let mut tree = BinaryTree::new();
 fn check_has(tree: &BinaryTree<i32>, exp: &[bool]) {

```

```

 let got: Vec<bool> =
 (0..exp.len()).map(|i| tree.has(&(i as i32))).collect();
 assert_eq!(&got, exp);
}

check_has(&tree, &[false, false, false, false, false]);
tree.insert(0);
check_has(&tree, &[true, false, false, false, false]);
tree.insert(4);
check_has(&tree, &[true, false, false, false, true]);
tree.insert(4);
check_has(&tree, &[true, false, false, false, true]);
tree.insert(3);
check_has(&tree, &[true, false, false, true, true]);
}

#[test]
fn unbalanced() {
 let mut tree = BinaryTree::new();
 for i in 0..100 {
 tree.insert(i);
 }
 assert_eq!(tree.len(), 100);
 assert!(tree.has(&50));
}
}

```

#### 21.4.1 Решение

```

use std::cmp::Ordering;

/// Узел двоичного дерева.
#[derive(Debug)]
struct Node<T: Ord> {
 value: T,
 left: Subtree<T>,
 right: Subtree<T>,
}

/// Возможно пустое поддерево.
#[derive(Debug)]
struct Subtree<T: Ord> (Option<Box<Node<T>>>);

/// Контейнер, хранящий набор значений с использованием двоичного дерева.
///
/// Если одно и то же значение добавляется несколько раз, оно сохраняется только один раз.
#[derive(Debug)]
pub struct BinaryTree<T: Ord> {
 root: Subtree<T>,
}

```

```

impl<T: Ord> BinaryTree<T> {
 fn new() -> Self {
 Self { root: Subtree::new() }
 }

 fn insert(&mut self, value: T) {
 self.root.insert(value);
 }

 fn has(&self, value: &T) -> bool {
 self.root.has(value)
 }

 fn len(&self) -> usize {
 self.root.len()
 }
}

impl<T: Ord> Subtree<T> {
 fn new() -> Self {
 Self(None)
 }

 fn insert(&mut self, value: T) {
 match &mut self.0 {
 None => self.0 = Some(Box::new(Node::new(value))),
 Some(n) => match value.cmp(&n.value) {
 Ordering::Less => n.left.insert(value),
 Ordering::Equal => {},
 Ordering::Greater => n.right.insert(value),
 },
 }
 }

 fn has(&self, value: &T) -> bool {
 match &self.0 {
 None => false,
 Some(n) => match value.cmp(&n.value) {
 Ordering::Less => n.left.has(value),
 Ordering::Equal => true,
 Ordering::Greater => n.right.has(value),
 },
 }
 }

 fn len(&self) -> usize {
 match &self.0 {
 None => 0,
 Some(n) => 1 + n.left.len() + n.right.len(),
 }
 }
}

```

```

}

impl<T: Ord> Node<T> {
 fn new(value: T) -> Self {
 Self { value, left: Subtree::new(), right: Subtree::new() }
 }
}

#[cfg(test)]
mod tests {
 use super::*;

#[test]
fn len() {
 let mut tree = BinaryTree::new();
 assert_eq!(tree.len(), 0);
 tree.insert(2);
 assert_eq!(tree.len(), 1);
 tree.insert(1);
 assert_eq!(tree.len(), 2);
 tree.insert(2); // неуникальный элемент
 assert_eq!(tree.len(), 2);
 tree.insert(3);
 assert_eq!(tree.len(), 3);
}

#[test]
fn has() {
 let mut tree = BinaryTree::new();
 fn check_has(tree: &BinaryTree<i32>, exp: &[bool]) {
 let got: Vec<bool> =
 (0..exp.len()).map(|i| tree.has(&(i as i32))).collect();
 assert_eq!(&got, exp);
 }

 check_has(&tree, &[false, false, false, false, false]);
 tree.insert(0);
 check_has(&tree, &[true, false, false, false, false]);
 tree.insert(4);
 check_has(&tree, &[true, false, false, false, true]);
 tree.insert(4);
 check_has(&tree, &[true, false, false, false, true]);
 tree.insert(3);
 check_has(&tree, &[true, false, false, true, true]);
}

#[test]
fn unbalanced() {
 let mut tree = BinaryTree::new();
 for i in 0..100 {
 tree.insert(i);
}
}

```

```
 }
 assert_eq!(tree.len(), 100);
 assert!(!tree.has(&50));
}
}
```

## **Часть VI**

### **День 3: Вторая половина дня**

## **Глава 22**

# **С возвращением**

С учётом 10-минутных перерывов эта сессия должна занять около 1 часа 55 минут. В неё входит:

| Сегмент       | Продолжительность |
|---------------|-------------------|
| Займствования | 55 минут          |
| Времена жизни | 50 минут          |

# Глава 23

## Заемствование

Этот раздел займет примерно 55 минут. В нем содержится:

| Слайд                                      | Продолжительность |
|--------------------------------------------|-------------------|
| Заемствование значения                     | 10 минут          |
| Проверка заемствований                     | 10 минут          |
| Ошибки заемствования                       | 3 минуты          |
| Внутренняя изменяемость                    | 10 минут          |
| Упражнение: Статистика здоровья — 20 минут |                   |

### 23.1 Заемствование значения

Как мы уже видели ранее, вместо передачи владения при вызове функции, можно позволить функции заемствовать значение:

```
#[derive(Debug)]
struct Point(i32, i32);

fn add(p1: &Point, p2: &Point) -> Point {
 Point(p1.0 + p2.0, p1.1 + p2.1)
}

fn main() {
 let p1 = Point(3, 4);
 let p2 = Point(10, 20);
 let p3 = add(&p1, &p2);
 println!("{} + {} = {}", p1, p2, p3);
}
```

- Функция `add` заемствует два объекта `Point` и возвращает новый объект `Point`.
- Вызывающая сторона сохраняет владение входными значениями.

На этот слайд должно уйти около 10 минут.

Этот слайд повторяет материал о ссылках с первого дня, с небольшим расширением, включающим аргументы функций и возвращаемые значения.

## Дополнительные материалы для изучения

Примечания по возврату значений в стеке и инлайнингу:

- Показать, что возврат из `add` недорогой, поскольку компилятор может устраниć операцию копирования, встроив вызов `add` непосредственно в `main`. Измените приведённый выше код так, чтобы он выводил адреса в стеке, и запустите его в Playground или просмотрите ассемблерный код в Godbolt. При уровне оптимизации «DEBUG» адреса должны изменяться, тогда как при переключении на настройку «RELEASE» они остаются неизменными:

```
#[derive(Debug)]
struct Point(i32, i32);

fn add(p1: &Point, p2: &Point) -> Point {
 let p = Point(p1.0 + p2.0, p1.1 + p2.1);
 println!("&p.0: {:p}", &p.0);
 p
}

pub fn main() {
 let p1 = Point(3, 4);
 let p2 = Point(10, 20);
 let p3 = add(&p1, &p2);
 println!("&p3.0: {:p}", &p3.0);
 println!("{} + {} = {}", p1.0, p2.0, p3.0);
}
```

- Компилятор Rust может выполнять автоматическое встроение функций, которое можно отключить на уровне конкретной функции с помощью атрибута `#[inline(never)]`.
- После отключения печати адрес изменится на всех уровнях оптимизации. Если посмотреть на Godbolt или Playground, можно увидеть, что в этом случае возврат значения зависит от ABI, например, на amd64 два `i32`, составляющие точку, будут возвращены в двух регистрах (`eax` и `edx`).

## 23.2 Проверка заимствований

Проверяющий заимствования Rust (*borrow checker*) накладывает ограничения на способы заимствования значений. Мы уже видели, что ссылка не может прожить дольше значения, которое она заимствует:

```
fn main() {
 let x_ref = {
 let x = 10;
 &x
 };
 dbg!(x_ref);
}
```

Существует также второе основное правило, которое проверяющий заимствования обеспечивает: правило *aliasing*. Для данного значения в любой момент времени:

- Вы можете иметь одну или несколько общих ссылок на значение, или
- Вы можете иметь ровно одну эксклюзивную ссылку на значение.

```
fn main() {
 let mut a = 10;
 let b = &a;

 {
 let c = & mut a;
 *c = 20;
 }

 dbg!(a);
 dbg!(b);
}
```

На этот слайд должно уйти около 10 минут.

- Правило «*outlives*» было продемонстрировано ранее, когда мы впервые рассматривали ссылки. Мы повторяем его здесь, чтобы показать студентам, что проверка заимствований следует нескольким различным правилам для валидации заимствований.
- Приведённый выше код не компилируется, потому что *a* заимствована как изменяемая (через *c*) и как неизменяемая (через *b*) одновременно.
  - Обратите внимание, что требование заключается в том, чтобы конфликтующие ссылки не существовали в одной и той же точке. Не имеет значения, где ссылка разыменовывается. Попробуйте закомментировать *\*c = 20* и показать, что ошибка компилятора всё равно возникает, даже если мы никогда не используем *c*.
  - Обратите внимание, что промежуточная ссылка сне обязательна для возникновения конфликта заимствований. Замените *c* на прямое изменение *a* и продемонстрируйте, что это вызывает аналогичную ошибку. Это происходит потому, что прямое изменение значения фактически создаёт временную изменяемую ссылку.
- Переместите оператор *dbg!* для *b* перед областью видимости, которая вводит *c*, чтобы код компилировался.
  - После этого изменения компилятор понимает, что *b* используется только до нового изменяемого заимствования *a* через *c*. Это особенность проверяющего заимствования, называемая «нелексическими временем жизни».

## Дополнительные материалы для изучения

- Технически несколько изменяемых ссылок на один и тот же фрагмент данных могут существовать одновременно через повторное заимствование. Именно это позволяет передавать изменяемую ссылку в функцию без инвалидирования исходной ссылки. Этот пример из playground демонстрирует такое поведение.
- Rust использует ограничение эксклюзивной ссылки, чтобы гарантировать отсутствие гонок данных в многопоточном коде, поскольку только один поток может иметь изменяемый доступ к данным в любой момент времени.
- Rust также применяет это ограничение для оптимизации кода. Например, значение за общей ссылкой может быть безопасно кэшировано в регистре на протяжении времени жизни этой ссылки.
- Поля структуры могут заимствоваться независимо друг от друга, но вызов метода структуры заимствует всю структуру целиком, что потенциально инвалидирует ссылки на отдельные поля. См. этот пример в playground для иллюстрации данного случая.

## 23.3 Ошибки заимствований

В качестве конкретного примера того, как эти правила заимствования предотвращают ошибки памяти, рассмотрим случай изменения коллекции при наличии ссылок на её элементы:

```
fn main() {
 let mut vec = vec![1, 2, 3, 4, 5];
 let elem = &vec[2];
 vec.push(6);
 dbg!(elem);
}
```

Аналогично рассмотрим случай инвалидирования итератора:

```
fn main() {
 let mut vec = vec![1, 2, 3, 4, 5];
 for elem in &vec {
 vec.push(elem * 2);
 }
}
```

Этот слайд должен занять около 3 минут.

- В обоих случаях изменение коллекции путём добавления новых элементов может потенциально привести к инвалидированию существующих ссылок на элементы коллекции, если коллекция вынуждена выполнить перераспределение памяти.

## 23.4 Внутренняя изменяемость

В некоторых ситуациях необходимо изменять данные, находящиеся за разделяемой (только для чтения) ссылкой. Например, разделяемая структура данных может иметь внутренний кэш и желать обновлять этот кэш из методов, доступных только для чтения.

Паттерн «внутренней изменяемости» позволяет получить эксклюзивный (изменяемый) доступ за разделяемой ссылкой. Стандартная библиотека предоставляет несколько способов сделать это, при этом обеспечивая безопасность, обычно посредством проверки во время выполнения.

Этот слайд и его подразделы должны занять примерно 10 минут.

Главное, что следует усвоить из этого слайда, — Rust предоставляет безопасные способы изменения данных через разделённую ссылку. Существует множество способов обеспечить такую безопасность, и следующие под-слайды демонстрируют некоторые из них.

### 23.4.1 Cell

`Cell` оберачивает значение и позволяет получать или устанавливать его, используя только разделённую ссылку на `Cell`. Однако он не допускает никаких ссылок на внутреннее значение. Поскольку ссылок нет, правила заимствования не могут быть нарушены.

```
use std::cell::Cell;

fn main() {
 // Обратите внимание, что `cell` НЕ объявлен как изменяемый.
 let cell = Cell::new(5);
```

```
 cell.set(123);
 dbg!(cell.get());
}
```

- Cell — это простой способ обеспечить безопасность: у него есть метод `set`, принимающий `&self`. Это не требует проверки во время выполнения, но требует перемещения значений, что может иметь свои издержки.

### 23.4.2 RefCell

RefCell позволяет получать доступ и изменять обёрнутое значение, предоставляя альтернативные типы `Ref` и `RefMut`, которые имитируют `&T` / `&mut T`, не являясь при этом настоящими ссылками Rust.

Эти типы выполняют динамические проверки с использованием счётчика в `RefCell`, чтобы предотвратить существование `RefMut` вместе с другим `Ref` или `RefMut`.

Реализуя `Deref` (и `DerefMut` для `RefMut`), эти типы позволяют вызывать методы внутреннего значения без возможности выхода ссылок за пределы.

```
use std::cell::RefCell;
```

```
fn main() {
 // Обратите внимание, что `cell` НЕ объявлен как изменяемый.
 let cell = RefCell::new(5);

 {
 let mut cell_ref = cell.borrow_mut();
 *cell_ref = 123;

 // Это вызывает ошибку во время выполнения.
 // let other = cell.borrow();
 // println!("{}", other);
 }

 println!("{}{:?}{}", cell, cell);
}
```

- `RefCell` обеспечивает соблюдение стандартных правил заимствования Rust (либо несколько общих ссылок, либо одна исключительная) с помощью проверки во время выполнения. В данном случае все заимствования очень короткие и никогда не пересекаются, поэтому проверки всегда проходят успешно.
- Дополнительный блок в примере необходим для завершения заимствования, созданного вызовом `borrow_mut`, перед выводом содержимого `cell`. Попытка вывести заимствованный `RefCell` просто отображает сообщение "`{borrowed}`".

### Дополнительные материалы для изучения

Существуют также `OnceCell` и `OnceLock`, которые позволяют инициализацию при первом использовании. Для эффективного применения этих инструментов требуется знание, превышающее текущий уровень студентов.

## 23.5 Упражнение: Статистика здоровья

Вы работаете над реализацией системы мониторинга здоровья. В рамках этого проекта необходимо отслеживать статистику здоровья пользователей.

Вы начнёте с заглушки функции в `impl` блоке, а также с определения структуры `User`. Ваша задача — реализовать заглушенный метод для структуры `User`, определённой в `impl` блоке.

Скопируйте приведённый ниже код на <https://play.rust-lang.org/> и заполните отсутствующий метод:

```
#![allow(dead_code)]
pub struct User {
 name: String,
 age: u32,
 height: f32,
 visit_count: u32,
 last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
 height: f32,
 blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
 patient_name: &'a str,
 visit_count: u32,
 height_change: f32,
 blood_pressure_change: Option<(i32, i32)>,
}

impl User {
 pub fn new(name: String, age: u32, height: f32) -> Self {
 Self { name, age, height, visit_count: 0, last_blood_pressure: None }
 }

 pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport {
 todo!("Обновить статистику пользователя на основе измерений, полученных во время визита к врачу")
 }
}

#[test]
fn test_visit() {
 let mut bob = User::new(String::from("Bob"), 32, 155.2);
 assert_eq!(bob.visit_count, 0);
 let report =
 bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (120, 80) });
 assert_eq!(report.patient_name, "Bob");
 assert_eq!(report.visit_count, 1);
 assert_eq!(report.blood_pressure_change, None);
 assert!((report.height_change - 0.9).abs() < 0.00001);
}
```

```

let report =
 bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115, 76) });

assert_eq!(report.visit_count, 2);
assert_eq!(report.blood_pressure_change, Some((-5, -4)));
assert_eq!(report.height_change, 0.0);
}

```

### 23.5.1 Решение

```

#[allow(dead_code)]
pub struct User {
 name: String,
 age: u32,
 height: f32,
 visit_count: u32,
 last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
 height: f32,
 blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
 patient_name: &'a str,
 visit_count: u32,
 height_change: f32,
 blood_pressure_change: Option<(i32, i32)>,
}

impl User {
 pub fn new(name: String, age: u32, height: f32) -> Self {
 Self { name, age, height, visit_count: 0, last_blood_pressure: None }
 }

 pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport {
 self.visit_count += 1;
 let bp = measurements.blood_pressure;
 let report = HealthReport {
 patient_name: &self.name,
 visit_count: self.visit_count,
 height_change: measurements.height - self.height,
 blood_pressure_change: match self.last_blood_pressure {
 Some(lbp) => {
 Some((bp.0 as i32 - lbp.0 as i32, bp.1 as i32 - lbp.1 as i32))
 }
 None => None,
 },
 };
 }
}

```

```

 };
 self.height = measurements.height;
 self.last_blood_pressure = Some(bp);
 report
}
}

#[test]
fn test_visit() {
 let mut bob = User::new(String::from("Bob"), 32, 155.2);
 assert_eq!(bob.visit_count, 0);
 let report =
 bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (120, 80) });
 assert_eq!(report.patient_name, "Bob");
 assert_eq!(report.visit_count, 1);
 assert_eq!(report.blood_pressure_change, None);
 assert!((report.height_change - 0.9).abs() < 0.00001);

 let report =
 bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115, 76) });

 assert_eq!(report.visit_count, 2);
 assert_eq!(report.blood_pressure_change, Some((-5, -4)));
 assert_eq!(report.height_change, 0.0);
}
}

```

## Глава 24

# Времена жизни

Этот раздел займет около 50 минут. В нем рассматривается:

| Слайд                                       | Продолжительность |
|---------------------------------------------|-------------------|
| Аннотации времени жизни                     | 10 минут          |
| Опущение времени жизни                      | 5 минут           |
| Времена жизни в структурах данных — 5 минут |                   |
| Упражнение: Разбор Protobuf — 30 минут      |                   |

### 24.1 Аннотации времени жизни

Ссылка имеет время жизни , которое не должно «превышать» время жизни значения, на которое она ссылается. Это проверяется системой заимствований.

Время жизни может быть неявным — именно это мы наблюдали до настоящего момента. Времена жизни также могут быть явными: &'a Point , &'document str . Времена жизни начинаются с символа ' , а 'a — типичное имя по умолчанию. Запись &'a Point означает «заимствованный Point , действительный как минимум в течение времени жизни a ».

Только владение, а не аннотации времени жизни, контролирует момент уничтожения значений и определяет конкретное время жизни данного значения. Проверяющий заимствования (borrow checker) лишь проверяет, что заимствования никогда не выходят за пределы конкретного времени жизни значения.

Явные аннотации времени жизни, как и типы, обязательны в сигнатурах функций (но могут быть опущены в распространённых случаях). Они предоставляют информацию для вывода типов в местах вызова и внутри тела функции, помогая проверяющему заимствования выполнять свою задачу.

```
#[derive(Debug)]
struct Point(i32, i32);

fn left_most(p1: &Point, p2: &Point) -> &Point {
 if p1.0 < p2.0 { p1 } else { p2 }
}

fn main() {
 let p1 = Point(10, 10);
```

```

let p2 = Point(20, 20);
let p3 = left_most(&p1, &p2); // Какова продолжительность жизни p3?
dbg!(p3);
}

```

На этот слайд должно уйти около 10 минут.

В этом примере компилятор не знает, какую продолжительность жизни следует вывести для p3 . Анализ тела функции показывает, что она может безопасно предположить, что продолжительность жизни p3 короче из двух — у p1 и у p2 . Однако, как и с типами, Rust требует явного указания продолжительности жизни в аргументах функции и возвращаемом значении.

Добавьте 'асоответственно в left\_most:

**fn** left\_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point { Это означает, что существует некоторая продолжительность жизни 'a , которую одновременно превосходят p1 и p2 , и которая превосходит возвращаемое значение. Проверяющий заимствования (borrow checker) подтверждает это внутри тела функции и использует эту информацию в main для определения продолжительности жизни p3 .

Попробуйте освободить p2 в main для вывода p3 .

## 24.2 Времена жизни в вызовах функций

Времена жизни для аргументов функции и возвращаемых значений должны быть полностью указаны, однако Rust позволяет опускать времена жизни в большинстве случаев, следуя нескольким простым правилам. Это не вывод типов — это просто синтаксический сокращённый вариант.

- Каждому аргументу без аннотации времени жизни присваивается одно время жизни.
- Если существует только одно время жизни аргумента, оно присваивается всем неаннотированным возвращаемым значениям.
- Если существует несколько времён жизни аргументов, но первое относится к self , это время жизни присваивается всем неаннотированным возвращаемым значениям.

```

#[derive(Debug)]
struct Point(i32, i32);

fn cab_distance(p1: &Point, p2: &Point) -> i32 {
 (p1.0 - p2.0).abs() + (p1.1 - p2.1).abs()
}

fn find_nearest<'a>(points: &'a [Point], query: &Point) -> Option<&'a Point> {
 let mut nearest = None;
 for p in points {
 if let Some((_, nearest_dist)) = nearest {
 let dist = cab_distance(p, query);
 if dist < nearest_dist {
 nearest = Some((p, dist));
 }
 } else {
 nearest = Some((p, cab_distance(p, query)));
 };
 }
 nearest.map(|(p, _)| p)
}

```

```

fn main() {
 let points = &[Point(1, 0), Point(1, 0), Point(-1, 0), Point(0, -1)];
 let nearest = {
 let query = Point(0, 2);
 find_nearest(points, &Point(0, 2))
 };
 println!("{}:?", nearest);
}

```

На этот слайд отводится примерно 5 минут.

В этом примере функция `cab_distant` виально опущена.

Функция `nearest` представляет собой ещё один пример функции с несколькими ссылками в аргументах, требующей явной аннотации. В `main` возвращаемое значение может жить дольше, чем `query`.

Попробуйте изменить сигнатуру, чтобы «согласиться» о возвращаемых временах жизни:

```
fn find_nearest<'a, 'q>(points: &'a [Point], query: &'q Point) -> Option<&'q Point> {
```

Это не скомпилируется, что демонстрирует проверку аннотаций компилятором на корректность. Обратите внимание, что это не относится к сырьем указателям (`unsafe`), и это является частой причиной ошибок при работе с `unsafe` Rust.

Студенты могут спросить, когда следует использовать времена жизни. Заимствования в Rust всегда имеют времена жизни. Большую часть времени благодаря элиминированию и выводу типов их не нужно явно указывать. В более сложных случаях аннотации времени жизни помогают разрешить неоднозначности. Часто, особенно при прототипировании, проще работать с владением данных, клонируя значения там, где это необходимо.

## 24.3 Времена жизни в структурах данных

Если тип данных хранит заимствованные данные, он должен быть аннотирован временем жизни:

```

#[derive(Debug)]
enum HighlightColor {
 Розовый,
 Жёлтый,
}

#[derive(Debug)]
struct Highlight<'document> {
 slice: &'document str,
 color: HighlightColor,
}

fn main() {
 let doc = String::from("The quick brown fox jumps over the lazy dog.");
 let noun = Highlight { slice: &doc[16..19], color: HighlightColor::Yellow };
 let verb = Highlight { slice: &doc[20..25], color: HighlightColor::Pink };
 // drop(doc);
 dbg!(noun);
}

```

```
 dbg!(verb);
}
```

На этот слайд отводится примерно 5 минут.

- В приведённом выше примере аннотация на `Highlight` гарантирует, что данные, лежащие в основе содержащейся `&str`, живут как минимум так же долго, как и любой экземпляр `Highlight`, использующий эти данные. Структура не может существовать дольше, чем данные, на которые она ссылается.
- Если `doc` будет уничтожен до окончания времени жизни `noun` или `verb`, проверяющий заимствования выдаст ошибку.
- Типы с заимствованными данными требуют, чтобы пользователи сохраняли исходные данные. Это может быть полезно для создания лёгких представлений, но обычно делает их несколько сложнее в использовании.
- По возможности делайте так, чтобы структуры данных напрямую владели своими данными.
- Некоторые структуры с несколькими ссылками внутри могут иметь более одной аннотации времени жизни. Это может быть необходимо, если нужно описать отношения времени жизни между самими ссылками, помимо времени жизни самой структуры. Это очень продвинутые случаи использования.

## 24.4 Упражнение: Разбор Protobuf

В этом упражнении вы создадите парсер для бинарного кодирования `protobuf`. Не волнуйтесь, это проще, чем кажется! Это иллюстрирует распространённый шаблон парсинга — передачу срезов данных. Исходные данные при этом никогда не копируются.

Полный разбор сообщения `protobuf` требует знания типов полей, индексированных по их номерам. Обычно эта информация предоставляется в `proto` файле. В этом упражнении мы закодируем эту информацию в `match` операторах в функциях, вызываемых для каждого поля.

Мы используем следующий `proto`:

```
message PhoneNumber {
 optional string number = 1;
 optional string тип = 2;
}

message Person {
 optional string name = 1;
 optional int32 id = 2;
 repeated PhoneNumber phones = 3;
}
```

### Сообщения

Proto-сообщение кодируется как последовательность полей, расположенных одно за другим. Каждое поле реализуется как «тег», за которым следует значение. Тег содержит номер поля (например, 2 для поля `id` сообщения `Person`) и тип провода, определяющий способ извлечения полезной нагрузки из потока байтов. Эти данные объединяются в одно целое число, как показано в функции `unpack_tag` ниже.

### Varint

Целые числа, включая тег, представлены с помощью кодирования переменной длины, называемого VARINT. К счастью, функция `parse_varint` определена для вас ниже.

## Типы провода

Proto определяет несколько типов провода, из которых в данном упражнении используются только два.

Тип провода Varint содержит один varint и используется для кодирования значений proto типа int32, таких как Person.id .

Тип провода Len содержит длину, выраженную в формате varint, за которой следует полезная нагрузка указанного количества байт. Это используется для кодирования значений proto типа string , таких как Person.name . Он также применяется для кодирования значений proto, содержащих подсообщения, таких как Person.phones , где полезная нагрузка содержит кодировку подсообщения.

## Упражнение

Данный код также определяет обратные вызовы для обработки полей Person и PhoneNumber , а также для разбора сообщения в серию вызовов этих обратных вызовов.

Ваша задача — реализовать функцию parse\_field и трейд ProtoMessage для Person и PhoneNumber .

```
// Тип провода, как он представлен в потоке .
enum WireType {
 // wireType Varint указывает, что значение представлено одним VARINT .
 Varint,
 // wireType I64 указывает, что значение занимает ровно 8 байт в // порядке
 // little-endian и содержит 64-битное знаковое целое число или тип double .
 // I64, -- не требуется для этого упражнения
 // Тип wireType Len указывает, что значение представляет собой длину, заданную как
 // VARINT, за которым следует ровно такое количество байт .
 Len,
 // Тип wireType I32 указывает, что значение занимает ровно 4 байта в
 // порядке little-endian и содержит 32-битное знаковое целое число или число с плавающей точкой .
 // I32, -- не требуется для этого упражнения
}

#[derive(Debug)]
// Значение поля, типизированное на основе wire type .
enum FieldValue<'a> {
 Varint(u64),
 // I64(i64), -- не требуется для этого упражнения
 Len(&'a [u8]),
 // I32(i32), -- не требуется для этого упражнения
}

#[derive(Debug)]
// Поле, содержащее номер поля и его значение .
struct Field<'a> {
 field_num: u64,
 value: FieldValue<'a>,
}

trait ProtoMessage<'a>: Default {
 fn add_field(&mut self, field: Field<'a>);
```

```

}

impl From<u64> for WireType {
 fn from(value: u64) -> Self {
 match value {
 0 => WireType::Varint,
 //1 => WireType::I64, -- не требуется для этого упражнения
 2 => WireType::Len,
 //5 => WireType::I32, -- не требуется для этого упражнения
 _ => panic!("Недопустимый тип wire: {value}"),
 }
 }
}

impl<'a> FieldValue<'a> {
 fn as_str(&self) -> &'a str {
 let FieldValue::Len(data) = self else {
 panic!("Ожидалась строка типа `Len`");
 };
 std::str::from_utf8(data).expect("Недопустимая строка")
 }

 fn as_bytes(&self) -> &'a [u8] {
 let FieldValue::Len(data) = self else {
 panic!("Ожидались байты типа `Len`");
 };
 data
 }

 fn as_u64(&self) -> u64 {
 let FieldValue::Varint(value) = self else {
 panic!("Ожидался `u64` типа `Varint`");
 };
 *value
 }
}

/// Разбор VARINT, возвращающий разобранное значение и оставшиеся байты.
fn parse_varint(data: &[u8]) -> (u64, &[u8]) {
 for i in 0..7 {
 let Some(b) = data.get(i) else { panic!("Недостаточно байтов для varint"); };
 if b & 0x80 == 0 {
 // Это последний байт VARINT, поэтому преобразуем его в u64 и
 // возвращаем.
 let mut value = 0u64;
 for b in data[..=i].iter().rev() {
 value = (value << 7) | (b & 0x7f) as u64;
 }
 return (value, &data[i + 1..]);
 }
 }
}

```

```

 }
 }

 // Более 7 байтов является недопустимым.
 panic!("Слишком много байт для varint");
}

/// Преобразовать тег в номер поля и WireType.
fn unpack_tag(tag: u64) -> (u64, WireType) {
 let field_num = tag >> 3;
 let wire_type = WireType::from(tag & 0x7);
 (field_num, wire_type)
}

/// Разобрать поле, возвращая оставшиеся байты
fn parse_field(data: &[u8]) -> (Field, &[u8]) {
 let (tag, remainder) = parse_varint(data);
 let (field_num, wire_type) = unpack_tag(tag);
 let (fieldvalue, remainder) = match wire_type {
 _ => todo!("На основе wire type построить Field, потребляя столько байт, сколько необходимо");
 todo!("Вернуть поле и любые неиспользованные байты.")
 }
}

/// Разбор сообщения в заданных данных с вызовом `T::add_field` для каждого поля ///
/// сообщения.
///
/// Входные данные полностью обработаны.
fn parse_message<'a, T: ProtoMessage<'a>>(& mut data: &'a [u8]) -> T {
 let mut result = T:: default();
 while !data.is_empty() {
 let parsed = parse_field(data);
 result.add_field(parsed.0);
 data = parsed.1;
 }
 result
}

#[derive(Debug, Default)]
struct PhoneNumber<'a> {
 number: &'a str,
 type_: &'a str,
}

#[derive(Debug, Default)]
struct Person<'a> {
 name: &'a str,
 id: u64,
 phone: Vec<PhoneNumber<'a>>,
}

```

```

// TODO: Реализовать ProtoMessage для Person и PhoneNumber.

#[test]
fn test_id() {
 let person_id: Person = parse_message(&[0x10, 0x2a]);
 assert_eq!(person_id, Person { name: "", id: 42, phone: vec![] });
}

#[test]
fn test_name() {
 let person_name: Person = parse_message(&[
 0x0a, 0x0e, 0x62, 0x65, 0x61, 0x75, 0x74, 0x69, 0x66, 0x75, 0x6c, 0x20,
 0x6e, 0x61, 0x6d, 0x65,
]);
 assert_eq!(person_name, Person { name: "beautiful name", id: 0, phone: vec![] });
}

#[test]
fn test_just_person() {
 let person_name_id: Person =
 parse_message(&[0x0a, 0x04, 0x45, 0x76, 0x61, 0x6e, 0x10, 0x16]);
 assert_eq!(person_name_id, Person { name: "Evan", id: 22, phone: vec![] });
}

#[test]
fn test_phone() {
 let phone: Person = parse_message(&[
 0x0a, 0x00, 0x10, 0x00, 0x1a, 0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x33,
 0x34, 0x2d, 0x37, 0x37, 0x37, 0x2d, 0x39, 0x30, 0x39, 0x30, 0x12, 0x04,
 0x68, 0x6f, 0x6d, 0x65,
]);
 assert_eq!(
 phone,
 Person {
 name: "",
 id: 0,
 phone: vec![PhoneNumber { number: "+1234-777-9090", type_: "home" }],
 }
);
}

// Объедините всё это в один парсер.
#[test]
fn test_full_person() {
 let person: Person = parse_message(&[
 0x0a, 0x07, 0x6d, 0x61, 0x78, 0x77, 0x65, 0x6c, 0x6c, 0x10, 0x2a, 0x1a,
 0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0x35,
 0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0x1a,
 0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0x37,
 0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0x6c,
]);
}

```

```

 0x65,
]);
assert_eq!(
 person,
 Person {
 name: "maxwell",
 id: 42,
 phone: vec![
 PhoneNumber { number: "+1202-555-1212", type_: "home" },
 PhoneNumber { number: "+1800-867-5308", type_: "mobile" },
]
 }
);
}

```

Этот слайд и его подразделы должны занять примерно 30 минут.

- В этом упражнении существуют различные случаи, когда разбор protobufl может завершиться неудачей, например, если вы пытаетесь разобрать `i32`, а в буфере данных осталось менее 4 байт. В обычном Rust-коде мы бы обрабатывали это с помощью перечисления `Result`, но для упрощения в этом упражнении вызываем `panic!` при возникновении любых ошибок. На четвёртый день мы рассмотрим обработку ошибок в Rust более подробно.

#### 24.4.1 Решение

```

/// Тип провода, как он представлен в потоке.
enum WireType {
 /// WireType Varint указывает, что значение представлено одним VARINT.
 Varint,
 // WireType I64 указывает, что значение занимает ровно 8 байт в // порядке
 // little-endian и содержит 64-битное знаковое целое число или тип double.
 //I64, -- не требуется для этого упражнения
 // Тип WireType Len указывает, что значение представляет собой длину, заданную как
 // VARINT, за которым следует ровно такое количество байт.
 Len,
 // Тип WireType I32 указывает, что значение занимает ровно 4 байта в
 // порядке little-endian и содержит 32-битное знаковое целое число или число с плавающей точкой.
 //I32, -- не требуется для этого упражнения
}

#[derive(Debug)]
// Значение поля, типизированное на основе wire type.
enum FieldValue<'a> {
 Varint(u64),
 //I64(i64), -- не требуется для этого упражнения
 Len(&'a [u8]),
 //I32(i32), -- не требуется для этого упражнения
}

#[derive(Debug)]
// Поле, содержащее номер поля и его значение.
struct Field<'a> {

```

```

 field_num: u64,
 value: FieldValue<'a>,
 }

trait ProtoMessage<'a>: Default {
 fn add_field(&mut self, field: Field<'a>);
}

impl From<u64> for WireType {
 fn from(value: u64) -> Self {
 match value {
 0 => WireType::Varint,
 //1 => WireType::I64, -- не требуется для этого упражнения
 2 => WireType::Len,
 //5 => WireType::I32, -- не требуется для этого упражнения
 _ => panic!("Недопустимый тип wire: {value}"),
 }
 }
}

impl<'a> FieldValue<'a> {
 fn as_str(&self) -> &'a str {
 let FieldValue::Len(data) = self else {
 panic!("Ожидалась строка типа `Len`");
 };
 std::str::from_utf8(data).expect("Недопустимая строка")
 }

 fn as_bytes(&self) -> &'a [u8] {
 let FieldValue::Len(data) = self else {
 panic!("Ожидались байты типа `Len`");
 };
 data
 }

 fn as_u64(&self) -> u64 {
 let FieldValue::Varint(value) = self else {
 panic!("Ожидался `u64` типа `Varint`");
 };
 *value
 }
}

/// Разбор VARINT, возвращающий разобранное значение и оставшиеся байты.
fn parse_varint(data: &[u8]) -> (u64, &[u8]) {
 for i in 0..7 {
 let Some(b) = data.get(i) else { panic!("Недостаточно байтов для varint"); };
 if b & 0x80 == 0 {
 // Это последний байт VARINT, поэтому преобразуем его в

```

```

 // u64 и возвращаем.
 let mut value = 0u64;
 for b in data[..=i].iter().rev() {
 value = (value << 7) | (b & 0x7f) as u64;
 }
 return (value, &data[i + 1..]);
 }

 // Более 7 байтов является недопустимым.
 panic!("Слишком много байт для varint");
}

/// Преобразовать тег в номер поля и WireType.
fn unpack_tag(tag: u64) -> (u64, WireType) {
 let field_num = tag >> 3;
 let wire_type = WireType::from(tag & 0x7);
 (field_num, wire_type)
}

/// Разобрать поле, возвращая оставшиеся байты
fn parse_field(data: &[u8]) -> (Field, &[u8]) {
 let (tag, remainder) = parse_varint(data);
 let (field_num, wire_type) = unpack_tag(tag);
 let (fieldvalue, remainder) = match wire_type {
 WireType::Varint => {
 let (value, remainder) = parse_varint(remainder);
 (FieldValue::Varint(value), remainder)
 }
 WireType::Len => {
 let (len, remainder) = parse_varint(remainder);
 let len: usize = len.try_into().expect("len не является допустимым `usize`");
 if remainder.len() < len {
 panic!("Неожиданный EOF");
 }
 let (value, remainder) = remainder.split_at(len);
 (FieldValue::Len(value), remainder)
 }
 };
 (Field { field_num, value: fieldvalue }, remainder)
}

/// Разбор сообщения в заданных данных с вызовом `T::add_field` для каждого поля ///
/// сообщения.
///
/// Входные данные полностью обработаны.
fn parse_message<'a, T: ProtoMessage<'a>>(& mut data: &'a [u8]) -> T {
 let mut result = T::default();
 while !data.is_empty() {
 let parsed = parse_field(data);
 result.add_field(parsed.0);
 }
}

```

```

 data = parsed.1;
 }
 result
}

#[derive(PartialEq)]
#[derive(Debug, Default)]
struct PhoneNumber<'a> {
 number: &'a str,
 type_: &'a str,
}

#[derive(PartialEq)]
#[derive(Debug, Default)]
struct Person<'a> {
 name: &'a str,
 id: u64,
 phone: Vec<PhoneNumber<'a>>,
}

impl<'a> ProtoMessage<'a> for Person<'a> {
 fn add_field(&mut self, field: Field<'a>) {
 match field.field_num {
 1 => self.name = field.value.as_str(),
 2 => self.id = field.value.as_u64(),
 3 => self.phone.push(parse_message(field.value.as_bytes())),
 _ => {} // пропустить всё остальное
 }
 }
}

impl<'a> ProtoMessage<'a> для PhoneNumber<'a> {
 fn add_field(&mut self, field: Field<'a>) {
 match field.field_num {
 1 => self.number = field.value.as_str(),
 2 => self.type_ = field.value.as_str(),
 _ => {} // пропустить всё остальное
 }
 }
}

#[test]
fn test_id() {
 let person_id: Person = parse_message(&[0x10, 0x2a]);
 assert_eq!(person_id, Person { name: "", id: 42, phone: vec![] });
}

#[test]
fn test_name() {
 let person_name: Person = parse_message(&[
 0x0a, 0x0e, 0x62, 0x65, 0x61, 0x75, 0x74, 0x69, 0x66, 0x75, 0x6c, 0x20,
]);
}

```

```

 0x6e, 0x61, 0x6d, 0x65,
]);
 assert_eq!(person_name, Person { name: "beautiful name", id: 0, phone: vec![] });
}

#[test]
fn test_just_person() {
 let person_name_id: Person =
 parse_message(&[0x0a, 0x04, 0x45, 0x76, 0x61, 0x6e, 0x10, 0x16]);
 assert_eq!(person_name_id, Person { name: "Evan", id: 22, phone: vec![] });
}

#[test]
fn test_phone() {
 let phone: Person = parse_message(&[
 0x0a, 0x00, 0x10, 0x00, 0x1a, 0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x33,
 0x34, 0x2d, 0x37, 0x37, 0x37, 0x2d, 0x39, 0x30, 0x39, 0x30, 0x12, 0x04,
 0x68, 0x6f, 0x6d, 0x65,
]);
 assert_eq!(
 phone,
 Person {
 name: "",
 id: 0,
 phone: vec![PhoneNumber { number: "+1234-777-9090", type_: "home" }],
 }
);
}

// Объедините всё это в один парсер.
#[test]
fn test_full_person() {
 let person: Person = parse_message(&[
 0x0a, 0x07, 0x6d, 0x61, 0x78, 0x77, 0x65, 0x6c, 0x6c, 0x10, 0x2a, 0x1a,
 0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0x35,
 0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0x1a,
 0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0x37,
 0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0x6c,
 0x65,
]);
 assert_eq!(
 person,
 Person {
 name: "maxwell",
 id: 42,
 phone: vec![
 PhoneNumber { number: "+1202-555-1212", type_: "home" },
 PhoneNumber { number: "+1800-867-5308", type_: "mobile" },
]
 }
);
}

```

}

## **Часть VII**

# **День 4: Утро**

## Глава 25

# Добро пожаловать в День 4

Сегодня мы рассмотрим темы, связанные с созданием масштабного программного обеспечения на Rust:

- Итераторы: глубокое изучение трейта `Iterator`.
- Модули и видимость.
- Тестирование.
- Обработка ошибок: паники, `Result` и оператор `try ?`.
- Unsafe Rust: выход из ситуации, когда невозможно выразить логику безопасным Rust.

## Расписание

С учётом 10-минутных перерывов, эта сессия должна занять примерно 2 часа 50 минут. В неё входит:

| Длительность сегмента      |
|----------------------------|
| Добро пожаловать, 3 минуты |
| Итераторы, 55 минут        |
| Модули 45 минут            |
| Тестирование 45 минут      |

# Глава 26

## Итераторы

Этот раздел займет примерно 55 минут. В нем содержится:

| Слайд                                           | Продолжительность |
|-------------------------------------------------|-------------------|
| Мотивация                                       | 3 минуты          |
| Трейт Iterator                                  | 5 минут           |
| Вспомогательные методы итератора                | 5 минут           |
| collect                                         | 5 минут           |
| IntoIterator                                    | 5 минут           |
| Упражнение: Цепочка методов итератора, 30 минут |                   |

### 26.1 Мотивация итераторов

Если вы хотите итерироваться по содержимому массива, необходимо определить:

- Состояние для отслеживания текущей позиции в процессе итерации, например, индекс.
- Условие, определяющее завершение итерации.
- Логику обновления состояния итерации на каждой итерации цикла.
- Логику получения каждого элемента с использованием текущего состояния итерации.

В цикле `for` в стиле C эти элементы объявляются напрямую:

```
for (int i = 0; i < array_len; i += 1) {
 int elem = array[i];
}
```

В Rust мы объединяем это состояние и логику в объект, известный как «итератор».

Этот слайд должен занять около 3 минут.

- Этот слайд предоставляет контекст того, как итераторы Rust работают «под капотом». Мы используем (надеюсь, знакомую) конструкцию цикла `for` в стиле C, чтобы показать, что итерация требует некоторого состояния и логики, чтобы на следующем слайде продемонстрировать, как итератор объединяет их вместе.
- В Rust отсутствует цикл `for` в стиле C, но мы можем выразить то же самое с помощью `while`:

```
let array = [2, 4, 6, 8];
let mut i = 0;
while i < array.len() {
 let elem = array[i];
 i += 1;
}
```

## Дополнительные материалы для изучения

Существует другой способ итерации по массиву с использованием `for` в C и C++: можно применять указатель на начало и указатель на конец массива, а затем сравнивать эти указатели, чтобы определить момент завершения цикла.

```
for (int *ptr = array; ptr < array + len; ptr += 1) {
 int elem = *ptr;
}
```

Если студенты спросят, вы можете отметить, что именно так работают итераторы срезов и массивов в Rust (хотя реализованы они как итераторы Rust).

## 26.2 Iterator Trait

Трейт `Iterator` определяет, как объект может использоваться для генерации последовательности значений. Например, если мы хотим создать итератор, который может выдавать элементы среза, он может выглядеть примерно так:

```
struct SliceIter<'s> {
 slice: &'s [i32],
 i: usize,
}

impl<'s> Iterator for SliceIter<'s> {
 type Item = &'s i32;

 fn next(&mut self) -> Option<Self::Item> {
 if self.i == self.slice.len() {
 None
 } else {
 let next = &self.slice[self.i];
 self.i += 1;
 Some(next)
 }
 }
}

fn main() {
 let slice = &[2, 4, 6, 8];
 let iter = SliceIter { slice, i: 0 };
 for elem in iter {
 dbg!(elem);
 }
}
```

```
 }
}
```

На этот слайд отводится примерно 5 минут.

- Пример `SliceIter` реализует ту же логику, что и цикл `for` в стиле C, продемонстрированный на предыдущем слайде.
- Обратите внимание студентов, что итераторы ленивы: создание итератора лишь инициализирует структуру, но не выполняет никакой работы. Работа не начинается, пока не вызывается метод `next`.
- Итераторы не обязательно должны быть конечными! Абсолютно допустимо иметь итератор, который будет генерировать значения бесконечно. Например, полуоткрытый диапазон вида `0..` будет продолжаться до тех пор, пока не произойдет переполнение целочисленного типа.

## Дополнительные материалы для изучения

- «Реальная» версия `SliceIter` — это тип `slice::Iter` в стандартной библиотеке, однако реальная версия использует указатели на низком уровне вместо индекса, чтобы устранить проверки границ.
- Пример `SliceIter` хорошо иллюстрирует структуру, содержащую ссылку, и поэтому использующую аннотации времени жизни.
- Вы также можете продемонстрировать добавление универсального параметра к `SliceIter` для обеспечения работы с любым типом среза (не только с `&[i32]`).

## 26.3 Iterator — вспомогательные методы

Помимо метода `next`, который определяет поведение итератора, трейд `Iterator` предоставляет более 70 вспомогательных методов для создания настраиваемых итераторов.

```
fn main() {
 let result: i32 = (1..=10) // Создаём диапазон от 1 до 10
 .filter(|x| x % 2 == 0) // Оставляем только чётные числа
 .map(|x| x * x) // Возводим каждое число в квадрат
 .sum(); // Суммируем все возведённые в квадрат числа

 println!("Сумма квадратов чётных чисел от 1 до 10: {}", result);
}
```

На этот слайд отводится примерно 5 минут.

- Трейт `Iterator` реализует множество распространённых операций функционального программирования над коллекциями (например, `map`, `filter`, `reduce` и др.). Именно в этом трейте содержится вся документация по ним.
- Многие из этих вспомогательных методов принимают исходный итератор и создают новый итератор с изменённым поведением. Такие методы называются «методами-адаптерами итератора».
- Некоторые методы, например `sum` и `count`, потребляют итератор, извлекая из него все элементы.
- Эти методы предназначены для последовательного вызова, что облегчает создание пользовательского итератора, выполняющего именно необходимые операции.

## Дополнительные материалы для изучения

- Итераторы Rust чрезвычайно эффективны и высоко оптимизируются. Даже сложные итераторы, созданные путём комбинирования множества адаптерных методов, по-прежнему приводят к коду, столь же эффективному, как и эквивалентные императивные реализации.

## 26.4 collect

Метод `collect` позволяет создать коллекцию из `Iterator`.

```
fn main() {
 let primes = vec![2, 3, 5, 7];
 let prime_squares = primes.into_iter().map(|p| p * p).collect::<Vec<_>>();
 println!("prime_squares: {prime_squares:?}");
}
```

На этот слайд отводится примерно 5 минут.

- Любой итератор можно собрать в `Vec`, `VecDeque` или `HashSet`. Итераторы, возвращающие пары ключ-значение (то есть кортеж из двух элементов), также можно собрать в `HashMap` и `BTreeMap`.

Покажите студентам определение `collect` в документации стандартной библиотеки. Существует два способа указать универсальный тип В для этого метода:

- С помощью «turbofish»: `some_iterator.collect::<COLLECTION_TYPE>()`, как показано. Используемый здесь сокращённый синтаксис `_` позволяет Rust вывести тип элементов `Vec`.
- С выводом типа: `let prime_squares: Vec<_> = some_iterator.collect()` . Перепишите пример, используя эту форму.

## Дополнительные материалы для изучения

- Если студентам интересно, как это работает, вы можете упомянуть трейд `FromIterator`, который определяет, как каждый тип коллекции создаётся из итератора.
- Помимо базовых реализаций `FromIterator` для `Vec`, `HashMap` и других, существуют также более специализированные реализации, позволяющие выполнять такие операции, как преобразование `Iterator<Item = Result<V, E>>` в `Result<Vec<V>, E>`.
- Причина, по которой часто требуются аннотации типа при использовании `collect`, заключается в том, что этот метод обобщён по возвращаемому типу. Это затрудняет компилятору вывод правильного типа во многих случаях.

## 26.5 IntoIterator

Трейд `Iterator` описывает, как выполнять итерацию после создания итератора. Связанный трейд `IntoIterator` определяет, как создать итератор для данного типа. Он используется автоматически циклом `for`.

```
struct Grid {
 x_coords: Vec<u32>,
 y_coords: Vec<u32>,
}
```

```

impl IntoIterator для Grid {
 type Item = (u32, u32);
 type IntoIter = GridIter;
 fn into_iter(self) -> GridIter {
 GridIter { grid: self, i: 0, j: 0 }
 }
}

struct GridIter {
 grid: Grid,
 i: usize,
 j: usize,
}

impl Iterator для GridIter {
 type Item = (u32, u32);

 fn next(&mut self) -> Option<(u32, u32)> {
 если self.i >= self.grid.x_coords.len() { self.i = 0;

 self.j += 1;
 если self.j >= self.grid.y_coords.len() {
 return None;
 }
 }
 let res = Some((self.grid.x_coords[self.i], self.grid.y_coords[self.j]));
 self.i += 1;
 res
 }
}

fn main() {
 let grid = Grid { x_coords: vec![3, 5, 7, 9], y_coords: vec![10, 20, 30, 40] };
 for (x, y) in grid {
 println!("точка = {x}, {y}");
 }
}

```

На этот слайд отводится примерно 5 минут.

- IntoIterator — это трейд, обеспечивающий работу циклов flor. Он реализован для типов коллекций, таких как Vec<T> и ссылок на них, например &Vec<T> и &[T]. Диапазоны также реализуют этот трейд. Поэтому можно итерироваться по вектору с помощью flor i in some\_vec { .. }, но метода some\_vec.next() не существует.

Перейдите по ссылке на документацию для IntoIterator. Каждая реализация IntoIterator должна объявлять два типа:

- Item: тип для итерации, например, i8,
- IntoIter: тип Iterator, возвращаемый методом into\_iter .

Обратите внимание, что IntoIter и Item связаны: итератор должен иметь тот же Item тип, то есть возвращать Option<Item>.

Пример перебирает все комбинации координат x и y.

Попробуйте выполнить итерацию по сетке дважды в `main`. Почему это не работает? Обратите внимание, что `IntoIterator::into_iter` захватывает владение `self`.

Исправьте эту проблему, реализовав `IntoIterator` для `&Grid` и создав `GridRefIter`, который итерирует по ссылке. Версия с обеими `GridIter` и `GridRefIter` доступна в этом [playground](#).

Та же проблема может возникнуть для типов стандартной библиотеки: конструкция `for e in some_vector` захватит владение `some_vector` и будет итерировать по владимым элементам этого вектора. Используйте `for e in &some_vector` вместо этого, чтобы итерировать по ссылкам на элементы `some_vector`.

## 26.6 Упражнение: Цепочки методов итератора

В этом упражнении вам необходимо найти и использовать некоторые из предоставленных методов трейта `Iterator` для реализации сложного вычисления.

Скопируйте следующий код на <https://play.rust-lang.org/> и обеспечьте успешное прохождение тестов. Используйте выражение итератора и `collect` результат для формирования возвращаемого значения.

```
/// Вычисляет разности между элементами `values`, смещёнными на `offset`,
/// с циклическим переходом от конца `values` к началу.
///
/// Элемент `n` результата равен `values[(n+offset)%len] - values[n]`.
fn offset_differences(offset: usize, values: Vec<i32>) -> Vec<i32> {
 todo!()
}

#[test]
fn test_offset_one() {
 assert_eq!(offset_differences(1, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
 assert_eq!(offset_differences(1, vec![1, 3, 5]), vec![2, 2, -4]);
 assert_eq!(offset_differences(1, vec![1, 3]), vec![2, -2]);
}

#[test]
fn test_larger_offsets() {
 assert_eq!(offset_differences(2, vec![1, 3, 5, 7]), vec![4, 4, -4, -4]);
 assert_eq!(offset_differences(3, vec![1, 3, 5, 7]), vec![6, -2, -2, -2]);
 assert_eq!(offset_differences(4, vec![1, 3, 5, 7]), vec![0, 0, 0, 0]);
 assert_eq!(offset_differences(5, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
}

#[test]
fn test_degenerate_cases() {
 assert_eq!(offset_differences(1, vec![0]), vec![0]);
 assert_eq!(offset_differences(1, vec![1]), vec![0]);
 let empty: Vec<i32> = vec![];
 assert_eq!(offset_differences(1, empty), vec![]);
}
```

## 26.6.1 Решение

```
// Вычисляет разности между элементами `values`, смещёнными на `offset`,
// с циклическим переходом от конца `values` к началу.
///
/// Элемент `n` результата равен `values[(n+offset)%len] - values[n]`.
fn offset_differences(offset: usize, values: Vec<i32>) -> Vec<i32> {
 let a = values.iter();
 let b = values.iter().cycle().skip(offset);
 a.zip(b).map(|(a, b)| *b - *a).collect()
}

#[test]
fn test_offset_one() {
 assert_eq!(offset_differences(1, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
 assert_eq!(offset_differences(1, vec![1, 3, 5]), vec![2, 2, -4]);
 assert_eq!(offset_differences(1, vec![1, 3]), vec![2, -2]);
}

#[test]
fn test_larger_offsets() {
 assert_eq!(offset_differences(2, vec![1, 3, 5, 7]), vec![4, 4, -4, -4]);
 assert_eq!(offset_differences(3, vec![1, 3, 5, 7]), vec![6, -2, -2, -2]);
 assert_eq!(offset_differences(4, vec![1, 3, 5, 7]), vec![0, 0, 0, 0]);
 assert_eq!(offset_differences(5, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
}

#[test]
fn test_degenerate_cases() {
 assert_eq!(offset_differences(1, vec![0]), vec![0]);
 assert_eq!(offset_differences(1, vec![1]), vec![0]);
 let empty: Vec<i32> = vec![];
 assert_eq!(offset_differences(1, empty), vec![]);
}
```

## Глава 27

# Модули

Этот раздел займет примерно 45 минут. В нем содержится:

| Слайд                                           | Продолжительность |
|-------------------------------------------------|-------------------|
| Модули                                          | 3 минуты          |
| Иерархия файловой системы                       | 5 минут           |
| Видимость                                       | 5 минут           |
| Инкапсуляция                                    | 5 минут           |
| use, super, self                                | 10 минут          |
| Упражнение: Модули для GUI-библиотеки, 15 минут |                   |

### 27.1 Модули

Мы уже видели, как `impl`блоки позволяют нам организовывать функции в пространство имён типа.

Аналогично, `mod`позволяет организовывать типы и функции в пространство имён:

```
mod foo {
 pub fn do_something() {
 println!("В модуле foo");
 }
}

mod bar {
 pub fn do_something() {
 println!("В модуле bar");
 }
}

fn main() {
 foo::do_something();
 bar::do_something();
}
```

Этот слайд должен занять около 3 минут.

- Пакеты предоставляют функциональность и включают файл `Cargo.toml`, который описывает, как собрать набор из одного или нескольких `crate`.
- Crate представляют собой дерево модулей, где бинарный `crate` создаёт исполняемый файл, а библиотечный `crate` компилируется в библиотеку.
- Модули определяют организацию, область видимости и являются центральной темой данного раздела.

## 27.2 Иерархия файловой системы

Пропуск содержимого модуля указывает Rust искать его в другом файле:

```
mod garden;
```

Это сообщает Rust, что содержимое модуля `garden` находится в файле `src/garden.rs`. Аналогично, модуль `garden::vegetables` можно найти в файле `src/garden/vegetables.rs`.

Корень `crate` находится в:

- `src/lib.rs` (для библиотечного `crate`)
- `src/main.rs` (для бинарного `crate`)

Модули, определённые в файлах, также могут быть документированы с помощью «внутренних комментариев документации». Эти документы описывают элемент, который их содержит — в данном случае модуль.

```
// ! Этот модуль реализует сад, включая высокопроизводительную реализацию прорастания
// ! .
```

```
// Повторный экспорт типов из этого модуля .
pub use garden::Garden;
pub use seeds::SeedPacket;

// Посейте указанные пакеты семян .
pub fn sow(seeds: Vec<SeedPacket>) {
 todo!()
}

// Соберите урожай в саду, готовый к сбору .
pub fn harvest(garden: & mut Garden) {
 todo!()
}
```

На этот слайд отводится примерно 5 минут.

- До Rust 2018 модули должны были располагаться в `module/mod.rs` вместо `module.rs`, и это по-прежнему рабочая альтернатива для изданий после 2018 года.
- Основная причина введения `filename.rs` в качестве альтернативы `filename/mod.rs` заключалась в том, что множество файлов с именем `mod.rs` трудно различать в IDE.
- Более глубокое вложение может использовать папки, даже если основной модуль — файл:

```
src/
└── main.rs
└── top_module.rs
└── top_module/
 └── sub_module.rs
```

- Место, где Rust будет искать модули, можно изменить с помощью директивы компилятора:

```
#[path = "some/path.rs"]
mod some_module;
```

Это полезно, например, если вы хотите разместить тесты для модуля в файле с именем `some_module_test.rs`, аналогично соглашению в Go.

## 27.3 Видимость

Модули являются границей приватности:

- Элементы модуля по умолчанию являются приватными (скрывают детали реализации).
- Элементы родительского и соседних модулей всегда видимы.
- Другими словами, если элемент виден в модуле `foo`, он виден во всех потомках `foo`.

```
mod outer {
 fn private() {
 println!("outer::private");
 }

 pub fn public() {
 println!("outer::public");
 }
}

mod inner {
 fn private() {
 println!("outer::inner::private");
 }

 pub fn public() {
 println!("outer::inner::public");
 super::private();
 }
}

fn main() {
 outer::public();
}
```

На этот слайд отводится примерно 5 минут.

- Используйте ключевое слово `pub` для объявления модулей как публичных.

Кроме того, существуют расширенные спецификаторы `pub(...)` для ограничения области видимости публичных элементов.

- См. Rust Reference.
- Настройка видимости `pub(crate)` является распространённой практикой.
- Реже можно задать видимость для конкретного пути.
- В любом случае видимость должна быть предоставлена родительскому модулю (и всем его потомкам).

## 27.4 Видимость и инкапсуляция

Как и элементы в модуле, поля struct по умолчанию являются приватными. Приватные поля также видимы внутри остальной части модуля (включая дочерние модули). Это позволяет инкапсулировать детали реализации struct, контролируя, какие данные и функциональность доступны извне.

```
use outer::Foo;

mod outer {
 pub struct Foo {
 pub val: i32,
 is_big: bool,
 }

 impl Foo {
 pub fn new(val: i32) -> Self {
 Self { val, is_big: val > 100 }
 }
 }

 pub mod inner {
 use super::Foo;

 pub fn print_foo(foo: &Foo) {
 println!("Является ли {} большим? {}", foo.val, foo.is_big);
 }
 }
}

fn main() {
 let foo = Foo::new(42);
 println!("foo.val = {}", foo.val);
 // let foo = Foo { val: 42, is_big: true };

 outer::inner::print_foo(&foo);
 // println!("Является ли {} большим? {}", foo.val, foo.is_big);
}
```

На этот слайд отводится примерно 5 минут.

- Этот слайд демонстрирует, что приватность в структурах определяется на уровне модулей. Студенты, знакомые с объектно-ориентированными языками, могут привыкнуть к тому, что типы являются границей инкапсуляции, поэтому здесь показано, как Rust ведёт себя иначе, при этом демонстрируя, как можно достичь инкапсуляции.
- Обратите внимание, что поле `is_big` полностью контролируется `Foo`, что позволяет `Foo` управлять его инициализацией и обеспечивать соблюдение любых инвариантов (например, что `is_big` становится `true` только если `val > 100`).
- Укажите, как вспомогательные функции могут быть определены в том же модуле (включая дочерние модули) для получения доступа к приватным полям и методам типа.
- Первая закомментированная строка демонстрирует, что нельзя инициализировать структуру с приватными полями. Вторая демонстрирует, что также нельзя напрямую получить доступ к приватным

полям.

- Перечисления не поддерживают приватность: варианты и данные внутри этих вариантов всегда публичны.

## Дополнительные материалы для изучения

- Если студентам нужна дополнительная информация о приватности (или её отсутствии) в перечислениях, можно упомянуть `#[doc_hidden]` и `#[non_exhaustive]` и показать, как они используются для ограничения возможностей работы с перечислением.
- Приватность модуля сохраняется, даже если существуют `impl` блоки в других модулях (пример в [playground](#)).

## 27.5 use, super, self

Модуль может импортировать символы из другого модуля с помощью `use`. Обычно в начале каждого модуля можно увидеть что-то подобное:

```
use std::collections::HashSet;
use std::process::abort;
```

### Пути

Пути разрешаются следующим образом:

1. В качестве относительного пути:

- `foo` или `self::foo` ссылается на `foo` в текущем модуле,
- `super::foo` ссылается на `foo` в родительском модуле.

2. В качестве абсолютного пути:

- `crate::foo` ссылается на `foo` в корне текущего `crate`,
- `bar::foo` ссылается на `foo` в `crate bar`.

На этот слайд следует отвести около 8 минут.

- Часто символы «реэкспортируют» по более короткому пути. Например, верхневальный файл `lib.rs` в `crate` может содержать `mod storage;`

```
pub use storage::disk::DiskStorage;
pub use storage::network::NetworkStorage;
```

делая `DiskStorage` и `NetworkStorage` доступными для других `crate` с удобным и коротким путем.

- В основном, только элементы, которые объявлены в модуле, нужно импортировать с помощью `use`. Однако трейд должен находиться в области видимости, чтобы вызвать любые методы этого трейта, даже если тип, реализующий этот трейд, уже находится в области видимости. Например, чтобы использовать метод `read_to_string` у типа, реализующего трейд `Read`, необходимо выполнить `use std::io::Read`.
- Оператор `use` может содержать подстановочный знак: `use std::io::*;`. Это не рекомендуется, поскольку неясно, какие элементы импортируются, и они могут измениться со временем.

## 27.6 Упражнение: Модули для GUI-библиотеки

В этом упражнении вы реорганизуете реализацию небольшой GUI-библиотеки. Эта библиотека определяет трейт `Widget` и несколько его реализаций, а также функцию `main`.

Обычно каждый тип или набор тесно связанных типов помещают в отдельный модуль, поэтому для каждого типа виджета должен быть свой модуль.

### Настройка Cargo

Rust playground поддерживает только один файл, поэтому вам необходимо создать проект Cargo в вашей локальной файловой системе:

```
cargo init gui-modules
cd gui-modules
cargo run
```

Отредактируйте полученный файл `src/main.rs`, чтобы добавить операторы `mod`, и добавьте дополнительные файлы в каталог `src`.

### Исходный код

Ниже приведена реализация GUI-библиотеки с одним модулем:

```
pub trait Widget {
 /// Естественная ширина `self`.
 fn width(&self) -> usize;

 /// Отрисовать виджет в буфер.
 fn draw_into(&self, buffer: & mut dyn std::fmt::Write);

 /// Отрисовать виджет на стандартный вывод.
 fn draw(&self) {
 let mut buffer = String::new();
 self.draw_into(&mut buffer);
 println!("{}{buffer}");
 }
}

pub struct Label {
 label: String,
}

impl Label {
 fn new(label: &str) -> Label {
 Label { label: label.to_owned() }
 }
}

pub struct Button {
 label: Label,
}
```

```

impl Button {
 fn new(label: &str) -> Button {
 Button { label: Label::new(label) }
 }
}

pub struct Window {
 title: String,
 widgets: Vec<Box<dyn Widget>>,
}

impl Window {
 fn new(title: &str) -> Window {
 Window { title: title.to_owned(), widgets: Vec::new() }
 }

 fn add_widget(&mut self, widget: Box<dyn Widget>) {
 self.widgets.push(widget);
 }

 fn inner_width(&self) -> usize {
 std::cmp::max(
 self.title.chars().count(),
 self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
)
 }
}

impl Widget for Window {
 fn width(&self) -> usize {
 // Добавить 4 отступа для границ
 self.inner_width() + 4
 }

 fn draw_into(&self, buffer: & mut dyn std::fmt::Write) {
 let mut inner = String::new();
 for widget in &self.widgets {
 widget.draw_into(&mut inner);
 }

 let inner_width = self.inner_width();

 // TODO : изменить draw_into, чтобы возвращала Result<(), std::fmt::Error>. Затем
 // используйте здесь оператор // ? вместо .unwrap().
 writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
 writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
 writeln!(buffer, "+{:=<inner_width$}=+", "").unwrap();
 for line in inner.lines() {
 writeln!(buffer, "| {:inner_width$} |", line).unwrap();
 }
 }
}

```

```

 writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
 }
}

impl Widget for Button {
 fn width(&self) -> usize {
 self.label.width() + 8 // добавляем немного отступа
 }

 fn draw_into(&self, buffer: & mut dyn std::fmt::Write) {
 let width = self.width();
 let mut label = String::new();
 self.label.draw_into(&mut label);

 writeln!(buffer, "+{:-<width$}+", "").unwrap();
 for line in label.lines() {
 writeln!(buffer, "|{:^width$}|", &line).unwrap();
 }
 writeln!(buffer, "+{:-<width$}+", "").unwrap();
 }
}

impl Widget for Label {
 fn width(&self) -> usize {
 self.label.lines().map(|line| line.chars().count()).max().unwrap_or(0)
 }

 fn draw_into(&self, buffer: & mut dyn std::fmt::Write) {
 writeln!(buffer, "{}", &self.label).unwrap();
 }
}

fn main() {
 let mut window = Window::new("Rust GUI Demo 1.23");
 window.add_widget(Box::new(Label::new("This is a small text GUI demo.")));
 window.add_widget(Box::new(Button::new("Click me!")));
 window.draw();
}

```

На этот слайд и его под-слайды следует выделить примерно 15 минут.

Поощряйте студентов делить код так, как им кажется естественным, и привыкать к необходимым объявлениям mod, use и pub. После этого обсудите, какие организации являются наиболее идиоматичными.

### 27.6.1 Решение

```

src
├── main.rs
└── widgets
 └── button.rs

```

```

| └── label.rs
| └── window.rs
└── widgets.rs

// ---- src/widgets.rs ----
pub use button::Button;
pub use label::Label;
pub use window::Window;

mod button;
mod label;
mod window;

pub trait Widget {
 // Естественная ширина `self`.
 fn width(&self) -> usize;

 // Отрисовать виджет в буфер.
 fn draw_into(&self, buffer: & mut dyn std::fmt::Write);

 // Отрисовать виджет на стандартный вывод.
 fn draw(&self) {
 let mut buffer = String::new();
 self.draw_into(&mut buffer);
 println!("{}buffer");
 }
}

// ---- src/widgets/label.rs ----
use super::Widget;

pub struct Label {
 label: String,
}

impl Label {
 pub fn new(label: &str) -> Label {
 Label { label: label.to_owned() }
 }
}

impl Widget for Label {
 fn width(&self) -> usize {
 // ANCHOR_END: Label-width
 self.label.lines().map(|line| line.chars().count()).max().unwrap_or(0)
 }

 // ANCHOR: Label-draw_into
 fn draw_into(&self, buffer: & mut dyn std::fmt::Write) {
 // ANCHOR_END: Label-draw_into
 writeln!(buffer, "{}", &self.label).unwrap();
 }
}

```

```

}

// ---- src/widgets/button.rs ----
use super::{Label, Widget};

pub struct Button {
 label: Label,
}

impl Button {
 pub fn new(label: &str) -> Button {
 Button { label: Label::new(label) }
 }
}

impl Widget for Button {
 fn width(&self) -> usize {
 // ANCHOR_END: Button-width
 self.label.width() + 8 // добавляем немного отступа
 }

 // ANCHOR: Button-draw_into
 fn draw_into(&self, buffer: & mut dyn std::fmt::Write) {
 // ANCHOR_END: Button-draw_into
 let width = self.width();
 let mut label = String::new();
 self.label.draw_into(&mut label);

 writeln!(buffer, "+{:<{}+}", "", width).unwrap();
 for line in label.lines() {
 writeln!(buffer, "|{:width$}|", &line).unwrap();
 }
 writeln!(buffer, "+{:<{}+}", "", width).unwrap();
 }
}

// ---- src/widgets/window.rs ----
use super::Widget;

pub struct Window {
 title: String,
 widgets: Vec<Box<dyn Widget>>,
}

impl Window {
 pub fn new(title: &str) -> Window {
 Window { title: title.to_owned(), widgets: Vec::new() }
 }

 pub fn add_widget(&mut self, widget: Box<dyn Widget>) {
 self.widgets.push(widget);
 }
}

```

```

fn inner_width(&self) -> usize {
 std::cmp::max(
 self.title.chars().count(),
 self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
)
}

impl Widget for Window {
 fn width(&self) -> usize {
 // ANCHOR_END: Window-width
 // Добавить 4 отступа для границ
 self.inner_width() + 4
 }

 // ЯКОРЬ: Window-draw_into
 fn draw_into(&self, buffer: &mut dyn std::fmt::Write) { // ЯКОРЬ_КОНЕЦ: Window-draw_into let mut
 inner = String::new();
 for widget in &self.widgets {
 widget.draw_into(&mut inner);
 }

 let inner_width = self.inner_width();

 // TODO : после изучения обработки ошибок можно изменить // draw_
 // into, чтобы возвращать Result<(), std::fmt::Error>. Затем ис-
 // пользуйте // оператор ? здесь вместо .unwrap().
 writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
 writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
 writeln!(buffer, "+{:=<inner_width$}=+", "").unwrap();
 for line in inner.lines() {
 writeln!(buffer, "| {:inner_width$} |", line).unwrap();
 }
 writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
 }
}

// ---- src/main.rs ----
mod widgets;

use widgets::{Button, Label, Widget, Window};

fn main() {
 let mut window = Window::new("Rust GUI Demo 1.23");
 window.add_widget(Box::new(Label::new("This is a small text GUI demo.")));
 window.add_widget(Box::new(Button::new("Click me!")));
 window.draw();
}

```

## Глава 28

# Тестирование

Этот раздел займет примерно 45 минут. В нем содержится:

| Слайд                                    | Продолжительность |
|------------------------------------------|-------------------|
| Модульные тесты                          | 5 минут           |
| Другие виды тестов                       | 5 минут           |
| Проверки компилятора и Clippy — 3 минуты |                   |
| Упражнение: алгоритм Луна — 30 минут     |                   |

### 28.1 Модульные тесты

Rust и Cargo поставляются с простым фреймворком для модульного тестирования. Тесты помечаются атрибутом `#[test]`. Модульные тесты часто размещают во вложенном модуле `tests`, используя `#[cfg(test)]` для условной компиляции только при сборке тестов.

```
fn first_word(text: &str) -> &str {
 match text.find(' ') {
 Some(idx) => &text[..idx],
 None => &text,
 }
}

#[cfg(test)]
mod tests {
 use super::*;

 #[test]
 fn test_empty() {
 assert_eq!(first_word(""), "");
 }

 #[test]
 fn test_single_word() {
 assert_eq!(first_word("Hello"), "Hello");
 }
}
```

```

 }

#[test]
fn test_multiple_words() {
 assert_eq!(first_word("Hello World"), "Hello");
}
}

```

- Это позволяет выполнять модульное тестирование приватных вспомогательных функций.
- Атрибут `#[cfg(test)]` активен только при запуске `cargo test`.

## 28.2 Другие виды тестов

### Интеграционные тесты

Если вы хотите протестировать библиотеку как клиент, используйте интеграционный тест.

Создайте файл `.rs` в каталоге `tests/`:

```

// tests/my_library.rs
use my_library::init;

#[test]
fn test_init() {
 assert!(init().is_ok());
}

```

Эти тесты имеют доступ только к публичному API вашего крейта.

### Тесты документации

Rust обладает встроенной поддержкой тестов документации:

```

/// Укорачивает строку до заданной длины.
///
/// ``
/// # use playground::shorten_string;
/// assert_eq!(shorten_string("Hello World", 5), "Hello");
/// assert_eq!(shorten_string("Hello World", 20), "Hello World");
/// ``
pub fn shorten_string(s: &str, length: usize) -> &str {
 &s[..std::cmp::min(length, s.len())]
}

```

- Блоки кода в `///` комментариях автоматически распознаются как код на Rust.
- Код будет скомпилирован и выполнен в рамках `cargo test`.
- Добавление `#` в коде скроет его из документации, но код по-прежнему будет скомпилирован и выполнен.
- Проверьте приведённый выше код в Rust Playground.

## 28.3 Линтеры компилятора и Clippy

Компилятор Rust выдаёт отличные сообщения об ошибках, а также полезные встроенные линтеры. Clippy предоставляет ещё больше линтеров, организованных в группы, которые можно включать для каждого проекта.

```
#[deny(clippy::cast_possible_truncation)]
fn main() {
 let mut x = 3;
 while (x < 70000) {
 x *= 2;
 }
 println!("X probably fits in a u16, right? {}", x as u16);
}
```

Этот слайд должен занять около 3 минут.

Здесь видны предупреждения компилятора, но не предупреждения Clippy. Запустите Clippy на сайте [Playground](#), чтобы показать предупреждения Clippy. Clippy обладает обширной документацией по своим предупреждениям и постоянно добавляет новые (включая предупреждения с дефолтным запретом).

Обратите внимание, что ошибки или предупреждения `help: ...` можно исправить с помощью `cargo fix` или через ваш редактор.

## 28.4 Упражнение: Алгоритм Луна

Алгоритм Луна используется для проверки номеров кредитных карт. Алгоритм принимает строку на вход и выполняет следующие действия для проверки номера кредитной карты:

- Игнорировать все пробелы. Отклонять номера с менее чем двумя цифрами. Отклонять буквы и другие недопустимые символы, не являющиеся цифрами.
- Двигаясь справа налево, удваивать каждую вторую цифру: для числа **1234** мы удваиваем 3 и 1. Для числа 98765 мы удваиваем 6 и 5.
- После удвоения цифры, если результат больше 9, сложите цифры этого результата. Таким образом, удвоение 7 даёт 14, что становится  $1 + 4 = 5$ .
- Сложите все не удвоенные и удвоенные цифры.
- Номер кредитной карты считается действительным, если сумма заканчивается на 0.

Предоставленный код содержит ошибочную реализацию алгоритма Луна, а также два базовых модульных теста, подтверждающих, что большая часть алгоритма реализована корректно.

Скопируйте код ниже на <https://play.rust-lang.org/> и напишите дополнительные тесты, чтобы выявить ошибки в предоставленной реализации и исправить найденные баги.

```
pub fn luhn(cc_number: &str) -> bool {
 let mut sum = 0;
 let mut double = false;

 for c in cc_number.chars().rev() {
 if let Some(digit) = c.to_digit(10) {
 if double {
 let double_digit = digit * 2;
 sum += if double_digit > 9 { double_digit - 9 } else { double_digit };
 } else {
 sum += digit;
 }
 double = !double;
 }
 }
 sum % 10 == 0
}
```

```

 } else {
 sum += digit;
 }
 double = !double;
 } else {
 continue;
 }
}

sum % 10 == 0
}

#[cfg(test)]
mod test {
 use super::*;

#[test]
fn test_valid_cc_number() {
 assert!(luhn("4263 9826 4026 9299"));
 assert!(luhn("4539 3195 0343 6467"));
 assert!(luhn("7992 7398 713"));
}

#[test]
fn test_invalid_cc_number() {
 assert!(!luhn("4223 9826 4026 9299"));
 assert!(!luhn("4539 3195 0343 6476"));
 assert!(!luhn("8273 1232 7352 0569"));
}
}
}

```

#### 28.4.1 Решение

```

pub fn luhn(cc_number: &str) -> bool {
 let mut sum = 0;
 let mut double = false;
 let mut digits = 0;

 for c in cc_number.chars().rev() {
 if let Some(digit) = c.to_digit(10) {
 digits += 1;
 if double {
 let double_digit = digit * 2;
 sum +=
 if double_digit > 9 { double_digit - 9 } else { double_digit };
 } else {
 sum += digit;
 }
 double = !double;
 } else if c.is_whitespace() {
 // Новое: принимаем пробельные символы.
 }
 }
}

```

```

 continue;
 } else {
 // Новое: отклоняем все остальные символы.
 return false;
 }
}

// Новое: проверяем, что у нас как минимум две цифры
digits >= 2 && sum % 10 == 0
}

#[cfg(test)]
mod test {
 use super::*;

 #[test]
 fn test_valid_cc_number() {
 assert!(luhn("4263 9826 4026 9299"));
 assert!(luhn("4539 3195 0343 6467"));
 assert!(luhn("7992 7398 713"));
 }

 #[test]
 fn test_invalid_cc_number() {
 assert!(!luhn("4223 9826 4026 9299"));
 assert!(!luhn("4539 3195 0343 6476"));
 assert!(!luhn("8273 1232 7352 0569"));
 }

 #[test]
 fn test_non_digit_cc_number() {
 assert!(!luhn("foo"));
 assert!(!luhn("foo 0 0"));
 }

 #[test]
 fn test_empty_cc_number() {
 assert!(!luhn(""));
 assert!(!luhn(" "));
 assert!(!luhn(" "));
 assert!(!luhn(" "));
 }

 #[test]
 fn test_single_digit_cc_number() {
 assert!(!luhn("0"));
 }

 #[test]
 fn test_two_digit_cc_number() {
 assert!(luhn(" 0 0 "));
 }
}

```

} }

## **Часть VIII**

# **День 4: После полудня**

## **Глава 29**

# **С возвращением**

С учётом 10-минутных перерывов эта сессия займет около 2 часов 20 минут. В ней содержится:

| Сегмент                    | Продолжительность |
|----------------------------|-------------------|
| Обработка ошибок, 55 минут |                   |
| Unsafe Rust                | 1 час 15 минут    |

# Глава 30

# Обработка ошибок

Этот раздел займет примерно 55 минут. В нем содержится:

| Слайд                                             | Продолжительность |
|---------------------------------------------------|-------------------|
| Паники                                            | 3 минуты          |
| Result                                            | 5 минут           |
| Оператор try                                      | 5 минут           |
| Преобразования try                                | 5 минут           |
| Трейт Error                                       | 5 минут           |
| thiserror                                         | 5 минут           |
| anyhow                                            | 5 минут           |
| Упражнение: Переписывание с использованием Result | 20 минут          |

## 30.1 Паники

Rust обрабатывает фатальные ошибки с помощью «panic».

Rust вызовет panic, если во время выполнения произойдет фатальная ошибка:

```
fn main() {
 let v = vec![10, 20, 30];
 dbg!(v[100]);
}
```

- Паники предназначены для необратимых и неожиданных ошибок.
  - Паники являются симптомами ошибок в программе.
  - Сбои во время выполнения, такие как неудачные проверки границ, могут вызвать panic
  - Утверждения (например, assert!) вызывают panic при ошибке
  - Специфичные по назначению panic могут использовать макрос panic!.
- Panic «разворачивает» стек, вызывая деструкторы значений так, как если бы функции вернули управление.
- Используйте API без panic (например, Vec::get), если аварийное завершение недопустимо.

Этот слайд должен занять около 3 минут.

По умолчанию panic вызывает разворачивание стека. Разворачивание можно перехватить:

```

use std::panic;

fn main() {
 let result = panic::catch_unwind(|| "Здесь проблем нет!");
 dbg!(result);

 let result = panic::catch_unwind(|| {
 panic!("ой нет!");
 });
 dbg!(result);
}

```

- Перехват panic — редкая практика; не пытайтесь реализовывать исключения с помощью `catch_unwind!`
- Это может быть полезно на серверах, которые должны продолжать работу, даже если один запрос завершился сбоем.
- Это не работает, если в вашем `Cargo.toml` установлено `panic = 'abort'`.

## 30.2 Результат

Основным механизмом обработки ошибок в Rust является перечисление `Result`, которое мы кратко рассмотрели при обсуждении типов стандартной библиотеки.

```

use std::fs::File;
use std::io::Read;

fn main() {
 let file: Result<File, std::io::Error> = File::open("diary.txt");
 match file {
 Ok(mut file) => {
 let mut contents = String::new();
 if let Ok(bytes) = file.read_to_string(&mut contents) {
 println!("Дорогой дневник: {contents} ({bytes} байт)");
 } else {
 println!("Не удалось прочитать содержимое файла");
 }
 }
 Err(err) => {
 println!("Дневник не удалось открыть: {err}");
 }
 }
}

```

На этот слайд отводится примерно 5 минут.

- `Result` имеет два варианта: `Ok`, содержащий значение успеха, и `Err`, содержащий значение ошибки.
- Возможность возникновения ошибки в функции отражается в её сигнатуре через возвращаемое значение типа `Result`.
- Как и с `Option`, невозможно забыть обработать ошибку: вы не можете получить доступ ни к значению успеха, ни к значению ошибки без предварительного сопоставления с образцом `Result` для определения варианта. Методы, такие как `unwrap`, упрощают написание быстрого и небрежного кода, который не выполняет надёжную обработку ошибок, но позволяют всегда видеть в

исходном коде места, где пропускается корректная обработка ошибок.

## Дополнительные материалы для изучения

Полезно сравнить обработку ошибок в Rust с конвенциями обработки ошибок, знакомыми студентам из других языков программирования.

### Исключения

- Многие языки используют исключения, например C++, Java, Python.
- В большинстве языков программирования, за исключением некоторых, возможность функции выбрасывать исключение не отражается в её сигнатуре типа. Это обычно означает, что при вызове функции нельзя определить, может ли она выбросить исключение или нет.
- Исключения, как правило, разворачиваются стек вызовов, распространяясь вверх до тех пор, пока не будет достигнут try блок. Ошибка, возникшая глубоко в стеке вызовов, может повлиять на несвязанную функцию выше по стеку.

### Номера ошибок

- В некоторых языках функции возвращают номер ошибки (или другое значение ошибки) отдельно от успешного результата функции. Примерами являются C и Go.
- В зависимости от языка возможно забыть проверить значение ошибки, в результате чего может использоваться неинициализированное или иным образом недопустимое успешное значение.

## 30.3 Оператор try

Ошибки времени выполнения, такие как отказ в соединении или файл не найден, обрабатываются с помощью типа Result, но сопоставление с этим типом при каждом вызове может быть громоздким. Оператор try ? используется для возврата ошибок вызывающему коду. Он позволяет пре-

```
образовать распространённое сопоставление some_expression {
 Ok(value) => value,
 Err(err) => return Err(err),
}
```

в гораздо более простое

```
some_expression?
```

Мы можем использовать это для упрощения кода обработки ошибок:

```
use std::io::Read;
use std::{fs, io};

fn read_username(path: &str) -> Result<String, io::Error> {
 let username_file_result = fs::File::open(path);
 let mut username_file = match username_file_result {
 Ok(file) => file,
 Err(err) => return Err(err),
```

```

};

let mut username = String::new();
match username_file.read_to_string(&mut username) {
 Ok(_) => Ok(username),
 Err(err) => Err(err),
}
}

fn main() {
 //fs::write("config.dat", "alice").unwrap();
 let username = read_username("config.dat");
 println!("username or error: {username:?}");
}

```

На этот слайд отводится примерно 5 минут.

Упростите функцию `read_username` с помощью `?`.

Ключевые моменты:

- Переменная `username` может принимать значение либо `Ok(string)`, либо `Err(error)`.
- Используйте вызов `fs::write` для проверки различных сценариев: отсутствия файла, пустого файла и файла с именем пользователя.
- Обратите внимание, что функция `main` может возвращать `Result<(), E>`, при условии, что `E` реализует `std::process::Termination`. На практике это означает, что `E` реализует трейд `Debug`. Исполняемый файл выведет вариант `Err` и вернёт ненулевой код завершения в случае ошибки.

## 30.4 Преобразования `try`

Фактическое расширение оператора `?` сложнее, чем было указано ранее:

выражение?

работает так же, как конструкция

```

match expression { Ok(value)
 => value,
 Err(err) => return Err(From::from(err)),
}

```

Вызов `From::from` означает, что мы пытаемся преобразовать тип ошибки в тип, возвращаемый функцией. Это облегчает инкапсуляцию ошибок в ошибки более высокого уровня.

### Пример

```

use std::error::Error;
use std::io::Read;
use std::{fmt, fs, io};

#[derive(Debug)]
enum ReadUsernameError {
 IoError(io::Error),
 EmptyUsername(String),
}

```

```

}

impl Error for ReadUsernameError {}

impl fmt::Display for ReadUsernameError {
 fn fmt(&self, f: & mut fmt::Formatter) -> fmt::Result {
 match self {
 Self::IoError(e) => write!(f, "I/O error: {}", e),
 Self::EmptyUsername(path) => write!(f, "Имя пользователя не найдено в {}", path),
 }
 }
}

impl From<io::Error> for ReadUsernameError {
 fn from(err: io::Error) -> Self {
 Self::IoError(err)
 }
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
 let mut username = String::with_capacity(100);
 fs::File::open(path)?.read_to_string(&mut username)?;
 if username.is_empty() {
 return Err(ReadUsernameError::EmptyUsername(String::from(path)));
 }
 Ok(username)
}

fn main() {
 //std::fs::write("config.dat", "").unwrap();
 let username = read_username("config.dat");
 println!("username or error: {}", username);
}

```

На этот слайд отводится примерно 5 минут.

Оператор `?` должен возвращать значение, совместимое с типом возвращаемого значения функции. Для `Result` это означает, что типы ошибок должны быть совместимы. Функция, возвращающая `Result<T, ErrorOuter>`, может использовать оператор `?` для значения типа `Result<U, ErrorInner>` только если `ErrorOuter` и `ErrorInner` являются одним и тем же типом или если `ErrorOuter` реализует `From<ErrorInner>`.

Распространённой альтернативой реализации `From` является использование `Result::map_err`, особенно когда преобразование происходит только в одном месте.

Требований к совместимости для `Option` не существует. Функция, возвращающая `Option<T>`, может использовать оператор `?` для `Option<U>` с произвольными типами `T` и `U`.

Функция, возвращающая `Result`, не может использовать `?` на `Option` и наоборот. Однако `Option::ok_or` преобразует `Option` в `Result`, в то время как `Result::ok` превращает `Result` в `Option`.

## 30.5 Динамические типы ошибок

Иногда необходимо разрешить возвращать ошибку любого типа без создания собственного enum, охватывающего все возможные варианты. Трейт std::error::Error облегчает создание объекта трейта, способного содержать любую ошибку.

```
use std::error::Error;
use std::fs;
use std::io::Read;

fn read_count(path: &str) -> Result<i32, Box<dyn Error>> {
 let mut count_str = String::new();
 fs::File::open(path)?.read_to_string(&mut count_str)?;
 let count: i32 = count_str.parse()?;
 Ok(count)
}

fn main() {
 fs::write("count.dat", "1i3").unwrap();
 match read_count("count.dat") {
 Ok(count) => println!("Count: {}", count),
 Err(err) => println!("Error: {}", err),
 }
}
```

На этот слайд отводится примерно 5 минут.

Функция `read_count` может возвращать `std::io::Error` (из операций с файлами) или `std::num::ParseIntError` (из `String::parse`).

Упаковка ошибок в `Box` экономит код, но лишает возможности корректно обрабатывать различные случаи ошибок по-разному в программе. Поэтому обычно не рекомендуется использовать `Box<dyn Error>` в публичном API библиотеки, однако это может быть хорошим решением в программе, где требуется лишь вывести сообщение об ошибке.

Обязательно реализуйте трейд `std::error::Error` при определении собственного типа ошибки, чтобы его можно было упаковать в `Box`.

## 30.6 thiserror

Трейт `thiserror` предоставляет макросы, которые помогают избежать шаблонного кода при определении типов ошибок. Он предоставляет derive-макросы, которые упрощают реализацию `From<T>`, `Display` и трейда `Error`.

```
use std::io::Read;
use std::{fs, io};
use thiserror::Error;

#[derive(Debug, Error)]
enum ReadUsernameError {
 #[error("I/O error: {0}")]
 IoError(#[from] io::Error),
 #[error("Found no username in {0}")]
 EmptyUsername(String),
```

```

}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
 let mut username = String::with_capacity(100);
 fs::File::open(path)?.read_to_string(&mut username)?;
 if username.is_empty() {
 return Err(ReadUsernameError::EmptyUsername(String::from(path)));
 }
 Ok(username)
}

fn main() {
 //fs::write("config.dat", "").unwrap();
 match read_username("config.dat") {
 Ok(username) => println!("Username: {}", username),
 Err(err) => println!("Error: {:?}", err),
 }
}

```

На этот слайд отводится примерно 5 минут.

- Макрос деривации `Error` предоставляетя `thiserror` и содержит множество полезных атрибутов для компактного определения типов ошибок.
- Сообщение из `#[error]` используется для реализации трейта `Display`.
- Обратите внимание, что макрос деривации `Error` (`thiserror:::`), хотя и реализует трейт `Error` (`std::error:::`), не является тем же самым; трейты и макросы не разделяют пространство имён.

## 30.7 anyhow

Крейт `anyhow` предоставляет расширенный тип ошибки с поддержкой дополнительной контекстной информации, которая может использоваться для семантического отслеживания действий программы, предшествующих ошибке.

Это можно комбинировать с удобными макросами из `thiserror`, чтобы избежать явного написания реализаций трейтов для пользовательских типов ошибок.

```

use anyhow::{Context, Result, bail};
use std::fs;
use std::io::Read;
use thiserror::Error;

#[derive(Clone, Debug, Eq, Error, PartialEq)] #[

error("Не найдено ИМЯ ПОЛЬЗОВАТЕЛЯ в {0}")]
struct EmptyUsernameError(String);

fn read_username(path: &str) -> Result<String> {
 let mut username = String::with_capacity(100);
 fs::File::open(path)
 .with_context(|| format!("Не удалось открыть {}", path))?
 .read_to_string(&mut username)
 .context("Не удалось прочитать ")?;

```

```

if username.is_empty() {
 bail!(EmptyUsernameError(path.to_string()));
}
Ok(username)
}

fn main() {
 //fs::write("config.dat", "").unwrap();
 match read_username("config.dat") {
 Ok(username) => println!("Username: {}", username),
 Err(err) => println!("Error: {}", err),
 }
}

```

На этот слайд отводится примерно 5 минут.

- anyhow: :Error фактически является обёрткой вокруг Box<dyn Error>. Таким образом, он обычно не является хорошим выбором для публичного API библиотеки, но широко используется в приложениях.
- anyhow: :Result<V> является псевдонимом типа для Result<V, anyhow: :Error>.
- Функциональность, предоставляемая anyhow: :Error, может быть знакома разработчикам на Go, поскольку она обеспечивает поведение, аналогичное типу error в Go, а Result<T, anyhow: :Error> очень похож на Go (T, error) (с условием, что только один элемент пары имеет смысл).
- anyhow: :Context — это трейд, реализованный для стандартных типов Result и Option. use anyhow: :Context необходим для активации методов .context() и .with\_context() для этих типов.

## Дополнительные материалы для изучения

- anyhow: :Error поддерживает downcasting, аналогично std: :any: :Any; конкретный тип ошибки, хранящийся внутри, может быть извлечён для анализа с помощью Error: :downcast.

## 30.8 Упражнение: переписывание с Result

В этом упражнении мы возвращаемся к упражнению по вычислению выражений, которое выполняли во второй день. Наше первоначальное решение игнорирует возможный случай ошибки: деление на ноль! Перепишите функцию eval так, чтобы она использовала идиоматическую обработку ошибок для обработки этого случая и возвращала ошибку при её возникновении. Мы предоставляем простой тип DivideByZeroError для использования в качестве типа ошибки в функции eval.

```

/// Операция, выполняемая над двумя подвыражениями.
#[derive(Debug)]
enum Operation {
 Add,
 Sub,
 Mul,
 Div,
}

```

```

/// Выражение в виде дерева.
#[derive(Debug)]
enum Expression {
 /// Операция над двумя подвыражениями.
 Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

 /// Литеральное значение
 Value(i64),
}

#[derive(PartialEq, Eq, Debug)]
struct DivideByZeroError;

// Исходная реализация вычислителя выражений. Обновите этот код, чтобы // возвращать `Result` и выдавать ошибку при делении на 0.
fn eval(e: Expression) -> i64 {
 match e {
 Expression::Op { op, left, right } => {
 let left = eval(*left);
 let right = eval(*right);
 match op {
 Operation::Add => left + right,
 Operation::Sub => left - right,
 Operation::Mul => left * right,
 Operation::Div => if right != 0 {
 left / right
 } else {
 panic!("Нельзя делить на ноль!");
 },
 }
 }
 Expression::Value(v) => v,
 }
}

#[cfg(test)]
mod test {
 use super::*;

 #[test]
 fn test_error() {
 assert_eq!(
 eval(Expression::Op {
 op: Operation::Div,
 left: Box::new(Expression::Value(99)),
 right: Box::new(Expression::Value(0)),
 }),
 Err(DivideByZeroError)
);
 }
}

```

```

#[test]
fn test_ok() {
 let expr = Expression::Op {
 op: Operation::Sub,
 left: Box::new(Expression::Value(20)),
 right: Box::new(Expression::Value(10)),
 };
 assert_eq!(eval(expr), Ok(10));
}

```

Этот слайд и его под-слайды должны занять около 20 минут.

- Начальный код здесь не полностью совпадает с решением предыдущего упражнения: мы добавили явный panic, чтобы показать студентам, где возникает ошибка. Обратите на это внимание, если у студентов возникнет путаница.

### 30.8.1 Решение

```

/// Операция, выполняемая над двумя подвыражениями.
#[derive(Debug)]
enum Operation {
 Add,
 Sub,
 Mul,
 Div,
}

/// Выражение в виде дерева.
#[derive(Debug)]
enum Expression {
 /// Операция над двумя подвыражениями.
 Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

 /// Литеральное значение
 Value(i64),
}

#[derive(PartialEq, Eq, Debug)]
struct DivideByZeroError;

fn eval(e: Expression) -> Result<i64, DivideByZeroError> {
 match e {
 Expression::Op { op, left, right } => {
 let left = eval(*left)?;
 let right = eval(*right)?;
 Ok(match op {
 Operation::Add => left + right,
 Operation::Sub => left - right,
 Operation::Mul => left * right,
 Operation::Div => {
 if right == 0 {
 DivideByZeroError
 } else {
 left / right
 }
 }
 })
 }
 }
}

```

```

 return Err(DivideByZeroError);
 } else {
 left / right
 }
}
)
}
Expression::Value(v) => Ok(v),
}
}

#[cfg(test)]
mod test {
 use super::*;

 #[test]
 fn test_error() {
 assert_eq!(
 eval(Expression::Op {
 op: Operation::Div,
 left: Box::new(Expression::Value(99)),
 right: Box::new(Expression::Value(0)),
 }),
 Err(DivideByZeroError)
);
 }

 #[test]
 fn test_ok() {
 let expr = Expression::Op {
 op: Operation::Sub,
 left: Box::new(Expression::Value(20)),
 right: Box::new(Expression::Value(10)),
 };
 assert_eq!(eval(expr), Ok(10));
 }
}
}

```

## Глава 31

# Небезопасный Rust

Этот раздел должен занять около 1 часа 15 минут. Он включает в себя:

| Слайд                                             | Продолжительность |
|---------------------------------------------------|-------------------|
| Небезопасный код                                  | 5 минут           |
| Дереференция необработанных указателей — 10 минут |                   |
| Изменяемые статические переменные                 | 5 минут           |
| Объединения                                       | 5 минут           |
| Небезопасные функции                              | 15 минут          |
| Небезопасные трейты                               | 5 минут           |
| Упражнение: обёртка FFI                           | 30 минут          |

### 31.1 Небезопасный Rust

Язык Rust состоит из двух частей:

- Безопасный **Rust**: безопасность памяти, невозможность неопределённого поведения.
- **Небезопасный Rust**: может вызвать неопределённое поведение при нарушении предусловий.

В этом курсе мы в основном рассматривали безопасный Rust, но важно понимать, что такое небезопасный Rust.

Небезопасный код обычно небольшой и изолирован, и его корректность должна быть тщательно документирована. Обычно он обёрнут в слой безопасной абстракции.

Небезопасный Rust предоставляет доступ к пяти новым возможностям:

- Разыменовывать необработанные указатели.
- Получать доступ к изменяемым статическим переменным или изменять их.
- Получать доступ к полям объединений.
- Вызывать небезопасные функции, включая `extern`-функции.
- Реализовывать небезопасные трейты.

Далее мы кратко рассмотрим возможности небезопасного **Rust**. Для получения полной информации, пожалуйста, обратитесь к главе 19.1 в Книге по **Rust** и **Rustonomicon**.

На этот слайд отводится примерно 5 минут.

Unsafe Rust не означает, что код является некорректным. Это означает, что разработчики отключили некоторые функции безопасности компилятора и должны самостоятельно писать корректный код. Это означает, что компилятор больше не применяет правила безопасности памяти Rust.

## 31.2 Разыменование сырых указателей

Создание указателей безопасно, но их разыменование требует unsafe:

```
fn main() {
 let mut x = 10;

 let p1: *mut i32 = &raw mut x;
 let p2 = p1 as *const i32;

 // БЕЗОПАСНОСТЬ: p1 и p2 были созданы путем получения сырых указателей на локальную
 // переменную , поэтому // гарантируется , что они не равны null , выровнены и указывают
 // на один (стековый) // выделенный объект .
 //
 // Объект , лежащий в основе сырых указателей , существует на протяжении всей функции , поэтому
 // он не dealloцируется , пока сырые указатели всё ещё существуют . К нему не об-
 //ращаются // через ссылки , пока существуют необработанные указатели , и к нему /
 // не обращаются из других потоков одновременно .
 unsafe {
 dbg! (*p1);
 *p1 = 6;
 // Мутация может корректно наблюдаться через необработанный указатель , как в C .
 dbg! (*p2);
 }

 // НЕБЕЗОПАСНО . НЕ ДЕЛАЙТЕ ТАК .
 /*
 let r: &i32 = unsafe { &*p1 };
 dbg!(r);
 x = 50;
 dbg!(r); // Объект , на который ссылается ссылка , был изменён . Это неопределенное поведение .
 */
}
```

На этот слайд должно уйти около 10 минут.

Хорошей практикой (и требованием руководства по стилю Rust для Android) является написание комментария для каждого unsafe блока, объясняющего, как код внутри него удовлетворяет требованиям безопасности выполняемых небезопасных операций.

В случае разыменования указателей это означает, что указатели должны быть валидными, то есть:

- Указатель должен быть ненулевым.
- Указатель должен быть *derefereable* (в пределах границ одного выделенного объекта).
- Объект не должен быть dealloцирован.
- Не должно быть одновременных обращений к одному и тому же адресу.
- Если указатель был получен приведением ссылки, базовый объект должен быть жив, и ни одна ссылка не может использоваться для доступа к памяти.

В большинстве случаев указатель также должен быть корректно выровнен.

Раздел «UNSAFE» приводит пример распространённой ошибки UB: наивное получение ссылки на разыменование сырого указателя обходит знания компилятора о том, на какой объект фактически указывает ссылка. Таким образом, borrow checker не блокирует `x`, и мы можем изменить его, несмотря на существование ссылки на него. Создание ссылки из указателя требует большой осторожности .

### 31.3 Изменяемые статические переменные

Безопасно читать неизменяемую статическую переменную:

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
 println!("HELLO_WORLD: {}", HELLO_WORLD);
}
```

Однако чтение и запись изменяемых статических переменных небезопасны, поскольку несколько потоков могут делать это одновременно без синхронизации, что приводит к состоянию гонки данных.

Безопасное использование изменяемых статических переменных требует рассуждений о конкурентности без поддержки компилятора:

```
static mut COUNTER: u32 = 0;

fn add_to_counter(inc: u32) {
 // БЕЗОПАСНОСТЬ: Нет других потоков, которые могут обращаться к `COUNTER`.
 unsafe {
 COUNTER += inc;
 }
}

fn main() {
 add_to_counter(42);

 // БЕЗОПАСНОСТЬ: Нет других потоков, которые могут обращаться к `COUNTER`.
 unsafe {
 dbg!(COUNTER);
 }
}
```

На этот слайд отводится примерно 5 минут.

- Данная программа корректна, поскольку она однопоточная. Однако компилятор Rust анализирует функции по отдельности и не может этого предполагать. Попробуйте убрать `unsafe` и посмотрите, как компилятор объясняет, что доступ к изменяемой статической переменной из нескольких потоков является неопределенным поведением.
- Редакция Rust 2024 года идет дальше и по умолчанию считает ошибкой доступ к изменяемой статической переменной по ссылке.
- Использование изменяемой статической переменной почти всегда является плохой практикой, вместо этого следует применять внутреннюю изменяемость.

- Существуют случаи, когда это может быть необходимо в низкоуровневом `no_std` коде, например, при реализации аллокатора кучи или работе с некоторыми C API. В таких случаях следует использовать указатели, а не ссылки.

## 31.4 Объединения

Объединения похожи на перечисления, но активное поле необходимо отслеживать самостоятельно:

```
#[repr(C)]
union MyUnion {
 i: u8,
 b: bool,
}

fn main() {
 let u = MyUnion { i: 42 };
 println!("int: {}", unsafe { u.i });
 println!("bool: {}", unsafe { u.b }); // Неопределённое поведение!
}
```

На этот слайд отводится примерно 5 минут.

Объединения крайне редко нужны в Rust, поскольку обычно можно использовать перечисления. Они иногда необходимы для взаимодействия с API библиотек на C.

Если вы хотите просто интерпретировать байты как другой тип, вероятно, вам подойдёт `std::mem::transmute` или безопасная оболочка, такая как `crate zerocopy`.

## 31.5 Небезопасные функции

Функция или метод могут быть помечены как `unsafe`, если для них существуют дополнительные предусловия, которые необходимо соблюдать, чтобы избежать неопределенного поведения.

Unsafe-функции могут происходить из двух источников:

- Функции Rust, объявленные как `unsafe`.
- Unsafe-внешние функции в блоках `extern "C"`.

На этот слайд и его под-слайды следует выделить примерно 15 минут.

Далее мы рассмотрим два типа unsafe-функций.

### 31.5.1 Небезопасные функции Rust

Вы можете пометить свои функции как `unsafe`, если они требуют определённых предусловий для предотвращения неопределенного поведения.

```
/// Меняет местами значения, на которые указывают заданные указатели.
///
/// # Безопасность
///
/// Указатели должны быть действительными, правильно выровненными и не использоваться иным образом
/// в течение вызова функции.
```

```

unsafe fn swap(a: *mut u8, b: *mut u8) {
 // БЕЗОПАСНОСТЬ: Нашзывающий гарантировал, что указатели действительны, пра-
 // вильно выровнены // и не имеют других обращений.
 unsafe {
 let temp = *a;
 *a = *b;
 *b = temp;
 }
}

fn main() {
 let mut a = 42;
 let mut b = 66;

 // БЕЗОПАСНОСТЬ: Указатели должны быть действительными, выровненными и уни-
 // кальными, поскольку они получены из ссылок.
 unsafe {
 swap(&mut a, &mut b);
 }

 println!("a = {}, b = {}", a, b);
}

```

На самом деле мы не использовали бы указатели для функции `swap`— это можно безопасно сделать с помощью ссылок.

Обратите внимание, что Rust 2021 и более ранние версии позволяют использовать небезопасный код внутри небезопасной функции без небезопасного блока. Это изменилось в издании 2024 года. Мы можем запретить это в более старых изданиях с помощью `#[deny(unsafe_op_in_unsafe_fn)]`. Попробуйте добавить это и посмотрите, что произойдет.

### 31.5.2 Небезопасные внешние функции

Вы можете объявлять внешние функции для доступа из Rust с помощью `unsafe extern`. Это небезопасно, потому что компилятор не может анализировать их поведение. Функции, объявленные в `extern` блоке, должны быть помечены как `safe` или `unsafe` в зависимости от того, есть ли у них предусловия для безопасного использования:

```

use std::ffi::c_char;

unsafe extern "C" {
 // `abs` не работает с указателями и не имеет требований по безопасности.
 safe fn abs(input: i32) -> i32;

 /// # Безопасность
 ///
 /// `s` должен быть указателем на корректную C-строку с завершающим нулём,
 /// которая не изменяется в течение вызова этой функции.
 unsafe fn strlen(s: *const c_char) -> usize;
}

fn main() {
 println!("Абсолютное значение -3 согласно C: {}", abs(-3));
}

```

```

unsafe {
 // БЕЗОПАСНОСТЬ: Мы передаём указатель на литерал С-строки, который действи-
 // телен // на протяжении всего времени работы программы.
 println!("Длина строки: {}", strlen(c"String".as_ptr()));
}
}

```

- Ранее Rust считал все `extern`-функции небезопасными, но это изменилось в Rust 1.82 с введением `unsafe extern` блоков.
- `abs` должна быть явно помечена как `safe`, поскольку это внешняя функция (FFI). Вызов внешних функций обычно представляет проблему только тогда, когда эти функции работают с указателями, что может нарушить модель памяти Rust, однако в целом любая C-функция может иметь неопределенное поведение при любых произвольных обстоятельствах.
- "C" в этом примере — это ABI; Также доступны другие ABI.
- Обратите внимание, что проверка соответствия сигнатуры функции Rust определению функции не выполняется — это ваша ответственность!

### 31.5.3 Вызов небезопасных функций

Несоблюдение требований безопасности приводит к нарушению безопасности памяти!

```

#[derive(Debug)]
#[repr(C)]
struct KeyPair { pk
 : [u16; 4], // 8 байт sk: [u
 16; 4], // 8 байт
}

const PK_BYTE_LEN: usize = 8;

fn log_public_key(pk_ptr: *const u16) {
 let pk: &[u16] = unsafe { std::slice::from_raw_parts(pk_ptr, PK_BYTE_LEN) };
 println!("{}{}", pk);
}

fn main() {
 let key_pair = KeyPair { pk: [1, 2, 3, 4], sk: [0, 0, 42, 0] };
 log_public_key(key_pair.pk.as_ptr());
}

```

Всегда включайте комментарий о безопасности для каждого `unsafe` блока. Он должен объяснять, почему код действительно безопасен. В этом примере отсутствует комментарий о безопасности, и он является ненадёжным.

Ключевые моменты:

- Второй аргумент функции `slice::from_raw_parts` — это количество элементов, а не байтов! Этот пример демонстрирует неожиданное поведение, считывая данные за пределами одного массива и в другой.
- Это неопределенное поведение, поскольку мы читаем за пределами массива, из которого был получен указатель.
- Функция `log_public_key` должна быть `unsafe`, поскольку `pk_ptr` должен удовлетворять определенным требованиям, чтобы избежать неопределенного поведения. Безопасная функция, которая может вызвать неопределенное поведение, называется ненадёжной. Что должно содержаться в её документации по безопасности?

- Стандартная библиотека содержит множество низкоуровневых unsafe-функций. Предпочитайте безопасные альтернативы, когда это возможно!
- Если вы используете unsafe-функцию в качестве оптимизации, обязательно добавьте бенчмарк для демонстрации прироста производительности.

## 31.6 Реализация небезопасных трейтов

Как и функции, трейты можно помечать как `unsafe`, если реализация должна гарантировать определённые условия для избежания неопределенного поведения.

Например, crate `zerocopy` содержит unsafe-трейт, который выглядит примерно так:

```
use std::mem, slice;

/// ...
/// # Безопасность
/// Тип должен иметь определённое представление и не содержать заполнения.
pub unsafe trait IntoBytes {
 fn as_bytes(&self) -> &[u8] {
 let len = mem::size_of_val(self);
 let slf: *const Self = self;
 unsafe { slice::from_raw_parts(slf.cast::<u8>(), len) }
 }
}

// БЕЗОПАСНОСТЬ: `u32` имеет определённое представление и не содержит заполнения.
unsafe impl IntoBytes for u32 {} Этот
слайд должен занять около 5 минут.
```

В Rustdoc для трейта должна быть секция `# Safety`, объясняющая требования для безопасной реализации трейта.

Фактическая секция безопасности для `IntoBytes` гораздо длиннее и сложнее.

Встроенные трейты `Send` и `Sync` являются небезопасными.

## 31.7 Безопасная оболочка FFI

Rust обладает отличной поддержкой вызова функций через интерфейс внешних функций (*foreign function interface*, FFI). Мы будем использовать это для создания безопасной оболочки для функций `libc`, которые применяются из C для чтения имён файлов в каталоге.

Рекомендуется обратиться к страницам руководства:

- `opendir(3)`
- `readdir(3)`
- `closedir(3)`

Также следует ознакомиться с модулем `std::ffi`. В нём вы найдёте несколько строковых типов, необходимых для выполнения упражнения:

| Типы                                       | Кодировка                            | Использование                          |
|--------------------------------------------|--------------------------------------|----------------------------------------|
| <code>str</code> и <code>String</code>     | UTF-8                                | Обработка текста в Rust                |
| <code>CStr</code> и <code>CString</code>   | NUL-терминированные                  | Взаимодействие с функциями С           |
| <code>OsStr</code> и <code>OsString</code> | Специфичные для операционной системы | Взаимодействие с операционной системой |

Вы будете выполнять преобразование между всеми этими типами:

- `&str` в `CString`: необходимо выделить память для завершающего символа `\0`,
- `CString` в `*const i8`: требуется указатель для вызова функций на C,
- `*const i8` в `&CStr`: нужен объект, способный определить завершающий символ `\0`,
- `&CStr` в `&[u8]`: срез байтов является универсальным интерфейсом для «некоторых неизвестных данных»,
- `&[u8]` в `&OsStr`: &OsStr является промежуточным шагом к `OsString`, используйте `OsStrExt` для его создания,
- `&OsStr` в `OsString`: необходимо клонировать данные из `&OsStr`, чтобы иметь возможность вернуть их и снова вызвать `readdir`.

В Nomicon также есть очень полезная глава, посвящённая FFI.

Скопируйте приведённый ниже код на <https://play.rust-lang.org/> и заполните отсутствующие функции и методы:

```
// TODO: удалить это после завершения реализации.
#![allow(unused_imports, unused_variables, dead_code)]
```

```
mod ffi {
 use std::os::raw::{c_char, c_int};
 #[cfg(not(target_os = "macos"))]
 use std::os::raw::{c_long, c_uchar, c_ulong, c_ushort};

 // Непрозрачный тип. См. https://doc.rust-lang.org/nomicon/ffi.html.
 #[repr(C)]
 pub struct DIR {
 _data: [u8; 0],
 _marker: core::marker::PhantomData<(* mut u8, core::marker::PhantomPinned)>,
 }

 // Макет согласно странице руководства Linux для readdir(3), где ino_t и off_t
 // определяются согласно определениям в /usr/include/x86_64-
 // linux-gnu/{sys/types.h, bits/typesizes.h}.
 #[cfg(not(target_os = "macos"))]
 #[repr(C)]
 pub struct dirent {
 pub d_ino: c_ulong,
 pub d_off: c_long,
 pub d_reclen: c_ushort,
 pub d_type: c_uchar,
 pub d_name: [c_char; 256],
 }

 // Структура согласно man-странице macOS для dir(5).
 #[cfg(all(target_os = "macos"))]
 #[repr(C)]
 pub struct dirent {
```

```

 pub d_FILENO: u64,
 pub d_seekoff: u64,
 pub d_reclen: u16,
 pub d_namlen: u16,
 pub d_type: u8,
 pub d_name: [c_char; 1024],
}

unsafe extern "C" {
 pub unsafe fn opendir(s: * const c_char) -> *mut DIR;

#[cfg(not(all(target_os = "macos", target_arch = "x86_64")))]
pub unsafe fn readdir(s: * mut DIR) -> *const dirent;

// См. https://github.com/rust-lang/libc/issues/414 и раздел о // _DARWIN_
FEATURE_64_BIT_INODE в man-странице macOS для stat(2).
//
// «Платформы, существовавшие до появления этих обновлений», относятся // к
macOS (в отличие от iOS, watchOS и др.) на Intel и PowerPC. #[cfg(all(target_os
= "macos", target_arch = "x86_64"))] #[link_name = "readdir$INODE64"]
pub unsafe fn readdir(s: * mut DIR) -> *const dirent;

 pub unsafe fn closedir(s: * mut DIR) -> c_int;
}
}

use std::ffi::{CString, OsString, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
 path: CString,
 dir: * mut ffi::DIR,
}

impl DirectoryIterator {
 fn new(path: &str) -> Result<DirectoryIterator, String> { //
 // Вызов opendir и возврат значения Ok, если вызов успешен, //
 // иначе возврат Err с сообщением.
 todo!()
 }
}

impl Iterator for DirectoryIterator {
 type Item = OsString;
 fn next(&mut self) -> Option<OsString> {
 // Продолжаем вызывать readdir, пока не получим NULL-указатель.
 todo!()
 }
}

```

```

impl Drop for DirectoryIterator {
 fn drop(&mut self) {
 // Вызов closedir по необходимости.
 todo!()
 }
}

fn main() -> Result<(), String> {
 let iter = DirectoryIterator::new(".")?;
 println!("files: {:?}", iter.collect::<Vec<_>>());
 Ok(())
}

```

### 31.7.1 Решение

```

mod ffi {
 use std::os::raw::{c_char, c_int};
#[cfg(not(target_os = "macos"))]
 use std::os::raw::{c_long, c_uchar, c_ulong, c_ushort};

 // Непрозрачный тип. См. https://doc.rust-lang.org/nomicon/ffi.html.
#[repr(C)]
 pub struct DIR {
 _data: [u8; 0],
 _marker: core::marker::PhantomData<(* mut u8, core::marker::PhantomPinned)>,
 }

 // Макет согласно странице руководства Linux для readdir(3), где ino_t и off_t
 // определяются согласно определениям в /usr/include/x86_64-
 // linux-gnu/{sys/types.h, bits/typesizes.h}.
#[cfg(not(target_os = "macos"))]
#[repr(C)]
 pub struct dirent {
 pub d_ino: c_ulong,
 pub d_off: c_long,
 pub d_reclen: c_ushort,
 pub d_type: c_uchar,
 pub d_name: [c_char; 256],
 }

 // Структура согласно man-странице macOS для dir(5).
#[cfg(all(target_os = "macos"))]
#[repr(C)]
 pub struct dirent {
 pub d_fileno: u64,
 pub d_seekoff: u64,
 pub d_reclen: u16,
 pub d_namlen: u16,
 pub d_type: u8,
 pub d_name: [c_char; 1024],
 }
}

```

```

}

unsafe extern "C" {
 pub unsafe fn opendir(s: * const c_char) -> *mut DIR;
#[cfg(not(all(target_os = "macos", target_arch = "x86_64")))]
 pub unsafe fn readdir(s: * mut DIR) -> *const dirent;

 // См. https://github.com/rust-lang/libc/issues/414 и раздел о // _DARWIN_
 // FEATURE_64_BIT_INODE в ман-странице macOS для stat(2).
 //
 // «Платформы, существовавшие до появления этих обновлений», относятся // к
 // macOS (в отличие от iOS, wearOS и др.) на Intel и PowerPC. #[cfg(all(target_os
 // = "macos", target_arch = "x86_64"))] #[link_name = "readdir$INODE64"]
 pub unsafe fn readdir(s: * mut DIR) -> *const dirent;

 pub unsafe fn closedir(s: * mut DIR) -> c_int;
}

use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
 path: CString,
 dir: * mut ffi::DIR,
}

impl DirectoryIterator {
 fn new(path: &str) -> Result<DirectoryIterator, String> { // Вызов opendir и возврат значения Ok, если вызов успешен, //
 // иначе возврат Err с сообщением.
 let path =
 CString::new(path).map_err(|err| format!("Invalid path: {}", err))?;
 // БЕЗОПАСНОСТЬ: path.as_ptr() не может быть NULL.
 let dir = unsafe { ffi::opendir(path.as_ptr()) };
 if dir.is_null() {
 Err(format!("Не удалось открыть {}", path))
 } else {
 Ok(DirectoryIterator { path, dir })
 }
 }
}

impl Iterator for DirectoryIterator {
 type Item = OsString;
 fn next(&mut self) -> Option<OsString> {
 // Продолжаем вызывать readdir, пока не получим NULL-указатель.
 // БЕЗОПАСНОСТЬ: self.dir никогда не равен NULL.
 }
}

```

```

let dirent = unsafe { ffi::readdir(self.dir) };
if dirent.is_null() {
 // Достигнут конец каталога.
 return None;
}
// БЕЗОПАСНОСТЬ: dirent не равен NULL, а dirent.d_name завершается NUL-символом.
let d_name = unsafe { CStr::from_ptr((*dirent).d_name.as_ptr()) };
let os_str = OsStr::from_bytes(d_name.to_bytes());
Some(os_str.to_owned())
}

impl Drop for DirectoryIterator {
fn drop(&mut self) {
 // Вызов closedir по необходимости.
 // БЕЗОПАСНОСТЬ: self.dir никогда не равен NULL.
 if unsafe { ffi::closedir(self.dir) } != 0 { panic!
 ("Не удалось закрыть {:?}", self.path);
 }
}
}

fn main() -> Result<(), String> {
 let iter = DirectoryIterator::new(".")?;
 println!("files: {:?}", iter.collect::<Vec<_>>());
 Ok(())
}

#[cfg(test)]
mod tests {
 use super::*;
 use std::error::Error;

 #[test]
 fn test_nonexisting_directory() {
 let iter = DirectoryIterator::new("no-such-directory");
 assert!(iter.is_err());
 }

 #[test]
 fn test_empty_directory() -> Result<(), Box

```

```
#[test]
fn test_nonempty_directory() -> Result<(), Box<dyn Error>> {
 let tmp = tempfile::TempDir::new()?;
 std::fs::write(tmp.path().join("foo.txt"), "The Foo Diaries\n")?;
 std::fs::write(tmp.path().join("bar.png"), "<PNG>\n")?;
 std::fs::write(tmp.path().join("crab.rs"), "///! Crab\n")?;
 let iter = DirectoryIterator::new(
 tmp.path().to_str().ok_or("В пути обнаружен символ, не являющийся UTF-8")?,
)?;
 let mut entries = iter.collect::<Vec<_>>();
 entries.sort();
 assert_eq!(entries, &[".", "..", "bar.png", "crab.rs", "foo.txt"]);
 Ok(())
}
```

**Часть IX**

**Android**

## Глава 32

# Добро пожаловать в Rust на Android

Rust поддерживается для системного программного обеспечения на Android. Это означает, что вы можете писать новые сервисы, библиотеки, драйверы или даже прошивки на Rust (или улучшать существующий код по мере необходимости).

Докладчик может упомянуть любое из следующего в связи с возросшим использованием Rust в Android:

- Пример сервиса: DNS через HTTP.
- Библиотеки: Rutabaga Virtual Graphics Interface.
- Драйверы ядра: Binder.
- Прошивка: прошивка pKVM.

## Глава 33

# Настройка

Мы будем использовать виртуальное устройство Android Cuttlefish для тестирования нашего кода. Убедитесь, что у вас есть доступ к такому устройству или создайте новое с помощью:

```
source build/envsetup.sh
lunch aosp_cf_x86_64_phone-trunk_staging-userdebug
acloud create
```

Пожалуйста, ознакомьтесь с Android Developer Codelab для получения подробной информации.

Код на следующих страницах находится в каталоге [src/android/ учебных материалов](#). Пожалуйста, выполните `git clone` репозиторий для практического изучения.

Ключевые моменты:

- Cuttlefish — эталонное Android-устройство, предназначенное для работы на универсальных Linux-десктопах. Поддержка MacOS также планируется.
- Системный образ Cuttlefish сохраняет высокую точность соответствия реальным устройствам и является идеальным эмулятором для запуска многих сценариев использования Rust.

## Глава 34

# Правила сборки

Система сборки Android (Soong) поддерживает Rust через ряд модулей:

| Module Type                  | Description                                                                                                                      |
|------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <code>rust_binary</code>     | Создаёт исполняемый файл Rust.                                                                                                   |
| <code>rust_library</code>    | Создаёт библиотеку Rust и предоставляет варианты <code>rlib</code> и <code>dylib</code> .                                        |
| <code>rust_ffi</code>        | Создаёт библиотеку Rust C, используемую модулями <code>cc</code> , и предоставляет как статические, так и динамические варианты. |
| <code>rust_proc_macro</code> | Создаёт библиотеку Rust типа <code>proc-macro</code> . Они аналогичны плагинам компилятора.                                      |
| <code>rust_test</code>       | Создаёт тестовый исполняемый файл Rust, использующий стандартный тестовый каркас Rust.                                           |
| <code>rust_fuzz</code>       | Создаёт Rust fuzz бинарный файл с использованием <code>libfuzzer</code> .                                                        |
| <code>rust_protobuf</code>   | Генерирует исходный код и создаёт Rust-библиотеку, предоставляющую интерфейс для конкретного <code>protobuf</code> .             |
| <code>rust_bindgen</code>    | Генерирует исходный код и создаёт Rust-библиотеку, содержащую Rust-биндинги к С-библиотекам.                                     |

Далее мы рассмотрим `rust_binary` и `rust_library`.

Дополнительные пункты, которые может упомянуть докладчик:

- Cargo не оптимизирован для мультиязыковых репозиториев и также загружает пакеты из интернета.
- Для соответствия требованиям и производительности Android должен иметь `crates` внутри дерева . Он также должен взаимодействовать с кодом на C/C++/Java. Soong заполняет этот пробел.
- Soong имеет много сходств с Bazel, который является открытой версией Blaze (используемой в google3).
- Интересный факт: персонаж Data из Star Trek — андроид типа Soong.

### 34.1 Rust бинарные файлы

Давайте начнём с простого приложения. В корне AOSP-репозитория создайте следующие файлы:

```

hello_rust/Android.bp:
rust_binary {
 name: "hello_rust",
 crate_name: "hello_rust",
 srcs: ["src/main.rs"],
}

hello_rust/src/main.rs:
//! Демонстрация Rust.

/// Выводит приветствие в стандартный вывод.
fn main() {
 println!("Hello from Rust!");
}

```

Теперь вы можете собрать, загрузить и запустить исполняемый файл:

```

m hello_rust
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust" /data/local/tmp
adb shell /data/local/tmp/hello_rust
Hello from Rust!

```

- Пройдитесь по этапам сборки и продемонстрируйте их выполнение в вашем эмуляторе.
- Обратите внимание на обширные комментарии документации? Правила сборки Android требуют, чтобы все модули имели документацию. Попробуйте удалить её и посмотрите, какую ошибку вы получите.
- Подчеркните, что правила сборки Rust похожи на другие правила Soong. Это сделано специально, чтобы использование Rust было таким же простым, как C++ или Java.

## 34.2 Rust библиотеки

Вы используете `rust_library` для создания новой библиотеки Rust для Android.

Здесь мы объявляем зависимость от двух библиотек:

- `libgreeting`, которую мы определяем ниже,
- `libtextwrap`, которая является `crate`, уже включённым в `external/rust/android-crates-io/crates/`.

*hello\_rust/Android.bp:*

```

rust_binary {
 name: "hello_rust_with_dep",
 crate_name: "hello_rust_with_dep",
 srcs: ["src/main.rs"],
 rustlibs: [
 "libgreetings",
 "libtextwrap",
],
 prefer_rlib: true, // Необходимо для предотвращения ошибки динамической линковки.
}

```

```

rust_library {
 name: "libgreetings",
 crate_name: "greetings",
 srcs: ["src/lib.rs"],
}

hello_rust/src/main.rs:
//! Демонстрация Rust.

use greetings::greeting;
use textwrap::fill;

/// Выводит приветствие в стандартный вывод.
fn main() {
 println!("{}", fill(&greeting("Bob"), 24));
}

hello_rust/src/lib.rs:
//! Библиотека приветствий.

/// Приветствует `name`.
pub fn greeting(name: &str) -> String {
 format!("Hello {} , it is very nice to meet you!")
}

```

Вы собираете, загружаете и запускаете бинарный файл так же, как и ранее:

```

$ hello_rust_with_dep
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_with_dep" /data/local/tmp
adb shell /data/local/tmp/hello_rust_with_dep

```

Hello Bob, it is very  
nice to meet you!

- Пройдитесь по этапам сборки и продемонстрируйте их выполнение в вашем эмуляторе.
- Rust crate с именем `greetings` должен быть собран правилом под названием `libgreetings`. Обратите внимание, как Rust-код использует имя `crate`, что является нормой в Rust.
- Правила сборки требуют добавления комментариев документации ко всем публичным элементам.

# Глава 35

## AIDL

Язык определения интерфейсов Android (AIDL) поддерживается в Rust:

- Rust-код может вызывать существующие AIDL-серверы,
- Вы можете создавать новые AIDL-серверы на Rust.
- AIDL обеспечивает взаимодействие Android-приложений друг с другом.
- Поскольку Rust поддерживается как полноценный язык в этой экосистеме, Rust-сервисы могут вызываться любым другим процессом на телефоне.

### 35.1 Учебник по сервису дня рождения

Чтобы продемонстрировать использование Rust с Binder, мы пройдем процесс создания интерфейса Binder. Затем мы реализуем описанный сервис и напишем клиентский код, который взаимодействует с этим сервисом.

#### 35.1.1 Интерфейсы AIDL

Вы объявляете API вашего сервиса с помощью интерфейса AIDL:

*birthday\_service/aidl/com/example/birthdayservice/IBirthdayService.aidl*:

```
package com.example.birthdayservice;

/** Интерфейс сервиса дней рождения. */
interface IBirthdayService {
 /** Генерирует сообщение с поздравлением с Днём Рождения. */
 String wishHappyBirthday(String name, int years);
}

birthday_service/aidl/Android.bp:

aidl_interface {
 name: "com.example.birthdayservice",
 srcs: ["com/example/birthdayservice/*.aidl"],
 unstable: true,
 backend: {
```

```

 rust: { // Rust по умолчанию не включён
 enabled: true,
 },
 },
}

```

- Обратите внимание, что структура каталогов в директории aidl/ должна соответствовать имени пакета, используемому в AIDL-файле, то есть пакет com.example.birthdayservice, а файл расположен по пути aidl/com/example/IBirthdayService.aidl.

### 35.1.2 Сгенерированный API сервиса

Binder генерирует трейты для каждого определения интерфейса.

*birthday\_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:*

```

/** Интерфейс сервиса дней рождения. */
interface IBirthdayService {
 /** Генерирует сообщение с поздравлением с Днём Рождения. */
 String wishHappyBirthday(String name, int years);
}

out/soong/.intermediates/.../com_example_birthdayservice.rs:

trait IBirthdayService {
 fn wishHappyBirthday(&self, name: &str, years: i32) -> binder::Result<String>;
}

```

Ваш сервис должен реализовать этот trait, а клиент будет использовать этот trait для взаимодействия с сервисом.

- Укажите, как сгенерированная сигнатура функции, в частности типы аргументов и возвращаемого значения, соответствует определению интерфейса.
  - String в качестве аргумента соответствует другому типу Rust, чем String в качестве возвращаемого значения.

### 35.1.3 Реализация сервиса

Теперь мы можем реализовать AIDL-сервис:

*birthday\_service/src/lib.rs:*

```

//! Реализация интерфейса AIDL `IBirthdayService`.
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService;
use com_example_birthdayservice::binder;

/// Реализация `IBirthdayService`.
pub struct BirthdayService;

impl binder::Interface for BirthdayService {}

impl IBirthdayService for BirthdayService {
 fn wishHappyBirthday(&self, name: &str, years: i32) -> binder::Result<String> {
 Ok(format!("Happy Birthday {}!, congratulations with the {} years!"))
 }
}

```

```

 }
 }

birthday_service/Android.bp:

rust_library {
 name: "libbirthdayservice",
 srcs: ["src/lib.rs"],
 crate_name: "birthdayservice",
 rustlibs: [
 "com.example.birthdayservice-rust",
 "libbinder_rs",
],
}

```

- Укажите путь к сгенерированному IBirthdayService трейту и объясните, почему каждый из сегментов необходим.
- Обратите внимание, что `wishHappyBirthday` и другие методы AIDL IPC принимают `&self` (вместо `&mut self`).
  - Это необходимо, поскольку binder обрабатывает входящие запросы в пуле потоков, что позволяет выполнять несколько запросов параллельно. Это требует, чтобы методы сервиса получали только разделяемую ссылку на `self`.
  - Любое состояние, которое необходимо изменять в сервисе, должно быть помещено в структуру вроде Mutex для обеспечения безопасной мутации.
  - Правильный подход к управлению состоянием сервиса во многом зависит от особенностей вашего сервиса.
- TODO: Что делает `trайдbinder::Interface`? Есть ли методы для переопределения? Где исходный код?

### 35.1.4 Сервер AIDL

Наконец, мы можем создать сервер, который предоставляет сервис:

*birthday\_service/src/server.rs:*

```

//! Сервис дня рождения.
use birthdayservice::BirthdayService;
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService;
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Точка входа для сервиса дня рождения.
fn main() {
 let birthday_service = BirthdayService;
 let birthday_service_binder = BnBirthdayService::new_binder(
 birthday_service,
 binder::BinderFeatures::default(),
);
 binder::add_service(SERVICE_IDENTIFIER, birthday_service_binder.as_binder());
 expect("Не удалось зарегистрировать сервис");
 binder::ProcessState::join_thread_pool();
}

```

*birthday\_service/Android.bp:*

```
rust_binary {
 name: "birthday_server",
 crate_name: "birthday_server",
 srcs: ["src/server.rs"],
 rustlibs: [
 "com.example.birthdayservice-rust",
 "libbinder_rs",
 "libbirthdayservice",
],
 prefer_rlib: true, // Чтобы избежать ошибки динамической линковки.
}
```

Процесс запуска пользовательской реализации сервиса (в данном случае типа `BirthdayService`, который реализует интерфейс `IBirthdayService`) в качестве Binder-сервиса включает несколько этапов и может показаться более сложным, чем студенты привыкли, если они использовали Binder на C++ или другом языке. Объясните студентам, почему каждый шаг необходим.

1. Создайте экземпляр вашего типа сервиса (`BirthdayService`).
2. Обёрните объект сервиса в соответствующий тип `Bn*`(в данном случае `BnBirthdayService`). Этот тип генерируется Binder и предоставляет общую функциональность Binder, которую в C++ обеспечивал бы базовый класс `BnBinder`. В Rust отсутствует наследование, поэтому вместо этого используется композиция: наш `BirthdayService` включается внутрь сгенерированного `BnBinderService`.
3. Вызовите `add_service`, передав ему идентификатор сервиса и объект вашего сервиса (в примере — объект `BnBirthdayService`).
4. Вызовите `join_thread_pool`, чтобы добавить текущий поток в пул потоков Binder и начать прослушивание подключений.

### 35.1.5 Развёртывание

Теперь мы можем собрать, отправить и запустить сервис:

```
m birthday_server
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_server" /data/local/tmp
adb root
adb shell /data/local/tmp/birthday_server
```

В другом терминале проверьте, что сервис запущен:

```
adb shell service check birthdayservice
```

Сервис birthdayservice: найден

Вы также можете вызвать сервис с помощью `service call`:

```
adb shell service call birthdayservice 1 s16 Bob i32 24
```

Результат: `Parcel`

|                                                 |                    |
|-------------------------------------------------|--------------------|
| 0x00000000: 00000000 00000036 00610048 00700070 | '....6...Н.а.р.р.' |
| 0x00000010: 00200079 00690042 00740072 00640068 | 'у. .В.и.р.т.х.д.' |
| 0x00000020: 00790061 00420020 0062006f 0020002c | 'а.у..Б.о.б.,..'   |
| 0x00000030: 006f0063 0067006e 00610072 00750074 | 'с.о.н.г.р.а.т.у.' |
| 0x00000040: 0061006c 00690074 006e006f 00200073 | 'л.а.т.и.о.н.с..'  |
| 0x00000050: 00690077 00680074 00740020 00650068 | 'w.i.t.h. .t.h.e.' |

```
0x00000060: 00320020 00200034 00650079 00720061 ' .2.4..y.e.a.r.'
0x00000070: 00210073 00000000 's.!.....')
```

### 35.1.6 Клиент AIDL

Наконец, мы можем создать Rust-клиент для нашего нового сервиса.

*birthday\_service/src/client.rs:*

```
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService;
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Вызов сервиса дня рождения.
fn main() -> Result<(), Box< dyn Error>> {
 let name = std::env::args().nth(1).unwrap_or_else(|| String::from("Bob"));
 let years = std::env::args()
 .nth(2)
 .and_then(|arg| arg.parse::<i32>().ok())
 .unwrap_or(42);

 binder::ProcessState::start_thread_pool();
 let service = binder::get_interface::<dyn IBirthdayService>(SERVICE_IDENTIFIER).
 map_err(|_| "Не удалось подключиться к BirthdayService");

 // Вызов сервиса.
 let msg = service.wishHappyBirthday(&name, years)?;
 println!("{}{}", msg);
}
```

*birthday\_service/Android.bp:*

```
rust_binary {
 name: "birthday_client",
 crate_name: "birthday_client",
 srcs: ["src/client.rs"],
 rustlibs: [
 "com.example.birthdayservice-rust",
 "libbinder_rs",
],
 prefer_rlib: true, // Чтобы избежать ошибки динамическойリンクовки.
}
```

Обратите внимание, что клиент не зависит от `libbirthdayservice`.

Соберите, загрузите и запустите клиент на вашем устройстве:

```
m birthday_client
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_client" /data/local/tmp
adb shell /data/local/tmp/birthday_client Charlie 60 C
```

днём рождения, Charlie, поздравляем с 60-летием!

- Strong<dyn IBirthayService> является объектом трейта, представляющим сервис, к которому подключился клиент.
  - Strong — это пользовательский умный указатель для Binder. Он управляет как внутренним подсчётом ссылок в процессе для объекта трейта сервиса, так и глобальным подсчётом ссылок Binder, который отслеживает, сколько процессов имеют ссылку на объект.
  - Обратите внимание, что объект трейта, который клиент использует для взаимодействия с сервисом, использует точно такой же трейт, который реализует сервер. Для данного интерфейса Binder генерируется один Rust-трейт, который используют и клиент, и сервер.
- Используйте тот же идентификатор сервиса, который применялся при регистрации сервиса. Идеально, если это будет определено в общем crate, от которого могут зависеть и клиент, и сервер.

### 35.1.7 Изменение API

Давайте расширим API дополнительным функционалом: мы хотим позволить клиентам указывать список строк для поздравительной открытки:

```
package com.example.birthdayservice;

/** Интерфейс сервиса дней рождения. */
interface IBirthayService {
 /** Генерирует сообщение с поздравлением с Днём Рождения. */
 String wishHappyBirthday(String name, int years, in String[] text);
}
```

Это приводит к обновлённому определению трейта для IBirthayService:

```
trait IBirthayService {
 fn wishHappyBirthday(
 &self,
 name: &str,
 years: i32,
 text: &[String],
) -> binder::Result<String>;
}
```

- Обратите внимание, что String[] в определении AIDL переводится как &[String] в Rust, то есть в сгенерированных биндингах используются идиоматичные типы Rust, где это возможно:
  - входные array аргументы переводятся в срезы.
  - выходные out и inout аргументы переводятся в &mut Vec<T> .
  - Возвращаемые значения переводятся в возвращаемый Vec<T>.

### 35.1.8 Обновление клиента и сервиса

Обновите клиентский и серверный код с учётом нового API.

*birthday\_service/src/lib.rs:*

```
impl IBirthayService for BirthdayService {
 fn wishHappyBirthday(
 &self,
 name: &str,
 years: i32,
 text: &[String],
```

```

) -> binder::Result<String> {
 let mut msg = format!(
 "С днём рождения, {name}, поздравляем с {years} годами!",
);
 for line in text {
 msg.push('\n');
 msg.push_str(line);
 }
 Ok(msg)
}
}

birthday_service/src/client.rs:

let msg = service.wishHappyBirthday(
 &name,
 years,
 &[
 String::from("Счастливого дня рождения тебе"),
 String::from("И также: ещё много всего"),
],
)?;

```

• TODO: Перенести фрагменты кода в файлы проекта, где они действительно будут собираться?

## 35.2 Работа с типами AIDL

Типы AIDL преобразуются в соответствующие идиоматичные типы Rust:

- Примитивные типы (в основном) отображаются на идиоматичные типы Rust.
- Поддерживаются типы коллекций, такие как срезы, Vec и строковые типы.
- Ссылки на объекты AIDL и дескрипторы файлов могут передаваться между клиентами и сервисами.
- Дескрипторы файлов и parcelable полностью поддерживаются.

### 35.2.1 Примитивные типы

Примитивные типы отображаются (в основном) идиоматично:

| Тип AIDL                 | Тип Rust | Примечание                                        |
|--------------------------|----------|---------------------------------------------------|
| логический               | bool     |                                                   |
| байт                     | i8       | Обратите внимание, что байты являются знаковыми.  |
| char                     | u16      | Обратите внимание на использование u16, а не u32. |
| целое                    | i32      |                                                   |
| длинное                  | i64      |                                                   |
| число с плавающей точкой | f32      |                                                   |
| число двойной точности   | f64      |                                                   |
| Строка                   | String   |                                                   |

### 35.2.2 Типы массивов

Типы массивов ( `T[]` , `byte[]` и `List<T>` ) преобразуются в соответствующий тип массива Rust в зависимости от того, как они используются в сигнатуре функции:

| Позиция                                     | Тип Rust                             |
|---------------------------------------------|--------------------------------------|
| входной аргумент                            | <code>&amp;[T]</code> out / inout    |
| аргумент <code>&amp;mut Vec&lt;T&gt;</code> | Возврат<br><code>Vec&lt;T&gt;</code> |

- В Android 13 и выше поддерживаются массивы фиксированного размера, то есть `T[N]` превращается в `[T; N]`. Массивы фиксированного размера могут иметь несколько измерений (например, `int[3][4]`). В Java-бэкенде массивы фиксированного размера представлены как типы массивов.
- Массивы в полях `Parcelable` всегда преобразуются в `Vec<T>`.

### 35.2.3 Отправка объектов

Объекты AIDL могут передаваться либо как конкретный тип AIDL, либо как типизированный интерфейс `IBinder`:

*birthday\_service/aidl/com/example/birthdayservice/IBirthdayInfoProvider.aidl:*

```
package com.example.birthdayservice;
```

```
interface IBirthdayInfoProvider {
 String name();
 int years();
}
```

*birthday\_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:*

```
import com.example.birthdayservice.IBirthdayInfoProvider;
```

```
interface IBirthdayService {
 /** То же самое, но с использованием объекта binder. */
 String wishWithProvider(IBirthdayInfoProvider provider);
```

```
 /** То же самое, но с использованием `IBinder`. */
 String wishWithErasedProvider(IBinder provider);
}
```

*birthday\_service/src/client.rs:*

```
// Rust-структура, реализующая интерфейс `IBirthdayInfoProvider`.
```

```
struct InfoProvider {
 name: String,
 age: u8,
}
```

```
impl binder::Interface for InfoProvider {}
```

```
impl IBirthdayInfoProvider for InfoProvider {
 fn name(&self) -> binder::Result<String> {
```

```

 Ok(self.name.clone())
 }

 fn years(&self) -> binder::Result<i32> {
 Ok(self.age as i32)
 }
}

fn main() {
 binder::ProcessState::start_thread_pool();
 let service = connect().expect("Не удалось подключиться к BirthdayService");

 // Создайте объект binder для интерфейса `IBirthdayInfoProvider`.
 let provider = BnBirthdayInfoProvider::new_binder(
 InfoProvider { name: name.clone(), age: years as u8 },
 BinderFeatures::default(),
);

 // Отправьте объект binder в сервис.
 service.wishWithProvider(&provider)?;

 // Выполните ту же операцию, передавая provider как `SpIBinder`.
 service.wishWithErasedProvider(&provider.as_binder())?;
}

```

- Обратите внимание на использование BnBirthdayInfoProvider. Это выполняет ту же функцию, что и BnBirthdayService, который мы рассматривали ранее.

### 35.2.4 Parcelable

Binder для Rust поддерживает отправку parcelable напрямую:

*birthday\_service/aidl/com/example/birthdayservice/BirthdayInfo.aidl*:

```
package com.example.birthdayservice;
```

```
parcelable BirthdayInfo {
 String name;
 int years;
}
```

*birthday\_service/aidl/com/example/birthdayservice/IBirthdayService.aidl*:

```
import com.example.birthdayservice.BirthdayInfo;
```

```
interface IBirthdayService {
 /* То же самое, но с parcelable. */ String
 wishWithInfo(in BirthdayInfo info);
}
```

*birthday\_service/src/client.rs*:

```
fn main() {
 binder::ProcessState::start_thread_pool();
```

```

let service = connect().expect("Не удалось подключиться к BirthdayService");

let info = BirthdayInfo { name: "Alice".into(), years: 123 };
service.wishWithInfo(&info)?;
}

```

### 35.2.5 Отправка файлов

Файлы могут передаваться между клиентами и серверами Binder с использованием типа `ParcelFileDescriptor`:

*birthday\_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:*

```

interface IBirthdayService {
 /** То же самое, но загружает информацию из файла.*/
 String wishFromFile(in ParcelFileDescriptor infoFile);
}

```

*birthday\_service/src/client.rs:*

```

fn main() {
 binder::ProcessState::start_thread_pool();
 let service = connect().expect("Не удалось подключиться к BirthdayService");

 // Открыть файл и записать в него информацию о дне рождения.
 let mut file = File::create("/data/local/tmp/birthday.info").unwrap();
 writeln!(file, "{name}")?;
 writeln!(file, "{years}")?;

 // Создать `ParcelFileDescriptor` из файла и отправить его.
 let file = ParcelFileDescriptor::new(file);
 service.wishFromFile(&file)?;
}

```

*birthday\_service/src/lib.rs:*

```

impl IBirthdayService for BirthdayService {
 fn wishFromFile(
 &self,
 info_file: &ParcelFileDescriptor,
) -> binder::Result<String> {
 // Преобразовать дескриптор файла в `File`. `ParcelFileDescriptor` оборачивает
 // `OwnedFd`, который можно клонировать и затем использовать для создания объ-
 // екта `File`.
 let mut info_file = info_file
 .as_ref()
 .try_clone()
 .map(File::from)
 .expect("Недопустимый дескриптор файла");

 let mut contents = String::new();
 info_file.read_to_string(&mut contents).unwrap();

 let mut lines = contents.lines();
 let name = lines.next().unwrap();
 }
}

```

```
let years: i32 = lines.next().unwrap().parse().unwrap();

Ok(format!("Happy Birthday {name}, congratulations with the {years} years!"))
}
```

- `ParcelFileDescriptor` обворачивает `OwnedFd`, поэтому может быть создан из `File` (или любого другого типа, который обворачивает `OwnedFd`), и может использоваться для создания нового `File` дескриптора на другой стороне.
- Другие типы файловых дескрипторов могут быть обернуты и переданы, например, TCP, UDP и UNIX-сокеты.

## Глава 36

# Тестирование в Android

Основываясь на тестировании, рассмотрим, как работают модульные тесты в AOSP. Используйте модуль `rust_test` для ваших модульных тестов:

*testing/Android.bp*:

```
rust_library {
 name: "libletpad",
 crate_name: "letpad",
 srcs: ["src/lib.rs"],
}

rust_test {
 name: "libletpad_test",
 crate_name: "letpad_test",
 srcs: ["src/lib.rs"],
 host_supported: true,
 test_suites: ["general-tests"],
}

rust_test {
 name: "libgoogletest_example",
 crate_name: "googletest_example",
 srcs: ["googletest.rs"],
 rustlibs: ["libgoogletest_rust"],
 host_supported: true,
}

rust_test {
 name: "libmockall_example",
 crate_name: "mockall_example",
 srcs: ["mockall.rs"],
 rustlibs: ["libmockall"],
 host_supported: true,
}

testing/src/lib.rs:
```

```
//! Библиотека для добавления отступов слева.

/// Добавляет отступ слева к `s` до ширины `width`.
pub fn letpad(s: &str, width: usize) -> String {
 format!("{}{:width$}")
}

#[cfg(test)]
mod tests {
 use super::*;

 #[test]
 fn short_string() {
 assert_eq!(letpad("foo", 5), " foo");
 }

 #[test]
 fn long_string() {
 assert_eq!(letpad("foobar", 6), "foobar");
 }
}
```

Теперь вы можете запустить тест с помощью `atest --host libletpad_test`

Вывод выглядит следующим образом:

```
INFO: Затраченное время: 2.666s, Критический путь: 2.40s
INFO: 3 процесса: 2 внутренних, 1 linux-sandbox.
INFO: Сборка успешно завершена, всего 3 действия
//comprehensive-rust-android/testing:libletpad_test_host ПРОЙДЕНО за 2.3s
 ПРОЙДЕНО libletpad_test.tests::long_string (0.0s)
 ПРОЙДЕНО libletpad_test.tests::short_string (0.0s)
Тесты завершены: 2 успешных, 0 неудачных из 2
```

Обратите внимание, что указывается только корень библиотеки `crate`. Тесты обнаруживаются рекурсивно во вложенных модулях.

## 36.1 GoogleTest

Crate `GoogleTest` позволяет использовать гибкие утверждения тестов с помощью *matchers*:

```
use googletest::prelude::*;

#[googletest::test]
fn test_elements_are() {
 let value = vec!["foo", "bar", "baz"];
 expect_that!(value, elements_are!(eq(&"foo"), lt(&"xyz"), starts_with("b")));
}
```

Если мы изменим последний элемент на "!", тест завершится с ошибкой с подробным сообщением, указывающим на источник ошибки:

```
---- test_elements_are_stdout ----
Значение: value
Ожидается: содержит элементы:
 0. равно "foo"
 1. меньше "хуз"
 2. начинается с префикса "!" Фактическое
значение: ["foo", "бар", "baz"],
где элемент №2 — "baz", который не начинается с "!" в src/
testing/googletest.rs:6:5 Ошибка: см.
вывод ошибки выше. Этот слайд рассчитан при-
мерно на 5 минут.
```

- GoogleTest не входит в состав Rust Playground, поэтому данный пример необходимо запускать в локальной среде. Используйте cargo add googletest для быстрого добавления его в существующий проект Cargo.
- Стока use googletest::prelude::\*; импортирует ряд часто используемых макросов и типов.
- Это лишь поверхностный обзор, существует множество встроенных матчеров. Рекомендуется пройти первый раздел курса «Продвинутое тестирование приложений на Rust», самостоятельного курса по Rust: он предлагает пошаговое введение в библиотеку с упражнениями, которые помогут вам освоить googletest макросы, его матчера и общую концепцию.
- Особенно удобной функцией является отображение различий в многострочных строках в виде diff:

```
#[test]
fn test_multiline_string_diff() {
 let haiku = "Memory safety found,\n\
 Rust's strong typing guides the way,\n\
 Secure code you'll write.";
 assert_that!(

 haiku,

 eq("Memory safety found,\n\
 Rust's silly humor guides the way,\n\
 Secure code you'll write.")
);
}
```

показывает цветовую дифференциацию (цвета здесь не отображаются):

Значение: haiku  
Ожидается: равно "Memory safety found,\nRust's silly humor guides the way,\nSecure  
Фактическое: "Memory safety found,\nRust's strong typing guides the way,\nSecure code you'll  
которое не равно "Memory safety found,\nRust's silly humor guides the way,\nSecure  
Разница (-фактическое / +ожидаемое):  
Обнаружена безопасность памяти,  
-Rust's strong typing guides the way,  
+Rust's silly humor guides the way,  
Вы напишете безопасный код.  
в src/testing/googletest.rs:17:5

- Крейт является портом GoogleTest для C++ на Rust.

## 36.2 Мокирование

Для мокирования широко используется библиотека Mockall. Вам необходимо рефакторить код с использованием трейтов, которые затем можно быстро мокировать:

используйте `std::time::Duration`;

```
#[mockall::automock]
публичный трейт Pet {
 fn is_hungry(&self, since_last_meal: Duration) -> bool;
}

#[test]
fn test_robot_dog() {
 let mut mock_dog = MockPet::new();
 mock_dog.expect_is_hungry().return_const(true);
 assert!(mock_dog.is_hungry(Duration::from_secs(10)));
}
```

На этот слайд отводится примерно 5 минут.

- Mockall — рекомендуемая библиотека для мокирования в Android (AOSP). Существуют и другие библиотеки для мокирования, доступные на [crates.io](#), в частности в области мокирования HTTP-сервисов. Другие библиотеки для мокирования работают аналогично Mockall, то есть упрощают получение мок-реализации заданного трейта.
- Обратите внимание, что мокирование является несколько спорным: моки позволяют полностью изолировать тест от его зависимостей. Непосредственным результатом этого является более быстрое и стабильное выполнение тестов. С другой стороны, моки могут быть неправильно настроены и возвращать результаты, отличающиеся от тех, что дали бы реальные зависимости.

Если это возможно, рекомендуется использовать реальные зависимости. В качестве примера многие базы данных позволяют настроить backend в памяти. Это означает, что вы получаете корректное поведение в ваших тестах, при этом они быстрые и автоматически очищают за собой ресурсы.

Аналогично, многие веб-фреймворки позволяют запустить сервер в процессе, который привязывается к случайному порту на `localhost`. Всегда отдавайте предпочтение этому, а не имитации работы фреймворка, поскольку это помогает тестировать ваш код в реальной среде.

- Mockall не входит в состав Rust Playground, поэтому данный пример необходимо запускать в локальной среде. Используйте `cargo add mockall` для быстрого добавления Mockall в существующий проект Cargo.
- Mockall обладает значительно большим функционалом. В частности, вы можете настроить ожидания, зависящие от переданных аргументов. Здесь мы используем это, чтобы замокать кота, который становится голодным через 3 часа после последнего кормления:

```
#[test]
fn test_robot_cat() {
 let mut mock_cat = MockPet::new();
 mock_cat
 .expect_is_hungry()
 .with(mockall::predicate::gt(Duration::from_secs(3 * 3600)))
 .return_const(true);
```

```
mock_cat.expect_is_hungry().return_const(false);
assert!(mock_cat.is_hungry(Duration::from_secs(5 * 3600)));
assert!(!mock_cat.is_hungry(Duration::from_secs(5)));
}
```

- Вы можете использовать `.times(n)` для ограничения количества вызовов метода `mock` до `n`  
--- `mock` автоматически вызовет `panic` при уничтожении, если это условие не выполнено.

## Глава 37

# Логирование

Рекомендуется использовать crate log для автоматического логирования в logcat(на устройстве) или в stdout (на хосте):

```
hello_rust_logs/Android.bp:
rust_binary {
 name: "hello_rust_logs",
 crate_name: "hello_rust_logs",
 srcs: ["src/main.rs"],
 rustlibs: [
 "liblog_rust",
 "liblogger",
],
 host_supported: true,
}

hello_rust_logs/src/main.rs:
//! Демонстрация логирования на Rust.

use log::{debug, error, info};

/// Логирует приветствие.
fn main() {
 logger::init(
 logger::Config::default()
 .with_tag_on_device("rust")
 .with_max_level(log::LevelFilter::Trace),
);
 debug!("Запуск программы.");
 info!("Всё идёт хорошо .");
 error!("Произошла ошибка !");
}
```

Соберите, загрузите и запустите бинарный файл на вашем устройстве:

```
m hello_rust_logs
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_logs" /data/local/tmp
```

```
adb shell /data/local/tmp/hello_rust_logs
```

Логи отображаются в adb logcat :

```
adb logcat -s rust
```

```
09-08 08:38:32.454 2420 2420 D rust: hello_rust_logs: Запуск программы.
```

```
09-08 08:38:32.454 2420 2420 I rust: hello_rust_logs: Всё идёт хорошо.
```

```
09-08 08:38:32.454 2420 2420 E rust: hello_rust_logs: Произошла ошибка!
```

- Реализация логгера в `liblogger` необходима только в конечном бинарном файле; если вы ведёте логирование из библиотеки, вам нужен только фасадный `crate log`.

## Глава 38

# Взаимодействие

Rust обладает отличной поддержкой взаимодействия с другими языками. Это означает, что вы можете:

- Вызывать функции Rust из других языков.
- Вызывать функции, написанные на других языках, из Rust.

Когда вы вызываете функции на иностранном языке, говорят, что вы используете *foreign function interface* (внешний интерфейс функций), также известный как FFI.

- Это ключевая возможность Rust: скомпилированный код становится неотличимым от скомпилиированного кода на C или C++.
- Технически говорят, что Rust может быть скомпилирован с тем же ABI (application binary interface), что и код на C.

### 38.1 Взаимодействие с C

Rust полностью поддерживает связывание объектных файлов с соглашением вызова C. Аналогично, вы можете экспорттировать функции Rust и вызывать их из C.

Вы можете сделать это вручную, если хотите:

```
unsafe extern "C" {
 safe fn abs(x: i32) -> i32;
}

fn main() {
 let x = -42;
 let abs_x = abs(x);
 println!("{} , {}", x, abs_x);
}
```

Мы уже рассматривали это в упражнении Safe FFI Wrapper.

Это предполагает полное знание целевой платформы. Не рекомендуется для использования в производстве.

Далее мы рассмотрим более подходящие варианты.

- Часть "C" в блоке `extern` **указывает** Rust, что `abs` можно вызвать с использованием C ABI (application binary interface).
- Объявление `safe fn abs` сообщает Rust, что `abs` является безопасной функцией. По умолчанию `extern`-функции считаются небезопасными, но поскольку `abs(x)` корректна для любого `x`, мы можем объявить её безопасной.

### 38.1.1 Простая библиотека на C

Сначала создадим небольшую C-библиотеку:

*interoperability/bindgen/libbirthday.h:*

```
typedef struct card {
 const char* name;
 int years;
} card;

void print_card(const card* card);
```

*interoperability/bindgen/libbirthday.c:*

```
#include <stdio.h>
#include "libbirthday.h"

void print_card(const card* card) {
 printf("-----\n");
 printf("| С днём рождения, %s!\n", card->name);
 printf("| Поздравляем с %i годами!\n", card->years);
 printf("-----\n");
}
```

Добавьте это в файл `Android.bp`:

*interoperability/bindgen/Android.bp:*

```
cc_library {
 name: "libbirthday",
 srcs: ["libbirthday.c"],
}
```

### 38.1.2 Использование Bindgen

Инструмент `bindgen` может автоматически сгенерировать биндинги из заголовочного файла C.

Создайте обёрточный заголовочный файл для библиотеки (необязательно в данном примере):

*interoperability/bindgen/libbirthday\_wrapper.h:*

```
#include "libbirthday.h"
```

*interoperability/bindgen/Android.bp:*

```
rust_bindgen {
 name: "libbirthday_bindgen",
 crate_name: "birthday_bindgen",
 wrapper_src: "libbirthday_wrapper.h",
```

```
 source_stem: "bindings",
 static_libs: ["libbirthday"],
 }
```

Наконец, мы можем использовать биндинги в нашей программе на Rust:

*interoperability/bindgen/Android.bp*:

```
rust_binary {
 name: "print_birthday_card",
 srcs: ["main.rs"],
 rustlibs: ["libbirthday_bindgen"],
 static_libs: ["libbirthday"],
}
```

*interoperability/bindgen/main.rs*:

```
//! Демонстрация Bindgen.
```

```
use birthday_bindgen::{card, print_card};
```

```
fn main() {
 let name = std::ffi::CString::new("Peter").unwrap();
 let card = card { name: name.as_ptr(), years: 42 };
 // БЕЗОПАСНОСТЬ: Указатель, который мы передаём, действителен, поскольку
 он получен из ссылки Rust, а поле `name` содержит ссылку на `name` выше, который
 также остаётся действительным. `print_card` не сохраняет ни один из указателей
 для последующего использования после возврата.
 unsafe {
 print_card(&card as *const card);
 }
}
```

- Правила сборки Android автоматически вызовут bindgen за вас в фоновом режиме.
- Обратите внимание, что код Rust в `main` всё ещё сложно писать. Хорошей практикой является инкапсуляция вывода bindgen в библиотеку Rust, которая предоставляет безопасный интерфейс вызывающему коду.

### 38.1.3 Запуск нашего бинарного файла

Соберите, загрузите и запустите бинарный файл на вашем устройстве:

```
m print_birthday_card
adb push "$ANDROID_PRODUCT_OUT/system/bin/print_birthday_card" /data/local/tmp
adb shell /data/local/tmp/print_birthday_card. Наконец, мы можем запустить автоматически сгенерированные тесты, чтобы убедиться, что биндинги работают:
```

*interoperability/bindgen/Android.bp*:

```
rust_test {
 name: "libbirthday_bindgen_test",
 srcs: [":libbirthday_bindgen"],
 crate_name: "libbirthday_bindgen_test",
 test_suites: ["general-tests"],
 auto_gen_config: true,
```

```
 clippy_lints: "none", // Сгенерированный файл, пропустить
 // проверку lints: "none",
}
atest libbirthday_bindgen_test
```

### 38.1.4 Простая библиотека на Rust

Экспорт функций и типов Rust в C — это просто. Вот простая библиотека на Rust:

*interoperability/rust/libanalyze/analyze.rs*

```
///! Демонстрация FFI в Rust.
#![deny(improper_ctypes_definitions)]

use std::os::raw::c_int;

/// Анализ чисел.
// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
pub extern "C" fn analyze_numbers(x: c_int, y: c_int) {
 if x < y {
 println!("x ({}) является наименьшим!", x);
 } else {
 println!("y ({}), вероятно, больше, чем x ({})", y, x);
 }
}
```

*interoperability/rust/libanalyze/Android.bp*

```
rust_ffi {
 name: "libanalyze_ffi",
 crate_name: "analyze_ffi",
 srcs: ["analyze.rs"],
 include_dirs: ["."],
}
```

`#[unsafe(no_mangle)]` отключает стандартное манглирование имён в Rust, поэтому экспортируемый символ будет просто именем функции. Вы также можете использовать `#[unsafe(export_name = "some_name")]` для указания любого желаемого имени.

### 38.1.5 Вызов Rust

Теперь мы можем вызвать эту функцию из C-биарника:

*interoperability/rust/libanalyze/analyze.h*

```
#ifndef ANALYZE_H
#define ANALYZE_H

void analyze_numbers(int x, int y);

#endif
```

*interoperability/rust/analyze/main.c*

```

#include "analyze.h"

int main() {
 analyze_numbers(10, 20);
 analyze_numbers(123, 123);
 return 0;
}

```

*interoperability/rust/analyze/Android.bp*

```

cc_binary {
 name: "analyze_numbers",
 srcs: ["main.c"],
 static_libs: ["libanalyze_ffi"],
}

```

Соберите, загрузите и запустите бинарный файл на вашем устройстве:

```

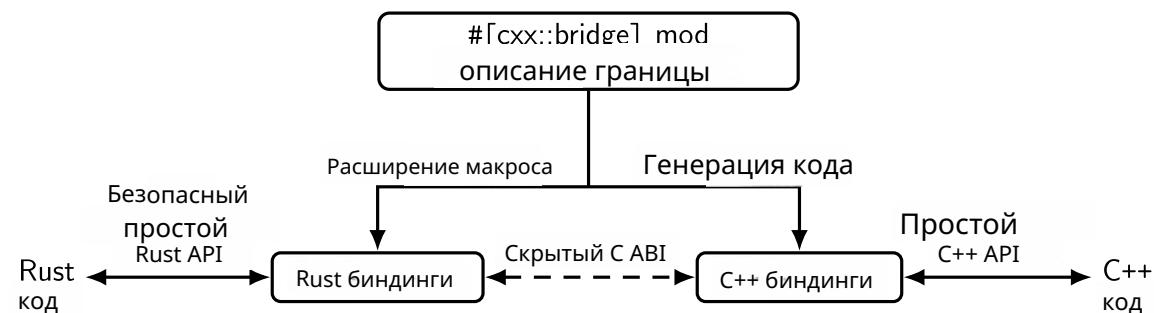
m analyze_numbers
adb push "$ANDROID_PRODUCT_OUT/system/bin/analyze_numbers" /data/local/tmp
adb shell /data/local/tmp/analyze_numbers

```

## 38.2 C C++

Крейт CXX обеспечивает безопасную и безопасную интероперабельность между Rust и C++.

Общий подход представлен следующим образом:



### 38.2.1 Модуль-мост

CXX основывается на описании сигнатур функций, которые будут доступны из одного языка для другого. Вы предоставляете это описание, используя `extern`-блоки в Rust-модуле, аннотированном макросом атрибута `#\[cxx::bridge]`.

```

#[allow(unsafe_op_in_unsafe_fn)]
#[cxx::bridge(namespace = "org::blobstore")]
mod ffi {
 // Общие структуры с полями, доступными для обоих языков.
 struct BlobMetadata {
 size: usize,
 tags: Vec<String>,
 }
}

```

```

// Типы и сигнатуры Rust, доступные для C++.
extern "Rust" {
 тип MultiBuf;

 fn next_chunk(buf: &mut MultiBuf) -> &[u8];
}

// Типы и сигнатуры C++, доступные для Rust.
unsafe extern "C++" {
 include!("include/blobstore.h");

 тип BlobstoreClient;

 fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
 fn put(self: Pin<&mut BlobstoreClient>, parts: &mut MultiBuf) -> u64;
 fn tag(self: Pin<&mut BlobstoreClient>, blobid: u64, tag: &str);
 fn metadata(&self, blobid: u64) -> BlobMetadata;
}
}

```

- Мост обычно объявляется в модуле `ffib` внутри вашего crate.
- На основе объявлений, сделанных в модуле моста, CXX генерирует соответствующие определения типов и функций для Rust и C++, чтобы предоставить доступ к этим элементам в обоих языках.
- Для просмотра генерированного кода Rust используйте `cargo-expand`, чтобы увидеть расширенный `proc macro`. В большинстве примеров используется команда `cargo expand ::ffi` для расширения только модуля `ffi` (хотя это не применяется к проектам Android).
- Для просмотра генерированного кода C++ откройте каталог `target/cxxbridge`.

### 38.2.2 Объявления Rust Bridge

```

#[cxx::bridge]
mod ffi {
 extern "Rust" {
 type MyType; // Непрозрачный тип
 fn foo(&self); // Метод для `MyType`
 fn bar() -> Box<MyType>; // Свободная функция
 }
}

struct MyType(i32);

impl MyType {
 fn foo(&self) {
 println!("{}", self.0);
 }
}

fn bar() -> Box<MyType> {
 Box::new(MyType(123))
}

```

- Элементы, объявленные в `extern "Rust"`, ссылаются на элементы, находящиеся в области видимости родительского модуля.
- Генератор кода CXX использует ваши `extern "Rust"` секции для создания заголовочного файла C++, содержащего соответствующие объявления C++. Сгенерированный заголовочный файл имеет тот же путь, что и исходный файл Rust, содержащий мост, за исключением расширения `.rs.h`.

### 38.2.3 Сгенерированный C++

```
#[cxx::bridge]
mod ffi {
 // Типы и сигнатуры Rust, доступные для C++.
 extern "Rust" {
 тип MultiBuf;

 fn next_chunk(buf: &mut MultiBuf) -> &[u8];
 }
}
```

В результате получается (примерно) следующий код на C++:

```
struct MultiBuf final : public ::rust::Opaque {
 ~MultiBuf() = delete;

private:
 friend ::rust::layout;
 struct layout {
 static ::std::size_t size() noexcept;
 static ::std::size_t align() noexcept;
 };
};

::rust::Slice<::std::uint8_t const> next_chunk(::org::blobstore::MultiBuf &buf) noexcept
```

### 38.2.4 Объявления моста C++

```
#[cxx::bridge]
mod ffi {
 // Типы и сигнатуры C++, доступные для Rust.
 unsafe extern "C++" {
 include!("include/blobstore.h");

 тип BlobstoreClient;

 fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
 fn put(self: Pin<& mut BlobstoreClient>, parts: & mut MultiBuf) -> u64;
 fn tag(self: Pin<& mut BlobstoreClient>, blobid: u64, tag: &str);
 fn metadata(&self, blobid: u64) -> BlobMetadata;
 }
}
```

В результате получается (примерно) следующий код на Rust:

```

#[repr(C)]
pub struct BlobstoreClient {
 _private: ::cxx::private::Opaque,
}

pub fn new_blobstore_client() -> ::cxx::UniquePtr<BlobstoreClient> {
 extern "C" {
 #[link_name = "org$blobstore$cxxbridge1$new_blobstore_client"]
 fn __new_blobstore_client() -> *mut BlobstoreClient;
 }
 unsafe { ::cxx::UniquePtr::from_raw(__new_blobstore_client()) }
}

impl BlobstoreClient {
 pub fn put(&self, parts: & mut MultiBuf) -> u64 {
 extern "C" {
 #[link_name = "org$blobstore$cxxbridge1$BlobstoreClient$put"]
 fn __put(
 _: &BlobstoreClient,
 parts: * mut ::cxx::core::ffi::c_void,
) -> u64;
 }
 unsafe {
 __put(self, parts as *mut MultiBuf as *mut ::cxx::core::ffi::c_void)
 }
 }
}

// ...

```

- Программисту не требуется гарантировать точность подписей, которые он указал. CXX выполняет статические проверки, чтобы убедиться, что подписи точно соответствуют объявленным в C++.
- unsafe extern блоки позволяют объявлять функции C++, которые безопасно вызывать из Rust.

### 38.2.5 Общие типы

```

#[cxx::bridge]
mod ffi {
 #[derive(Clone, Debug, Hash)]
 struct PlayingCard {
 масть: Suit,
 значение: u8, // A=1, J=11, Q=12, K=13
 }

 enum Suit {
 Трефы,
 Бубны,
 Червы,
 Пики,
 }
}

```

```
 }
}
```

- Поддерживаются только перечисления, подобные C (unit).
- Ограничено число трейтов поддерживается для #[derive()] на общих типах. Соответствующая функциональность также генерируется для кода C++, например, если вы используете derive Hash, то также создаётся реализация std::hash для соответствующего типа C++.

### 38.2.6 Общие перечисления

```
#[cxx::bridge]
mod ffi {
 enum Suit {
 Трефы,
 Бубны,
 Черви,
 Пики,
 }
}
```

Сгенерировано Rust:

```
#[derive(Copy, Clone, PartialEq, Eq)]
#[repr(transparent)]
pub struct Suit {
 pub repr: u8,
}

#[allow(non_upper_case_globals)]
impl Suit {
 pub const Clubs: Self = Suit { repr: 0 };
 pub const Diamonds: Self = Suit { repr: 1 };
 pub const Hearts: Self = Suit { repr: 2 };
 pub const Spades: Self = Suit { repr: 3 };
}
```

Сгенерировано C++:

```
enum class Suit : uint8_t {
 Clubs = 0,
 Diamonds = 1,
 Hearts = 2,
 Spades = 3,
};
```

- На стороне Rust сгенерированный код для общих перечислений фактически представляет собой структуру, инкапсулирующую числовое значение. Это связано с тем, что в C++ не является неопределенным поведением (UB), если enum class содержит значение, отличное от всех перечисленных вариантов, и наше представление в Rust должно иметь такое же поведение.

### 38.2.7 Обработка ошибок в Rust

```
#[cxx::bridge]
mod ffi {
 extern "Rust" {
 fn fallible(depth: usize) -> Result<String>;
 }
}

fn fallible(depth: usize) -> anyhow::Result<String> {
 if depth == 0 {
 return Err(anyhow::Error::msg("fallible1 требует, чтобы depth > 0"));
 }

 Ok("Success!".into())
}
```

- Функции Rust, возвращающие `Result`, на стороне C++ транслируются в исключения.
- Брошенное исключение всегда будет типа `rust::Error`, который в первую очередь предоставляет способ получить строку сообщения об ошибке. Сообщение об ошибке будет получено из реализации `Display` для типа ошибки.
- Паника с разворачиванием стека (unwinding) из Rust в C++ всегда приводит к немедленному завершению процесса.

### 38.2.8 Обработка ошибок в C++

```
#[cxx::bridge]
mod ffi {
 unsafe extern "C++" {
 include!("example/include/example.h");
 fn fallible(depth: usize) -> Result<String>;
 }
}

fn main() {
 if let Err(err) = ffi::fallible(99) { eprintln!(
 !"Ошибка: {}", err);
 process::exit(1);
 }
}
```

- Функции C++, объявленные с возвращаемым типом `Result`, перехватывают любые исключения, выброшенные на стороне C++, и возвращают их как значение `Err` вызывающей функции Rust.
- Если исключение выбрасывается из внешней функции "C++", которая не объявлена через мост CXX с возвращаемым типом `Result`, программа вызывает `std::terminate` в C++. Поведение эквивалентно выбрасыванию того же исключения через функцию C++ с квалификатором `noexcept`.

### 38.2.9 Дополнительные типы

| Тип Rust      | Тип C++            |
|---------------|--------------------|
| String        | rust::String       |
| &str          | rust::Str          |
| CxxString     | std::string        |
| &[T]/&mut [T] | rust::Slice        |
| Box<T>        | rust::Box<T>       |
| UniquePtr<T>  | std::unique_ptr<T> |
| Vec<T>        | rust::Vec<T>       |
| CxxVector<T>  | std::vector<T>     |

- Эти типы могут использоваться в полях общих структур, а также в аргументах и возвращаемых значениях внешних функций.
- Обратите внимание, что Rust-тип String не соответствует напрямую std::string . Существует несколько причин для этого:
  - std::string не поддерживает инвариант UTF-8, который требует String.
  - Два типа имеют разное расположение в памяти, поэтому их нельзя напрямую передавать между языками.
  - std::string требует конструкторы перемещения, не соответствующие семантике перемещения Rust, поэтому std::string нельзя передавать по значению в Rust.

### 38.2.10 Сборка для Android

Создайте два genrule: один для генерации заголовочного файла CXX, другой — для генерации исходного файла CXX. Затем они используются в качестве входных данных для cc\_library\_static.

```
// Генерация заголовочного файла C++, содержащего привязки
// к экспортимым функциям Rust в lib.rs.
genrule {
 name: "libcxx_test_bridge_header",
 tools: ["cxxbridge"],
 cmd: "$(location cxxbridge) $(in) --header > $(out)",
 srcs: ["lib.rs"],
 out: ["lib.rs.h"],
}

// Генерация C++ кода, в который вызывает Rust .
genrule {
 name: "libcxx_test_bridge_code",
 tools: ["cxxbridge"],
 cmd: "$(location cxxbridge) $(in) > $(out)",
 srcs: ["lib.rs"],
 out: ["lib.rs.cc"],
}
```

- Инструмент cxxbridge является автономным средством, которое генерирует C++ часть модуля-моста. Он включён в состав Android и доступен как инструмент Soong.
- По соглашению, если ваш исходный файл Rust называется lib.rs, то заголовочный файл будет называться lib.rs.h, а исходный файл — lib.rs.cc . Однако это соглашение об именовании не является обязательным.

### 38.2.11 Сборка в Android

Создайте `cc_library_static` для сборки C++ библиотеки, включая сгенерированные CXX заголовочный и исходный файлы.

```
cc_library_static {
 name: "libcxx_test_cpp",
 srcs: ["cxx_test.cpp"],
 generated_headers: [
 "cxx-bridge-header",
 "libcxx_test_bridge_header"
],
 generated_sources: ["libcxx_test_bridge_code"],
}
```

- Укажите, что `libcxx_test_bridge_header` и `libcxx_test_bridge_code` являются зависимостями для C++ биндингов, сгенерированных CXX. Мы покажем, как это настраивается, на следующем слайде.
- Обратите внимание, что также необходимо зависеть от библиотеки `cxx-bridge-header` для подключения общих определений CXX.
- Полная документация по использованию CXX в Android доступна в документации Android. Возможно, стоит поделиться этой ссылкой с классом, чтобы студенты знали, где они смогут снова найти эти инструкции в будущем.

### 38.2.12 Сборка в Android

Создайте `rust_binary`, который зависит от `libcxx` вашей `cc_library_static`.

```
rust_binary {
 name: "cxx_test",
 srcs: ["lib.rs"],
 rustlibs: ["libcxx"],
 static_libs: ["libcxx_test_cpp"],
}
```

## 38.3 Взаимодействие с Java

Java может загружать разделяемые объекты через Java Native Interface (JNI). Крейт `jni` позволяет создать совместимую библиотеку.

Сначала создадим функцию на Rust для экспорта в Java:

`interoperability/java/src/lib.rs`:

```
//! Демонстрация FFI Rust <-> Java.

use jni::JNIEnv;
use jni::objects::{JClass, JString};
use jni::sys::jstring;

/// Реализация метода HelloWorld::hello.
// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
```

```
pub extern "system" fn Java_HelloWorld_hello(
 mut env: JNIEnv,
 _class: JClass,
 name: JString,
) -> jstring {
 let input: String = env.get_string(&name).unwrap().into();
 let greeting = format!("Hello, {}!".format(input));
 let output = env.new_string(greeting).unwrap();
 output.into_raw()
}
```

*interoperability/java/Android.bp:*

```
rust_ffi_shared {
 name: "libhello_jni",
 crate_name: "hello_jni",
 srcs: ["src/lib.rs"],
 rustlibs: ["libjni"],
}
```

Далее вызываем эту функцию из Java:

*interoperability/java/HelloWorld.java:*

```
class HelloWorld {
 private static native String hello(String name);

 static {
 System.loadLibrary("hello_jni");
 }

 public static void main(String[] args) {
 String output = HelloWorld.hello("Alice");
 System.out.println(output);
 }
}
```

*interoperability/java/Android.bp:*

```
java_binary {
 name: "helloworld_jni",
 srcs: ["HelloWorld.java"],
 main_class: "HelloWorld",
 jni_libs: ["libhello_jni"],
}
```

Наконец, вы можете собрать, синхронизировать и запустить бинарный файл:

```
m helloworld_jni
adb sync # требует adb root && adb remount
adb shell /system/bin/helloworld_jni
```

- Атрибут `unsafe(no_mangle)` указывает Rust сгенерировать символ `Java_HelloWorld_hello` точно в том виде, в каком он написан. Это важно, чтобы Java могла распознать символ как метод `hello` класса `HelloWorld`.

- По умолчанию Rust изменяет (*mangle*) имена символов, чтобы бинарный файл мог содержать две версии одного и того же Rust-крайта.

# **Часть X**

# **Chromium**

## Глава 39

# Добро пожаловать в Rust в Chromium

Rust поддерживается для сторонних библиотек в Chromium, с кодом первой стороны для интеграции между Rust и существующим C++ кодом Chromium.

Сегодня мы вызовем Rust, чтобы выполнить простую операцию со строками. Если в вашем коде есть участок, где вы отображаете UTF8-строку пользователю, вы можете использовать этот рецепт в своей части кодовой базы вместо той, о которой мы говорим.

## Глава 40

# Настройка

Убедитесь, что вы можете собрать и запустить Chromium. Любая платформа и набор флагов сборки подходят, при условии, что ваш код относительно свежий (позиция коммита 1223636 и выше, соответствующая ноябрю 2023 года):

```
gn gen out/Debug
autoninja -C out/Debug chrome
out/Debug/chrome # или на Mac: out/Debug/Chromium.app/Contents/MacOS/Chromium
```

(Рекомендуется сборка компонента в режиме отладки для максимально быстрой итерации. Это значение по умолчанию!)

Изучите, как собрать Chromium, если вы ещё не достигли этого этапа. Обратите внимание: настройка окружения для сборки Chromium занимает значительное время.

Также рекомендуется установить Visual Studio Code.

# Об упражнениях

В этой части курса представлена серия упражнений, которые последовательно развивают навыки. Мы будем выполнять их на протяжении всего курса, а не только в конце. Если у вас не будет времени завершить определённый раздел, не беспокойтесь — вы сможете наверстать упущенное в следующем занятии.

## Глава 41

# Сравнение экосистем Chromium и Cargo

Сообщество Rust обычно использует `cargo` и библиотеки с `crates.io`. Chromium собирается с помощью `gn` и `ninja` и набора проверенных зависимостей.

При написании кода на Rust у вас есть следующие варианты:

- Использовать `gn` и `ninja` с помощью шаблонов из `//build/rust/*.gni` (например, `rust_static_library`, с которым мы познакомимся позже). Это использует проверенный инструментарий и пакеты Chromium.
- Использовать `cargo`, но ограничиться проверенным инструментарием и пакетами Chromium.
- Используйте `cargo`, доверяя инструментарию и/или пакетам, загруженным из интернета.

Далее мы сосредоточимся на `gn` и `ninja`, поскольку именно так код Rust может быть собран в браузер Chromium. В то же время Cargo является важной частью экосистемы Rust, и его следует иметь в своём арсенале.

### Мини-упражнение

Разделитесь на небольшие группы и:

- Обсудите сценарии, в которых `cargo` может предоставить преимущество, и оцените профиль рисков этих сценариев.
- Обсудите, каким инструментам, библиотекам и группам людей необходимо доверять при использовании `gn` и `ninja`, автономного `cargo` и т.д.

Попросите студентов не заглядывать в заметки лектора до завершения упражнения. Предполагая, что участники курса находятся вместе, попросите их обсудить в небольших группах по 3–4 человека.

Заметки/подсказки, относящиеся к первой части упражнения («сценарии, в которых Cargo может предоставить преимущество»):

- Прекрасно, что при написании инструмента или прототипирования части Chromium есть доступ к богатой экосистеме библиотек с `crates.io`. Существует `crate` практически для всего, и ими обычно очень приятно пользоваться. (`clap` для разбора командной строки, `serde` для

серIALIZации/десериализации в/из различных форматов, `itertools` для работы с итераторами и т.д.).

- `cargo` облегчает использование библиотеки (просто добавьте одну строку в `Cargo.toml` и начните писать код).
- Стоит сравнить, как CPAN помог сделать `perl` популярным выбором. Или сравнить с `python + pip`.
- Опыт разработки становится действительно приятным не только благодаря основным инструментам Rust (например, использование `rustup` для переключения на другую версию `rustc` при тестировании `crate`, который должен работать на `nightly`, текущей стабильной и более старых стабильных версиях), но и благодаря экосистеме сторонних инструментов (например, Mozilla предоставляет `cargo vet` для упрощения и совместного использования аудитов безопасности; `crate criterion` предлагает упрощённый способ запуска бенчмарков).
  - `cargo` облегчает добавление инструмента через `cargo install --locked cargo-vet`.
  - Возможно, стоит сравнить с расширениями для Chrome или расширениями для VScode.
- Широкие, общие примеры проектов, где `cargo` может быть правильным выбором:
  - Возможно, это удивительно, но Rust становится всё более популярным в индустрии для написания командных утилит. Объём и эргономика библиотек сопоставимы с Python, при этом Rust более надёжен (благодаря богатой системе типов) и работает быстрее (как компилируемый, а не интерпретируемый язык).
  - Для участия в экосистеме Rust необходимо использовать стандартные инструменты Rust, такие как `Cargo`. Библиотеки, которые хотят получать внешние вклады и использоваться вне Chromium (например, в средах сборки `Bazel` или `Android/Soong`), вероятно, должны использовать `Cargo`.
- Примеры проектов, связанных с Chromium, основанных на `cargo`:
  - `serde_json_lenient` (экспериментировалось в других подразделениях Google, что привело к PR с улучшениями производительности)
  - Библиотеки шрифтов, такие как `font-types`
  - Инструмент `gnrt` (мы познакомимся с ним позже в курсе), который зависит от `clap` для разбора командной строки и от `toml` для конфигурационных файлов.
    - \* Отказ от ответственности: уникальной причиной использования `cargo` была недоступность `gn` при сборке и начальной загрузке стандартной библиотеки Rust в процессе сборки инструментальной цепочки Rust.
    - \* `run_gnrt.py` использует копию `cargo` и `rustc` из Chromium. `gnrt` зависит от сторонних библиотек, загружаемых из интернета, но `run_gnrt.py` требует от `cargo` разрешать только содержимое с флагом `--locked` через `Cargo.lock`.

Студенты могут определить следующие элементы как явно или неявно доверенные:

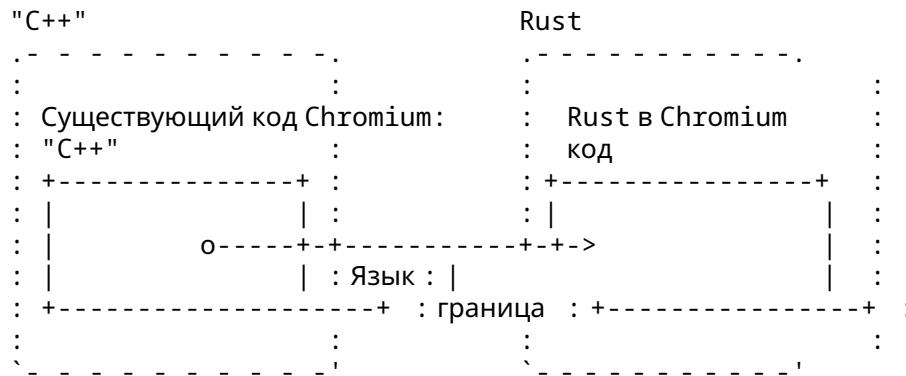
- `rustc` (компилятор Rust), который, в свою очередь, зависит от библиотек LLVM, компилятора Clang, исходников `rustc` (загружаемых с GitHub, проверяемых командой компилятора Rust), бинарного компилятора Rust, загружаемого для начальной загрузки.
- `rustup` (стоит отметить, что `rustup` разрабатывается под эгидой организации <https://github.com/rust-lang/> — той же, что и `rustc` ).
- `cargo`, `rustfmt` и другие.
- Различная внутренняя инфраструктура (боты, которые собирают `rustc`, система распространения предварительно собранной инструментальной цепочки для инженеров Chromium и др.).
- Инструменты `Cargo`, такие как `cargo audit`, `cargo vet` и др.
- Библиотеки Rust, включённые в //third\_party/rust (проверены службой безопасности [security@chromium.org](mailto:security@chromium.org))
- Другие библиотеки Rust (некоторые нишевые, некоторые достаточно популярные и широко используемые)

## Глава 42

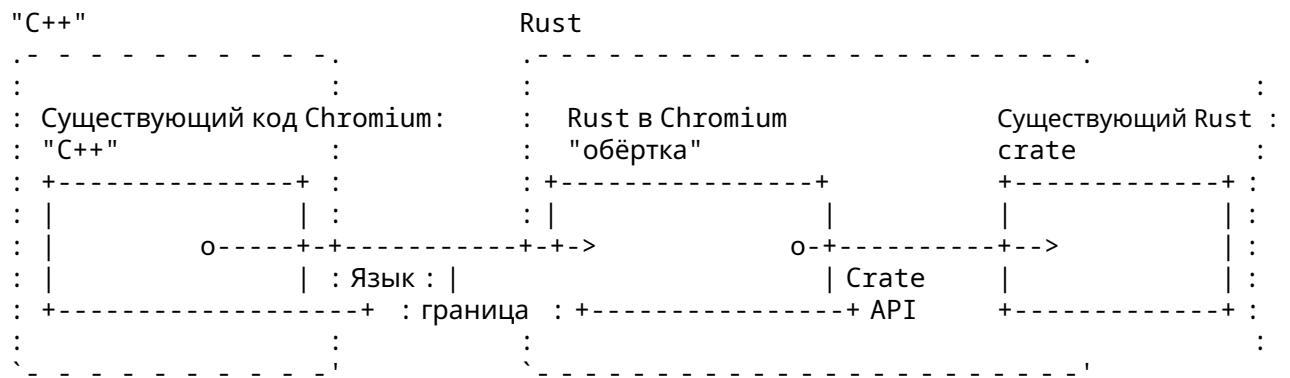
### Политика использования Rust в Chromium

Политика использования Rust в Chromium доступна здесь. Rust может использоваться как для кода первой стороны, так и для кода третьих сторон.

Использование Rust для чистого кода первой стороны выглядит следующим образом:



Случай с кодом третьих сторон также распространён. Вероятно, вам также потребуется небольшое количество кода первой стороны для связки, поскольку очень немногие библиотеки Rust напрямую предоставляют API на C/C++.



Сценарий использования стороннего crate является более сложным, поэтому сегодняшний курс будет сосредоточен на следующем:

- Подключении сторонних библиотек Rust ("crates")
- Написании промежуточного кода для возможности использования этих crate из Chromium C++. (Те же методы применяются при работе с собственным Rust-кодом).

## Глава 43

# Правила сборки

Код на Rust обычно собирается с помощью `cargo`. Chromium собирается с использованием `gn` и `ninja` для повышения эффективности — его статические правила обеспечивают максимальный уровень параллелизма. Rust не является исключением.

### Добавление Rust-кода в Chromium

В существующем файле Chromium `BUILD.gn` добавьте `rust_static_library`:

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
 crate_root = "lib.rs"
 sources = ["lib.rs"]
}
```

Вы также можете добавить `deps` для других Rust-целей. Позже мы используем это для зависимости от стороннего кода.

Вы должны указать как корень `crate`, так и полный список исходных файлов. Корень `crate_root` — это файл, передаваемый компилятору Rust, представляющий корневой файл единицы компиляции — обычно `lib.rs`. `sources` — это полный список всех исходных файлов, которые `ninja` требуются для определения необходимости пересборки.

(В Rust не существует понятия `source_set`, поскольку вся `crate` является единицей компиляции. Наименьшей единицей является `static_library`.)

Студенты могут задаться вопросом, почему нам нужен шаблон `gn`, а не встроенная поддержка `Rust static libraries` в `gn`. Ответ в том, что этот шаблон обеспечивает поддержку взаимодействия с `CXX`, функций `Rust` и модульных тестов, некоторые из которых мы используем позже.

### 43.1 Включение unsafe-кода на Rust

Небезопасный код Rust по умолчанию запрещён в `rust_static_library` — он не скомпилируется. Если вам нужен небезопасный код Rust, добавьте `allow_unsafe = true` в цель `gn`. (Позже в курсе мы рассмотрим ситуации, когда это необходимо.)

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
 crate_root = "lib.rs"
 sources = [
 "lib.rs",
 "hippopotamus.rs"
]
 allow_unsafe = true
}
```

## 43.2 Зависимость от Rust-кода из C++ в Chromium

Просто добавьте указанный выше target в deps некоторой цели C++ в Chromium.

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
 crate_root = "lib.rs"
 sources = ["lib.rs"]
}

или source_set, static_library и т.д.
component("preexisting_cpp") {
 deps = [":my_rust_lib"]
}
```

Мы увидим, что эта связь работает только в том случае, если Rust-код предоставляет простые C API, которые могут быть вызваны из C++, или если мы используем инструмент взаимодействия C++/Rust.

## 43.3 Visual Studio Code

В Rust-коде типы опускаются, что делает хороший IDE ещё более полезным, чем для C++. Visual Studio Code хорошо подходит для Rust в Chromium. Чтобы использовать его,

- Убедитесь, что в вашем VSCode установлен rust-analyzerextension, а не более ранние формы поддержки Rust.
- gn gen out/Debug --export-rust-project (или эквивалент для вашей выходной директории)
- ln -s out/Debug/rust-project.json rust-project.json

```

actual_version = match qr_code.version() {
 Version::Micro | Version::Normal | Version::High | Version::QrCode => {
 pub struct QrCode {
 content: Vec<Color, Global>,
 version: Version,
 ec_level: EcLevel,
 width: usize,
 }
 }
}

Some(min_version) => Some(min_version)
Some(min_version) => Some(min_version)
// Если `act` реализации
qr_code = QrCode::with_version(data, Version::Micro);
}

```

Демонстрация некоторых возможностей аннотирования кода и исследования с помощью rust-analyzer может быть полезна, если аудитория по своей природе скептически относится к IDE.

Следующие шаги могут помочь с демонстрацией (но вы можете использовать любой другой фрагмент Rust, связанный с Chromium, с которым вы наиболее знакомы):

- Откройте components/qr\_code\_generator/qr\_code\_generator\_ffi\_glue.rs
- Поместите курсор на вызов QrCode::new (примерно на строке 26) в файле ‘qr\_code\_generator\_ffi\_glue.rs’
- Демонстрация **показа документации** (стандартные сочетания клавиш: vscode = Ctrl+K I; vim/CoC = K).
- Демонстрация **перехода к определению** (стандартные сочетания клавиш: vscode = F12; vim/CoC = g d). (Это приведёт вас к //third\_party/rust/.../qr\_code-.../src/lib.rs.)
- Демонстрация структуры и перехода к методу QrCode::with\_bits (примерно на строке 164; структура отображается в панели проводника файлов в vscode; стандартные сочетания клавиш vim/CoC = Space O)
- Демонстрация аннотаций типов (в методе QrCode::with\_bits есть несколько хороших примеров)

Стоит отметить, что gn gen ... --export-rust-project необходимо запускать заново после редактирования файлов BUILD.gn (что мы будем делать несколько раз в ходе упражнений этой сессии).

## 43.4 Практическое задание по правилам сборки

В вашей сборке Chromium добавьте новую цель Rust в файл //ui/base/BUILD.gn со следующим содержимым:

```

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
pub extern "C" fn hello_from_rust() {
 println!("Привет из Rust!")
}

```

Важно: обратите внимание, что no\_mangle считается компилятором Rust видом небезопасности, поэтому необходимо разрешить unsafe-код в вашей gn цели.

Добавьте эту новую цель Rust как зависимость для //ui/base:base . Объявите эту функцию в начале файла ui/base/resource/resource\_bundle.cc (позже мы увидим, как это можно автоматизировать с помощью инструментов генерации биндингов):

```
extern "C" void hello_from_rust();
```

Вызовите эту функцию в любом месте файла ui/base/resource/resource\_bundle.cc — рекомендуем в начале ResourceBundle::MaybeMangleLocalizedString . Соберите и запустите Chromium, убедитесь, что "Hello from Rust!" выводится многократно.

Если вы используете VSCode, настройте Rust для эффективной работы в VSCode. Это будет полезно в последующих упражнениях. Если у вас получилось, вы сможете использовать правый клик "Перейти к определению" на println! .

## Где найти помощь

- Опции, доступные для шаблонаrust\_static\_librarygn
- Информация о#[unsafe(no\_mangle)]
- Информация оextern "C"
- Информация о переключателе--export-rust-projectв gn
- Как установить rust-analyzer в VSCode

Очень важно, чтобы студенты смогли запустить это, поскольку последующие упражнения будут строиться на этом.

Этот пример необычен тем, что сводится к языку межязыкового взаимодействия с наименьшим общим знаменателем — С. И C++, и Rust могут нативно объявлять и вызывать функции с С ABI. Позже в курсе мы напрямую свяжем C++ с Rust.

allow\_unsafe = true требуется здесь, потому что #[unsafe(no\_mangle)] может позволить Rust сгенерировать две функции с одинаковым именем, и Rust больше не может гарантировать вызов нужной.

Если вам нужен чисто Rust-исполняемый файл, это также можно сделать с помощью шаблонаrust\_executable gn.

## Глава 44

# Тестирование

Сообщество Rust обычно пишет модульные тесты в модуле, размещённом в том же исходном файле, что и тестируемый код. Это было рассмотрено ранее в курсе и выглядит следующим образом:

```
#[cfg(test)]
mod tests {
 #[test]
 fn my_test() {
 todo!()
 }
}
```

В Chromium модульные тесты размещаются в отдельном исходном файле, и мы продолжаем следовать этой практике для Rust — это обеспечивает последовательное обнаружение тестов и помогает избежать повторной сборки .rs файлов во время конфигурации test.

Это приводит к следующим вариантам тестирования кода Rust в Chromium:

- Нативные тесты Rust (т.е. #[test]). Не рекомендуется использовать вне каталога //third\_party/rust.
- gtest тесты, написанные на C++ и вызывающие Rust через FFI. Достаточно, когда код Rust является лишь тонким слоем FFI, а существующие модульные тесты обеспечивают достаточное покрытие функционала.
- gtest тесты, написанные на Rust и использующие тестируемый crate через его публичный API (при необходимости с использованием pub mod for\_testing { ... }). Это тема следующих нескольких слайдов.

Упомяните, что нативные тесты Rust для сторонних crate в конечном итоге должны запускаться ботами Chromium. (Такое тестирование требуется редко — только после добавления или обновления сторонних crates.)

Некоторые примеры могут помочь проиллюстрировать, когда следует использовать C++ gtest или Rust gtest:

- QR обладает очень ограниченным функционалом в первом уровне Rust (это всего лишь тонкий FFI-клей). Поэтому он использует существующие модульные тесты C++ для проверки как C++, так и Rust реализаций (параметризую тесты таким образом, чтобы они включали или отключали Rust с помощью ScopedFeatureList ).
- Гипотетическая или находящаяся в разработке интеграция PNG может потребовать реализации безопасных с точки зрения памяти преобразований пикселей, которые предоставляет libpng, но отсутствуют в png crate — например,

RGBA => BGRA или гамма-коррекция. Такой функционал может выиграть от отдельных тестов, написанных на Rust.

## 44.1 rust\_gtest\_interop библиотека

Библиотека `rust_gtest_interop` предоставляет возможность:

- Использовать функцию Rust в качестве gtest тесткейса (с помощью атрибута `#[gtest(...)]`)
- Используйте `expect_eq!` и аналогичные макросы (похожие на `assert_eq!`, но не вызывающие панику и не завершающие тест при неудачном утверждении).

Пример:

```
use rust_gtest_interop::prelude::*;

#[gtest(MyRustTestSuite, MyAdditionTest)]
fn test_addition() {
 expect_eq!(2 + 2, 4);
}
```

## 44.2 Правила GN для тестов на Rust

Самый простой способ создавать тесты Rust gtest — добавлять их в существующий тестовый бинарный файл, который уже содержит тесты, написанные на C++. Например:

```
test("ui_base_unittests") {
 ...
 sources += ["my_rust_lib_unittest.rs"]
 deps += [":my_rust_lib"]
}
```

Создание тестов Rust в отдельной `static_library` также возможно, но требует ручного объявления зависимости от вспомогательных библиотек:

```
rust_static_library("my_rust_lib_unittests") {
 testonly = true
 is_gtest_unittests = true
 crate_root = "my_rust_lib_unittest.rs"
 sources = ["my_rust_lib_unittest.rs"]
 deps = [
 ":my_rust_lib",
 "//testing/rust_gtest_interop",
]
}

test("ui_base_unittests") {
 ...
 deps += [":my_rust_lib_unittests"]
}
```

## 44.3 chromium::import! Macro

После добавления `:my_rust_lib` в `GN deps`, нам всё ещё необходимо изучить, как импортировать и использовать `my_rust_lib` из `my_rust_lib_unittest.rs`. Мы не указали явное `crate_name` для `my_rust_lib`, поэтому имя `crate` вычисляется на основе полного пути и имени цели. К счастью, мы можем избежать работы с таким громоздким именем, используя макрос `chromium::import!` из автоматически импортируемого `chromium` `crate`:

```
chromium::import! {
 "/ui/base:my_rust_lib";
}

use my_rust_lib::my_function_under_test;
```

Внутри макроса разворачивается во что-то подобное:

```
extern crate ui_sbase_cmy_urust_ulib as my_rust_lib;

use my_rust_lib::my_function_under_test;
```

Более подробная информация содержится в комментарии документации к макросу `chromium::import`.

`rust_static_library` поддерживает указание явного имени через свойство `crate_name`, но этого делать не рекомендуется. Это не рекомендуется, поскольку имя `crate` должно быть уникальным в глобальном масштабе. `crates.io` гарантирует уникальность имен своих `crate`, поэтому цели `cargo_crate` `GN` (генерируемые инструментом `gnrt`, рассмотренным в последующем разделе) используют короткие имена `crate`.

## 44.4 Практическое задание по тестированию

Время для следующего упражнения!

В вашей сборке Chromium:

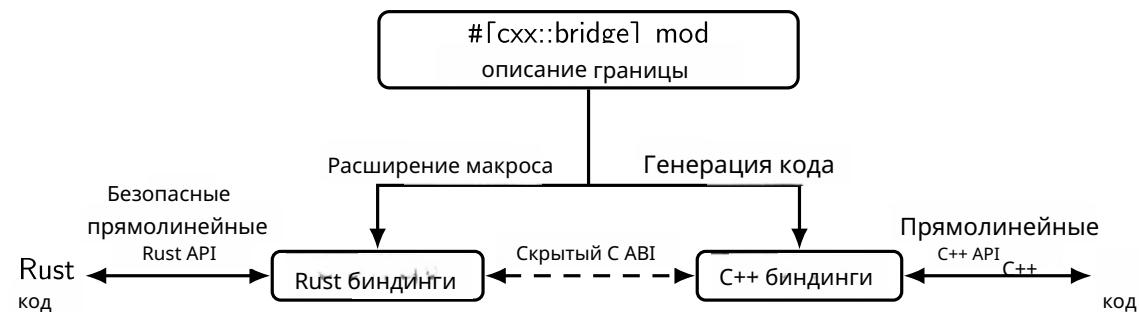
- Добавьте тестируемую функцию рядом с `hello_from_rust`. Некоторые предложения: сложение двух целых чисел, переданных в качестве аргументов, вычисление n-го числа Фибоначчи, суммирование целых чисел в срезе и т.д.
- Добавьте отдельный файл `..._unittest.rsc` тестом для новой функции.
- Добавьте новые тесты в `BUILD.gn`.
- Соберите тесты, запустите их и убедитесь, что новый тест работает.

## Глава 45

# Взаимодействие с C++

Сообщество Rust предлагает несколько вариантов взаимодействия C++ и Rust, при этом постоянно разрабатываются новые инструменты. В настоящее время Chromium использует инструмент под названием CXX.

Вы описываете всю границу языка в языке определения интерфейсов (который очень похож на Rust), а затем инструменты CXX генерируют объявления функций и типов как для Rust, так и для C++.



Смотрите учебник по CXX для полного примера использования.

Пройдитесь по диаграмме. Объясните, что за кулисами происходит то же самое, что вы делали ранее. Укажите, что автоматизация процесса имеет следующие преимущества:

- Инструмент гарантирует соответствие сторон C++ и Rust (например, вы получите ошибки компиляции, если #[cxx::bridge] не совпадает с фактическими определениями C++ или Rust, в то время как при несинхронизированных ручных привязках возникает неопределенное поведение).
- Инструмент автоматизирует генерацию FFI-прокси (небольших, совместимых с C-ABI, свободных функций) для не-C функций (например, обеспечивая вызовы FFI в методы Rust или C++; при ручных привязках такие верхнеуровневые свободные функции пришлось бы писать вручную).
- Инструмент и библиотека могут обрабатывать набор основных типов — например:
  - &[T] может передаваться через границу FFI, хотя это и не гарантирует конкретный ABI или расположение в памяти. При ручном связывании std::span<T> / &[T] необходимо вручную разбирать и восстанавливать из указателя и длины — это подвержено ошибкам, учитывая, что каждый язык немного по-разному представляет пустые срезы.
  - Умные указатели, такие как std::unique\_ptr<T>, std::shared\_ptr<T> и/или Box, поддерживаются нативно. При ручном связывании необходимо передавать сырье указатели, совместимые с C-ABI, что увеличивает риски, связанные с временем жизни и безопасностью памяти.

риски.

- Типы `rust::String` и `CxxString` понимают и учитывают различия в представлении строк между языками (например, `rust::String::lossy` может создавать Rust-строку из входных данных, не являющихся UTF-8, а `rust::String::c_str` может добавлять завершающий NUL-символ).

## 45.1 Пример биндингов

CXX требует, чтобы вся граница C++/Rust была объявлена в `cxx::bridge` модулях внутри `.rs` исходного кода.

```
#[cxx::bridge]
mod ffi {
 extern "Rust" {
 тип MultiBuf;

 fn next_chunk(buf: & mut MultiBuf) -> &[u8];
 }

 unsafe extern "C++" {
 include!("example/include/blobstore.h");

 тип BlobstoreClient;

 fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
 fn put(self: &BlobstoreClient, buf: & mut MultiBuf) -> Result<u64>;
 }
}

// Здесь размещаются определения Rust-типов и функций
```

Обратите внимание:

- Хотя это выглядит как обычный Rust `mod`, процедурный макрос `#[cxx::bridge]` выполняет с ним сложные операции. Сгенерированный код значительно более сложен — однако в вашем коде всё равно появляется `mod` с именем `ffi`.
- Нативная поддержка `std::unique_ptr` из C++ в Rust
- Нативная поддержка срезов Rust в C++
- Вызовы из C++ в Rust и Rust-типы (в верхней части)
- Вызовы из Rust в C++ и C++-типы (в нижней части)

Распространённое заблуждение: кажется, что Rust парсит заголовочный файл C++, но это вводит в заблуждение. Этот заголовок никогда не интерпретируется Rust, а просто `#include` в сгенерированном C++ коде для поддержки компиляторов C++.

## 45.2 Ограничения CXX

Безусловно, самая полезная страница при использовании CXX — это справочник по типам.

CXX в основном подходит для случаев, когда:

- Ваш интерфейс Rust-C++ достаточно прост, чтобы вы могли объявить его полностью.

- Вы используете только типы, уже нативно поддерживаемые CXX, например `std::unique_ptr`, `std::string`, `&[u8]` и т.д.

У него много ограничений — например, отсутствие поддержки типа Rust `Option`.

Эти ограничения заставляют нас использовать Rust в Chromium только для хорошо изолированных «листовых узлов», а не для произвольного взаимодействия Rust-C++. При рассмотрении варианта использования Rust в Chromium хорошей отправной точкой является создание CXX биндингов для языкового интерфейса, чтобы оценить, достаточно ли он прост.

Кроме того, в настоящее время Rust-код в одном компоненте не может зависеть от Rust-кода в другом из-за особенностей линковки в нашей компонентной сборке. Это ещё одна причина ограничить использование Rust только листовыми узлами.

Также следует обсудить некоторые другие проблемные моменты с CXX, например:

- Обработка ошибок основана на исключениях C++ (подробности на следующем слайде).
- Указатели на функции неудобны в использовании.

## 45.3 Обработка ошибок в CXX

Поддержка `Result<T, E>` в CXX базируется на исключениях C++, поэтому мы не можем использовать это в Chromium.

Альтернативы:

- Часть T в `Result<T, E>` может быть:

- Возвращена через выходные параметры (например, через `&mut T`). Это требует, чтобы `T` мог быть передан через границу FFI — например, чтобы `T` был:
  - \* Примитивным типом (например, `u32` или `usize`).
  - \* Типом, нативно поддерживаемым CXX (например, `UniquePtr<T>`), который имеет подходящее значение по умолчанию для использования в случае ошибки (в отличие от `Box<T>`).
- Сохраняется на стороне Rust и предоставляется через ссылку. Это может потребоваться, когда `T` является типом Rust, который нельзя передавать через границу FFI и который нельзя хранить в `UniquePtr<T>`.

- Часть E в `Result<T, E>` может быть:

- Возвращена в виде булевого значения (например, `true` означает успех, а `false` — неудача).
- Сохранение деталей ошибки теоретически возможно, но на практике пока не требовалось.

### 45.3.1 Обработка ошибок в CXX: пример QR

Генератор QR-кодов — пример, где булево значение используется для передачи информации об успехе или неудаче, и где успешный результат может быть передан через границу FFI:

```
#[cxx::bridge(namespace = "qr_code_generator")]
mod ffi {
 extern "Rust" {
 fn generate_qr_code_using_rust(
 data: &[u8],
 min_version: i16,
 out_pixels: Pin<& mut CxxVector<u8>>,
 out_qr_size: &mut usize,
```

```

) -> bool;
 }
}

```

Студенты могут интересоваться семантикой вывода `out_qr_size`. Это не размер вектора, а размер QR-кода (и, признаться, это несколько избыточно — это квадратный корень из размера вектора).

Стоит подчеркнуть важность инициализации `out_qr_size` до вызова функции на Rust. Создание ссылки Rust, указывающей на неинициализированную память, приводит к неопределённому поведению (в отличие от C++, где неопределенное поведение возникает только при разыменовании такой памяти).

Если студенты спрашивают о `Pin`, объясните, почему CXX требует его для изменяемых ссылок на данные C++: Ответ в том, что данные C++ нельзя перемещать так же, как данные Rust, поскольку они могут содержать самоссыльочные указатели.

### 45.3.2 Обработка ошибок CXX: пример с PNG

Прототип декодера PNG иллюстрирует, что можно сделать, когда успешный результат нельзя передать через границу FFI:

```

#[cxx::bridge(namespace = "gfx::rust_bindings")]
mod ffi {
 extern "Rust" {
 /// Возвращает FFI-совместимый эквивалент `Result<PngReader<'a>,`

 /// `()`.
 fn new_png_reader<'a>(input: &'a [u8]) -> Box<ResultOfPngReader<'a>>;

 /// C++-биндинги для типа `crate::png::ResultOfPngReader`.
 type ResultOfPngReader<'a>;
 fn is_err(self: &ResultOfPngReader) -> bool;
 fn unwrap_as_mut<'a, 'b>(

 self: &'b mut ResultOfPngReader<'a>,

) -> &'b mut PngReader<'a>;

 /// C++-биндинги для типа `crate::png::PngReader`.
 type PngReader<'a>;
 fn height(self: &PngReader) -> u32;
 fn width(self: &PngReader) -> u32;
 fn read_rgba8(self: &mut PngReader, output: & mut [u8]) -> bool;
 }
}

```

`PngReader` и `ResultOfPngReader` — это типы Rust; объекты этих типов не могут пересекать границу FFI без косвенной ссылки через `Box<T>`. Мы не можем использовать параметр `out_parameter: &mut PngReader`, поскольку CXX не позволяет C++ хранить объекты Rust по значению

Этот пример демонстрирует, что, несмотря на отсутствие поддержки произвольных дженериков или шаблонов в CXX, мы можем передавать их через границу FFI, вручную специализируя или мономорфизируя в `nogeneric` тип. В примере `ResultOfPngReader` является `nogeneric` типом, который перенаправляет вызовы к соответствующим методам `Result<T, E>` (например, `is_err`, `unwrap` и/или `as_mut`).

## 45.4 Использование cxx в Chromium

В Chromium мы определяем независимый `#[cxx::bridge] mod` для каждого конечного узла, где планируем использовать Rust. Обычно для каждой `rust_static_library` со-

здаётся отдельный модуль. Просто добавьте `cxx_bindings = [ "my_rust_file.rs" ]`  
# список файлов, содержащих `#[cxx::bridge]`, не все исходные файлы  
имеют `allow_unsafe = true`

к вашему существующему `rust_static_library` target вместе с `crate_root` и `sources`.

C++ заголовки будут генерированы в подходящем месте, поэтому

вы можете просто `#include "ui/base/my_rust_file.rs.h"`

Вы найдете некоторые вспомогательные функции в `/base` для преобразования из/в типы Chromium C++ в типы CXX Rust — например, `SpanToRustSlice`.

Студенты могут спросить — зачем нам всё ещё нужен `allow_unsafe = true`?

Общий ответ таков: никакой C/C++ код не является «безопасным» по стандартам Rust. Вызовы туда и обратно между C/C++ и Rust могут выполнять произвольные операции с памятью и нарушать безопасность собственных структур данных Rust. Наличие слишком большого количества `unsafe` ключевых слов в C/C++ взаимодействии может ухудшать соотношение сигнал/шум для такого ключевого слова и является спорным, но строго говоря, включение любого стороннего кода в Rust-бинарь может вызывать непредсказуемое поведение с точки зрения Rust.

Краткий ответ содержится в диаграмме в верхней части этой страницы — за кулисами CXX генерирует Rust `unsafe` и `extern "C"` функции точно так же, как мы делали вручную в предыдущем разделе.

## 45.5 Упражнение: взаимодействие с C++

### Часть первая

- В ранее созданном вами Rust-файле добавьте `#[cxx::bridge]`, который определяет одну функцию с именем `hello_from_rust`, вызываемую из C++, не принимающую параметров и не возвращающую значения.
- Измените вашу предыдущую функцию `hello_from_rust`, удалив `extern "C"` и атрибут `#[unsafe(no_mangle)]`. Теперь это обычная функция Rust.
- Измените вашу цель сборки для создания этих биндингов.
- В вашем C++ коде удалите предварительное объявление функции `hello_from_rust`. Вместо этого включите генерированный заголовочный файл.
- Соберите и запустите!

### Часть вторая

Полезно немного поэкспериментировать с CXX. Это помогает понять, насколько гибок Rust в Chromium.

Некоторые задания для практики:

- Вызов C++ из Rust. Для этого потребуется:
  - Дополнительный заголовочный файл, который вы можете `include!` из вашего `cxx::bridge`. Необходимо объявить вашу C++ функцию в этом новом заголовочном файле.

- Небезопасный блок для вызова такой функции или, альтернативно, указать ключевое слово `unsafe` в вашем `#[cxx::bridge]`, как описано здесь.
- Возможно, также потребуется `#include "third_party/rust/cxx/v1/crate/include/cxx.h"`
- Передать строку C++ из C++ в Rust.
- Передать ссылку на объект C++ в Rust.
- Преднамеренно сделать сигнатуры функций Rust несовместимыми с `#[cxx::bridge]` и привыкнуть к возникающим ошибкам.
- Преднамеренно сделать сигнатуры функций C++ несовместимыми с `#[cxx::bridge]`, и привыкнуть к возникающим ошибкам.
- Передать `std::unique_ptr` некоторого типа из C++ в Rust, чтобы Rust мог владеть объектом C++.
- Создайте объект Rust и передайте его в C++, чтобы владение перешло к C++. (Подсказка: вам нужен Box).
- Объявите несколько методов для C++ типа. Вызовите их из Rust.
- Объявите несколько методов для Rust типа. Вызовите их из C++.

## Часть третья

Теперь, когда вы понимаете сильные и слабые стороны взаимодействия CXX, придумайте несколько вариантов использования Rust в Chromium, где интерфейс будет достаточно простым. Набросайте, как вы могли бы определить этот интерфейс.

## Где найти помощь

- Справочник по [cxxbinding](#)
- Шаблон [rust\\_static\\_librarygn](#)

Когда студенты изучают Часть вторую, у них наверняка возникнет множество вопросов о том, как реализовать эти задачи, а также о том, как CXX работает за кулисами.

Некоторые из вопросов, с которыми вы можете столкнуться:

- У меня возникает проблема при инициализации переменной типа X значением типа Y, где X и Y — оба функциональные типы. Это происходит потому, что ваша C++ функция не совсем соответствует объявлению в вашем `cxx::bridge`.
- Кажется, я могу свободно преобразовывать ссылки C++ в ссылки Rust. Разве это не грозит неопределенным поведением? Для *оради* типов CXX — нет, поскольку они имеют нулевой размер. Для тривиальных типов CXX — да, возможно вызвать неопределенное поведение, хотя архитектура CXX значительно усложняет создание такого примера.

## Глава 46

# Добавление сторонних crate

Библиотеки Rust называются «crate» и размещаются на crates.io. Очень просто, чтобы Rust crate зависели друг от друга. И они действительно зависят!

| Свойство                   | Библиотека C++ Rust crate |                          |
|----------------------------|---------------------------|--------------------------|
| Система сборки             | Много                     | Единообразно: Cargo.toml |
| Типичный размер библиотеки | Довольно большой          | Малый                    |
| Транзитивные зависимости   | Мало                      | Много                    |

Для инженера Chromium это имеет свои плюсы и минусы:

- Все crate используют общую систему сборки, поэтому мы можем автоматизировать их включение в Chromium...
- ... Однако crate обычно имеют транзитивные зависимости, поэтому, вероятно, придётся подключать несколько библиотек.

Мы обсудим:

- Как добавить crate в исходный код Chromium
- Как создавать правила сборки для этого
- Как проводить аудит исходного кода на предмет достаточной безопасности.

Все пункты в таблице на этом слайде являются обобщениями, и можно найти контрпримеры. Однако в целом важно, чтобы студенты понимали, что большая часть кода на Rust зависит от других библиотек Rust, поскольку это легко реализовать, и это имеет как преимущества, так и издержки.

### 46.1 Настройка файла Cargo.toml для добавления Crates

В Chromium существует единый набор централизованно управляемых прямых зависимостей crate. Они управляются через единый `Cargo.toml`:

```
[dependencies]
bitflags = "1"
cfg-if = "1"
cxx = "1"
много других...
```

Как и в любом другом `Cargo.toml`, вы можете указать дополнительные параметры зависимостей — чаще всего вы захотите указать `features`, которые хотите включить в `crate`.

При добавлении `crate` в Chromium часто требуется предоставить дополнительную информацию в отдельном файле, `gnrt_config.toml`, с которым мы познакомимся далее.

## 46.2 Настройка `gnrt_config.toml`

Рядом с `Cargo.toml` находится `gnrt_config.toml`. Этот файл содержит расширения, специфичные для Chromium, в обработке `crate`.

Если вы добавляете новый `crate`, необходимо указать как минимум `group`. Это одно из следующих значений:

```
'safe' : библиотека удовлетворяет правилу-2 и может использоваться в любом процессе.
'sandbox' : библиотека не удовлетворяет правилу-2 и должна использоваться в
песочном процессе, таком как renderer или utility process.
'test' : библиотека используется только в тестах.
```

Например,

```
[crate.my-new-crate]
group = 'test' # используется только в тестовом коде
```

В зависимости от структуры исходного кода `crate`, возможно, потребуется указать в этом файле расположение его `LICENSE` файлов.

Позже мы рассмотрим другие параметры, которые необходимо настроить в этом файле для решения возникающих проблем.

## 46.3 Загрузка Crates

Инструмент под названием `gnrt` знает, как загружать `crates` и как генерировать правила `BUILD.gn`.

Для начала загрузите нужный `crate` следующим образом:

```
cd chromium/src
vpython3 tools/crates/run_gnrt.py -- vendor
```

Хотя инструмент `gnrt` является частью исходного кода Chromium, при выполнении этой команды вы загрузите и запустите его зависимости с `crates.io`.

См. предыдущий раздел, посвящённый этому решению по безопасности.

Команда `vendor` может загрузить:

- Ваш `crate`
- Прямые и транзитивные зависимости
- Новые версии других `crate`, необходимые `cargo` для разрешения полного набора `crate`, требуемых Chromium.

Chromium поддерживает патчи для некоторых `crate`, которые хранятся в каталоге `//third_party/rust/chromium_crates_io/patches`. Они будут применены автоматически, но если применение патчей не удастся, возможно, потребуется вмешательство вручную.

## 46.4 Генерация правил сборки gn

После загрузки crate сгенерируйте файлы BUILD.gn следующим образом:

```
vpython3 tools/crates/run_gnrt.py -- gen
```

Теперь выполните команду `git status`. Вы должны увидеть:

- По крайней мере один новый исходный код crate в каталоге `third_party/rust/chromium_crates_io/vendor`
- По крайней мере один новый `BUILD.gn` в `third_party/rust/<crate name>/v<major semver version>`
- Соответствующий `README.chromium`

«`major semver version`» — это версия Rust по системе «semver».

Внимательно изучите, особенно элементы, сгенерированные в `third_party/rust`.

Немного о `semver` — и в частности о том, что в Chromium это используется для поддержки нескольких несовместимых версий одного crate, что не рекомендуется, но иногда необходимо в экосистеме Cargo.

## 46.5 Решение проблем

Если сборка не удалась, возможно, причина в `build.rs`: программах, выполняющих произвольные действия во время сборки. Это принципиально противоречит дизайну `gn` и `ninja`, которые ориентированы на статичные, детерминированные правила сборки для максимизации параллелизма и повторяемости сборок.

Некоторые действия `build.rs` поддерживаются автоматически; другие требуют вмешательства:

| Эффект скрипта сборки                                                  | Поддерживается нашими шаблонами <code>gn</code> | Требуется ваша работа                        |
|------------------------------------------------------------------------|-------------------------------------------------|----------------------------------------------|
| Проверка версии <code>rustc</code> для включения и отключения функций  | Да                                              | Нет                                          |
| Проверка платформы или процессора для включения или отключения функций | Да                                              | Нет                                          |
| Генерация кода                                                         | Да                                              | Да — укажите в <code>gnrt_config.toml</code> |
| Сборка C/C++                                                           | Нет                                             | Обходное решение                             |
| Произвольные другие действия                                           | Нет                                             | Обходное решение                             |

К счастью, большинство crate не содержат скриптов сборки, и большинство из них выполняют только первые два действия.

### 46.5.1 Скрипты сборки, генерирующие код

Если `ninja` жалуется на отсутствие файлов, проверьте `build.rs`, чтобы узнать, записывает ли он исходные файлы.

Если да, измените `gnrt_config.toml`, добавив `build-script-outputs` в crate. Если это транзитивная зависимость, то есть такая, от которой код Chromium не должен напрямую зависеть, также добавьте `allow-first-party-usage=false`. В этом файле уже есть несколько примеров:

```
[crate_unicode-linebreak]
allow-first-party-usage = false
build-script-outputs = ["tables.rs"]
```

Теперь повторно запустите `gnrt.py -- gen`, чтобы регенерировать файлы `BUILD.gn` и уведомить `ninja`, что этот конкретный выходной файл является входным для последующих этапов сборки.

#### 46.5.2 Скрипты сборки, компилирующие C++ или выполняющие произвольные действия

Некоторые crates используют `cc crate` для сборки и связывания библиотек C/C++. Другие crates анализируют C/C++ с помощью `bindgen` в своих скриптах сборки. Эти действия не поддерживаются в контексте Chromium — наша система сборки `gn`, `ninja` и LLVM очень специфична в выражении взаимосвязей между действиями сборки.

Итак, ваши варианты:

- Избегать использования этих crates
- Применить патч к crate.

Патчи должны храниться в `third_party/rust/chromium_crates_io/patches/<crate>` — см., например, патчи для `cxx crate` — и будут автоматически применяться `gnrt` при каждом обновлении crate.

## 46.6 Зависимость от Crate

После добавления стороннего crate и генерации правил сборки зависит от crate просто . Найдите вашу цель `rust_static_library` и добавьте `dep` на цель `:lib` внутри вашего crate.

Конкретно,

```
+-----+ +-----+
//third_party/rust" | имя crate | "/v" | основная версия semver | ":lib"
+-----+ +-----+
```

Например,

```
rust_static_library("my_rust_lib") {
 crate_root = "lib.rs"
 sources = ["lib.rs"]
 deps = ["//third_party/rust/example_rust_crate/v1:lib"]
}
```

## 46.7 Аудит сторонних Crates

Добавление новых библиотек подчиняется стандартным политикам Chromium, но, разумеется, также подлежит проверке безопасности. Поскольку вы можете подключать не только один crate, но и транзитивные зависимости, может потребоваться проверка большого объема кода. С другой стороны, безопасный код на Rust может иметь ограниченные негативные побочные эффекты. Как следует его проверять?

Со временем Chromium стремится перейти на процесс, основанный на `cargo vet`.

Тем временем для каждого нового добавления crate мы проверяем следующее:

- Понять, зачем используется каждый crate. Каковы взаимосвязи между crate? Если в системе сборки каждого crate присутствует `build.rs` или процедурные макросы, выясните, для чего они нужны. Совместимы ли они с обычным способом сборки Chromium?
- Проверить, что каждый crate поддерживается на должном уровне
- Используйте `cd third-party/rust/chromium_crates_io; cargo audit` для проверки известных уязвимостей (сначала необходимо выполнить `cargo install cargo-audit`, что, иронично, требует загрузки множества зависимостей из интернета)
- Убедитесь, что любой unsafe код соответствует правилу двух
- Проверьте использование любых `fs` или `net` API
- Внимательно прочитайте весь код, чтобы выявить любые аномалии, которые могли быть внедрены с вредоносными намерениями. (Реалистично невозможно достичь 100% совершенства: зачастую кода слишком много.)

Это лишь рекомендации — сотрудничайте с рецензентами из `security@chromium.org` для определения правильного подхода к уверенности в crate.

## 46.8 Проверка Crates в исходном коде Chromium

`git status` должен показать:

- Код crate в `//third_party/rust/chromium_crates_io`
- Метаданные (`BUILD.gn` и `README.chromium`) находятся в `//third_party/rust/<crate>`

`/<version>`. Пожалуйста, также добавьте файл `OWNERS` в этом каталоге.

Всё это вместе с изменениями в `Cargo.toml` и `gnrt_config.toml` следует внести в репозиторий Chromium.

Важно: необходимо использовать `git add -f`, так как в противном случае файлы, указанные в `.gitignore`, могут быть пропущены.

При этом вы можете столкнуться с тем, что проверки `presubmit` не проходят из-за использования неинклюзивной лексики. Это связано с тем, что данные Rust crate обычно содержат имена веток `git`, и во многих проектах там до сих пор используется неинклюзивная терминология. Поэтому вам может понадобиться выполнить:

```
infra/update_inclusive_language_presubmit_exempt_dirs.sh > infra/inclusive_language_pres git add -p infra/inclusive_language_presubmit_exempt_dirs.txt # добавьте любые имеющиеся изменения
```

## 46.9 Обновление Crates

Как ВЛАДЕЦ любой сторонней зависимости Chromium, вы обязаны поддерживать её в актуальном состоянии с учётом всех исправлений безопасности. Ожидается, что вскоре этот процесс будет автоматизирован для **Rust crate**, но пока это остаётся вашей ответственностью, как и для любой другой сторонней зависимости.

## 46.10 Упражнение

Добавьте `uwuify` в Chromium, отключив стандартные функции crate. Предположим, что crate будет использоваться в поставляемом Chromium, но не будет обрабатывать недоверенный ввод.

(В следующем упражнении мы будем использовать `uwuify` из Chromium, но вы можете пропустить вперёд и сделать это сейчас, если хотите. Или вы можете создать новую **trust\_executable target**, которая использует `uwuify`).

Студентам потребуется загрузить множество транзитивных зависимостей.

Общее количество необходимых crates:

- instant,
- lock\_api,
- parking\_lot,
- parking\_lot\_core,
- redox\_syscall,
- scopeguard,
- smallvec, и
- uwuify.

Если студенты загружают ещё больше, вероятно, они забыли отключить стандартные функции.

Благодарим Дэниела Лю за этот crate!

## Глава 47

# Объединение знаний — упражнение

В этом упражнении вы добавите совершенно новую функцию Chromium, объединив всё, что уже изучили.

### Задание от отдела управления продуктом

Обнаружено сообщество пикси, живущее в удалённом тропическом лесу. Важно как можно скорее доставить им Chromium для пикси.

Требуется перевести все строки пользовательского интерфейса Chromium на язык Pixie.

Нет времени ждать корректных переводов, но, к счастью, язык `rīxie` очень близок к английскому, и, как оказалось, существует Rust crate, который выполняет этот перевод.

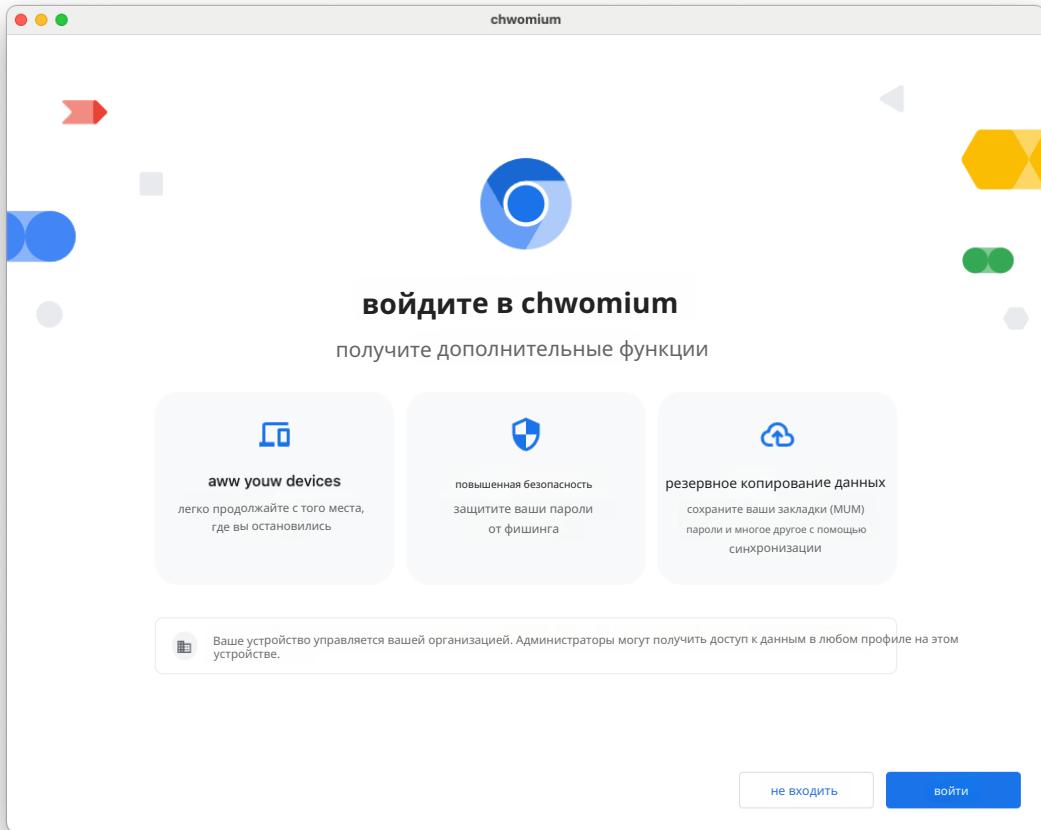
Фактически, вы уже импортировали этот crate в предыдущем упражнении.

(Очевидно, что настоящие переводы Chrome требуют исключительной тщательности и аккуратности. Не публикуйте это!)

### Шаги

Измените `ResourceBundle::MaybeMangleLocalizedString` так, чтобы он инициализировал все строки перед отображением. В этой специальной сборке Chromium это должно происходить всегда, независимо от значения `mangle_localized_strings_`.

Если вы всё сделали правильно во всех этих упражнениях, поздравляем — вы создали Chrome для pixies!



Студентам, вероятно, понадобятся подсказки. Подсказки включают:

- UTF16 и UTF8. Студенты должны знать, что строки в Rust всегда в UTF8, и, вероятно, решат, что лучше выполнять конвертацию на стороне C++ с помощью `base::UTF16ToUTF8` и обратно.
- Если студенты решат выполнить преобразование на стороне Rust, им потребуется рассмотреть использование `String::from_utf16`, обработку ошибок и определить, какие типы, поддерживаемые CXX, могут передавать большое количество `u16`.
- Студенты могут спроектировать границу C++/Rust несколькими способами, например, принимая и возвращая строки по значению или принимая изменяемую ссылку на строку. Если используется изменяемая ссылка, CXX, вероятно, сообщит студенту, что необходимо использовать `Pin`. Возможно, потребуется объяснить, что делает `Pin`, а затем объяснить, почему CXX требует его для изменяемых ссылок на данные C++: ответ в том, что данные C++ нельзя перемещать так же, как данные Rust, поскольку они могут содержать самоссылочные указатели.
- Целевой C++-модуль, содержащий `ResourceBundle::MaybeMangleLocalizedString`, должен зависеть от цели `rust_static_library`. Вероятно, студент уже выполнил это.
- Цель `rust_static_library` должна зависеть от `//third_party/rust/uwufy/v0_2:lib`.

## **Глава 48**

# **Решения упражнений**

Решения упражнений по Chromium можно найти в этой серии CL.

**Часть XI**

**Bare Metal: Утро**

## Глава 49

# Добро пожаловать в Bare Metal Rust

Это автономный однодневный курс по bare-metal Rust, предназначенный для людей, знакомых с основами Rust (возможно, после прохождения курса Comprehensive Rust), и, желательно, имеющих некоторый опыт bare-metal программирования на других языках, таких как C.

Сегодня мы поговорим о «bare-metal» Rust: запуске Rust-кода без операционной системы под нами. Курс будет разделён на несколько частей:

- Что такое no\_std Rust?
- Написание прошивки для микроконтроллеров.
- Написание загрузчика и кода ядра для процессоров приложений.
- Некоторые полезные crates для разработки bare-metal Rust.

Для части курса, посвящённой микроконтроллерам, мы будем использовать BBC micro:bit v2 в качестве примера. Это плата разработки на базе микроконтроллера Nordic nRF52833 с несколькими светодиодами и кнопками, акселерометром и компасом, подключёнными по I2C, а также встроенным отладчиком SWD.

Для начала установите инструменты, которые нам понадобятся позже. В gLinux или Debian:

```
sudo apt install gdb-multiarch libudev-dev picocom pkg-config qemu-system-arm build-essential
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils
curl --proto '=https' --tlsv1.2 -LsSf https://github.com/probe-rs/probe-rs/releases/latest
```

Предоставьте пользователям из группы plugdev доступ к программатору micro:bit:

```
echo 'SUBSYSTEM=="hidraw", ATTRS{idVendor}=="0d28", MODE="0660", GROUP="logindev", TAG+="udevadm-control"' |
 sudo tee /etc/udev/rules.d/50-microbit.rules && sudo udevadm control --reload-rules
```

В выводе команды lsusb вы должны увидеть "NXP ARM mbed", если устройство доступно. Если вы используете среду Linux на Chromebook, необходимо предоставить доступ к USB-устройству для Linux через chrome://os-settings/crostini/sharedUsbDevices .

В macOS:

```
xcode-select --install
brew install gdb picocom qemu
rustup update
```

```
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils
curl --proto '=https' --tlsv1.2 -LsSf https://github.com/probe-rs/probe-rs/releases/latest
```

# Глава 50

## no\_std

core

alloc

std

- Срезы, &str, CStr
- NonZeroU8...
- Option, Result
- Display, Debug, write!...
- Iterator
- Error
- panic!, assert\_eq!...
- NonNull и все стандартные функции, связанные с указателями
- Future и async/await
- fence, AtomicBool, AtomicPtr, AtomicU32...
- Продолжительность
- Box, Cow, Arc, Rc
- Vec, BinaryHeap, BtreeMap, LinkedList, VecDeque
- String, CString, format!
- HashMap
- Mutex, Condvar, Barrier, Once, RwLock, mpsc
- File и остальное из fs
- println!, Read, Write, Stdin, Stdout и остальное из io
- Path, OsString
- net
- Command, Child, ExitCode
- spawn, sleep и остальное из thread
- SystemTime, Instant
- HashMap зависит от RNG.
- std реэкспортирует содержимое как core, так и alloc.

## 50.1 Минимальная no\_std программа

```
#! [no_main]
#! [no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo) -> ! {
 loop {}
}
```

- Это скомпилируется в пустой бинарный файл.
- std предоставляет обработчик паники; без него необходимо предоставить собственный.
- Он также может быть предоставлен другим crate, например, panic-halt.
- В зависимости от целевой платформы может потребоваться компиляция с параметром `panic = "abort"` для предотвращения ошибки, связанной с `eh_personality`.
- Обратите внимание, что отсутствует `main` или какая-либо другая точка входа; вам необходимо определить собственную точку входа. Обычно это включает скрипт компоновщика и некоторый ассемблерный код для подготовки среды к выполнению кода на Rust.

## 50.2 alloc

Для использования alloc необходимо реализовать глобальный (кучевый) аллокатор.

```
#! [no_main]
#! [no_std]

extern crate alloc;
extern crate panic_halt as _;

use alloc::string::ToString;
use alloc::vec::Vec;
use buddy_system_allocator::LockedHeap;

#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap<32> = LockedHeap::<32>::new();

const HEAP_SIZE: usize = 65536;
static mut HEAP: [u8; HEAP_SIZE] = [0; HEAP_SIZE];

pub fn entry() {
 // БЕЗОПАСНОСТЬ: `HEAP` используется только здесь, а `entry` вызывается единожды.
 unsafe {
 // Выделяем аллокатору некоторый объём памяти для распределения.
 HEAP_ALLOCATOR.lock().init(&raw mut HEAP as usize, HEAP_SIZE);
 }

 // Теперь можно выполнять операции, требующие выделения памяти в куче.
 let mut v = Vec::new();
 v.push("A string".to_string());
```

}

- `buddy_system_allocator` — crate, реализующий базовый аллокатор на основе buddy-системы. Доступны и другие crates, либо вы можете написать собственный аллокатор или интегрировать существующий.
- Константный параметр `LockedHeap` определяет максимальный порядок аллокатора; то есть в данном случае он может выделять области размером до  $2^{**32}$  байт.
- Если любой crate в вашем дереве зависимостей использует `alloc`, то в вашем бинарном файле должен быть определён ровно один глобальный аллокатор. Обычно это выполняется в верхнеуровневом бинарном crate.
- `extern crate panic_halt as _` необходимо для обеспечения связывания краты `panic_halt`, чтобы получить его обработчик паники.
- Этот пример скомпилируется, но не запустится, так как не содержит точки входа.

## Глава 51

# Микроконтроллеры

Крат cortex\_m\_rt предоставляет (в числе прочего) обработчик сброса для микроконтроллеров Cortex M.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
 loop {}
}
```

Далее мы рассмотрим, как получить доступ к периферийным устройствам с различными уровнями абстракции.

- Макрос cortex\_m\_rt::entry требует, чтобы функция имела тип fn() -> !, поскольку возврат к обработчику сброса не имеет смысла.
- Запустите пример с помощью cargo embed --bin minimal

### 51.1 Raw MMIO

Большинство микроконтроллеров обращаются к периферии через память с отображением ввода-вывода. Попробуем включить светодиод на нашем micro:bit:

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;

use core::mem::size_of;
```

```

use cortex_m_rt::entry;

/// Адрес периферийного устройства порта GPIO 0
const GPIO_P0: usize = 0x5000_0000;

// Смещения периферийных регистров GPIO
const PIN_CNF: usize = 0x700;
const OUTSET: usize = 0x508;
const OUTCLR: usize = 0x50c;

// Поля регистра PIN_CNF
const DIR_OUTPUT: u32 = 0x1;
const INPUT_DISCONNECT: u32 = 0x1 << 1;
const PULL_DISABLED: u32 = 0x0 << 2;
const DRIVE_S0S1: u32 = 0x0 << 8;
const SENSE_DISABLED: u32 = 0x0 << 16;

#[entry]
fn main() -> ! {
 // Настройка выводов 21 и 28 порта GPIO 0 как выходов с push-pull.
 let pin_cnf_21 = (GPIO_P0 + PIN_CNF + 21 * size_of::<u32>()) as *mut u32;
 let pin_cnf_28 = (GPIO_P0 + PIN_CNF + 28 * size_of::<u32>()) as *mut u32;
 // БЕЗОПАСНОСТЬ: Указатели указывают на корректные регистры управления периферией, и отсутствуют
 // алиасы.
 unsafe {
 pin_cnf_21.write_volatile(
 DIR_OUTPUT
 | INPUT_DISCONNECT
 | PULL_DISABLED
 | DRIVE_S0S1
 | SENSE_DISABLED,
);
 pin_cnf_28.write_volatile(
 DIR_OUTPUT
 | INPUT_DISCONNECT
 | PULL_DISABLED
 | DRIVE_S0S1
 | SENSE_DISABLED,
);
 }

 // Установить вывод 28 в низкий уровень, а вывод 21 в высокий для включения светодиода.
 let gpio0_outset = (GPIO_P0 + OUTSET) as *mut u32;
 let gpio0_outclr = (GPIO_P0 + OUTCLR) as *mut u32;
 // БЕЗОПАСНОСТЬ: Указатели указывают на корректные регистры управления периферией, и отсутствуют
 // алиасы.
 unsafe {
 gpio0_outclr.write_volatile(1 << 28);
 gpio0_outset.write_volatile(1 << 21);
 }
}

```

```
 loop {}
}
```

- Пин GPIO 0 номер 21 подключён к первому столбцу LED-матрицы, а пин 28 — к первой строке.

Запустите пример с помощью:

```
cargo embed --bin mmio
```

## 51.2 Peripheral Access Crates

**svd2rust** генерирует преимущественно безопасные обёртки Rust для периферийных устройств с отображением в память на основе файлов CMSIS-SVD.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_pac::Peripherals;

#[entry]
fn main() -> ! {
 let p = Peripherals::take().unwrap();
 let gpio0 = p.P0;

 // Настройка выводов 21 и 28 порта GPIO 0 как выходов с push-pull.
 gpio0.pin_cnf[21].write(|w| {
 w.dir().output();
 w.input().disconnect();
 w.pull().disabled();
 w.drive().s0s1();
 w.sense().disabled();
 w
 });
 gpio0.pin_cnf[28].write(|w| {
 w.dir().output();
 w.input().disconnect();
 w.pull().disabled();
 w.drive().s0s1();
 w.sense().disabled();
 w
 });

 // Установить вывод 28 в низкий уровень, а вывод 21 в высокий для включения светодиода.
 gpio0.outclr.write(|w| w.pin28().clear());
 gpio0.outset.write(|w| w.pin21().set());

 loop {}
}
```

- Файлы SVD (System View Description) — это XML-файлы, обычно предоставляемые производителями микросхем, которые описывают карту памяти устройства.
  - Они организованы по периферийным устройствам, регистрами, полями и значениями, с указанием имён, описаний, адресов и прочего.
  - Файлы SVD часто содержат ошибки и неполные данные, поэтому существуют различные проекты, которые исправляют ошибки, добавляют недостающие детали и публикуют генерированные crates.
- cortex-m-rt предоставляет таблицу векторов, среди прочего.
- Если вы cargo install cargo-binutils, то можете выполнить cargo objdump --bin рас -- -d --no-show-raw-insn чтобы увидеть результирующий бинарный файл.

Запустите пример с помощью:

```
cargo embed --bin рас
```

### 51.3 HAL crates

HAL crates для многих микроконтроллеров предоставляют обёртки вокруг различных периферийных устройств. Они, как правило, реализуют трейты из embedded-hal .

```
#!/[no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use nrf52833_hal::gpio::{Level, p0};
use nrf52833_hal::pac::Peripherals;

#[entry]
fn main() -> ! {
 let p = Peripherals::take().unwrap();

 // Создаём обёртку HAL для порта GPIO 0.
 let gpio0 = p0::Parts::new(p.P0);

 // Настройка выводов 21 и 28 порта GPIO 0 как выходов с push-pull.
 let mut col1 = gpio0.p0_28.into_push_pull_output(Level::High);
 let mut row1 = gpio0.p0_21.into_push_pull_output(Level::Low);

 // Установите вывод 28 в низкий уровень, а вывод 21 в высокий,
 // чтобы включить светодиод. col1.set_low().unwrap();
 row1.set_high().unwrap();

 loop {}
}

 • set_low и set_high — методы трейта embedded_hal OutputPin.
 • HAL-крайтсы существуют для многих устройств на базе Cortex-M и RISC-V, включая различные микроконтроллеры STM32, GD32, nRF, NXP, MSP430, AVR и PIC.
```

Запустите пример с помощью:

```
cargo embed --bin hal
```

## 51.4 Board support crates

Крейтсы поддержки плат обеспечивают дополнительный уровень обёртки для конкретной платы для удобства.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use microbit::Board;

#[entry]
fn main() -> ! {
 let mut board = Board::take().unwrap();

 board.display_pins.col1.set_low().unwrap();
 board.display_pins.row1.set_high().unwrap();

 loop {}
}
```

- В данном случае crate поддержки платы просто предоставляет более удобные имена и некоторую инициализацию.
- Crate также может включать драйверы для некоторых встроенных устройств за пределами самого микроконтроллера.
  - microbit-v2 включает простой драйвер для LED-матрицы.

Запустите пример с помощью:

```
cargo embed --bin board_support
```

## 51.5 Паттерн состояния типа

```
#[entry]
fn main() -> ! {
 let p = Peripherals::take().unwrap();
 let gpio0 = p::Parts::new(p.P0);

 let pin: P0_01<Disconnected> = gpio0.p0_01;

 // let gpio0_01_again = gpio0.p0_01; // Ошибка, значение было перемещено.
 let mut pin_input: P0_01<Input<Floating>> = pin.into_floating_input();
 if pin_input.is_high().unwrap() {
 // ...
 }
 let mut pin_output: P0_01<Output<OpenDrain>> = pin_input
 .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level::Low);
```

```

pin_output.set_high().unwrap();
// pin_input.is_high(); // Ошибка, перемещено.

let _pin2: P0_02<Output<OpenDrain>> = gpio0
 .p0_02
 .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level::Low);
let _pin3: P0_03<Output<PushPull>> =
 gpio0.p0_03.into_push_pull_output(Level::Low);

loop {}
}

```

- Пины не реализуют `Copy` или `Clone`, поэтому может существовать только один экземпляр каждого. После того как пин перемещён из структуры порта, никто другой не может его получить.
- Изменение конфигурации пина потребляет старый экземпляр пина, поэтому его нельзя использовать после этого.
- Тип значения указывает на состояние, в котором оно находится: например, в данном случае — состояние конфигурации GPIO-пина. Это кодирует конечный автомат в систему типов и гарантирует, что вы не попытаетесь использовать пин определённым образом без предварительной правильной настройки. Незаконные переходы состояний обнаруживаются на этапе компиляции.
- Вы можете вызвать `is_high` на входном контакте и `set_high` на выходном, но не наоборот.
- Многие HAL-крайты следуют этой модели.

## 51.6 embedded-hal

Крейт `embedded-hal` предоставляет ряд трейтов, охватывающих распространённые периферийные устройства микроконтроллеров:

- GPIO
- PWM
- Таймеры задержки
- Шины и устройства I2C и SPI

Аналогичные трейты для потоков байтов (например, UART), CAN-шин и генераторов случайных чисел выделены в отдельные крейты `embedded-io`, `embedded-can` и `xrand_core` соответственно.

Другие крейты реализуют драйверы на основе этих трейтов, например, драйвер акселерометра может требовать экземпляр устройства I2C или SPI.

- Трейты охватывают использование периферии, но не её инициализацию или конфигурацию, так как инициализация и настройка обычно сильно зависят от платформы.
- Существуют реализации для многих микроконтроллеров, а также для других платформ, таких как Linux на Raspberry Pi.
- `embedded-hal-async` предоставляет асинхронные версии трейтов.
- `embedded-hal-nb` предоставляет альтернативный подход к неблокирующему вводу-выводу, основанный на `nb` crate.

## 51.7 probe-rs и cargo-embed

`probe-rs` — удобный набор инструментов для отладки встроенных систем, аналогичный OpenOCD, но с лучшей интеграцией.

- SWD (Serial Wire Debug) и JTAG через программаторы CMSIS-DAP, ST-Link и J-Link

- GDB stub и сервер Microsoft DAP (Debug Adapter Protocol)
- Интеграция с Cargo

cargo-embed — подкоманда cargo для сборки и прошивки бинарных файлов, логирования вывода RTT (Real Time Transfers) и подключения GDB. Настраивается с помощью файла `Embed.toml` в каталоге проекта.

- CMSIS-DAP — стандартный протокол Arm по USB для отладчика в цепи, обеспечивающий доступ к CoreSight Debug Access Port различных процессоров Arm Cortex. Именно этот протокол использует встроенный отладчик на плате BBC micro:bit.
- ST-Link — линейка отладчиков в цепи от ST Microelectronics, J-Link — линейка от SEGGER.
- Debug Access Port обычно представляет собой либо 5-контактный интерфейс JTAG, либо 2-контактный Serial Wire Debug.
- probe-rs — это библиотека, которую можно интегрировать в собственные инструменты при необходимости.
- Протокол Microsoft Debug Adapter позволяет VSCode и другим IDE отлаживать код, выполняющийся на любом поддерживаемом микроконтроллере.
- cargo-embed — это исполняемый файл, созданный с использованием библиотеки probe-rs.
- RTT (Real Time Transfers) — механизм передачи данных между отладочным хостом и целевым устройством через несколько кольцевых буферов.

### 51.7.1 Отладка

*Embed.toml:*

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true
```

В одном терминале в каталоге `src/bare-metal/microcontrollers/examples/`:

`cargo embed --bin board_support debug`

В другом терминале в том же каталоге:

В gLinux или Debian:

`gdb-multiarch target/thumbv7em-none-eabihf/debug/board_support --eval-command="target re`

On MacOS:

`arm-none-eabi-gdb target/thumbv7em-none-eabihf/debug/board_support --eval-command="targe`

В GDB попробуйте выполнить:

```
b src/bin/board_support.rs:29
b src/bin/board_support.rs:30
b src/bin/board_support.rs:32
c
c
c
```

## 51.8 Другие проекты

- RTIC
  - «Real-Time Interrupt-driven Concurrency».

- Управление общими ресурсами, передача сообщений, планирование задач, очередь таймеров.
- Embassy
  - async исполнители с приоритетами, таймеры, сетевые возможности, USB.
- TockOS
  - Ориентированная на безопасность RTOS с вытесняющим планированием и поддержкой Memory Protection Unit.
- Hubris
  - Микроядерная RTOS от Oxide Computer Company с защитой памяти, непривилегированными драйверами и IPC.
- Привязки для FreeRTOS.

Некоторые платформы имеют stdреализации, например esp-idf.

- RTIC можно рассматривать как RTOS или как фреймворк для конкурентности.
  - Не включает в себя HAL.
  - Для планирования используется Cortex-M NVIC (Nested Virtual Interrupt Controller), а не полноценное ядро.
  - Только Cortex-M.
- Google использует TockOS на микроконтроллере Haven для ключей безопасности Titan.
- FreeRTOS в основном написан на C, но существуют Rust-обёртки для разработки приложений.

## Глава 52

# Упражнения

Мы будем считывать направление с компаса I2C и записывать показания в последовательный порт.

После выполнения упражнений вы можете ознакомиться с предоставленными решениями.

### 52.1 Компас

Мы будем считывать направление с компаса I2C и записывать показания в последовательный порт. Если у вас есть время, попробуйте также отобразить это на светодиодах или использовать кнопки каким-либо образом.

Подсказки:

- Изучите документацию для `lsm303agr` и `microbit-v2` крейтов, а также аппаратное обеспечение `micro:bit`.
- Инерциальный измерительный блок LSM303AGR подключён к внутренней шине I2C.
- TWI — это другое название I2C, поэтому периферийное устройство I2C master называется TWIM.
- Драйвер LSM303AGR требует реализации трейта `embedded_hal::i2c::I2c`. Структура `microbit::hal::Twim` реализует этот трейт.
- У вас есть структурой `microbit::Board` с полями для различных пинов и периферийных устройств.
- Вы также можете ознакомиться с даташитом на nRF52833, если хотите, но это не обязательно для данного упражнения.

Скачайте шаблон упражнения и посмотрите в каталоге `compass` следующие файлы.

`src/main.rs`:

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use microbit::{hal::{Delay, uarte::{Baudrate, Parity, Uarte}}, Board};

#[entry]
fn main() -> ! {
 let mut board = Board::take().unwrap();
```

```

// Настройка последовательного порта.
let mut serial = Uarte::new(
 board.UARTE0, board
 .uart.into(), Parity
 ::EXCLUDED, Baudrate
 ::BAUD115200);

// Используйте системный таймер в качестве поставщика задержек.
let mut delay = Delay::new(board.SYST);

// Настройка контроллера I2C и инерциального измерительного блока.
// TODO

writeln!(serial, "Ready.").unwrap();

loop {
 // Считать данные компаса и записать их в последовательный порт.
 // TODO
}
}

```

*Cargo.toml* (вам не нужно это менять):

```

[workspace]

[package]
name = "compass"
version = "0.1.0"
edition = "2024"
publish = false

```

```

[dependencies]
cortex-m-rt = "0.7.5"
embedded-hal = "1.0.0"
lsm303agr = "1.1.0"
microbit-v2 = "0.15.1"
panic-halt = "1.0.0"

```

*Embed.toml* (вам не нужно это менять):

```

[default.general]
chip = "nrf52833_xxAA"

```

```

[debug.gdb]
enabled = true

```

```

[debug.reset]
halt_afterwards = true

```

*.cargo/config.toml* (вам не нужно это менять):

```

[build]

```

```

target = "thumbv7em-none-eabihf" # Cortex-M4F

[target.'cfg(all(target_arch = "arm", target_os = "none"))']
rustflags = ["-C", "link-arg=-Tlink.x"] Просмотрите по-
следовательный вывод в Linux с помощью:
picocom --baud 115200 --imap lfcrlf /dev/ttyACM0
Или на Mac OS что-то вроде (имя устройства может немного отличаться):
picocom --baud 115200 --imap lfcrlf /dev/tty.usbmodem14502
Используйте Ctrl+A Ctrl+Q для выхода из picocom.

```

## 52.2 Упражнение Bare Metal Rust Morning

### Компас

(возврат к упражнению)

```

#!/[no_main]
#!/[no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use embedded_hal::digital::InputPin;
use lsm303agr::{
 AccelMode, AccelOutputDataRate, Lsm303agr, MagMode, MagOutputDataRate,
};
use microbit::Board;
use microbit::display::blocking::Display;
use microbit::hal::twim::Twim;
use microbit::hal::{Baudrate, Parity, Uarte};
use microbit::hal::{Delay, Timer};
use microbit::pac::twim0::frequency::FREQUENCY_A;

const COMPASS_SCALE: i32 = 30000;
const ACCELEROMETER_SCALE: i32 = 700;

#[entry]
fn main() -> ! {
 let mut board = Board::take().unwrap();

 // Настройка последовательного порта.
 let mut serial = Uarte::new(
 board.UARTE0, board
 .uart.into(), Parity
 ::EXCLUDED, Baudrate
 ::BAUD115200);
}

```

```

// Используйте системный таймер в качестве поставщика задержек.
let mut delay = Delay::new(board.SYST);

// Настройка контроллера I2C и инерциального измерительного блока.
writeln!(serial, "Настройка IMU...").unwrap();
let i2c = Twim::new(board.TWIM0, board.i2c_internal.into(), FREQUENCY_A::K100);
let mut imu = Lsm303agr::new_with_i2c(i2c);
imu.init().unwrap();
imu.set_mag_mode_and_odr(
 &mut delay,
 MagMode::HighResolution,
 MagOutputDataRate::Hz50,
)
.unwrap();
imu.set_accel_mode_and_odr(
 &mut delay,
 AccelMode::Normal,
 AccelOutputDataRate::Hz50,
)
.unwrap();
let mut imu = imu.into_mag_continuous().ok().unwrap();

// Настройка дисплея и таймера.
let mut timer = Timer::new(board.TIMER0);
let mut display = Display::new(board.display_pins);

let mut mode = Mode::Compass;
let mut button_pressed = false;

writeln!(serial, "Ready.").unwrap();

loop {
 // Считать данные компаса и записать их в последовательный порт.
 while !(imu.mag_status().unwrap().xyz_new_data()
 && imu.accel_status().unwrap().xyz_new_data())
 {}
 let compass_reading = imu.magnetic_field().unwrap();
 let accelerometer_reading = imu.acceleration().unwrap();
 writeln!(
 serial,
 "{} , {} , {} \t {} , {} , {}",
 compass_reading.x_nt(),
 compass_reading.y_nt(),
 compass_reading.z_nt(),
 accelerometer_reading.x_mg(),
 accelerometer_reading.y_mg(),
 accelerometer_reading.z_mg(),
)
 .unwrap();
}

```

```

let mut image = [[0; 5]; 5];
let (x, y) = match mode {
 Mode::Compass => (
 scale(-compass_reading.x_nt(), -COMPASS_SCALE, COMPASS_SCALE, 0, 4)
 as usize,
 scale(compass_reading.y_nt(), -COMPASS_SCALE, COMPASS_SCALE, 0, 4)
 as usize,
),
 Mode::Accelerometer => (
 scale(
 accelerometer_reading.x_mg(),
 -ACCELEROMETER_SCALE,
 ACCELEROMETER_SCALE,
 0,
 4,
) as usize,
 scale(
 -accelerometer_reading.y_mg(),
 -ACCELEROMETER_SCALE,
 ACCELEROMETER_SCALE,
 0,
 4,
) as usize,
),
};

image[y][x] = 255;
display.show(&mut timer, image, 100);

// Если кнопка A нажата, переключиться на следующий режим и кратковременно мигнуть всеми светодиодами
// включено.
if board.buttons.button_a.is_low().unwrap() {
 if !button_pressed {
 mode = mode.next();
 display.show(&mut timer, [[255; 5]; 5], 200);
 }
 button_pressed = true;
} else {
 button_pressed = false;
}
}

#[derive(Copy, Clone, Debug, Eq, PartialEq)]
enum Mode {
 Compass,
 Accelerometer,
}

impl Mode {
 fn next(self) -> Self {
 match self {

```

```
 Self::Compass => Self::Accelerometer,
 Self::Accelerometer => Self::Compass,
)
}

fn scale(value: i32, min_in: i32, max_in: i32, min_out: i32, max_out: i32) -> i32 {
 let range_in = max_in - min_in;
 let range_out = max_out - min_out;
 let scaled = min_out + range_out * (value - min_in) / range_in;
 scaled.clamp(min_out, max_out)
}
```

## **Часть XII**

# **Bare Metal: Aflternoon**

## Глава 53

# Процессоры приложений

До настоящего момента мы рассматривали микроконтроллеры, такие как серия Arm Cortex-M. Обычно это небольшие системы с очень ограниченными ресурсами.

Более крупные системы с большим объёмом ресурсов обычно называются процессорами приложений, построенными на базе процессоров, таких как ARM Cortex-A или Intel Atom.

Для упрощения мы будем работать только с платой 'virt' aarch64 в QEMU.

- В общем, микроконтроллеры не имеют MMU или нескольких уровней привилегий (исключительные уровни на процессорах Arm, кольца на x86).
- Процессоры приложений обладают большими ресурсами и часто запускают операционную систему, вместо непосредственного выполнения целевого приложения при запуске.
- QEMU поддерживает эмуляцию различных моделей машин или плат для каждой архитектуры. Плата 'virt' не соответствует какому-либо конкретному реальному оборудованию, а предназначена исключительно для виртуальных машин.
- Мы по-прежнему будем рассматривать эту плату как bare-metal, как если бы писали операционную систему.

### 53.1 Подготовка к Rust

Прежде чем мы сможем начать выполнение кода на Rust, необходимо выполнить начальную инициализацию.

```
/**
 * Эта универсальная точка входа для образа. Она выполняет
 * операции, необходимые для подготовки загруженного образа к запуску.
 * В частности, она
 *
 * - настраивает MMU с идентичной картой виртуальных и физических
 * адресов и включает кэширование,
 * - включает поддержку плавающей точки,
 * - обнуляет секцию bss, используя регистры x25 и выше,
 * - подготавливает стек, указывая на секцию внутри образа,
 * - настраивает вектор исключений,
 * - переходит к функции Rust `main`.
 *
 * Она сохраняет регистры x0-x3 для точки входа Rust, так как они могут содержать
```

```

* параметры загрузки.
*/
.section .init.entry, "ax"
.global entry
entry:
/*
 * Загружает и применяет конфигурацию управления памятью, подготавливаясь к
 * включению MMU и кэшей.
 */
adrp x30, idmap
msr ttbr0_el1, x30

mov_i x30, .Lmairval
msr mair_el1, x30

mov_i x30, .Ltcrvval
/* Скопировать поддерживаемый диапазон PA в TCR_EL1.IPS. */
mrs x29, id_aa64mmfr0_el1
bfi x30, x29, #32, #4

msr tcr_el1, x30

mov_i x30, .Lsctlrval

/*
 * Убедитесь, что все операции до этого момента завершены, затем
 * аннулируйте любые потенциально устаревшие локальные записи TLB перед
 * началом их использования.
 */
isb
tlbi vmalle1
ic iallu
dsb nsh
isb

/*
 * Настройте sctlr_el1 для включения MMU и кэша и не продолжайте
 * до тех пор, пока это не будет завершено.
 */
msr sctlr_el1, x30
isb

/* Отключить перехват доступа к плавающей точке в EL1. */
mrs x30, cracr_el1
orr x30, x30, #(0x3 << 20)
msr cracr_el1, x30
isb

/* Обнуление секции bss. */
adr_l x29, bss_begin
adr_l x30, bss_end

```

```

0: cmp x29, x30
b.hs 1f
stp xzr, xzr, [x29], #16
b 0b

1: /* Подготовка стека. */
 adr_l x30, boot_stack_end
 mov sp, x30

 /* Настройка вектора исключений. */
 adr x30, vector_table_el1
 msr vbar_el1, x30

 /* Вызов кода на Rust. */
 b1 main

 /* Бесконечный цикл ожидания прерываний. */
2: wfi
b 2b

```

Этот код находится в `src/bare-metal/aps/examples/src/entry.S`. Нет необходимости понимать это подробно — главное, что обычно требуется некоторый низкоуровневый настрой для соответствия ожиданиям Rust относительно системы.

- Это то же самое, что и для C: инициализация состояния процессора, обнуление BSS и настройка указателя стека.
  - BSS (block starting symbol, по историческим причинам) — это часть объектного файла, содержащая статически выделенные переменные, инициализированные нулюм. Они исключаются из образа, чтобы избежать пустой траты места на нули. Компилятор предполагает, что загрузчик позаботится об их обнулении.
- BSS может быть уже обнулён, в зависимости от того, как инициализируется память и загружается образ, но мы обнуляем её для уверенности.
- Необходимо включить MMU и кэш перед чтением или записью любой памяти. Если этого не сделать:
  - Несогласованные обращения вызовут ошибку. Мы собираем Rust-код для цели `aarch64-unknown-none` с установленным `+strict-align`, чтобы предотвратить генерацию компилятором несогласованных обращений, так что в этом случае всё должно быть в порядке, но в общем случае это не обязательно так.
  - Если бы это выполнялось в виртуальной машине, это могло привести к проблемам с когерентностью кэша. Проблема заключается в том, что виртуальная машина обращается к памяти напрямую с отключённым кэшем, в то время как хост имеет кэшируемые алиасы на ту же область памяти. Даже если хост явно не обращается к памяти, спекулятивные обращения могут привести к заполнению кэша, и тогда изменения с одной или другой стороны будут потеряны при очистке кэша или включении кэша виртуальной машиной.

(Кэш индексируется по физическому адресу, а не по VA или IPA.)
- Для упрощения мы используем жёстко заданную таблицу страниц (см. `idmap.S`), которая идентично отображает первые 1 ГиБ адресного пространства для устройств, следующие 1 ГиБ для DRAM и ещё 1 ГиБ выше для дополнительных устройств. Это соответствует расположению памяти, используемому QEMU.
- Мы также настраиваем вектор исключений (`vbar_el1`), о котором поговорим подробнее позже.
- Все примеры сегодня днём предполагают выполнение на уровне исключений 1 (EL1). Если необходимо запускаться на другом уровне исключений, потребуется соответствующим образом изменить `entry.S`.

## 53.2 Встроенная ассемблерная вставка

Иногда необходимо использовать ассемблер для выполнения операций, которые невозможно реализовать на Rust. Например, чтобы выполнить HVC (вызов гипервизора) для указания прошивке выключить систему:

```
#! [no_main]
#! [no_std]

use core::arch::asm;
use core::panic::PanicInfo;

mod asm;
mod exceptions;

const PSCI_SYSTEM_OFF: u32 = 0x84000008;

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn main(_x0: u64, _x1: u64, _x2: u64, _x3: u64) {
 // БЕЗОПАСНОСТЬ: используются только объявленные регистры, и не выполняется
 // никаких операций с памятью.
 unsafe {
 asm!("hvc #0",
 inout("w0") PSCI_SYSTEM_OFF => _,
 inout("w1") 0 => _,
 inout("w2") 0 => _,
 inout("w3") 0 => _,
 inout("w4") 0 => _,
 inout("w5") 0 => _,
 inout("w6") 0 => _,
 inout("w7") 0 => _,
 options(nomem, nostack)
);
 }

 loop {}
}
```

(Если вы действительно хотите это сделать, используйте crate [smccc](#), который содержит обёртки для всех этих функций.)

- PSCI — это Arm Power State Coordination Interface, стандартный набор функций для управления состояниями питания системы и процессора, среди прочего. Он реализован прошивкой EL3 и гипервизорами на многих системах.
- Синтаксис `0 => _` означает инициализировать регистр значением 0 перед выполнением встроенным ассемблерным кодом и игнорировать его содержимое после. Необходимо использовать `inout` вместо `in`, поскольку вызов потенциально может повредить содержимое регистров.
- Эта функция `main` должна быть помечена как `#[unsafe(no_mangle)]` и иметь объявление `extern "C"`, поскольку она вызывается из нашей точки входа в `entry.S`.
  - Просто `[no_mangle]` было бы достаточно, но RFC3325 использует эту нотацию, чтобы привлечь внимание рецензентов к атрибутам, которые могут вызвать неопределённое поведение при неправильном использовании.
- `_x0-_x3` — это значения регистров `x0-x3`, которые традиционно используются загрузчиком-

Загрузчик для передачи таких данных, как указатель на дерево устройств. Согласно стандартному соглашению о вызовах aarch64 (которое задаёт extern "C" ), регистры x0 – x7 используются для первых восьми аргументов, передаваемых функции, поэтому entry.S не требуется выполнять ничего особенного, кроме как гарантировать, что эти регистры не изменятся.

- Запустите пример в QEMU с помощью команды make qemu\_psc в каталоге src/bare-metal/aps/examples.

### 53.3 Нестабильный доступ к памяти для MMIO

- Используйте pointer::read\_volatile и pointer::write\_volatile.
- Никогда не храните ссылку на область памяти, к которой осуществляется доступ с помощью этих методов. Rust может читать из (или записывать в, для &mut) ссылку в любое время.
- Используйте &хаш для получения полей структур без создания промежуточной ссылки.

```
const SOME_DEVICE_REGISTER: *mut u64 = 0x800_0000 as _;
// БЕЗОПАСНОСТЬ: Некоторое устройство отображено по этому адресу.
unsafe {
 SOME_DEVICE_REGISTER.write_volatile(0xff);
 SOME_DEVICE_REGISTER.write_volatile(0x80);
 assert_eq!(SOME_DEVICE_REGISTER.read_volatile(), 0xaa);
}
```

- Доступ с использованием volatile: операции чтения или записи могут иметь побочные эффекты, поэтому необходимо предотвратить переупорядочивание, дублирование или устранение этих операций компилятором или аппаратным обеспечением.
  - Обычно, если вы сначала записываете, а затем читаете, например, через изменяющую ссылку, компилятор может предположить, что прочитанное значение совпадает с только что записанным, и не выполнять фактическое чтение из памяти.
- Некоторые существующие crates для доступа к аппаратуре с использованием volatile действительно хранят ссылки, но это является небезопасным. Всякий раз, когда существует ссылка, компилятор может выбрать её разыменование.
- Используйте &хаш для получения указателей на поля структуры из указателя на структуру.
- Для совместимости со старыми версиями Rust можно использовать макрос addr\_of!

### 53.4 Напишем драйвер UART

В виртуальной машине QEMU 'virt' имеется UART PL011, поэтому давайте напишем драйвер для него.

```
const FLAG_REGISTER_OFFSET: usize = 0x18;
const FR_BUSY: u8 = 1 << 3;
const FR_TXFF: u8 = 1 << 5;

/// Минимальный драйвер для UART PL011.
#[derive(Debug)]
pub struct Uart {
 base_address: *mut u8,
}

impl Uart {
 /// Создаёт новый экземпляр драйвера UART для устройства PL011 по заданному базовому адресу.
 ///
 /// # Безопасность
```

```

/// Указанный базовый адрес должен указывать на 8 MMIO регистров управления
/// устройства PL011, которые должны быть отображены в адресное пространство процесса
/// как память устройства и не иметь других алиасов.
pub unsafe fn new(base_address: *mut u8) -> Self {
 Self { base_address }
}

// Записывает один байт в UART.
pub fn write_byte(&self, byte: u8) {
 // Ожидание освобождения места в буфере передачи (TX).
 while self.read_flag_register() & FR_TXFF != 0 {}

 // БЕЗОПАСНОСТЬ: Известно, что базовый адрес указывает на управляющие
 // регистры устройства PL011, корректно отображённого в память.
 unsafe {
 // Запись в буфер передачи (TX).
 self.base_address.write_volatile(byte);
 }

 // Ожидание, пока UART перестанет быть занятым.
 while self.read_flag_register() & FR_BUSY != 0 {}
}

fn read_flag_register(&self) -> u8 {
 // БЕЗОПАСНОСТЬ: Известно, что базовый адрес указывает на управляющие
 // регистры устройства PL011, корректно отображённого в память.
 unsafe { self.base_address.add(FLAG_REGISTER_OFFSET).read_volatile() }
}
}

```

- Обратите внимание, что `Uart::new` является `unsafe`, в то время как остальные методы безопасны. Это связано с тем, что если вызывающий `Uart::new` гарантирует выполнение требований безопасности (то есть существует только один экземпляр драйвера для данного UART и отсутствуют другие ссылки на его адресное пространство), то вызов `write_byte` впоследствии всегда безопасен, поскольку можно считать, что необходимые предусловия выполнены.
- Мы могли бы сделать наоборот (сделав `new` безопасным, а `write_byte` — небезопасным), но это было бы значительно менее удобно, поскольку каждое место, вызывающее `write_byte`, должно было бы учитывать вопросы безопасности.
- Это распространённый шаблон для написания безопасных обёрток над небезопасным кодом: перенос бремени доказательства корректности с множества мест на меньшее число.

### 53.4.1 Дополнительные трейты

Мы вывели трейд `Debug`. Было бы полезно реализовать ещё несколько трейтов . Используйте `core::fmt::{self, Write}`:

```

impl Write для Uart {
 fn write_str(&mut self, s: &str) -> fmt::Result { для c in s.
 as_bytes() { self.write
 _byte(*c); }
 }
}

```

```

 }
 Ok(())
 }

// БЕЗОПАСНОСТЬ: `Uart` содержит только указатель на память устройства , к которой
// можно // обращаться из любого контекста .
unsafe impl Send for Uart {}

```

- Реализация Write позволяет использовать макросы write! и writeln! с нашим типом Uart
- Send является авто-трейтом, но не реализуется автоматически, поскольку не реализован для указателей.

### 53.4.2 Использование

Давайте напишем небольшую программу, используя наш драйвер для записи в последовательную консоль.

```

#!/[no_main]
#!/[no_std]

mod asm;
mod exceptions;
mod pl011_minimal;

use crate::pl011_minimal::Uart;
use core::fmt::Write;
use core::panic::PanicInfo;
use log::error;
use smccc::Hvc;
use smccc::psc::system_off;

/// Базовый адрес основного UART PL011 .
const PL011_BASE_ADDRESS: *mut u8 = 0x900_0000 as _;

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем .
#[unsafe(no_mangle)]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
 // БЕЗОПАСНОСТЬ: `PL011_BASE_ADDRESS` — базовый адрес устройства PL011 , и // ни-
 // кто другой не обращается к этому диапазону адресов .
 let mut uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };

 writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

 system_off::<Hvc>().unwrap();
}

```

- Как и в примере с встроенным ассемблером, эта функция main вызывается из нашего кода точки входа в файле entry.S . Подробности смотрите в заметках докладчика.
- Запустите пример в QEMU с помощью команды make qemu\_minimal в каталоге src/bare-metal/aps/examples .

## 53.5 Улучшенный драйвер UART

На самом деле у PL011 гораздо больше регистров, и добавление смещений для формирования указателей для доступа к ним подвержено ошибкам и затрудняет чтение. Кроме того, некоторые из них являются битовыми полями, к которым было бы удобно обращаться структурированным способом.

| Смещение Имя регистра Ширина |       |    |
|------------------------------|-------|----|
| 0x00                         | DR    | 12 |
| 0x04                         | RSR   | 4  |
| 0x18                         | FR    | 9  |
| 0x20                         | ILPR  | 8  |
| 0x24                         | IBRD  | 16 |
| 0x28                         | FBRD  | 6  |
| 0x2c                         | LCR_H | 8  |
| 0x30                         | CR    | 16 |
| 0x34                         | IFLS  | 6  |
| 0x38                         | IMSC  | 11 |
| 0x3c                         | RIS   | 11 |
| 0x40                         | MIS   | 11 |
| 0x44                         | ICR   | 11 |
| 0x48                         | DMACR | 3  |

- Также существуют некоторые ID-регистры, опущенные для краткости.

### 53.5.1 Bitflags

Крейт `bitflags` полезен для работы с битовыми флагами.

```
use bitflags::bitflags;

bitflags! {
 /// Флаги из регистра флагов UART.
 #[repr(transparent)]
 #[derive(Copy, Clone, Debug, Eq, PartialEq)]
 struct Flags: u16 {
 /// Разрешение на передачу.
 const CTS = 1 << 0;
 /// Набор данных готов.
 const DSR = 1 << 1;
 /// Обнаружение носителя данных.
 const DCD = 1 << 2;
 /// UART занят передачей данных.
 const BUSY = 1 << 3;
 /// Приёмный FIFO пуст.
 const RXFE = 1 << 4;
 /// Передающий FIFO заполнен.
 const TXFF = 1 << 5;
 /// Приёмный FIFO заполнен.
 const RXFF = 1 << 6;
 /// Передающий FIFO пуст.
 }
}
```

```

 const TXFE = 1 << 7;
 // Индикатор звонка.
 const RI = 1 << 8;
}

```

- Макрос `bitflags!` создаёт newtype, аналогичный `struct Flags(u16)`, вместе с набором реализаций методов для получения и установки флагов.

### 53.5.2 Несколько регистров

Мы можем использовать `struct` для представления расположения в памяти регистров UART.

```

#[repr(C, align(4))]
pub struct Registers {
 dr: u16,
 _reserved0: [u8; 2],
 rsr: ReceiveStatus,
 _reserved1: [u8; 19],
 fr: Flags,
 _reserved2: [u8; 6],
 ilpr: u8,
 _reserved3: [u8; 3],
 ibrd: u16,
 _reserved4: [u8; 2],
 fbrd: u8,
 _reserved5: [u8; 3],
 lcr_h: u8,
 _reserved6: [u8; 3],
 cr: u16,
 _reserved7: [u8; 3],
 ifls: u8,
 _reserved8: [u8; 3],
 imsc: u16,
 _reserved9: [u8; 2],
 ris: u16,
 _reserved10: [u8; 2],
 mis: u16,
 _reserved11: [u8; 2],
 icr: u16,
 _reserved12: [u8; 2],
 dmacr: u8,
 _reserved13: [u8; 3],
}

```

- `#[repr(C)]` указывает компилятору расположить поля структуры в порядке, соответствующем правилам языка C. Это необходимо для того, чтобы структура имела предсказуемое расположение, поскольку стандартное представление Rust позволяет компилятору, среди прочего, переставлять поля по своему усмотрению.

### 53.5.3 Драйвер

Теперь давайте используем новую структуру Registers в нашем драйвере.

```
/// Драйвер для PL011 UART.
#[derive(Debug)]
pub struct Uart {
 registers: *mut Registers,
}

impl Uart {
 /// Создаёт новый экземпляр драйвера UART для устройства PL011 с заданным набором ре-
 /// гистров.
 ///
 /// # Безопасность
 ///
 /// Указанный указатель должен указывать на 8 управляющих регистров MMIO устройства PL011,
 /// которые должны быть отображены в адресное пространство процесса как
 /// память устройства и не иметь других алиасов.
 pub unsafe fn new(registers: *mut Registers) -> Self {
 Self { registers }
 }

 /// Записывает один байт в UART.
 pub fn write_byte(&mut self, byte: u8) {
 // Ожидание освобождения места в буфере передачи (TX).
 while self.read_flag_register().contains(Flags::TXFF) {}

 // БЕЗОПАСНОСТЬ: известно, что self.registers указывает на управляющие ре-
 /// гистры // устройства PL011, корректно отображённые в память .
 unsafe {
 // Запись в буфер передачи (TX).
 (&raw mut (*self.registers).dr).write_volatile(byte.into());
 }

 // Ожидание, пока UART перестанет быть занятым.
 while self.read_flag_register().contains(Flags::BUSY) {}
 }

 /// Считывает и возвращает ожидающий байт или `None` , если данных не по-
 /// лучено.
 pub fn read_byte(&mut self) -> Option<u8> {
 if self.read_flag_register().contains(Flags::RXFE) {
 Нет
 } else {
 // БЕЗОПАСНОСТЬ: известно, что self.registers указывает на управляющие
 // регистры устройства PL011, корректно отображённые в память .
 let data = unsafe { (&raw const (*self.registers).dr).read_volatile() };
 // TODO: проверить условия ошибок в битах 8-11.
 Some(data as u8)
 }
 }
}
```

```

fn read_flag_register(&self) -> Flags {
 // БЕЗОПАСНОСТЬ: известно, что self.registers указывает на управляющие ре-
 // гистры // устройства PL011, корректно отображённые в память.
 unsafe { (&raw const (*self.registers).fr).read_volatile() }
}
}

```

- Обратите внимание на использование `&raw const` / `&raw mut` для получения указателей на отдельные поля без создания промежуточной ссылки, что было бы небезопасно.
- Этот пример не включён в слайды, поскольку он очень похож на пример `safe-mmio`, который представлен далее. Вы можете запустить его в QEMU с помощью `make qemu` в каталоге `src/bare-metal/aps/examples` при необходимости.

## 53.6 safle-mmio

Крейт `safe-mmio` предоставляет типы для обёртки регистров, которые можно безопасно читать или записывать.

| Чтение не имеет побочных эффектов |                               |           |
|-----------------------------------|-------------------------------|-----------|
| Невозможно прочитать              | Чтение имеет побочные эффекты |           |
| Невозможно записать               | ReadPure                      | ReadOnly  |
| Можно записывать                  | WriteOnly                     | ReadWrite |

```

use safe_mmio::fields::{ReadPure, ReadPureWrite, ReadWrite, WriteOnly};

#[repr(C, align(4))]
pub struct Registers {
 dr: ReadWrite<u16>,
 _reserved0: [u8; 2],
 rsr: ReadPure<ReceiveStatus>,
 _reserved1: [u8; 19],
 fr: ReadPure<Flags>,
 _reserved2: [u8; 6],
 ilpr: ReadPureWrite<u8>,
 _reserved3: [u8; 3],
 ibrd: ReadPureWrite<u16>,
 _reserved4: [u8; 2],
 fbrd: ReadPureWrite<u8>,
 _reserved5: [u8; 3],
 lcr_h: ReadPureWrite<u8>,
 _reserved6: [u8; 3],
 cr: ReadPureWrite<u16>,
 _reserved7: [u8; 3],
 ifls: ReadPureWrite<u8>,
 _reserved8: [u8; 3],
 imsc: ReadPureWrite<u16>,
 _reserved9: [u8; 2],
 ris: ReadPure<u16>,
 _reserved10: [u8; 2],
}

```

```

 mis: ReadPure<u16>,
 _reserved11: [u8; 2],
 icr: WriteOnly<u16>,
 _reserved12: [u8; 2],
 dmacr: ReadPureWrite<u8>,
 _reserved13: [u8; 3],
}

```

- Чтение `dr` имеет побочный эффект: оно извлекает байт из приёмного FIFO.
- Чтение `rsr`(и других регистров) не имеет побочных эффектов. Это «чистое» чтение.
- Существует множество различных crates, обеспечивающих безопасные абстракции для операций MMIO; мы рекомендуем использовать crate `safle-mmio`.
- Разница между `ReadPure` и `ReadOnly` (а также между `ReadPureWrite` и `ReadWrite`) заключается в том, может ли чтение регистра вызывать побочные эффекты, изменяющие состояние устройства. Например, чтение регистра данных извлекает байт из приёмного FIFO. `ReadPure` означает, что чтение не имеет побочных эффектов, оно исключительно считывает данные.

### 53.6.1 Драйвер

Теперь давайте используем новую структуру `Registers` в нашем драйвере.

```

use safe_mmio::{UniqueMmioPointer, field, field_shared};

/// Драйвер для PL011 UART.
#[derive(Debug)]
pub struct Uart<'a> {
 registers: UniqueMmioPointer<'a, Registers>,
}

impl<'a> Uart<'a> {
 /// Создаёт новый экземпляр драйвера UART для устройства PL011 с заданным набором ре-
 гистров.
 pub fn new(registers: UniqueMmioPointer<'a, Registers>) -> Self {
 Self { registers }
 }

 /// Записывает один байт в UART.
 pub fn write_byte(&mut self, byte: u8) {
 // Ожидание освобождения места в буфере передачи (TX).
 while self.read_flag_register().contains(Flags::TXFF) {}

 // Запись в буфер передачи (TX).
 field!(self.registers, dr).write(byte.into());

 // Ожидание, пока UART перестанет быть занятым.
 while self.read_flag_register().contains(Flags::BUSY) {}
 }

 /// Считывает и возвращает ожидающий байт или `None`, если данных не по-
 //лучено.
 pub fn read_byte(&mut self) -> Option<u8> {

```

```

 if self.read_flag_register().contains(Flags::RXFE) {
 Нет
 } else {
 let data = field!(self.registers, dr).read();
 // TODO: проверить условия ошибок в битах 8-11.
 Some(data as u8)
 }
}

fn read_flag_register(&self) -> Flags {
 field_shared!(self.registers, fr).read()
}
}

```

- Драйвер больше не требует использования unsafe-кода!
- UniqueMmioPointer является оболочкой вокруг необработанного указателя на MMIO-устройство или регистр. Вызывающая сторона UniqueMmioPointer::new гарантирует, что указатель действителен и уникален в течение заданного времени жизни, что позволяет предоставлять безопасные методы для чтения и записи полей.
- Обратите внимание, что Uart::new теперь является безопасным; UniqueMmioPointer::new наоборот, является небезопасным.
- Доступы к MMIO обычно представляют собой оболочку вокруг read\_volatile и write\_volatile, хотя на aarch64 они реализованы на ассемблере для обхода ошибки, при которой компилятор может генерировать инструкции, препятствующие виртуализации MMIO.
- Макросы field! и field\_shared! внутри используют &raw mut и &raw const для получения указателей на отдельные поля без создания промежуточной ссылки, что было бы небезопасно.
- field! требует изменяемую ссылку на UniqueMmioPointer и возвращает UniqueMmioPointer, позволяющий выполнять операции чтения с побочными эффектами
- field\_shared! работает с общей ссылкой либо на UniqueMmioPointer, либо на SharedMmioPointer. Он возвращает SharedMmioPointer, который разрешает только чистое

### 53.6.2 Использование

Давайте напишем небольшую программу с использованием нашего драйвера для записи в последовательную консоль и эхо входящих байтов.

```

#!/usr/bin/rustc
#![no_main]
#![no_std]

mod asm;
mod exceptions;
mod pl011;

use crate::pl011::Uart;
use core::fmt::Write;
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::error;
use safe_mmio::UniqueMmioPointer;
use smccc::Hvc;
use smccc::psc::system_off;

```

```

/// Базовый адрес основного UART PL011.
const PL011_BASE_ADDRESS: NonNull<pl011::Registers> =
 NonNull::new(0x900_0000 as _).unwrap();

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
 // БЕЗОПАСНОСТЬ: `PL011_BASE_ADDRESS` — базовый адрес устройства PL011, и //
 // никто другой не обращается к этому диапазону адресов.
 let mut uart = Uart::new(unsafe { UniqueMmioPointer::new(PL011_BASE_ADDRESS) });

 writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

 loop {
 if let Some(byte) = uart.read_byte() {
 uart.write_byte(byte);
 match byte {
 b'r' => uart.write_byte(b'\n'),
 b'q' => break,
 _ => continue,
 }
 }
 }

 writeln!(uart, "\n\nBye!").unwrap();
 system_off::<Hvc>().unwrap();
}

```

- Запустите пример в QEMU с помощью команды make qemu\_safemrios каталога src/bare-metal/aps/examples.

## 53.7 Логирование

Было бы полезно иметь возможность использовать макросы логирования из crates `log`. Мы можем сделать это, реализовав трейд `Log`.

```

use crate::pl011::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
 uart: SpinMutex<Option<Uart<'static>>,
}

impl Log for Logger {
 fn enabled(&self, _metadata: &Metadata) -> bool {
 true
 }
}

```

```

fn log(&self, record: &Record) {
 writeln!(

 self.uart.lock().as_mut().unwrap(),

 "[{}]", {}

 record.level(),

 record.args()
)
 .unwrap();
}

fn flush(&self) {}

/// Инициализирует UART-логгер.
pub fn init(

 uart: Uart<'static>,

 max_level: LevelFilter,

) -> Result<(), SetLoggerError> {
 LOGGER.uart.lock().replace(uart);

 log::set_logger(&LOGGER)?;
 log::set_max_level(max_level);
 Ok(())
}

```

- Первый unwrap в log успешно выполнится, поскольку мы инициализируем LOGGER до вызова set\_logger. Второй unwrap выполнится успешно, так как Uart::write\_str всегда возвращает Ok.

### 53.7.1 Использование

Необходимо инициализировать логгер перед его использованием.

```

#! [no_main]
#! [no_std]

mod asm;
mod exceptions;
mod logger;
mod p1011;

use crate::p1011::Uart;
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::{LevelFilter, error, info};
use safe_mmio::UniqueMmioPointer;
use smccc::Hvc;
use smccc::psc::system_off;

/// Базовый адрес основного UART PL011.
const PL011_BASE_ADDRESS: NonNull<p1011::Registers> =
 NonNull::new(0x900_0000 as _) .unwrap();

```

```

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
 // БЕЗОПАСНОСТЬ: `PL011_BASE_ADDRESS` — базовый адрес устройства PL011, и // никто другой не обращается к этому диапазону адресов.
 let uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS)) };
 logger::init(uart, LevelFilter::Trace).unwrap();

 info!("main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})");
 assert_eq!(x1, 42);

 system_off::<Hvc>().unwrap();
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
 error!("{}");
 system_off::<Hvc>().unwrap();
 loop {}
}

```

- Обратите внимание, что теперь наш обработчик `panic` может логировать детали сбоев.
- Запустите пример в QEMU с помощью команды `make qemu_logger` в каталоге `src/bare-metal/aps/examples`.

## 53.8 Исключения

AArch64 определяет таблицу векторов исключений с 16 записями для 4 типов исключений (синхронные, IRQ, FIQ, SError) из 4 состояний (текущий EL с SP0, текущий EL с SPx, нижний EL с использованием AArch64, нижний EL с использованием AArch32). Мы реализуем это на ассемблере, чтобы сохранить изменяемые регистры в стеке перед вызовом кода на Rust:

```

use log::error;
use smccc::Hvc;
use smccc::psci::system_off;

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
 error!("sync_exception_current");
 system_off::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
 error!("irq_current");
 system_off::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.

```

```

#[unsafe(no_mangle)]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
 error!("fiq_current");
 system_off::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
 error!("serr_current");
 system_off::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
 error!("sync_lower");
 system_off::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
 error!("irq_lower");
 system_off::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
 error!("fiq_lower");
 system_off::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
 error!("serr_lower");
 system_off::<Hvc>().unwrap();
}

```

- EL — уровень исключения; Все наши примеры сегодня днем выполняются в EL1.
- Для упрощения мы не различаем SP0 и SPx для исключений текущего EL, а также не разделяем AArch32 и AArch64 для исключений нижнего EL.
- В этом примере мы просто регистрируем исключение и выключаем питание, поскольку не ожидаем, что они действительно произойдут.
- Мы можем рассматривать обработчики исключений и основной контекст выполнения как разные потоки. [Send](#) и [Sync](#) контролируют, что мы можем разделять между ними, так же как и в случае с потоками. Например, если мы хотим разделить некоторое значение между обработчиками исключений и остальной частью программы, и оно является Send, но не Sync, то нам потребуется обернуть его во что-то вроде Mutex и поместить в static.

## 53.9 aarch64-rt

Крейт aarch64-rt предоставляет точку входа на ассемблере и вектор исключений, которые мы реализовали ранее. Нам достаточно пометить основную функцию макросом entry!.

Он также предоставляет макрос initial\_pagetable!, который позволяет определить начальную статическую таблицу страниц на Rust, а не на ассемблере, как мы делали ранее.

Мы также можем использовать драйвер UART из крейта arm-pl011-uart вместо того, чтобы писать собственный.

```
#! [no_main]
#! [no_std]

mod exceptions;

use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::fmt::Write;
use core::panic::PanicInfo;
use core::ptr::NonNull;
use smccc::Hvc;
use smccc::psc::system_off;

/// Базовый адрес основного UART PL011.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
 NonNull::new(0x900_0000 as _).unwrap();

/// Атрибуты для использования с памятью устройств в начальной идентичной карте.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
 .union(Attributes::ATTRIBUTE_INDEX_0)
 .union(Attributes::ACCESSED)
 .union(Attributes::UXN);

/// Атрибуты для использования с обычной памятью в начальной идентичной карте.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
 .union(Attributes::ATTRIBUTE_INDEX_1)
 .union(Attributes::INNER_SHAREABLE)
 .union(Attributes::ACCESSED)
 .union(Attributes::NON_GLOBAL);

initial_pagetable!({
 let mut idmap = [0; 512];
 // 1 GiB памяти устройства.
 idmap[0] = DEVICE_ATTRIBUTES.bits();
 // 1 GiB обычной памяти.
 idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
 // Ещё 1 GiB памяти устройства, начиная с 256 GiB.
 idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
 InitialPagetable(idmap)
});
```

```

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
 // БЕЗОПАСНОСТЬ: `PL011_BASE_ADDRESS` — базовый адрес устройства PL011, и //
 // никто другой не обращается к этому диапазону адресов.
 let mut uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS)) };

 writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

 system_off::<Hvc>().unwrap();
 panic!("system_off returned");
}

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
 system_off::<Hvc>().unwrap();
 loop {}
}

```

- Запустите пример в QEMU с помощью команды make qemu\_rtb каталога src/bare-metal/aps/examples.

## 53.10 Другие проекты

- oreboot
  - «coreboot без C».
  - Поддерживает x86, aarch64 и RISC-V.
  - Опирается на LinuxBoot, а не содержит множество драйверов самостоятельно.
- Руководство по Rust для RaspberryPi OS
  - Инициализация, драйвер UART, простой загрузчик, JTAG, уровни исключений, обработка исключений, таблицы страниц.
  - Некоторая нестабильность в обслуживании и инициализации кэша в Rust, не обязательно хороший пример для промышленного кода.
- cargo-call-stack
  - Статический анализ для определения максимального использования стека.
- Руководство по RaspberryPi OS запускает код на Rust до включения MMU и кэшей. Это будет читать и записывать память (например, стек). Однако это имеет проблемы, упомянутые в начале этой сессии, связанные с невыравненным доступом и когерентностью кэша.

## Глава 54

# Полезные crates

Рассмотрим несколько crates, которые решают распространённые задачи в программировании без операционной системы.

### 54.1 zerocopy

Crate `zerocopy` (из Fuchsia) предоставляет трейты и макросы для безопасного преобразования между последовательностями байтов и другими типами.

```
use zerocopy::{Immutable, IntoBytes};

#[repr(u32)]
#[derive(Debug, Default, Immutable, IntoBytes)]
enum RequestType {
 #[default]
 In = 0,
 Out = 1,
 Flush = 4,
}

#[repr(C)]
#[derive(Debug, Default, Immutable, IntoBytes)]
struct VirtioBlockRequest {
 request_type: RequestType,
 reserved: u32,
 sector: u64,
}

fn main() {
 let request = VirtioBlockRequest {
 request_type: RequestType::Flush,
 sector: 42,
 ..Default::default()
 };

 assert_eq!(
```

```

 request.as_bytes(),
 &[4, 0, 0, 0, 0, 0, 0, 0, 42, 0, 0, 0, 0, 0, 0, 0]
);
}

```

Это не подходит для MMIO (поскольку не использует `volatile` чтение и запись), но может быть полезно при работе со структурами, разделяемыми с аппаратным обеспечением, например, через DMA, или передаваемыми по внешнему интерфейсу.

- `FromBytes` может быть реализован для типов, для которых любой байтовый шаблон является допустимым и поэтому может безопасно преобразовываться из недоверенной последовательности байтов.
- Попытка вывести `FromBytes` для этих типов завершится неудачей, поскольку `RequestType` не использует все возможные значения `u32` в качестве дискриминантов, и поэтому не все байтовые шаблоны являются допустимыми.
- `zerocopy::byteorder` содержит типы для числовых примитивов с учётом порядка байтов.
- Запустите пример с помощью `cargo run` в каталоге `src/bare-metal/useful-crates/zerocopy-example/`. (Он не запустится в Playground из-за зависимости от `crate`.)

## 54.2 aarch64-paging

Crate `aarch64-paging` позволяет создавать таблицы страниц в соответствии с архитектурой виртуальной памяти AArch64.

```

use aarch64_paging::{
 idmap::IdMap,
 paging::{Attributes, MemoryRegion},
};

const ASID: usize = 1;
const ROOT_LEVEL: usize = 1;

// Создаём новую таблицу страниц с идентичным отображением.
let mut idmap = IdMap::new(ASID, ROOT_LEVEL);
// Отобразить область памяти размером 2 МБ как только для чтения.
idmap.map_range(
 &MemoryRegion::new(0x80200000, 0x80400000),
 Attributes::NORMAL | Attributes::NON_GLOBAL | Attributes::READ_ONLY,
).unwrap();
// Установить `TTBR0_EL1` для активации таблицы страниц.
idmap.activate();

```

- Это используется в Android для защищённой виртуальной машины прошивки.
- Нет простого способа запустить этот пример самостоятельно, поскольку он должен выполняться на реальном оборудовании или под QEMU.

## 54.3 buddy\_system\_allocator

`buddy_system_allocator` — crate, реализующий базовый аллокатор системы `buddy`. Он может использоваться как для реализации `GlobalAlloc` (с помощью `LockedHeap`), чтобы можно было использовать стандартный `crate alloc` (как мы видели ранее), так и для выделения другого адресного пространства (с помощью `FrameAllocator`). Например, мы можем захотеть выделить пространство MMIO для PCI BAR:

```

use buddy_system_allocator::FrameAllocator;
use core::alloc::Layout;

fn main() {
 let mut allocator = FrameAllocator::new();
 allocator.add_frame(0x200_0000, 0x400_0000);

 let layout = Layout::from_size_align(0x100, 0x100).unwrap();
 let bar = allocator
 .alloc_aligned(layout)
 .expect("Не удалось выделить MMIO область размером 0x100 байт");
 println!("Выделена MMIO область размером 0x100 байт по адресу {:#x}", bar);
}

• PCI BAR всегда имеют выравнивание, равное их размеру.
• Запустите пример с помощью cargo run в каталоге src/bare-metal/useful-crates/
allocator-example/. (Он не запустится в Playground из-за зависимости от crate.)

```

## 54.4 tinyvec

Иногда требуется структура, которую можно изменять по размеру, как Vec, но без выделения памяти в куче. **tinyvec** предоставляет именно это: вектор, основанный на массиве или срезе, который может быть статически выделен или размещён в стеке, отслеживает количество используемых элементов и вызывает панику при попытке использовать больше выделенного объёма.

```

use tinyvec::{ArrayVec, array_vec};

fn main() {
 let mut numbers: ArrayVec<[u32; 5]> = array_vec!(42, 66);
 println!("{} numbers:?", numbers);
 numbers.push(7);
 println!("{} numbers:?", numbers);
 numbers.remove(1);
 println!("{} numbers:?", numbers);
}

• tinyvec требует, чтобы тип элемента реализовывал Default для инициализации.
• Rust Playground включает tinyvec, поэтому этот пример будет корректно выполняться inline.

```

## 54.5 spin

`std::sync::Mutex` и другие примитивы синхронизации из `std::sync` недоступны в `core` или `alloc`. Как можно управлять синхронизацией или внутренней изменяемостью, например, для совместного использования состояния между разными CPU?

crate `spin` предоставляет эквиваленты многих из этих примитивов на основе спинлоков.

```

use spin::mutex::SpinMutex;

static COUNTER: SpinMutex<u32> = SpinMutex::new(0);

```

```
fn main() {
 dbg!(COUNTER.lock());
 *COUNTER.lock() += 2;
 dbg!(COUNTER.lock());
}
```

- Будьте осторожны, чтобы избежать взаимной блокировки при захвате блокировок в обработчиках прерываний.
- `spin` также содержит реализацию мьютекса с блокировкой по билету; эквиваленты `RwLock`, `Barrier` и `Once` из `std::sync`; и `Lazy` для ленивой инициализации.
- Крейт `once_cell` также содержит полезные типы для поздней инициализации с несколько иным подходом по сравнению с `spin::once::Once`.
- Rust Playgroung включает `spin`, поэтому этот пример будет корректно выполняться `inline`.

## Глава 55

# Bare-Metal на Android

Для сборки bare-metal Rust-бинарника в AOSP необходимо использовать правило `rust_ffi_static` Soong для компиляции вашего Rust-кода, затем `cc_binary` с использованием скрипта линковщика для создания самого бинарника, и затем `raw_binary` для преобразования ELF в raw-бинарник, готовый к запуску.

```
rust_ffi_static {
 name: "libvmbase_example",
 defaults: ["vmbase_ffi_defaults"],
 crate_name: "vmbase_example",
 srcs: ["src/main.rs"],
 rustlibs: [
 "libvmbase",
],
}

cc_binary {
 name: "vmbase_example",
 defaults: ["vmbase_elf_defaults"],
 srcs: [
 "idmap.S",
],
 static_libs: [
 "libvmbase_example",
],
 linker_scripts: [
 "image.ld",
 ":vmbase_sections",
],
}

raw_binary {
 name: "vmbase_example_bin",
 stem: "vmbase_example.bin",
 src: ":vmbase_example",
 enabled: false,
 target: {
```

```
 android_arm64: {
 enabled: true,
 },
 },
}
```

## 55.1 vmbase

Для виртуальных машин, работающих под управлением crosvm на aarch64, библиотека vmbase предоставляет скрипт компоновщика и полезные настройки по умолчанию для правил сборки, а также точку входа, ведение журнала консоли UART и другие функции.

```
#! [no_main]
#! [no_std]

use vmbase::{main, println};

main!(main);

pub fn main(arg0: u64, arg1: u64, arg2: u64, arg3: u64) {
 println!("Hello world");
}
```

- Макрос `main!` обозначает вашу основную функцию, вызываемую из точки входа `vmbase`.
- Точка входа `vmbase` обеспечивает инициализацию консоли и отправляет команду `PSCI_SYSTEM_OFF` для завершения работы виртуальной машины, если ваша основная функция возвращает управление.

# Глава 56

## Упражнения

Мы напишем драйвер для устройства реального времени PL031.

После выполнения упражнений вы можете ознакомиться с предоставленными решениями.

### 56.1 Драйвер RTC

В виртуальной машине QEMU aarch64 virt имеется часы реального времени PL031 по адресу 0x9010000. Для этого упражнения вам следует написать драйвер для данного устройства.

1. Используйте его для вывода текущего времени на последовательную консоль. Вы можете использовать crate `chrono` для форматирования даты и времени.
2. Используйте регистр совпадения и статус прерывания для активного ожидания до заданного времени, например, через 3 секунды. (Вызовите `core::hint::spin_loop` внутри цикла.)
3. Дополнительно, если у вас есть время: включите и обработайте прерывание, генерируемое совпадением RTC. Вы можете использовать драйвер из crate `arm-gic` для настройки контроллера прерываний Arm Generic Interrupt Controller.
  - Используйте прерывание RTC, которое подключено к GIC как `IntId::spi(2)`.
  - После включения прерывания вы можете перевести ядро в спящий режим с помощью `arm_gic::wfi()`, что приведёт к переходу ядра в спящий режим до получения прерывания.

Скачайте шаблон упражнения и найдите в каталоге `rtc` следующие файлы.

`src/main.rs`:

```
#!/usr/bin/rustc

mod exceptions;
mod logger;

use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_gic::gicv3::GicV3;
use arm_gic::gicv3::registers::{Gicd, GicrSgi};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::panic::PanicInfo;
```

```

use core::ptr::NonNull;
use log::{LevelFilter, error, info, trace};
use smccc::Hvc;
use smccc::psc::system_off;

/// Базовые адреса GICv3.
const GICD_BASE_ADDRESS: *mut Gicd = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut GicrSgi = 0x80A_0000 as _;

/// Базовый адрес основного UART PL011.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
 NonNull::new(0x900_0000 as _).unwrap();

/// Атрибуты для использования с памятью устройств в начальной идентичной карте.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
 .union(Attributes::ATTRIBUTE_INDEX_0)
 .union(Attributes::ACCESSED)
 .union(Attributes::UXN);

/// Атрибуты для использования с обычной памятью в начальной идентичной карте.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
 .union(Attributes::ATTRIBUTE_INDEX_1)
 .union(Attributes::INNER_SHAREABLE)
 .union(Attributes::ACCESSED)
 .union(Attributes::NON_GLOBAL);

initial_pagetable!({
 let mut idmap = [0; 512];
 // 1 GiB памяти устройства.
 idmap[0] = DEVICE_ATTRIBUTES.bits();
 // 1 GiB обычной памяти.
 idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
 // Ещё 1 GiB памяти устройства, начиная с 256 GiB.
 idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
 InitialPagetable(idmap)
});

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
 // БЕЗОПАСНОСТЬ: `PL011_BASE_ADDRESS` — базовый адрес устройства PL011, и //
 // никто другой не обращается к этому диапазону адресов.
 let uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS)) };
 logger::init(uart, LevelFilter::Trace).unwrap();

 info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

 // БЕЗОПАСНОСТЬ: `GICD_BASE_ADDRESS` и `GICR_BASE_ADDRESS` являются //
 // адресами распределителя и перераспределителя GICv3 соответственно, //
 // и // никто другой не обращается к этим диапазонам адресов.
 let mut gic =
 unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS, 1, false) };
}

```

```

gic.setup(0);

// TODO: Создать экземпляр драйвера RTC и вывести текущее время .

// TODO: Подождать 3 секунды .

system_off::<Hvc>().unwrap();
panic!("system_off returned");
}

```

```

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
 error!("{}");
 system_off::<Hvc>().unwrap();
 loop {}
}

```

*src/exceptions.rs* (изменения потребуются только для третьей части упражнения):

```

// Copyright 2023 Google LLC
//
// Лицензировано согласно Apache License, Version 2.0 ("Лицензия");
// использование этого файла разрешено только в соответствии с Лицензией. /
// Копию Лицензии можно получить по адресу
//
// http://www.apache.org/licenses/LICENSE-2.0
//
// Если это не требуется применимым законодательством или не согласовано в письменной
// форме, программное обеспечение // распространяется по Лицензии на условиях "КАК
// ЕСТЬ", // БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ ИЛИ УСЛОВИЙ, явных или подразумеваемых.
// См. Лицензию для получения конкретных условий, регулирующих разрешения и
// ограничения по Лицензии.

use arm_gic::gicv3::GicV3;
use log::{error, info, trace};
use smccc::Hvc;
use smccc::psc::system_off;

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
 error!("sync_exception_current");
 system_off::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
 trace!("irq_current");
 let intid =
 GicV3::get_and_acknowledge_interrupt().expect("Нет ожидающего прерывания");
 info!("IRQ {}({intid:?})");
}

```

```

}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
 error!("fiq_current");
 system_off:::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
 error!("serr_current");
 system_off:::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
 error!("sync_lower");
 system_off:::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
 error!("irq_lower");
 system_off:::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
 error!("fiq_lower");
 system_off:::<Hvc>().unwrap();
}

// БЕЗОПАСНОСТЬ: Нет другой глобальной функции с таким именем.
#[unsafe(no_mangle)]
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
 error!("serr_lower");
 system_off:::<Hvc>().unwrap();
}

```

*src/logger.rs* (вам не потребуется изменять этот файл):

```

// Copyright 2023 Google LLC
//
// Лицензировано согласно Apache License, Version 2.0 ("Лицензия");
// использование этого файла разрешено только в соответствии с Лицензией. /
// Копию Лицензии можно получить по адресу
//

```

```

// http://www.apache.org/licenses/LICENSE-2.0
//
// Если это не требуется применимым законодательством или не согласовано в письменной
// форме, программное обеспечение // распространяется по Лицензии на условиях "КАК
// ЕСТЬ", // БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ ИЛИ УСЛОВИЙ, явных или подразумеваемых.
// См. Лицензию для получения конкретных условий, регулирующих разрешения и
// ограничения по Лицензии.

use arm_pl011_uart::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
 uart: SpinMutex<Option<Uart<'static>>>,
}

impl Log for Logger {
 fn enabled(&self, _metadata: &Metadata) -> bool {
 true
 }

 fn log(&self, record: &Record) {
 writeln!(
 self.uart.lock().as_mut().unwrap(),
 "[{}]",
 record.level(),
 record.args()
)
 .unwrap();
 }

 fn flush(&self) {}
}

/// Инициализирует UART-логгер.
pub fn init(
 uart: Uart<'static>,
 max_level: LevelFilter,
) -> Result<(), SetLoggerError> {
 LOGGER.uart.lock().replace(uart);

 log::set_logger(&LOGGER)?;
 log::set_max_level(max_level);
 Ok(())
}

```

*Cargo.toml* (вам не нужно это менять):

[workspace]

```

[package]
name = "rtc"
version = "0.1.0"
edition = "2024"
publish = false

[dependencies]
aarch64-paging = { version = "0.9.1", default-features = false }
aarch64-rt = "0.2.1"
arm-gic = "0.4.0"
arm-pl011-uart = "0.3.1"
bitflags = "2.9.1"
chrono = { version = "0.4.41", default-features = false }
log = "0.4.27"
safe-mmio = "0.2.5"
smccc = "0.2.2"
spin = "0.10.0"
zerocopy = "0.8.26"

```

*build.rs* (вам не потребуется изменять этот файл):

```

// Copyright 2025 Google LLC
//
// Лицензировано согласно Apache License, Version 2.0 ("Лицензия");
// использование этого файла разрешено только в соответствии с Лицензией. /
// Копию Лицензии можно получить по адресу
//
// http://www.apache.org/licenses/LICENSE-2.0
//
// Если это не требуется применимым законодательством или не согласовано в письменной
// форме, программное обеспечение // распространяется по Лицензии на условиях "КАК
// ЕСТЬ", // БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ ИЛИ УСЛОВИЙ, явных или подразумеваемых.
// См. Лицензию для получения конкретных условий, регулирующих разрешения
// и // ограничения по Лицензии.

fn main() {
 println!("cargo:rustc-link-arg=-Timage.ld");
 println!("cargo:rustc-link-arg=-Tmemory.ld");
 println!("cargo:rerun-if-changed=memory.ld");
}

```

*memory.ld* (вам не нужно изменять этот файл):

```

/*
 * Авторские права © 2023 Google LLC
 *
 * Лицензировано на условиях Лицензии Apache, версия 2.0 ("Лицензия");
 * вы не можете использовать этот файл иначе, чем в соответствии с Лицензией.
 * Вы можете получить копию Лицензии по адресу
 *
 * https://www.apache.org/licenses/LICENSE-2.0
*/

```

```
* Если иное не предусмотрено применимым законодательством или не согласовано письменно , программное обеспечение
* распространяемое на условиях Лицензии , предоставляется «КАК ЕСТЬ» ,
* БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ ИЛИ УСЛОВИЙ , явных или подразумеваемых .
* См. Лицензию для получения сведений о правах и
* ограничениях в рамках Лицензии .
*/
```

## MEMORY

```
{
 image : ORIGIN = 0x40080000 , LENGTH = 2M
}
```

*Makefile* (вам не нужно изменять этот файл):

```
Авторские права © 2023 Google LLC

Лицензировано на условиях Лицензии Apache , версия 2.0 ("Лицензия");
вы не можете использовать этот файл иначе, чем в соответствии с Лицензией .
Вы можете получить копию Лицензии по адресу

http://www.apache.org/licenses/LICENSE-2.0

Если иное не предусмотрено применимым законодательством или не согласовано письменно , программное обеспечение
распространяемое на условиях Лицензии , предоставляется «КАК ЕСТЬ» ,
БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ ИЛИ УСЛОВИЙ , явных или подразумеваемых .
См. Лицензию для получения сведений о правах и
ограничениях в рамках Лицензии .
```

```
.PHONY: build qemu_minimal qemu_logger
```

```
all: rtc.bin
```

```
build:
 cargo build
```

```
rtc.bin: build
 cargo objcopy -- -O binary $@
```

```
qemu: rtc.bin
 qemu-system-aarch64 -machine virt,gic-version=3 -cpu max -serial mon:stdio -display
```

```
clean:
 cargo clean
 rm -f *.bin
```

*.cargo/config.toml* (вам не нужно это менять):

```
[build]
target = "aarch64-unknown-none" Запустите
код в QEMU с помощью команды make qemu .
```

## 56.2 Bare Metal Rust: послеобеденное занятие

### Драйвер RTC

(возврат к упражнению)

*main.rs:*

```
#!/usr/bin/rustc

// Импорт необходимых модулей
mod exceptions;
mod logger;
mod pl031;

use crate::pl031::Rtc;
use arm_gic::{IntId, Trigger, irq_enable, wfi};
use chrono::{TimeZone, Utc};
use core::hint::spin_loop;
use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_gic::gicv3::GicV3;
use arm_gic::gicv3::registers::{Gicd, GicrSgi};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::{LevelFilter, error, info, trace};
use smccc::Hvc;
use smccc::psc::system_off;

// Базовые адреса GICv3.
const GICD_BASE_ADDRESS: *mut Gicd = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut GicrSgi = 0x80A_0000 as _;

// Базовый адрес основного UART PL011.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
 NonNull::new(0x900_0000 as _).unwrap();

// Атрибуты для использования с памятью устройств в начальной идентичной карте.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
 .union(Attributes::ATTRIBUTE_INDEX_0)
 .union(Attributes::ACCESSED)
 .union(Attributes::UXN);

// Атрибуты для использования с обычной памятью в начальной идентичной карте.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
 .union(Attributes::ATTRIBUTE_INDEX_1)
 .union(Attributes::INNER_SHAREABLE)
 .union(Attributes::ACCESSED)
 .union(Attributes::NON_GLOBAL);

initial_pagetable!({
```

```

let mut idmap = [0; 512];
// 1 GiB памяти устройства.
idmap[0] = DEVICE_ATTRIBUTES.bits();
// 1 GiB обычной памяти.
idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
// Ещё 1 GiB памяти устройства, начиная с 256 GiB.
idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
InitialPagetable(idmap)
);

/// Базовый адрес PL031 RTC.
const PL031_BASE_ADDRESS: NonNull<pL031::Registers> =
 NonNull::new(0x901_0000 as _).unwrap();
/// IRQ, используемый PL031 RTC.
const PL031_IRQ: IntId = IntId::spi(2);

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
 // БЕЗОПАСНОСТЬ: `PL011_BASE_ADDRESS` — базовый адрес устройства PL011, и //
 // никто другой не обращается к этому диапазону адресов.
 let uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS)) };
 logger::init(uart, LevelFilter::Trace).unwrap();

 info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

 // БЕЗОПАСНОСТЬ: `GICD_BASE_ADDRESS` и `GICR_BASE_ADDRESS` являются
 // базовыми // адресами распределителя и перераспределителя GICv3 соответственно,
 // и // никто другой не обращается к этим диапазонам адресов.
 let mut gic =
 unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS, 1, false) };
 gic.setup(0);

 // БЕЗОПАСНОСТЬ: `PL031_BASE_ADDRESS` — базовый адрес устройства PL031, и //
 // никакие другие компоненты не обращаются к этому диапазону адресов.
 let mut rtc = unsafe { Rtc::new(UniqueMmioPointer::new(PL031_BASE_ADDRESS)) };
 let timestamp = rtc.read();
 let time = Utc.timestamp_opt(timestamp.into(), 0).unwrap();
 info!("RTC: {time}");

 GicV3::set_priority_mask(0xff);
 gic.set_interrupt_priority(PL031_IRQ, None, 0x80);
 gic.set_trigger(PL031_IRQ, None, Trigger::Level);
 irq_enable();
 gic.enable_interrupt(PL031_IRQ, None, true);

 // Ожидание 3 секунды без прерываний.
 let target = timestamp + 3;
 rtc.set_match(target);
 info!("Ожидание для {}", Utc.timestamp_opt(target.into(), 0).unwrap());
 trace!(
 "matched={}, interrupt_pending={}",

```

```

 rtc.matched(),
 rtc.interrupt_pending()
);
 while !rtc.matched() {
 spin_loop();
 }
 trace!(
 "matched={}, interrupt_pending={}",
 rtc.matched(),
 rtc.interrupt_pending()
);
 info! ("Завершено ожидание ");

 // Подождать ещё 3 секунды для прерывания.
 let target = timestamp + 6;
 info! ("Ожидание для {}", Utc.timestamp_opt(target.into(), 0).unwrap());
 rtc.set_match(target);
 rtc.clear_interrupt();
 rtc.enable_interrupt(true);
 trace!(
 "matched={}, interrupt_pending={}",
 rtc.matched(),
 rtc.interrupt_pending()
);
 while !rtc.interrupt_pending() {
 wfi();
 }
 trace!(
 "matched={}, interrupt_pending={}",
 rtc.matched(),
 rtc.interrupt_pending()
);
 info! ("Завершено ожидание ");

 system_off::<Hvc>().unwrap();
 panic!("system_off returned");
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
 error!("{}");
 system_off::<Hvc>().unwrap();
 loop {}
}

pl031.rs:

#[repr(C, align(4))]
pub struct Registers {
 /// Регистр данных
 dr: ReadPure<u32>,
 /// Регистр совпадения

```

```

 mr: ReadPureWrite<u32>,
 /// Регистр загрузки
 lr: ReadPureWrite<u32>,
 /// Регистр управления
 cr: ReadPureWrite<u8>,
 _reserved0: [u8; 3],
 /// Регистр установки или сброса маски прерываний
 imsc: ReadPureWrite<u8>, _
 reserved1: [u8; 3],
 /// Регистр необработанного статуса прерываний
 ris: ReadPure<u8>, _
 reserved2: [u8; 3],
 /// Регистр маскированного статуса прерываний
 mis: ReadPure<u8>, _
 reserved3: [u8; 3],
 /// Регистр очистки прерываний icr
 : только для записи<u8>, _зарезервировано4: [u8; 3],
 }

 /// Драйвер для часов реального времени PL031.
#[derive(Debug)]
pub struct Rtc<'a> {
 registers: UniqueMmioPointer<'a, Registers>,
}

impl<'a> Rtc<'a> {
 /// Создаёт новый экземпляр драйвера RTC для устройства PL031 с заданным набором регистров.
 pub fn new(registers: UniqueMmioPointer<'a, Registers>) -> Self {
 Self { registers }
 }

 /// Считывает текущее значение RTC.
 pub fn read(&self) -> u32 {
 field_shared!(&self.registers, dr).read()
 }

 /// Записывает значение совпадения. /// Когда значение RTC совпадает с этим, будет сгенерировано прерывание (если оно включено).
 pub fn set_match(&mut self, value: u32) {
 field!(&self.registers, mr).write(value);
 }

 /// Возвращает, совпадает ли регистр совпадения со значением RTC, независимо от того, включено ли прерывание.
 pub fn matched(&self) -> bool {
 let ris = field_shared!(&self.registers, ris).read();
 (ris & 0x01) != 0
 }
}

```

```
 /// Возвращает, ожидается ли в данный момент прерывание.
 ///
 /// Это должно быть истинно тогда и только тогда, когда `matched` возвращает
 /// true и прерывание маскировано.
 pub fn interrupt_pending(&self) -> bool {
 let mis = field_shared!(self.registers, mis).read();
 (mis & 0x01) != 0
 }

 /// Устанавливает или сбрасывает маску прерывания.
 ///
 /// Когда маска равна true, прерывание включено; /// когда она равна false, пре-
 ///рывание отключено.
 pub fn enable_interrupt(&mut self, mask: bool) {
 let imsc = if mask { 0x01 } else { 0x00 };
 field!(self.registers, imsc).write(imsc);
 }

 /// Очищает ожидающее прерывание, если оно имеется.
 pub fn clear_interrupt(&mut self) {
 field!(self.registers, icr).write(0x01);
 }
}
```

## **Часть XIII**

# **Конкурентность: утро**

# Глава 57

## Добро пожаловать в раздел «Конкурентность в Rust»

Rust полностью поддерживает конкурентность с использованием потоков ОС, мьютексов и каналов.

Система типов Rust играет важную роль в превращении многих ошибок конкурентности в ошибки времени компиляции. Это часто называют *бессстрашной конкурентностью*, поскольку можно полагаться на компилятор для обеспечения корректности во время выполнения.

### Расписание

Включая 10-минутные перерывы, эта сессия должна занять около 3 часов 20 минут. Она включает:

| Сегмент         | Продолжительность |
|-----------------|-------------------|
| Потоки          | 30 минут          |
| Каналы          | 20 минут          |
| Send и Sync —   | 15 минут          |
| Общее состояние | 30 минут          |
| Упражнения      | 1 час 10 минут    |

- Rust предоставляет доступ к инструментам ОС для конкурентности: потоки, синхронизирующие примитивы и др.
- Система типов обеспечивает безопасность конкурентности без использования специальных возможностей.
- Те же инструменты, которые помогают с «конкурентным» доступом в одном потоке (например, вызываемая функция, которая может изменять аргумент или сохранять ссылки на него для последующего чтения), защищают нас от проблем многопоточности.

# Глава 58

## Потоки

Этот раздел займет примерно 30 минут. В нем рассматриваются:

| Слайд                       | Продолжительность |
|-----------------------------|-------------------|
| Обычные потоки              | 15 минут          |
| Потоки с областью видимости | 15 минут          |

### 58.1 Обычные потоки

Потоки Rust работают аналогично потокам в других языках:

```
use std::thread;
используйте std::time::Duration;

fn main() {
 thread::spawn(|| {
 for i in 0..10 {
 println!("Count in thread: {}", i);
 thread::sleep(Duration::from_millis(5));
 }
 });

 for i in 0..5 {
 println!("Main thread: {}", i);
 thread::sleep(Duration::from_millis(5));
 }
}
```

- Создание новых потоков не задерживает автоматическое завершение программы в конце `main`.
- Паники потоков независимы друг от друга.
  - Паники могут содержать полезную нагрузку, которую можно распаковать с помощью `Any::downcast_ref`.

На этот слайд потребуется около 15 минут.

- Запустите пример.
  - Тайминг в 5 мс достаточно свободный, чтобы main и созданные потоки в основном работали синхронно.
  - Обратите внимание, что программа завершается до того, как созданный поток достигнет 10!
  - Это происходит потому, что main завершает программу, и созданные потоки недерживают её выполнение.  
\* При желании сравните с pthreads /C++ std::thread / boost::thread.
- Как нам дождаться завершения созданного потока?
- `thread::spawn` возвращает JoinHandle. Ознакомьтесь с документацией.
  - JoinHandle имеет метод `.join()`, который блокирует выполнение.
- Используйте `let handle = thread::spawn(...)` и затем `handle.join()` чтобы дождаться завершения потока и чтобы программа досчитала до 10.
- А что если мы хотим вернуть значение?
- Снова взглянем на документацию:
  - Замыкание `thread::spawn` возвращает T
  - JoinHandle `.join()` возвращает `thread::Result<T>`
- Используйте `Result` возвращаемое значение из `handle.join()` для доступа к возвращённому значению.
- Хорошо, а как насчёт другого случая?
  - Вызовите `panic` в потоке. Обратите внимание, что это не вызывает `panic` в `main`.
  - Получите доступ к полезной нагрузке `panic`. Сейчас самое время поговорить о Any.
- Теперь мы можем возвращать значения из потоков! А как насчёт передачи входных данных?
  - Захватите что-то по ссылке в замыкании потока.
  - Сообщение об ошибке указывает, что значение нужно переместить.
  - Переместите его внутрь, посмотрите, мы можем вычислить и затем вернуть производное значение.
- Если мы хотим взять значение взаймы?
  - Main завершает дочерние потоки при возврате, но другая функция просто вернётся и оставит их работать.
  - Это будет использование стека после возврата, что нарушает безопасность памяти!
  - Как этого избежать? См. следующий слайд.

## 58.2 Scoped-потоки

Обычные потоки не могут заимствовать из своей среды:

```
use std::thread;

fn foo() {
 let s = String::from("Hello");
 thread::spawn(|| {
 dbg!(s.len());
 });
}

fn main() {
```

```
 foo();
}
```

Однако для этого можно использовать scoped thread:

```
use std::thread;

fn foo() {
 let s = String::from("Hello");
 thread::scope(|scope| {
 scope.spawn(|| {
 dbg!(s.len());
 });
 });
}

fn main() {
 foo();
}
```

На этот слайд следует выделить около 13 минут.

- Причина в том, что когда функция `thread::scope` завершается, гарантируется присоединение всех потоков, поэтому они могут возвращать заимствованные данные.
- Применяются обычные правила заимствования Rust: либо один поток может заимствовать изменяемо, либо любое количество потоков — неизменяемо.

# Глава 59

## Каналы

Этот сегмент займет около 20 минут. В него входит:

| Слайд                               | Продолжительность |
|-------------------------------------|-------------------|
| Отправители и получатели — 10 минут |                   |
| Неограниченные каналы 2 минуты      |                   |
| Ограниченнные каналы 10 минут       |                   |

### 59.1 Отправители и Получатели

Каналы в Rust состоят из двух частей: `Sender<T>` и `Receiver<T>`. Две части связаны через канал, но видны только конечные точки.

```
use std::sync::mpsc;

fn main() {
 let (tx, rx) = mpsc::channel();

 tx.send(10).unwrap();
 tx.send(20).unwrap();

 println!("Получено: {:?}", rx.recv());
 println!("Получено: {:?}", rx.recv());

 let tx2 = tx.clone();
 tx2.send(30).unwrap();
 println!("Получено: {:?}", rx.recv());
}
```

На этот слайд потребуется примерно 9 минут.

- `mpsc` означает Multi-Producer, Single-Consumer. `Sender` и `SyncSender` реализуют `Clone` (что позволяет создавать несколько производителей), но `Receiver` — нет.
- `send()` и `recv()` возвращают `Result`. Если они возвращают `Err`, это означает, что соответствующий `Sender` или `Receiver` был уничтожен, и канал закрыт.

## 59.2 Неограниченные каналы

Вы получаете неограниченный и асинхронный канал с помощью `mpsc::channel()`:

```
use std::sync::mpsc;
use std::thread;
используйте std::time::Duration;

fn main() {
 let (tx, rx) = mpsc::channel();

 thread::spawn(move || {
 let thread_id = thread::current().id();
 for i in 0..10 {
 tx.send(format!("Message {}", i)).unwrap();
 println!("{}: sent Message {}", thread_id, i);
 }
 println!("{}: done", thread_id);
 });
 thread::sleep(Duration::from_millis(100));

 for msg in rx.iter() {
 println!("Main: got {}", msg);
 }
}
```

Этот слайд рассчитан примерно на 2 минуты.

- Неограниченный канал выделяет столько памяти, сколько необходимо для хранения ожидающих сообщений. Метод `send()` не блокирует вызывающий поток.
- Вызов `send()` приведёт к ошибке (поэтому он возвращает `Result`), если канал закрыт. Канал считается закрытым, когда приёмник уничтожается.

## 59.3 Ограниченнные каналы

В случае ограниченных (синхронных) каналов `send()` может блокировать текущий поток:

```
use std::sync::mpsc;
use std::thread;
используйте std::time::Duration;

fn main() {
 let (tx, rx) = mpsc::sync_channel(3);

 thread::spawn(move || {
 let thread_id = thread::current().id();
 for i in 0..10 {
 tx.send(format!("Message {}", i)).unwrap();
 println!("{}: sent Message {}", thread_id, i);
 }
 println!("{}: done", thread_id);
 });
}
```

```
 thread::sleep(Duration::from_millis(100));

 for msg in rx.iter() {
 println!("Main: got {}", msg);
 }
}
```

На этот слайд следует отвести около 8 минут.

- Вызов `send()` заблокирует текущий поток до тех пор, пока в канале не появится место для нового сообщения. Поток может быть заблокирован бесконечно, если нет никого, кто читает из канала.
- Как и в случае с неограниченными каналами, вызов `send()` приведёт к ошибке, если канал закрыт.
- Канал с ограничением размера, равным нулю, называется «каналом свидания». Каждый вызов `send` будет блокировать текущий поток до тех пор, пока другой поток не вызовет `recv()`.

# Глава 60

## Send и Sync

Этот раздел займет около 15 минут. В него входит:

| Слайд            | Продолжительность |
|------------------|-------------------|
| Маркерные трейты | 2 минуты          |
| Send             | 2 минуты          |
| Sync             | 2 минуты          |
| Примеры          | 10 минут          |

### 60.1 Маркерные трейты

Как Rust узнаёт, что нужно запретить совместный доступ между потоками? Ответ содержится в двух трейтах:

- **Send**: тип Тявляется Send, если безопасно перемещать Т через границу потока.
- **Sync**: тип Тявляется Sync, если безопасно перемещать &Т через границу потока.

Send и Sync — это небезопасные трейты. Компилятор автоматически выведет их для ваших типов, если они содержат только Send и Sync типы. Вы также можете реализовать их вручную, если уверены в корректности такой реализации.

Этот слайд рассчитан примерно на 2 минуты.

- Эти трейты можно рассматривать как маркеры, указывающие, что тип обладает определёнными свойствами безопасности потоков.
- Они могут использоваться в ограничениях обобщений как обычные трейты.

### 60.2 Send

Тип Тявляется Send, если безопасно передавать значение Т в другой поток.

Передача владения в другой поток означает, что деструкторы будут выполняться в этом потоке. Вопрос заключается в том, когда можно выделить значение в одном потоке и освободить его в другом.

Этот слайд рассчитан примерно на 2 минуты.

Например, соединение с библиотекой SQLite должно использоваться только из одного потока.

## 60.3 Sync

Тип `T` является `Sync`, если безопасно одновременно обращаться к значению `T` из нескольких потоков.

Более точно, определение звучит так:

`T` является `Sync` тогда и только тогда, когда `&T` является `Send`

Этот слайд рассчитан примерно на 2 минуты.

Это утверждение является краткой формулировкой того, что если тип безопасен для совместного использования в потоках, то также безопасно передавать ссылки на него между потоками.

Это связано с тем, что если тип является `Sync`, это означает, что он может быть разделён между несколькими потоками без риска гонок данных или других проблем синхронизации, поэтому его безопасно перемещать в другой поток.

Ссылка на тип также безопасна для перемещения в другой поток, поскольку данные, на которые она ссылается, могут быть безопасно доступны из любого потока.

## 60.4 Примеры

### Send + Sync

Большинство типов, с которыми вы сталкиваетесь, являются `Send + Sync`:

- `i8, f32, bool, char, &str, ...`
- `(T1, T2), [T; N], &[T], struct { x: T } , ...`
- `String, Option<T>, Vec<T>, Box<T>, ...`
- `Arc<T>`: Явно потокобезопасен за счёт атомарного подсчёта ссылок.
- `Mutex<T>`: Явно потокобезопасен за счёт внутренней блокировки.
- `mpsc::Sender<T>`: Начиная с версии 1.72.0.
- `AtomicBool, AtomicU8, ...`: Использует специальные атомарные инструкции.

Обобщённые типы обычно являются `Send + Sync`, когда параметры типа `Send + Sync`.

### Send + !Sync

Эти типы могут быть перемещены в другие потоки, но они не являются потокобезопасными. Обычно из-за внутренней изменяемости:

- `mpsc::Receiver<T>`
- `Cell<T>`
- `RefCell<T>`

### !Send + Sync

Эти типы безопасны для доступа (через разделяемые ссылки) из нескольких потоков, но не могут быть перемещены в другой поток:

- `MutexGuard<T: Sync>` : использует примитивы уровня ОС, которые должны быть освобождены в том потоке, который их создал. Однако уже заблокированный мьютекс позволяет читать защищённую переменную любым потоком, с которым разделён `guard`.

## **!Send + !Sync**

Эти типы не являются потокобезопасными и не могут быть перемещены в другие потоки:

- `Rc<T>` : каждый `Rc<T>` имеет ссылку на `RcBox<T>`, который содержит неатомарный счётчик ссылок.
- `*const T`, `*mut T` : Rust предполагает, что сырье указатели могут иметь особые особенности конкурентности.

## Глава 61

# Общее состояние

Этот раздел займет примерно 30 минут. В нем рассматриваются:

| Слайд  | Продолжительность |
|--------|-------------------|
| Arc    | 5 минут           |
| Mutex  | 15 минут          |
| Пример | 10 минут          |

### 61.1 Arc

`Arc<T>` позволяет совместное владение только для чтения через `Arc::clone`:

```
use std::sync::Arc;
use std::thread;

/// Структура, которая выводит, какой поток её уничтожает.
#[derive(Debug)]
struct WhereDropped(Vec<i32>);

impl Drop для WhereDropped {
 fn drop(&mut self) {
 println!("Уничтожено потоком {:?}", thread::current().id())
 }
}

fn main() {
 let v = Arc::new(WhereDropped(vec![10, 20, 30]));
 let mut handles = Vec::new();
 for i in 0..5 {
 let v = Arc::clone(&v);
 handles.push(thread::spawn(move || {
 // Сон от 0 до 500 мс.
 std::thread::sleep(std::time::Duration::from_millis(500 - i * 100));
 let thread_id = thread::current().id();
 }));
 }
}
```

```

 println!("{}thread_id:{}: {}v:{}");
 }));
}

// Теперь только созданные потоки будут владеть клонами `v`.
drop(v);

// Когда последний созданный поток завершится, он освободит содержимое `v`.
handles.into_iter().for_each(|h| h.join().unwrap());
}

```

На этот слайд отводится примерно 5 минут.

- Arc означает «Atomic Reflerence Counted» — потокобезопасную версию Rc, которая использует атомарные операции.
- Arc<T> реализует Clone независимо от того, реализует ли это T. Он реализует Send и Sync только если T реализует их оба.
- Arc::clone() имеет стоимость атомарных операций, которые выполняются, но после этого использование T бесплатно.
- Будьте осторожны с циклическими ссылками — Arcne использует сборщик мусора для их обнаружения.
  - std::sync::Weak может помочь.

## 61.2 Mutex

**Mutex<T>** обеспечивает взаимное исключение и позволяет изменять T через интерфейс только для чтения (другая форма внутренней изменяемости):

```

use std::sync::Mutex;

fn main() {
 let v = Mutex::new(vec![10, 20, 30]);
 println!("v: {:?}", v.lock().unwrap());

 {
 let mut guard = v.lock().unwrap();
 guard.push(40);
 }

 println!("v: {:?}", v.lock().unwrap());
}

```

Обратите внимание, что существует универсальная реализация **impl<T: Send> Sync for Mutex<T>**.

На этот слайд потребуется примерно 14 минут.

- Mutex в Rust представляет собой коллекцию с одним элементом — защищёнными данными.
  - Невозможно забыть захватить мьютекс перед доступом к защищённым данным.
- Вы можете получить &mut T из &Mutex<T> путём захвата блокировки. MutexGuard гарантирует, что &mut T не будет существовать дольше, чем удерживается блокировка.
- Mutex<T> реализует как Send, так и Sync только если T реализует Send.
- Аналог с блокировкой для чтения-записи: RwLock.
- Почему lock() возвращает Result?

- Если поток, который владел Mutex, завершился с паникой, то Mutex становится «отравленным» (poisoned), чтобы сигнализировать, что защищённые им данные могут находиться в неконсистентном состоянии. Вызов lock() на отправленном mutex приводит к ошибке `PoisonError`. Вы можете вызвать `into_inner()` на ошибке, чтобы восстановить данные независимо от этого.

## 61.3 Пример

Рассмотрим Arcs Mutex действии:

```
use std::thread;
// используйте std::sync::{Arc, Mutex};

fn main() {
 let v = vec![10, 20, 30];
 let mut handles = Vec::new();
 for i in 0..5 {
 handles.push(thread::spawn(|| {
 v.push(10 * i);
 println!("v: {:?}", v);
 }));
 }

 handles.into_iter().for_each(|h| h.join().unwrap());
}
```

На этот слайд следует отвести около 8 минут.

Возможное решение:

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
 let v = Arc::new(Mutex::new(vec![10, 20, 30]));
 let mut handles = Vec::new();
 for i in 0..5 {
 let v = Arc::clone(&v);
 handles.push(thread::spawn(move || {
 let mut v = v.lock().unwrap();
 v.push(10 * i);
 println!("v: {:?}", v);
 }));
 }

 handles.into_iter().for_each(|h| h.join().unwrap());
}
```

Значимые части:

- v обернут в Arcs Mutex, поскольку их задачи ортогональны.
  - Оборачивание Mutex в Arc — распространённый шаблон для совместного использования изменяемого состояния между потоками.

- `v: Arc<_>` нужно клонировать, чтобы создать новую ссылку для каждого нового порождаемого потока.  
Обратите внимание, что `to` был добавлен в сигнатуру лямбда-выражения.
- Блоки вводятся для максимально узкого ограничения области видимости `LockGuard`.

## Глава 62

# Упражнения

Этот раздел займет примерно 1 час 10 минут. В нём содержится:

| Слайд                                     | Продолжительность |
|-------------------------------------------|-------------------|
| Обедающие философы                        | 20 минут          |
| Многопоточный проверщик ссылок — 20 минут |                   |
| Решения                                   | 30 минут          |

### 62.1 Обед философов

Проблема обедающих философов — классическая задача в области конкурентности:

Пять философов обедают вместе за одним столом. У каждого философа есть своё место за столом. Между каждой тарелкой находится одна палочка для еды. Подаваемое блюдо — спагетти, для поедания которых требуются две палочки. Каждый философ может поочерёдно думать и есть. Более того, философ может есть спагетти только тогда, когда у него есть обе палочки — левая и правая. Таким образом, две палочки для еды будут доступны только тогда, когда их два ближайших соседа думают, а не едят. После того как отдельный философ заканчивает есть, он кладёт обе палочки.

Для этого упражнения потребуется локальная установка Cargo. Скопируйте приведённый ниже код в файл с именем `src/main.rs`, заполните пропуски и убедитесь, что `cargo run` не приводит к взаимной блокировке:

```
use std::sync::{Arc, Mutex, mpsc};
use std::thread;
исользуйте std::time::Duration;

struct Chopstick;

struct Philosopher {
 name: String,
 // left_chopstick: ...
 // right_chopstick: ...
 // thoughts: ...
}
```

```

impl Philosopher {
 fn think(&self) {
 self.thoughts
 .send(format!("Eureka! {} has a new idea!", & self.name))
 .unwrap();
 }

 fn eat(&self) {
 // Pick up chopsticks...
 println!("{} is eating...", & self.name);
 thread::sleep(Duration::from_millis(10));
 }
}

static PHILOSOPHERS: [&[&str]] =
 &["Сократ", "Гипатия", "Платон", "Аристотель", "Пифагор"];

fn main() {
 // Создать палочки для еды

 // Создать философов

 // Заставить каждого из них думать и есть по 100 раз

 // Вывести их мысли
}

```

Вы можете использовать следующий Cargo.toml:

```

[package]
name = "dining-philosophers"
version = "0.1.0"
edition = "2024"

```

Этот слайд должен занять примерно 20 минут.

- Поощряйте студентов сначала сосредоточиться на реализации решения, которое «в основном» работает.
- Взаимная блокировка в самом простом решении является общей проблемой конкурентности и подчёркивает, что Rust не предотвращает автоматически такого рода ошибки.

## 62.2 Многопоточный проверщик ссылок

Давайте применим наши новые знания для создания многопоточного проверяющего ссылок. Он должен начинать с веб-страницы и проверять, что ссылки на странице действительны. Он должен рекурсивно проверять другие страницы на том же домене и продолжать это делать, пока все страницы не будут проверены.

Для этого вам понадобится HTTP-клиент, такой как `reqwest`. Вам также потребуется способ находить ссылки, мы можем использовать `scraper`. Наконец, нам понадобится механизм обработки ошибок, для этого мы используем `thiserror`.

Создайте новый проект Cargo и добавьте `reqwest` в качестве зависимости с помощью команды:

```
cargo new link-checker
```

```
cd link-checker
cargo add --features blocking,rustls-tls reqwest
cargo add scraper
cargo add thiserror
```

Если команда cargo add выдаёт ошибку error: no such subcommand , отредактируйте файл Cargo .toml вручную. Добавьте зависимости, указанные ниже.

Вызовы cargo add добавляют файл Cargo.toml до следующего вида:

```
[package]
name = "link-checker"
version = "0.1.0"
edition = "2024"
publish = false

[dependencies]
reqwest = { version = "0.11.12", features = ["blocking", "rustls-tls"] }
scraper = "0.13.0"
thiserror = "1.0.37"
```

Теперь вы можете загрузить стартовую страницу. Попробуйте с небольшим сайтом, например, <https://www.google.org/>.

Ваш файл src/main.rs должен выглядеть примерно так:

```
use reqwest::Url;
use reqwest::blocking::Client;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
 #[error("request error: {0}")]
 ReqwestError(#[from] reqwest::Error),
 #[error("bad http response: {0}")]
 BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
 url: Url,
 extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>, Error> {
 println!("Checking {}", command.url);
 let response = client.get(command.url.clone()).send()?;
 if !response.status().is_success() {
 return Err(Error::BadResponse(response.status().to_string()));
 }

 let mut link_urls = Vec::new();
 if !command.extract_links {
 return Ok(link_urls);
 }

 let html = response.text()?;
 let document = Html::parse_document(&html);
 let links = document.select(<a>);

 for link in links {
 let href = link.value().attr("href").unwrap();
 link_urls.push(href.to_string());
 }
}
```

```

 }

let base_url = response.url().to_owned();
let body_text = response.text()?;
let document = Html::parse_document(&body_text);

let selector = Selector::parse("a").unwrap();
let href_values = document
 .select(&selector)
 .filter_map(|element| element.value().attr("href"));
for href in href_values {
 match base_url.join(href) {
 Ok(link_url) => {
 link_urls.push(link_url);
 }
 Err(err) => {
 println!("На {base_url:#}: проигнорирован неразборчивый {href:#}: {err}");
 }
 }
}
Ok(link_urls)
}

fn main() {
 let client = Client::new();
 let start_url = Url::parse("https://www.google.org").unwrap();
 let crawl_command = CrawlCommand{ url: start_url, extract_links: true };
 match visit_page(&client, &crawl_command) {
 Ok(links) => println!("Links: {links:#?}"),
 Err(err) => println!("Could not extract links: {err:#}"),
 }
}

```

Запустите код в файле `src/main.rs` помошью команды  
`cargo run`

## Задания

- Используйте потоки для параллельной проверки ссылок: отправляйте URL-адреса для проверки в канал и позвольте нескольким потокам проверять URL-адреса параллельно.
- Расширьте это, чтобы рекурсивно извлекать ссылки со всех страниц домена `www.google.org`. Установите верхний предел примерно в 100 страниц, чтобы избежать блокировки со стороны сайта.

Этот слайд должен занять примерно 20 минут.

- Это сложное упражнение, предназначенное для того, чтобы дать студентам возможность работать над более крупным проектом, чем остальные. Условием успешного выполнения этого упражнения является столкновение с какой-либо «реальной» проблемой и её решение при поддержке других студентов или преподавателя.

## 62.3 Решения

### Обедающие философы

```
use std::sync::{Arc, Mutex, mpsc};
use std::thread;
используйте std::time::Duration;

struct Chopstick;

struct Philosopher {
 name: String,
 left_chopstick: Arc<Mutex<Chopstick>>,
 right_chopstick: Arc<Mutex<Chopstick>>,
 thoughts: mpsc::SyncSender<String>,
}

impl Philosopher {
 fn think(&self) {
 self.thoughts
 .send(format!("Eureka! {} has a new idea!", & self.name))
 .unwrap();
 }

 fn eat(&self) {
 println!("{} пытается поесть", & self.name);
 let _left = self.left_chopstick.lock().unwrap();
 let _right = self.right_chopstick.lock().unwrap();

 println!("{} is eating...", & self.name);
 thread::sleep(Duration::from_millis(10));
 }
}

static PHILOSOPHERS: [&[&str]] =
 &["Сократ", "Гипатия", "Платон", "Аристотель", "Пифагор"];

fn main() {
 let (tx, rx) = mpsc::sync_channel(10);

 let chopsticks = PHILOSOPHERS
 .iter()
 .map(|_| Arc::new(Mutex::new(Chopstick)))
 .collect::<Vec<_>>();

 for i in 0..chopsticks.len() {
 let tx = tx.clone();
 let mut left_chopstick = Arc::clone(&chopsticks[i]);
 let mut right_chopstick =
 Arc::clone(&chopsticks[(i + 1) % chopsticks.len()]);
 }
}
```

```

// Чтобы избежать взаимной блокировки, необходимо разорвать симметрию // где-то. Это позволит поменять палочки для еды местами без инициализации // ни одной из них.
if i == chopsticks.len() - 1 {
 std::mem::swap(&mut left_chopstick, & mut right_chopstick);
}

let philosopher = Philosopher {
 name: PHILOSOPHERS[i].to_string(),
 thoughts: tx,
 left_chopstick,
 right_chopstick,
};

thread::spawn(move || {
 for _ in 0..100 {
 philosopher.eat();
 philosopher.think();
 }
});

drop(tx);
for thought in rx {
 println!("{} thought");
}
}

```

## Проверка ссылок

```

use std::sync::{Arc, Mutex, mpsc};
use std::thread;

use reqwest::Url;
use reqwest::blocking::Client;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
 #[error("request error: {0}")]
 ReqwestError(#[from] reqwest::Error),
 #[error("bad http response: {0}")]
 BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
 url: Url,
 extract_links: bool,
}

```

```

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>, Error> {
 println!("Checking {:?}", command.url);
 let response = client.get(command.url.clone()).send()?;
 if !response.status().is_success() {
 return Err(Error::BadResponse(response.status().to_string()));
 }

 let mut link_urls = Vec::new();
 if !command.extract_links {
 return Ok(link_urls);
 }

 let base_url = response.url().to_owned();
 let body_text = response.text()?;
 let document = Html::parse_document(&body_text);

 let selector = Selector::parse("a").unwrap();
 let href_values = document
 .select(&selector)
 .filter_map(|element| element.value().attr("href"));
 for href in href_values {
 match base_url.join(href) {
 Ok(link_url) => {
 link_urls.push(link_url);
 }
 Err(err) => {
 println!("На {}: проигнорирован неразборчивый {}", base_url, err);
 }
 }
 }
 Ok(link_urls)
}

struct CrawlState {
 domain: String,
 visited_pages: std::collections::HashSet<String>,
}

impl CrawlState {
 fn new(start_url: &Url) -> CrawlState {
 let mut visited_pages = std::collections::HashSet::new();
 visited_pages.insert(start_url.as_str().to_string());
 CrawlState { domain: start_url.domain().unwrap().to_string(), visited_pages }
 }

 // Определяет, следует ли извлекать ссылки с указанной страницы.
 fn should_extract_links(&self, url: &Url) -> bool {
 let Some(url_domain) = url.domain() else {
 return false;
 };
 }
}

```

```

 url_domain == self.domain
 }

 // Отмечает указанную страницу как посещённую, возвращая false, если она
 // уже была посещена.
 fn mark_visited(&mut self, url: &Url) -> bool {
 self.visited_pages.insert(url.as_str().to_string())
 }
}

type CrawlResult = Result<Vec<Url>, (Url, Error)>;
fn spawn_crawler_threads(
 command_receiver: mpsc::Receiver<CrawlCommand>,
 result_sender: mpsc::Sender<CrawlResult>,
 thread_count: u32,
) {
 // Для мультиплексирования неповторяемого Receiver оберните его в Arc<Mutex<_>>.
 let command_receiver = Arc::new(Mutex::new(command_receiver));

 for _ in 0..thread_count {
 let result_sender = result_sender.clone();
 let command_receiver = Arc::clone(&command_receiver);
 thread::spawn(move || {
 let client = Client::new();
 loop {
 let command_result = {
 let receiver_guard = command_receiver.lock().unwrap();
 receiver_guard.recv()
 };
 let Ok(crawl_command) = command_result else { /
 / Отправитель был удалён. Новых команд не поступает. break ;
 };
 let crawl_result = match visit_page(&client, &crawl_command) {
 Ok(link_urls) => Ok(link_urls),
 Err(error) => Err((crawl_command.url, error)),
 };
 result_sender.send(crawl_result).unwrap();
 }
 });
 }
}

fn control_crawl(
 start_url: Url,
 command_sender: mpsc::Sender<CrawlCommand>,
 result_receiver: mpsc::Receiver<CrawlResult>,
) -> Vec<Url> {
 let mut crawl_state = CrawlState::new(&start_url);
 let start_command = CrawlCommand { url: start_url, extract_links: true };
 command_sender.send(start_command).unwrap();
}

```

```

let mut pending_urls = 1;

let mut bad_urls = Vec::new();
while pending_urls > 0 {
 let crawl_result = result_receiver.recv().unwrap();
 pending_urls -= 1;

 match crawl_result {
 Ok(link_urls) => {
 for url in link_urls {
 if crawl_state.mark_visited(&url) {
 let extract_links = crawl_state.should_extract_links(&url);
 let crawl_command = CrawlCommand { url, extract_links };
 command_sender.send(crawl_command).unwrap();
 pending_urls += 1;
 }
 }
 }
 Err((url, error)) => {
 bad_urls.push(url);
 println!("Обнаружена ошибка обхода: {:?}", error);
 continue;
 }
 }
}

fn check_links(start_url: Url) -> Vec<Url> {
 let (result_sender, result_receiver) = mpsc::channel::<CrawlResult>();
 let (command_sender, command_receiver) = mpsc::channel::<CrawlCommand>();
 spawn_crawler_threads(command_receiver, result_sender, 16);
 control_crawl(start_url, command_sender, result_receiver)
}

fn main() {
 let start_url = reqwest::Url::parse("https://www.google.org").unwrap();
 let bad_urls = check_links(start_url);
 println!("Плохие URLs: {:?}", bad_urls);
}

```

## **Часть XIV**

**Конкурентность: вторая половина дня**

## Глава 63

# Приветствие

„Async“ — это модель конкурентности, при которой несколько задач выполняются одновременно, выполняя каждую задачу до тех пор, пока она не заблокируется, после чего происходит переключение на другую задачу, готовую к выполнению. Эта модель позволяет запускать большее количество задач на ограниченном числе потоков. Это объясняется тем, что накладные расходы на каждую задачу обычно очень низки, а операционные системы предоставляют примитивы для эффективного определения ввода-вывода, который может продолжаться.

Асинхронные операции в Rust основаны на „futures“, которые представляют работу, которая может быть завершена в будущем. Futures опрашиваются до тех пор, пока не сигнализируют о своём завершении.

Futures опрашиваются асинхронным рантаймом, и существует несколько различных реализаций рантаймов.

## Сравнения

- Python имеет аналогичную модель в своём `asyncio`. Однако тип `Future` в Python основан на обратных вызовах и не опрашивается. Асинхронные программы на Python требуют «цикла», аналогичного рантайму в Rust.
- `Promise` в JavaScript похож, но, опять же, основан на `callback`. Среда выполнения языка реализует цикл событий, поэтому многие детали разрешения `Promise` скрыты.

## Расписание

Включая 10-минутные перерывы, эта сессия должна занять около 3 часов 30 минут. Она включает:

| Сегмент                     | Продолжительность |
|-----------------------------|-------------------|
| Основы Async                | 40 минут          |
| Каналы и управление потоком | — 20 минут        |
| Подводные камни             | 55 минут          |
| Упражнения                  | 1 час 10 минут    |

## Глава 64

# Основы Async

Этот сегмент рассчитан примерно на 40 минут. Он включает:

| Слайд            | Продолжительность |
|------------------|-------------------|
| async/await      | 10 минут          |
| Futures          | 4 минуты          |
| Машина состояний | 10 минут          |
| Среды выполнения | 10 минут          |
| Задания          | 10 минут          |

### 64.1 async/await

На высоком уровне async-код Rust выглядит очень похоже на «обычный» последовательный код:

```
use futures::executor::block_on;

async fn count_to(count: i32) {
 for i in 0..count {
 println!("Count is: {}!", i);
 }
}

async fn async_main(count: i32) {
 count_to(count).await;
}

fn main() {
 block_on(async_main(10));
}
```

Этот слайд должен занять около 6 минут.

Ключевые моменты:

- Обратите внимание, что это упрощённый пример для демонстрации синтаксиса. В этом нет длительных операций или какой-либо реальной конкурентности!
- Ключевое слово ”async” является синтаксическим сахаром. Компилятор заменяет возвращаемый тип на fluture.
- Вы не можете сделать `main` `async` без дополнительных указаний компилятору о том, как использовать возвращаемый `fluture`.
- Для выполнения `async`-кода необходим исполнитель. `block_on` блокирует текущий поток до тех пор, пока предоставленный `fluture` не завершится.
- `.await` асинхронно ожидает завершения другой операции. В отличие от `block_on`, `.await` не блокирует текущий поток.
- `.await` может использоваться только внутри `async`-функции (или блока; они будут рассмотрены позже).

## 64.2 Futures

`Future` — это трейд, реализуемый объектами, представляющими операцию, которая может быть ещё не завершена. `Future` можно опрашивать, и `poll` возвращает `Poll`.

```
use std::pin::Pin;
use std::task::Context;

pub trait Future {
 type Output;
 fn poll(self: Pin<& mut Self>, cx: & mut Context<'_>) -> Poll<Self::Output>;
}

pub enum Poll<T> {
 Ready(T),
 Pending,
}
```

Асинхронная функция возвращает `impl Future`. Также возможно (хотя и редко) реализовать `Future` для собственных типов. Например, `JoinHandle`, возвращаемый из `tokio::spawn`, реализует `Future`, что позволяет выполнять `join`.

Ключевое слово `.await`, применённое к `Future`, приостанавливает текущую асинхронную функцию до готовности этого `Future`, а затем возвращает его результат.

Этот слайд должен занять примерно 4 минуты.

- Типы `Future` и `Poll` реализованы точно так, как показано; нажмите на ссылки, чтобы увидеть реализации в документации.
- `Context` позволяет `Future` запланировать повторный опрос при возникновении события, например, таймаута.
- `Pin` гарантирует, что `Future` не будет перемещён в памяти, чтобы указатели на этот `fluture` оставались валидными. Это необходимо для сохранения валидности ссылок после `.await`. Мы рассмотрим `Pin` в разделе «Подводные камни».

## 64.3 Машина состояний

Rust преобразует асинхронную функцию или блок в скрытый тип, реализующий Future, используя конечный автомат для отслеживания прогресса функции. Детали этого преобразования сложны, но полезно иметь схематичное представление о происходящем. Следующая функция // Суммирует два броска D10 с учётом модификатора.

```
async fn two_d10(modifier: u32) -> u32 {
 let first_roll = roll_d10().await;
 let second_roll = roll_d10(). await;
 first_roll + second_roll + modifier
}

преобразуется во что-то вроде

use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};

// Суммирует два броска D10 с учётом модификатора .
fn two_d10(modifier: u32) -> TwoD10 {
 TwoD10::Init { modifier }
}

enum TwoD10 {
 // Функция ещё не началась .
 Init { modifier: u32 },
 // Ожидание завершения первого ` .await` .
 FirstRoll { modifier: u32, fut: RollD10Future },
 // Ожидание завершения второго ` .await` .
 SecondRoll { modifier: u32, first_roll: u32, fut: RollD10Future },
}

impl Future for TwoD10 {
 type Output = u32;
 fn poll(mut self: Pin<& mut Self>, ctx: &mut Context) -> Poll<Self::Output> {
 loop {
 match *self {
 TwoD10::Init { modifier } => { // Создание
 future для первого броска кубика .
 let fut = roll_d10();
 *self = TwoD10::FirstRoll { modifier, fut };
 }
 TwoD10::FirstRoll { modifier, ref mut fut } => { //
 Опрос под-future для первого броска кубика .
 if let Poll::Ready(first_roll) = fut.poll(ctx) {
 // Создание future для второго броска .
 let fut = roll_d10();
 *self = TwoD10::SecondRoll { modifier, first_roll, fut };
 } else {
 return Poll::Pending;
 }
 }
 }
 }
 }
}
```

```
 }
 TwoD10::SecondRoll { modifier, first_roll, ref mut fut } => { //+
 Опрос подфьючерса для второго броска кубика.
 if let Poll::Ready(second_roll) = fut.poll(ctx) {
 return Poll::Ready(first_roll + second_roll + modifier);
 } else {
 return Poll::Pending;
 }
 }
}
```

На этот слайд должно уйти около 10 минут.

Этот пример является иллюстративным и не представляет собой точного отображения трансформации компилятора Rust. Важные моменты, на которые следует обратить внимание:

- Вызов `async`-функции лишь создает и возвращает фьючерс.
  - Все локальные переменные сохраняются в фьючерсе функции с использованием `enum` для идентификации текущей точки приостановки выполнения.
  - Оператор `.await` в `async`-функции преобразуется в новое состояние, содержащее все активные переменные и ожидаемый фьючерс. Затем `loop` обрабатывает обновленное состояние, опрашивая фьючерс до тех пор, пока он не вернет `Poll::Ready`.
  - Выполнение продолжается активно до тех пор, пока не произойдет `Poll::Pending`. В этом простом примере каждый `fluture` готов немедленно.
  - `main` содержит наивный исполнитель, который просто выполняет `busy-loop` до готовности `fluture`. Мы вскоре рассмотрим реальные исполнители.

## **Дополнительные материалы для изучения**

Представьте структуру данных `Future` для глубоко вложенного стека асинхронных функций. Каждый `Future` функции содержит структуры `Future` для вызываемых ею функций. Это может привести к неожиданно большим типам `Future`, генерируемым компилятором.

Это также означает, что рекурсивные асинхронные функции представляют собой сложную задачу. Сравните с распространённой ошибкой построения рекур-

```
сивного типа, например enum LinkedList<T> {
 Node { value: T, next: LinkedList<T> },
 Nil,
}
```

Решение для рекурсивного типа — добавить уровень косвенности, например с помощью `Box`. Аналогично, рекурсивная асинхронная функция должна оборачивать рекурсивный `future` в `box`:

```
async fn count_to(n: u32) {
 if n > 0 {
 Box::pin(count_to(n - 1)).await;
 println!("{}"), n);
 }
}
```

## 64.4 Среды выполнения

Время выполнения (*runtime*) обеспечивает поддержку выполнения операций асинхронно (реактор) и отвечает за выполнение futures (исполнитель). В Rust отсутствует «встроенное» время выполнения, однако доступны несколько вариантов:

- Tokio: производительный, с хорошо развитой экосистемой, включающей такие инструменты, как [Нурег для HTTP](#) и [Tonic для gRPC](#).
- async-std: стремится стать «std для async» и включает базовое время выполнения в `async::task`.
- smol: простой и лёгкий

Некоторые крупные приложения имеют собственные времена выполнения. Например, Fuchsia уже располагает собственным временем выполнения.

Этот слайд и его подразделы должны занять примерно 10 минут.

- Обратите внимание, что из перечисленных времен выполнения в Rust playground поддерживается только Tokio. Playground также не разрешает ввод-вывод, поэтому большинство интересных асинхронных задач не могут быть выполнены в playground.
- Futures являются «инертными» в том смысле, что они ничего не делают (даже не запускают операцию ввода-вывода), если их не опрашивает исполнитель. Это отличается от JS Promises, которые, например, выполняются до завершения, даже если их никогда не используют.

### 64.4.1 Tokio

Tokio предоставляет:

- Многопоточный runtime для выполнения асинхронного кода.
- Асинхронную версию стандартной библиотеки.
- Обширную экосистему библиотек.

```
use tokio::time;

async fn count_to(count: i32) {
 for i in 0..count {
 println!("Count in task: {i}!");
 time::sleep(time::Duration::from_millis(5)).await;
 }
}

#[tokio::main]
async fn main() {
 tokio::spawn(count_to(10));

 for i in 0..5 {
 println!("Main task: {i}");
 time::sleep(time::Duration::from_millis(5)).await;
 }
}
```

- С помощью макроса `tokio::main` мы теперь можем сделать функцию `main` асинхронной.
- Функция `spawn` создаёт новую конкурентную «задачу».
- Примечание: `spawn` принимает Future, поэтому вы не вызываете `.await` на `count_to`.

### Дальнейшее изучение:

- Почему count\_to обычно не достигает 10? Это пример отмены async.  
tokio::spawn возвращает дескриптор, который можно ожидать для ожидания завершения.
- Попробуйте count\_to(10).await вместо создания задачи.
- Попробуйте ожидать задачу, возвращённую из tokio::spawn.

## 64.5 Задачи

В Rust существует система задач, представляющая собой форму лёгких потоков.

Задача имеет единственное верхнеуровневое future, которое исполнитель опрашивает для продвижения. Это future может содержать одно или несколько вложенных future, которые его метод poll опрашивает, что примерно соответствует стеку вызовов. Конкурентность внутри задачи возможна путём опроса нескольких дочерних future, например, гонки таймера и операции ввода-вывода.

```
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> io::Result<()> {
 let listener = TcpListener::bind("127.0.0.1:0").await?;
 println!("listening on port {}", listener.local_addr()?.port());

 loop {
 let (mut socket, addr) = listener.accept().await?;

 println!("connection from {addr:?}");

 tokio::spawn(async move {
 socket.write_all(b"Who are you?\n").await.expect("socket error");

 let mut buf = vec![0; 1024];
 let name_size = socket.read(&mut buf).await.expect("socket error");
 let name = std::str::from_utf8(&buf[..name_size]).unwrap().trim();
 let reply = format!("Thanks for dialing in, {name}!\n");
 socket.write_all(reply.as_bytes()).await.expect("socket error");
 });
 }
}
```

Этот слайд должен занять около 6 минут.

Скопируйте этот пример в подготовленный файл `src/main.rs` и запустите его оттуда.

Попробуйте подключиться к нему с помощью инструмента TCP-соединения, такого как nc или telnet.

- Попросите студентов представить, каким будет состояние примерного сервера при наличии нескольких подключённых клиентов. Какие задачи существуют? Каковы их Futures?
- Это первый раз, когда мы видим async блок. Это похоже на замыкание, но не принимает никаких аргументов. Его возвращаемое значение — Future, аналогично async fn.

- Рефакторинг `async`-блока в функцию и улучшение обработки ошибок с использованием `?`.

## Глава 65

# Каналы и управление потоком

Этот сегмент займет около 20 минут. В него входит:

| Слайд              | Продолжительность |
|--------------------|-------------------|
| Асинхронные каналы | 10 минут          |
| Присоединиться     | 4 минуты          |
| Select             | 5 минут           |

### 65.1 Async-каналы

Несколько crate поддерживают асинхронные каналы. Например, tokio:

```
use tokio::sync::mpsc;

async fn ping_handler(mut input: mpsc::Receiver<()>) {
 let mut count: usize = 0;

 while let Some(_) = input.recv().await {
 count += 1;
 println!("Получено {count} пингов на данный момент . ");
 }

 println!("ping_handler complete");
}

#[tokio::main]
async fn main() {
 let (sender, receiver) = mpsc::channel(32);
 let ping_handler_task = tokio::spawn(ping_handler(receiver));
 for i in 0..10 {
 sender.send(()).await.expect("Не удалось отправить ping.");
 println!("Отправлено {} ping'ов на данный момент . ", i + 1);
 }
}
```

```

 drop(sender);
ping_handler_task.await.expect("Произошла ошибка в задаче ping handler.");
}

```

На этот слайд следует отвести около 8 минут.

- Измените размер канала на 3 и посмотрите, как это повлияет на выполнение.
- В целом, интерфейс похож на syncканалы, как было показано на утреннем занятии.
- Попробуйте убрать вызов `std::mem::drop`. Что произойдет? Почему?
- В библиотеке Flume есть каналы, которые реализуют как `sync`, так и `async send/receive`. Это может быть удобно для сложных приложений с задачами как ввода-вывода, так и интенсивной обработки на CPU.
- Преимущество работы с `async` каналами заключается в возможности комбинировать их с другими `future` для создания сложных управляющих потоков.

## 65.2 Join

Операция `join` ожидает, пока все `flutures` из набора не будут готовы, и возвращает коллекцию их результатов. Это аналогично `Promise.all` в JavaScript или `asyncio.gather` в Python.

```

use anyhow::Result;
use futures::future;
use reqwest;
use std::collections::HashMap;

async fn size_of_page(url: &str) -> Result<usize> {
 let resp = reqwest::get(url).await?;
 Ok(resp.text().await?.len())
}

#[tokio::main]
async fn main() {
 let urls: [&str; 4] = [
 "https://google.com",
 "https://httpbin.org/ip",
 "https://play.rust-lang.org/",
 "BAD_URL",
];
 let futures_iter = urls.into_iter().map(size_of_page);
 let results = future::join_all(futures_iter).await;
 let page_sizes_dict: HashMap<&str, Result<usize>> =
 urls.into_iter().zip(results.into_iter()).collect();
 println!("{}page_sizes_dict: {}");
}

```

Этот слайд должен занять примерно 4 минуты.

Скопируйте этот пример в подготовленный файл `src/main.rs` и запустите его оттуда.

- Для нескольких `flutures` различных типов можно использовать `std::future::join!`, однако необходимо знать количество `flutures` на этапе компиляции. В настоящее время это находится в `futures`

crate, который вскоре будет стабилизирован в `std::future`.

- Риск использования `join` заключается в том, что один из `futures` может никогда не завершиться, что приведёт к зависанию программы.
- Также можно комбинировать `join_all` с `join!`, например, чтобы объединить все запросы к HTTP-сервису и запрос к базе данных. Попробуйте добавить `tokio::time::sleep` в `future`, используя `futures::join!`. Это не таймаут (для этого требуется `select!`, который объясняется в следующей главе), но демонстрирует работу `join!`.

## 65.3 Select

Операция `select` ожидает, пока любой из набора `futures` не будет готов, и реагирует на результат этого `future`. В JavaScript это аналогично `Promise.race`. В Python это сопоставимо с `asyncio.wait(task_set, return_when=asyncio.FIRST_COMPLETED)`.

Подобно оператору `match`, тело `select!` содержит несколько ветвей, каждая из которых имеет форму `pattern = future => statement`. Когда `future` готов, его возвращаемое значение де-структурируется с помощью `pattern`. Затем `statement` выполняется с полученными переменными. Результат выполнения `statement` становится результатом макроса `select!`.

```
use tokio::sync::mpsc;
use tokio::time::{Duration, sleep};

#[tokio::main]
async fn main() {
 let (tx, mut rx) = mpsc::channel(32);
 let listener = tokio::spawn(async move {
 tokio::select! {
 Some(msg) = rx.recv() => println!("got: {msg}"),
 _ = sleep(Duration::from_millis(50)) => println!("timeout"),
 };
 });
 sleep(Duration::from_millis(10)).await;
 tx.send(String::from("Hello!")).await.expect("Не удалось отправить приветствие");

 listener.await.expect("Ошибка слушателя");
}
```

На этот слайд отводится примерно 5 минут.

- Асинхронный блок `listener` здесь — распространённая форма: ожидание некоторого асинхронного события или тайм-аута. Измените `sleep` на более длительный, чтобы увидеть сбой. Почему в этой ситуации также не удаётся выполнить `send`?
- `select!` часто используется в цикле в архитектурах «актеров», где задача реагирует на события в цикле. Это имеет некоторые подводные камни, которые будут рассмотрены в следующем разделе.

# Глава 66

## Подводные камни

Async/await предоставляет удобную и эффективную абстракцию для конкурентного асинхронного программирования. Однако модель async/await в Rust также имеет свои подводные камни и ловушки. Некоторые из них мы иллюстрируем в этой главе.

Этот раздел займет примерно 55 минут. В нем содержится:

| Слайд                           | Продолжительность |
|---------------------------------|-------------------|
| Блокировка Executor на 10 минут | 20 минут          |
| Pin                             | 5 минут           |
| Асинхронные трейты              | 20 минут          |
| Отмена                          |                   |

### 66.1 Блокировка исполнителя

Большинство асинхронных рантаймов позволяют выполнять параллельно только IO-задачи. Это означает, что блокирующие задачи процессора будут блокировать исполнитель и препятствовать выполнению других задач. Простое решение — использовать асинхронные эквивалентные методы, где это возможно.

```
use futures::future::join_all;
use std::time::Instant;

async fn sleep_ms(start: &Instant, id: u64, duration_ms: u64) {
 std::thread::sleep(std::time::Duration::from_millis(duration_ms));
 println!(
 "future {} slept for {}ms, finished after {}ms",
 start.elapsed().as_millis()
);
}

#[tokio::main(flavor = "current_thread")]
async fn main() {
 let start = Instant::now();
 let sleep_futures = (1..=10).map(|t| sleep_ms(&start, t, t * 10));
}
```

```
 join_all(sleep_futures).await;
 }
```

На этот слайд должно уйти около 10 минут.

- Запустите код и убедитесь, что задержки происходят последовательно, а не одновременно.
- Вариант "current\_thread" размещает все задачи на одном потоке. Это делает эффект более очевидным, но ошибка всё ещё присутствует в многопоточном варианте.
- Замените `std::thread::sleep` на `tokio::time::sleep` и дождитесь его результата.
- Другим решением будет использование `tokio::task::spawn_blocking`, который создаёт реальный поток и преобразует его дескриптор в `future` без блокировки исполнителя.
- Не следует рассматривать задачи как потоки ОС. Они не соответствуют друг другу один к одному, и большинство исполнителей позволяют запускать множество задач на одном потоке ОС. Это особенно проблематично при взаимодействии с другими библиотеками через FFI, где библиотека может зависеть от локального хранилища потока или быть привязана к конкретным потокам ОС (например, CUDA). В таких случаях предпочтительно использовать `tokio::task::spawn_blocking`.
- Используйте синхронизирующие мьютексы с осторожностью. Удержание мьютекса во время `.await` может привести к блокировке другой задачи, которая может выполняться на том же потоке.

## 66.2 Pin

Напомним, что асинхронная функция или блок создаёт тип, реализующий `Future` и содержащий все локальные переменные. Некоторые из этих переменных могут содержать ссылки (указатели) на другие локальные переменные. Чтобы гарантировать их корректность, `future` никогда не может быть перемещён в другое место в памяти.

Чтобы предотвратить перемещение типа `future` в памяти, его можно опрашивать только через закреплённый указатель. `Pin` — это оболочка вокруг ссылки, которая запрещает все операции, приводящие к перемещению экземпляра, на который она указывает, в другое место в памяти.

```
use tokio::sync::{mpsc, oneshot};
use tokio::task::spawn;
use tokio::time::{Duration, sleep};

// Элемент работы. В данном случае просто подождать указанное время и ответить // сообщением в канал `respond_on`.
#[derive(Debug)]
struct Work {
 input: u32,
 respond_on: oneshot::Sender<u32>,
}

// Рабочий, который слушает очередь задач и выполняет их.
async fn worker(mut work_queue: mpsc::Receiver<Work>) {
 let mut iterations = 0;
 loop {
 tokio::select! {
 Some(work) = work_queue.recv() => {
 sleep(Duration::from_millis(10)).await; // Имитация работы.
 }
 }
 }
}
```

```

 work.respond_on
 .send(work.input * 1000)
 .expect("не удалось отправить ответ");
 iterations += 1;
 }
 // TODO: сообщать количество итераций каждые 100 мс
}
}

// Запросчик, который запрашивает работу и ожидает её завершения.
async fn do_work(work_queue: &mpsc::Sender<Work>, input: u32) -> u32 {
 let (tx, rx) = oneshot::channel();
 work_queue
 .send(Work { input, respond_on: tx })
 .await
 .expect("не удалось отправить в очередь работы");
 rx.await.expect("ошибка ожидания ответа")
}

#[tokio::main]
async fn main() {
 let (tx, rx) = mpsc::channel(10);
 spawn(worker(rx));
 for i in 0..100 {
 let resp = do_work(&tx, i).await;
 println!("результат работы для итерации {i}: {resp}");
 }
}

```

Этот слайд должен занять примерно 20 минут.

- Вы можете узнать это как пример паттерна актёра. Актёры обычно вызывают `select!` в цикле.
- Это служит обобщением нескольких предыдущих уроков, поэтому уделите этому достаточно времени.
  - Наивно добавьте `_ = sleep(Duration::from_millis(100)) => { println!(...) }` в `select!`. Это никогда не выполнится. Почему?
  - Вместо этого добавьте `timeout_fut`, содержащий этот `future`, вне `loop`:

```

let timeout_fut = sleep(Duration::from_millis(100));
loop {
 select! {
 ...
 _ = timeout_fut => { println!(...); },
 }
}

```

- Это всё ещё не работает. Следуйте ошибкам компилятора, добавляя `&mut` к `timeout_fut` в `select!` для обхода перемещения, затем используйте `Box::pin`:

```

let mut timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
loop {

```

```

 select! {
 ...
 _ = & mut timeout_fut => { println!(...); },
 }
}

```

- Этот код компилируется, но после истечения таймаута он становится `Poll::Ready` на каждой итерации (с этим помог бы `flused future`). Обновление для сброса `timeout_fut` каждый раз при его истечении:

```

let mut timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
loop {
 select! {
 _ = & mut timeout_fut => {
 println!(...);
 timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
 },
 }
}

```

- Вок выделяет память в куче. В некоторых случаях `std::pin::pin!` (недавно стабилизирован, в старом коде часто используется `tokio::pin!`) также является вариантом, но его сложно применять для `future`, который переназначается.
- Другой альтернативой является полное отсутствие использования `pin` и создание другой задачи, которая будет отправлять данные в однократный канал каждые 100 мс.
- Данные, содержащие указатели на самих себя, называются самоссылочными. Обычно проверяющий заимствования Rust предотвращает перемещение таких данных, поскольку ссылки не могут жить дольше данных, на которые они указывают. Однако преобразование кода для `async-blocks` и функций не проверяется проверяющим заимствования.
- `Pin` — это обёртка вокруг ссылки. Объект не может быть перемещён с помощью закреплённого указателя. Однако он всё ещё может быть перемещён через незакреплённый указатель.
- Метод `poll` трейта `Future` использует `Pin<&mut Self>` вместо `&mut Self` для обращения к экземпляру. Именно поэтому он может быть вызван только на закреплённом указателе.

## 66.3 Async-трейты

Асинхронные методы в трейтах были стабилизированы в релизе 1.75. Для этого потребовалась поддержка использования возвращаемого типа `impl Trait` в трейтах, поскольку десугаринг для `async fn` включает `-> impl Future<Output = ...>`.

Однако, даже при нативной поддержке, существуют некоторые подводные камни, связанные с `async fn`:

- Return-position `impl Trait` захватывает все жизненные циклы в области видимости (поэтому некоторые шаблоны заимствования не могут быть выражены).
- Асинхронные трейты не могут использоваться с объектами трейтов (`dyn Trait` поддержка).

Крейт `async_trait` предоставляет обходное решение для поддержки `dyn` через макрос, с некоторыми оговорками:

```

use async_trait::async_trait;
use std::time::Instant;

```

```

use tokio::time::{Duration, sleep};

#[async_trait]
trait Sleeper {
 async fn sleep(&self);
}

struct FixedSleeper {
 sleep_ms: u64,
}

#[async_trait]
impl Sleeper for FixedSleeper {
 async fn sleep(&self) {
 sleep(Duration::from_millis(self.sleep_ms)).await;
 }
}

async fn run_all_sleepers_multiple_times(
 sleepers: Vec<Box<dyn Sleeper>>,
 n_times: usize,
) {
 for _ in 0..n_times {
 println!("Запуск всех спящих... ");
 for sleeper in &sleepers {
 let start = Instant::now();
 sleeper.sleep().await;
 println!("Спал(а) {} мс", start.elapsed().as_millis());
 }
 }
}

#[tokio::main]
async fn main() {
 let sleepers: Vec<Box<dyn Sleeper>> = vec![
 Box::new(FixedSleeper { sleep_ms: 50 }),
 Box::new(FixedSleeper { sleep_ms: 100 }),
];
 run_all_sleepers_multiple_times(sleepers, 5).await;
}

```

На этот слайд отводится примерно 5 минут.

- `async_trait` легко использовать, но обратите внимание, что для этого применяются выделения в куче. Это выделение в куче вызывает накладные расходы на производительность.
- Проблемы поддержки `async trait` в языке слишком сложны, чтобы подробно рассматривать их в рамках этого курса. Если хотите изучить тему глубже, ознакомьтесь с этим блог-постом Нико Мацакса. Смотрите также следующие ключевые слова:
  - RPIT: сокращение от return-position `impl Trait`.
  - RPITIT: сокращение от return-position `impl Trait` в трейте (RPIT в трейте).

- Попробуйте создать новую структуру sleeper, которая будет приостанавливать выполнение на случайное время, и добавить её в Vec .

## 66.4 Отмена

Отбрасывание future означает, что её больше нельзя будет опрашивать. Это называется *отмена* и может произойти в любой точке await . Требуется осторожность, чтобы система работала корректно даже при отмене futures. Например, не должно возникать взаимных блокировок или потери данных.

```
use std::io;
используйте std::time::Duration;
use tokio::io::{AsyncReadExt, AsyncWriteExt, DuplexStream};

struct LinesReader {
 stream: DuplexStream,
}

impl LinesReader {
 fn new(stream: DuplexStream) -> Self {
 Self { stream }
 }

 async fn next(&mut self) -> io::Result<Option<String>> {
 let mut bytes = Vec::new();
 let mut buf = [0];
 while self.stream.read(&mut buf[..]).await? != 0 {
 bytes.push(buf[0]);
 if buf[0] == b'\n' {
 break;
 }
 }
 if bytes.is_empty() {
 return Ok(None);
 }
 let s = String::from_utf8(bytes)
 .map_err(|_| io::Error::new(io::ErrorKind::InvalidData, "not UTF-8"))?;
 Ok(Some(s))
 }

 async fn slow_copy(source: String, mut dest: DuplexStream) -> io::Result<()> {
 for b in source.bytes() {
 dest.write_u8(b).await?;
 tokio::time::sleep(Duration::from_millis(10)).await
 }
 Ok(())
 }
}

#[tokio::main]
async fn main() -> io::Result<()> {
```

```

let (client, server) = tokio::io::duplex(5);
let handle = tokio::spawn(slow_copy("hi\nthere\n".to_owned(), client));

let mut lines = LinesReader::new(server);
let mut interval = tokio::time::interval(Duration::from_millis(60));
loop {
 tokio::select! {
 _ = interval.tick() => println!("tick!"),
 line = lines.next() => if let Some(l) = line? {
 print!("{} ", l)
 } else {
 break
 },
 }
}
handle.await.unwrap()?;
Ok(())
}

```

Этот слайд должен занять примерно 18 минут.

- Компилятор не обеспечивает безопасность отмены. Необходимо изучить документацию API и учитывать состояние вашей async fn.
- В отличие от `panics?`, отмена является частью нормального потока управления (в отличие от обработки ошибок).
- В этом примере теряются части строки.
  - Всякий раз, когда ветка `tick()` завершается первой, `next()` и её `buf` отбрасываются.
  - `LinesReader` можно сделать безопасным для отмены, включив `buf` в структуру:

```

struct LinesReader {
 stream: DuplexStream,
 bytes: Vec<u8>,
 buf: [u8; 1],
}

impl LinesReader {
 fn new(stream: DuplexStream) -> Self {
 Self { stream, bytes: Vec::new(), buf: [0] }
 }
 async fn next(&mut self) -> io::Result<Option<String>> {
 // добавьте префиксы self. к buf и bytes.
 // ...
 let raw = std::mem::take(& mut self.bytes);
 let s = String::from_utf8(raw)
 .map_err(|_| io::Error::new(io::ErrorKind::InvalidData, "не UTF-8"))
 // ...
 }
}

```

- `Interval::tick` является безопасным для отмены, поскольку отслеживает, был ли тик «доставлен».

- `AsyncReadExt::read` является безопасным для отмены, так как либо возвращает результат, либо не читает данные.
- `AsyncBufReadExt::read_line` похожа на пример и не является безопасной для отмены.  
См. документацию для подробностей и альтернатив.

## Глава 67

# Упражнения

Этот раздел займет примерно 1 час 10 минут. В нём содержится:

| Слайд                          | Продолжительность |
|--------------------------------|-------------------|
| Обедающие философы             | 20 минут          |
| Приложение для трансляции чата | 30 минут          |
| Решения                        | 20 минут          |

### 67.1 Обед философов — Async

См. dining philosophers для описания задачи.

Как и ранее, для этого упражнения потребуется локальная установка Cargo. Скопируйте код ниже в файл с именем `src/main.rs`, заполните пропуски и убедитесь, что `cargo run` не приводит к взаимной блокировке:

```
use std::sync::Arc;
use tokio::sync::{Mutex, mpsc};
use tokio::time;

struct Chopstick;

struct Philosopher {
 name: String,
 // left_chopstick: ...
 // right_chopstick: ...
 // thoughts: ...
}

impl Philosopher {
 async fn think(&self) {
 self.thoughts
 .send(format!("Eureka! {} has a new idea!", & self.name))
 .await
 .unwrap();
 }
}
```

```

 }

 async fn eat(&self) {
 // Продолжайте попытки, пока не получите обе палочки
 println!("{} is eating...", &self.name);
 time::sleep(time::Duration::from_millis(5)).await;
 }
}

// Планировщик tokio не вызывает взаимной блокировки с 5 философами, поэтому используйте 2.
static PHILOSOPHERS: [&str] = &["Socrates", "Hypatia"];

#[tokio::main]
async fn main() {
 // Создайте палочки

 // Создать философов

 // Заставьте их думать и есть

 // Вывести их мысли
}

```

Поскольку на этот раз вы используете Async Rust, потребуется зависимость `tokio`. Вы можете использовать следующий `Cargo.toml`:

```

[package]
name = "dining-philosophers-async-dine"
version = "0.1.0"
edition = "2024"

[dependencies]
tokio = { version = "1.26.0", features = ["sync", "time", "macros", "rt-multi-thread"] } Также обратите внимание, что на этот раз необходимо использовать Mutex и модуль mpsc из крейта tokio.
Этот слайд должен занять примерно 20 минут.
• Можете ли вы сделать вашу реализацию однопоточной?

```

## 67.2 Приложение для группового чата

В этом упражнении мы хотим применить наши новые знания для реализации приложения вещательного чата. У нас есть сервер чата, к которому подключаются клиенты и публикуют свои сообщения. Клиент считывает сообщения пользователя из стандартного ввода и отправляет их на сервер. Сервер чата транслирует каждое полученное сообщение всем клиентам.

Для этого мы используем вещательный канал на сервере и `okio_websockets` для связи между клиентом и сервером.

Создайте новый проект Cargo и добавьте следующие зависимости:

`Cargo.toml`:

```

[package]
name = "chat-async"
version = "0.1.0"
edition = "2024"

[dependencies]
futures-util = { version = "0.3.31", features = ["sink"] }
http = "1.3.1"
tokio = { version = "1.45.1", features = ["full"] }
tokio-websockets = { version = "0.11.4", features = ["client", "fastrand", "server", "shuttle"] }

```

## Необходимые API

Вам потребуются следующие функции из `tokio` и `tokio_websockets`. Потратьте несколько минут на ознакомление с API.

- `StreamExt::next()` реализован в `WebSocketStream`: для асинхронного чтения сообщений из WebSocket Stream.
- `SinkExt::send()` реализован в `WebSocketStream`: для асинхронной отправки сообщений в WebSocket Stream.
- `Lines::next_line()`: для асинхронного чтения сообщений пользователя из стандартного ввода.
- `Sender::subscribe()`: для подписки на канал вещания.

## Два бинарных файла

Обычно в проекте Cargo может быть только один бинарный файл и один файл `src/main.rs`. В этом проекте нам нужны два бинарных файла. Один для клиента и один для сервера. Вы могли бы сделать их двумя отдельными проектами Cargo, но мы поместим их в один проект Cargo с двумя бинарными файлами. Для этого клиентский и серверный код должны находиться в папке `src/bin` (см. документацию).

Скопируйте следующий код сервера и клиента в файлы `src/bin/server.rs` и `src/bin/client.rs` соответственно. Ваша задача — дополнить эти файлы, как описано ниже.

`src/bin/server.rs:`

```

use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{Sender, channel};
use tokio_websockets::{Message, ServerBuilder, WebSocketStream};

async fn handle_connection(
 addr: SocketAddr,
 mut ws_stream: WebSocketStream<TcpStream>,
 bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {
 // TODO: Для подсказкисмотрите описание задачи ниже .
}

```

```

}

#[tokio::main]
async fn main() -> Result<(), Box< dyn Error + Send + Sync>> {
 let (bcast_tx, _) = channel(16);

 let listener = TcpListener::bind("127.0.0.1:2000").await?;
 println!("Прослушивание на порту 2000");

 loop {
 let (socket, addr) = listener.accept().await?;
 println!("Новое соединение от {addr:?}");
 let bcast_tx = bcast_tx.clone();
 tokio::spawn(async move {
 // Обернуть необработанный TCP-поток в WebSocket.
 let (_req, ws_stream) = ServerBuilder::new().accept(socket).await?;

 handle_connection(addr, ws_stream, bcast_tx).await
 });
 }
}

```

*src/bin/client.rs:*

```

use futures_util::SinkExt;
use futures_util::stream::StreamExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

#[tokio::main]
async fn main() -> Result<(), tokio_websockets::Error> {
 let (mut ws_stream, _) =
 ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
 .connect()
 .await?;

 let stdin = tokio::io::stdin();
 let mut stdin = BufReader::new(stdin).lines();

 // TODO: Для подсказкисмотрите описание задачи ниже.

}

```

## Запуск бинарных файлов

Запустите сервер с помощью:

`cargo run --bin server`

и клиент с помощью:

```
cargo run --bin client
```

## Задания

- Реализуйте функцию `handle_connection` в файле `src/bin/server.rs`.
  - Подсказка: используйте `tokio::select!` для одновременного выполнения двух задач в непрерывном цикле. Одна задача принимает сообщения от клиента и транслирует их. Другая отправляет сообщения, полученные сервером, клиенту.
- Завершите функцию `main` в файле `src/bin/client.rs`.
  - Подсказка: как и ранее, используйте `tokio::select!` в непрерывном цикле для одновременного выполнения двух задач: (1) чтение сообщений пользователя из стандартного ввода и отправка их серверу, и (2) получение сообщений от сервера и отображение их пользователю.
- Опционально: после завершения измените код так, чтобы сообщения транслировались всем клиентам, кроме отправителя.

## 67.3 Решения

### Обедающие философы — Async

```
use std::sync::Arc;
use tokio::sync::{Mutex, mpsc};
use tokio::time;

struct Chopstick;

struct Philosopher {
 name: String,
 left_chopstick: Arc<Mutex<Chopstick>>,
 right_chopstick: Arc<Mutex<Chopstick>>,
 thoughts: mpsc::Sender<String>,
}

impl Philosopher {
 async fn think(&self) {
 self.thoughts
 .send(format!("Eureka! {} has a new idea!", & self.name))
 .await
 .unwrap();
 }

 async fn eat(&self) {
 // Продолжайте попытки, пока не получите обе палочки
 // Взять палочки...
 let _left_chopstick = self.left_chopstick.lock().await;
 let _right_chopstick = self.right_chopstick.lock().await;

 println!("{} is eating...", & self.name);
 time::sleep(time::Duration::from_millis(5)).await;
 }
}
```

```

 // Здесь блокировки освобождаются
 }

}

// Планировщик tokio не вызывает взаимной блокировки с 5 философами, поэтому используйте .await.
static PHILOSOPHERS: &[&str] = &["Socrates", "Hypatia"];

#[tokio::main]
async fn main() {
 // Создать палочки
 let mut chopsticks = vec![];
 PHILOSOPHERS
 .iter()
 .for_each(|_| chopsticks.push(Arc::new(Mutex::new(Chopstick))));

 // Создать философов let
 (philosophers, mut rx) = { let
 mut philosophers = vec![];
 let (tx, rx) = mpsc::channel(10);
 for (i, name) in PHILOSOPHERS.iter().enumerate() {
 let mut left_chopstick = Arc::clone(&chopsticks[i]);
 let mut правый_палочка =
 Arc::clone(&палочки[(i + 1) % PHILOSOPHERS.len()]);
 if i == PHILOSOPHERS.len() - 1 {
 std::mem::swap(&mut left_chopstick, &mut right_chopstick);
 }
 philosophers.push(Philosopher {
 name: name.to_string(),
 левый_палочка, правый
 _палочка, мысли: tx.
 clone(), });
 }
 (philosophers, rx)
 };
 // tx здесь уничтожается, поэтому нам не нужно явно уничтожать его позже
};

// Заставьте их думать и есть
for phil in philosophers {
 tokio::spawn(async move {
 for _ in 0..100 {
 phil.think().await;
 phil.eat().await;
 }
 });
}

// Вывести их мысли
while let Some(мысль) = rx.recv().await {
 println!("Бот мысль: {мысль}");
}

```

```
}
```

## Приложение для группового чата

src/bin/server.rs:

```
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{Sender, channel};
use tokio_websockets::{Message, ServerBuilder, WebSocketStream};

async fn handle_connection(
 addr: SocketAddr,
 mut ws_stream: WebSocketStream<TcpStream>,
 bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {
 ws_stream
 .send(Message::text("Добро пожаловать в чат! Введите сообщение".to_string()))
 .await?;
 let mut bcast_rx = bcast_tx.subscribe();

 // Непрерывный цикл для параллельного выполнения двух задач: (1) получение // сообщений из `ws_stream` и их трансляция, и (2) получение // сообщений на `bcast_rx` и отправка их клиенту.
 loop {
 tokio::select! {
 incoming = ws_stream.next() => {
 match incoming {
 Some(Ok(msg)) => {
 if let Some(text) = msg.as_text() {
 println!("От клиента {addr:?} {text:?}");
 bcast_tx.send(text.into())?;
 }
 }
 Some(Err(err)) => return Err(err.into()),
 None => return Ok(()),
 }
 }
 msg = bcast_rx.recv() => {
 ws_stream.send(Message::text(msg?)).await?;
 }
 }
 }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
```

```

let (bcast_tx, _) = channel(16);

let listener = TcpListener::bind("127.0.0.1:2000").await?;
println!("Прослушивание на порту 2000");

loop {
 let (socket, addr) = listener.accept().await?;
 println!("Новое соединение от {addr:?}");
 let bcast_tx = bcast_tx.clone();
 tokio::spawn(async move {
 // Обернуть необработанный TCP-поток в WebSocket.
 let (_req, ws_stream) = ServerBuilder::new().accept(socket).await?;

 handle_connection(addr, ws_stream, bcast_tx).await
 });
}
}

src/bin/client.rs:

use futures_util::SinkExt;
use futures_util::stream::StreamExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::ClientBuilder, Message;

#[tokio::main]
async fn main() -> Result<(), tokio_websockets::Error> {
 let (mut ws_stream, _) =
 ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
 .connect()
 .await?;

 let stdin = tokio::io::stdin();
 let mut stdin = BufReader::new(stdin).lines();

 // Непрерывный цикл для параллельной отправки и получения сообщений.
 loop {
 tokio::select! {
 incoming = ws_stream.next() => {
 match incoming {
 Some(Ok(msg)) => {
 if let Some(text) = msg.as_text() {
 println!("От сервера: {}", text);
 }
 },
 Some(Err(err)) => return Err(err.into()),
 None => return Ok(()),
 }
 }
 res = stdin.next_line() => {
 match res {

```

```
Ok(None) => return Ok(),
Ok(Some(line)) => ws_stream.send(Message::text(line.to_string())).await,
Err(err) => return Err(err.into()),
}
}
}
}
}
```

## **Часть XV**

# **Идиоматичный Rust**

# Глава 68

## Добро пожаловать в «Идиоматичный Rust»

«Основы Rust» познакомили с синтаксисом Rust и ключевыми концепциями. Теперь мы хотим сделать следующий шаг: как эффективно использовать Rust в ваших проектах? Как выглядит идиоматичный Rust?

Этот курс имеет определённую позицию: мы будем направлять вас к некоторым паттернам и отдалять от других. Тем не менее, мы признаём, что у некоторых проектов могут быть иные потребности. Мы всегда предоставляем необходимую информацию, чтобы помочь вам принимать обоснованные решения в контексте и с учётом ограничений ваших собственных проектов.

 Этот курс находится в **активной разработке**.

Материал может часто меняться, и возможно наличие ошибок, которые ещё не были обнаружены. Тем не менее, мы рекомендуем вам ознакомиться с материалом и предложить раннюю обратную связь!

### Расписание

С учётом 10-минутных перерывов эта сессия должна занять около 25 минут. Она включает:

| Сегмент                                | Продолжительность |
|----------------------------------------|-------------------|
| Использование системы типов — 25 минут |                   |

Курс охватывает перечисленные ниже темы. Каждая тема может быть рассмотрена на одном или нескольких слайдах в зависимости от её сложности и значимости.

### Основы проектирования API

- Золотое правило: отдавайте приоритет ясности и читаемости в месте вызова. Люди проводят гораздо больше времени, читая места вызова, чем объявления вызываемых функций.
- Сделайте ваш API предсказуемым
  - Следуйте соглашениям об именовании (регистровым соглашениям, предпочтению лексики, принятой в стандартной библиотеке — например, методы должны называться «push», а не «push\_back», «is\_empty», а не «empty» и т.д.)

- Знайте типы и трейты стандартной библиотеки и используйте их в своих API. Если что-то кажется базовым типом или алгоритмом, сначала проверьте стандартную библиотеку.
- Используйте хорошо зарекомендовавшие себя шаблоны проектирования API, которые мы обсудим позже на занятии (например, newtype, пары типов owned/view, обработка ошибок).
- Пишите содержательные и эффективные комментарии для документации (например, не просто повторяйте имя метода с пробелами вместо подчёркиваний, не дублируйте одну и ту же информацию ради заполнения каждого markdown-тега, приводите примеры использования)

## Использование системы типов

- Краткое повторение перечислений, структур и псевдонимов типов
- Паттерн newtype и инкапсуляция: парсить, а не валидировать
- Трейты расширения: избегайте паттерна newtype, когда хотите добавить дополнительное поведение
- RAII, охранники области видимости и drop-бомбы: использование Drop для очистки ресурсов, запуска действий или обеспечения инвариантов
- «Токен»-типы: заставляют пользователей доказывать выполнение конкретного действия
- Паттерн типового состояния: обеспечение корректных переходов состояний на этапе компиляции
- Использование borrow checker для обеспечения инвариантов, не связанных с владением памятью
  - OwnedFd/BorrowedFd в стандартной библиотеке
  - **Брендированные типы**

## Не противьтесь borrow checker

- «Владельческие» типы и «просмотровые» типы: &str и String, Path и PathBuf и др.
- Не скрывайте требования владения: избегайте скрытого .clone(), научитесь ценить Cow
- Разделяйте типы по границам владения
- Структурируйте иерархию владения как дерево
- Стратегии управления циклическими зависимостями: подсчёт ссылок, использование индексов вместо ссылок
- Внутренняя изменяемость (Cell, RefCell)
- Работа с параметрами времени жизни в пользовательских типах данных

## Полиморфизм в Rust

- Краткое повторение трейтов и обобщённых функций
- В Rust отсутствует наследование: каковы последствия?
  - Использование перечислений для полиморфизма
  - Использование трейтов для полиморфизма
  - Использование композиции
  - Как выбрать наиболее подходящий шаблон?
- Работа с обобщениями
  - Обобщённый параметр типа в функции или объект трейта в качестве аргумента?
  - Ограничения трейтов не обязательно должны ссылаться на обобщённый параметр
  - Параметры типа в трейтах: должен ли это быть обобщённый параметр или связанный тип?
- Макросы: ценный инструмент для сокращения кода (DRY), когда трейтов недостаточно или они слишком сложны

## Обработка ошибок

- Какова цель ошибок? Восстановление или сообщение.
- Result и Option
- Проектирование качественных ошибок:
  - Определите область видимости ошибки.
  - Фиксируйте дополнительный контекст по мере распространения ошибки вверх, при пересечении границ областей видимости.
  - Используйте трей Err для отслеживания полной цепочки ошибок.
  - Используйте thiserr для сокращения шаблонного кода при определении типов ошибок.
  - anyhow
- Различайте фатальные ошибки и ошибки, поддающиеся восстановлению, используя Result<Result<T, RecoverableError>, FatalError> .

## Глава 69

# Использование системы типов

Система типов Rust является *expressive*: вы можете использовать типы и трейты для создания абстракций, которые затрудняют неправильное использование вашего кода.

В некоторых случаях вы можете обеспечить корректность на этапе *компиляции* без накладных расходов во время выполнения.

Типы и трейты могут моделировать концепции и ограничения из вашей предметной области. При тщательном проектировании вы можете повысить ясность и поддерживаемость всего кода.

На этот слайд отводится примерно 5 минут.

Дополнительные пункты, которые может упомянуть докладчик:

- Система типов Rust заимствует множество идей из функциональных языков программирования.

Например, перечисления Rust известны как «алгебраические типы данных» в таких языках, как Haskell и OCaml. Вы можете черпать вдохновение из учебных материалов, ориентированных на функциональные языки, при поиске рекомендаций по проектированию с использованием типов. «*Domain Modeling Made Functional*» — отличный ресурс по данной теме с примерами, написанными на F#.

- Несмотря на функциональные корни Rust, не все функциональные шаблоны проектирования могут быть легко перенесены в Rust.

Например, для проектирования API, использующих функции высшего порядка и типы высшего порядка в Rust, необходимо иметь глубокие знания широкого круга продвинутых тем.

Оценивайте в каждом конкретном случае, может ли более императивный подход быть проще для реализации. Рассмотрите возможность использования мутации на месте, полагаясь на borrow-checker и систему типов Rust для контроля того, что и где может изменяться.

- Такую же осторожность следует применять и к объектно-ориентированным шаблонам проектирования. Rust не поддерживает наследование, и декомпозиция объектов должна учитывать ограничения, введённые borrow-checker.
- Следует отметить, что программирование на уровне типов часто используется для создания «абстракций с нулевой стоимостью», хотя этот термин может вводить в заблуждение: влияние на время компиляции и сложность кода может быть значительным.

Этот раздел займет примерно 25 минут. В него входит:

| Слайд                                 | Продолжительность |
|---------------------------------------|-------------------|
| Использование системы типов — 5 минут |                   |
| Паттерн Newtype                       | 20 минут          |

## 69.1 Паттерн Newtype

Newtype — это оболочка вокруг существующего типа, часто примитивного:

```
// Уникальный идентификатор пользователя, реализованный как newtype вокруг `u64`.
pub struct UserId(u64);
```

В отличие от псевдонимов типов, newtype не взаимозаменяется с обернутым типом:

```
fn double(n: u64) -> u64 {
 n * 2
}
```

```
double(UserId(1)); // ❌
```

Компилятор Rust также не позволяет использовать методы или операторы, определённые для базового типа:

```
assert_ne!(UserId(1), UserId(2)); // ❌
```

Этот слайд и его под-слайды должны занять около 20 минут.

- Студенты должны были встретить паттерн newtype в курсе «Основы», когда изучали кортежные структуры.
- Запустите пример, чтобы продемонстрировать студентам сообщение об ошибке от компилятора.
- Измените пример, чтобы использовать typealias вместо newtype, например type MessageId = u64 . Изменённый пример должен успешно компилироваться, что подчёркивает различия между двумя подходами.
- Подчеркните, что newtypes изначально не обладают никаким поведением. Необходимо сознательно выбирать, какие методы и операторы вы хотите перенаправлять от базового типа. В нашем примере UserId разумно разрешить сравнения между UserId, но не имеет смысла разрешать арифметические операции, такие как сложение или вычитание.

### 69.1.1 Семантическая путаница

Когда функция принимает несколько аргументов одного типа, места вызова становятся неочевидными:

```
pub fn login(username: &str, password: &str) -> Result<(), LoginError> {
 // [...]
}

// В другой части кода мы по ошибке меняем аргументы местами.
// Ошибка (в лучшем случае), уязвимость безопасности (в худшем случае).
login(password, username);
```

Паттерн newtype может предотвратить этот класс ошибок на этапе компиляции:

```

pub struct Username(String);
pub struct Password(String);

pub fn login(username: &Username, password: &Password) -> Result<(), LoginError> {
 // [...]
}

login(password, username); // ✘

```

- Запустите оба примера, чтобы продемонстрировать студентам успешную компиляцию исходного примера, а также ошибку компилятора, возвращаемую изменённым примером.
- Особое внимание уделите семантическому аспекту. Паттерн newtype следует использовать для применения различных типов к разным концепциям, тем самым полностью исключая этот класс ошибок.
- Тем не менее, следует учитывать, что существуют легитимные сценарии, когда функция может принимать несколько аргументов одного типа. В таких случаях, если корректность имеет первостепенное значение, рассмотрите возможность использования структуры с именованными полями в качестве входных данных:

```

pub struct LoginArguments<'a> {
 pub username: &'a str,
 pub password: &'a str,
}

// Нет необходимости проверять определение функции `login`, чтобы выявить проблему.
login(LoginArguments {
 username: password,
 password: username,
})

```

Пользователи вынуждены на месте вызова присваивать значения каждому полю, что повышает вероятность обнаружения ошибок.

### 69.1.2 Парсить, а не валидировать

Паттерн newtype может быть использован для обеспечения инвариантов.

```

pub struct Username(String);

impl Username {
 pub fn new(username: String) -> Result<Self, InvalidUsername> {
 if username.is_empty() {
 return Err(InvalidUsername::CannotBeEmpty)
 }
 if username.len() > 32 {
 return Err(InvalidUsername::TooLong { len: username.len() })
 }
 // Другие проверки валидации...
 Ok(Self(username))
 }

 pub fn as_str(&self) -> &str {
 &self.0
 }
}

```

```
}
```

- Паттерн newtype в сочетании с системой модулей и видимости Rust может использоваться для гарантирования того, что экземпляры данного типа удовлетворяют набору инвариантов.

В приведённом выше примере необработанная `String`, хранящаяся внутри структуры `Username`, не может быть напрямую доступна из других модулей или пакетов, поскольку она не помечена как `pub` или `pub(in ...)`. Потребители типа `Username` вынуждены использовать метод `new` для создания экземпляров. В свою очередь, `new` выполняет проверку, обеспечивая, что все экземпляры `Username` удовлетворяют этим условиям.

- Метод `as_str` позволяет потребителям получить доступ к исходному строковому представлению (например, для сохранения в базе данных). Однако потребители не могут изменить исходное значение, поскольку возвращаемый тип `&str` ограничивает их только чтением.
- Инварианты на уровне типа имеют дополнительные преимущества.

Ввод проверяется один раз на границе, и остальная часть программы может полагаться на соблюдение инвариантов. Мы можем избежать избыточной проверки и «оборонительного программирования» по всему коду, снижая шум и повышая производительность.

### 69.1.3 Действительно ли это инкапсулировано?

Необходимо оценить всю поверхность API, предоставляемую `newtype`, чтобы убедиться, что инварианты действительно надёжны. Крайне важно учитывать все возможные взаимодействия, включая реализации трейтов, которые могут позволить пользователям обходить проверки валидации.

```
pub struct Username(String);

impl Username {
 pub fn new(username: String) -> Result<Self, InvalidUsername> { // Проверки
 валидации...
 Ok(Self(username))
 }
}

impl std::ops::DerefMut for Username { // !!
 fn deref_mut(&mut self) -> &mut Self::Target {
 &mut self.0
 }
}
```

- `DerefMut` позволяет пользователям получить изменяемую ссылку на обёрнутое значение. Изменяемая ссылка может использоваться для модификации базовых данных способами, которые могут нарушать инварианты, обеспечиваемые `Username::new!`
- При аудите API нового типа вы можете сузить область видимости до методов и трейтов, предоставляющих изменяемый доступ к базовым данным.
- Напомните студентам о границах приватности.

В частности, функции и методы, определённые в том же модуле, что и новый тип, могут напрямую обращаться к его базовым данным. Если возможно, переместите определение нового типа в отдельный модуль, чтобы сузить область видимости аудита.

## **Часть XVI**

# **Заключительные слова**

## Глава 70

# Спасибо!

*Благодарим вас за прохождение Comprehensive Rust!* Надеемся, что курс был для вас полезен и интересен.

Нам было очень приятно создавать этот курс. Курс не идеален, поэтому если вы заметили ошибки или у вас есть предложения по улучшению, пожалуйста, свяжитесь с нами на GitHub. Мы будем рады получить ваши отзывы.

- Спасибо за чтение заметок докладчика! Надеемся, они оказались полезными. Если вы обнаружите страницы без заметок, пожалуйста, отправьте нам PR и свяжите его с issue #1083. Мы также очень благодарны за исправления и улучшения существующих заметок.

# Глава 71

## Глоссарий

Ниже представлен глоссарий, который призван дать краткое определение многим терминам Rust. Для переводов это также служит для связи термина с его английским оригиналом.

- ```
h1#glossary ~ ul { list-style: none; padding-inline-start: 0; }

h1#glossary ~ ul > li { /* Упростить с помощью «text-indent: 2em hanging», когда поддерживается: https
://caniuse.com/mdn-css_properties_text-indent_hanging */ padding-left: 2em; text-indent: -2em;
}

h1#glossary ~ ul > li:first-line { font-weight: bold; }
```
- **allocate:**
Динамическое выделение памяти в куче.
 - **argument:**
Информация, передаваемая в функцию или метод.
 - **associated type:**
Тип, связанный с конкретным трейтом. Полезно для определения взаимосвязи между типами.
 - **Bare-metal Rust:**
Низкоуровневая разработка на Rust, часто применяемая в системах без операционной системы.
См. Bare-metal Rust.
 - **block:**
См. Блоки и область видимости.
 - **borrow:**
См. Заимствование.
 - **borrow checker:**
Часть компилятора Rust, проверяющая корректность всех заимствований.
 - **brace:**
{ и }. Также называется *фигурная скобка*, они ограничивают блоки.
 - **сборка:**
Процесс преобразования исходного кода в исполняемый код или пригодную для использования программу.
 - **вызов:**
Вызов или выполнение функции либо метода.
 - **канал:**
Используется для безопасной передачи сообщений между потоками.
 - **Комплексный Rust :** 🐛

Курсы здесь совместно называются Комплексный Rust

- конкурентность:

Выполнение нескольких задач или процессов одновременно.

- Конкурентность в Rust:

См. Конкурентность в Rust.

- константа:

Значение, которое не изменяется в ходе выполнения программы.

- управление потоком:

Порядок, в котором отдельные операторы или инструкции выполняются в программе.

- сбой:

Неожиданное и необработанное завершение или сбой программы.

- перечисление:

Тип данных, который содержит одно из нескольких именованных значений, возможно с ассоциированным кортежем или структурой.

- ошибка:

Неожиданное состояние или результат, отклоняющийся от ожидаемого поведения.

- обработка ошибок:

Процесс управления и реагирования на ошибки, возникающие во время выполнения программы.

- упражнение:

Задача или проблема, предназначенная для практики и проверки навыков программирования.

- функция:

Повторно используемый блок кода, выполняющий конкретную задачу.

- сборщик мусора:

Механизм, автоматически освобождающий память, занятую объектами, которые больше не используются.

- обобщения:

Функция, позволяющая писать код с параметрами типов, обеспечивая повторное использование кода с разными типами данных.

- неизменяемый:

Невозможно изменить после создания.

- интеграционный тест:

Тип теста, который проверяет взаимодействие между различными частями или компонентами системы.

- ключевое слово:

Зарезервированное слово в языке программирования, имеющее конкретное значение и не используемое в качестве идентификатора.

- библиотека:

Набор предварительно скомпилированных процедур или кода, используемых программами.

- макрос:

Макросы Rust можно распознать по символу ! в имени. Макросы применяются, когда обычных функций недостаточно. Типичный пример — `format!`, который принимает переменное число аргументов, что не поддерживается функциями Rust.

- функция `main`:

Программы на Rust начинают выполнение с функции `main`.

- `match`:

Конструкция управления потоком в Rust, позволяющая выполнять сопоставление с образцом по значению выражения.

- утечка памяти:

Ситуация, когда программа не освобождает память, которая больше не нужна, что приводит к постепенному увеличению использования памяти.

- метод:

Функция, связанная с объектом или типом в Rust.

- **модуль:**
Пространство имён, содержащее определения, такие как функции, типы или трейты, для организации кода в Rust.
- **move:**
Передача владения значением от одной переменной к другой в Rust.
- **mutable:**
Свойство в Rust, позволяющее изменять переменные после их объявления.
- **ownership:**
Концепция в Rust, определяющая, какая часть кода отвечает за управление памятью, связанной со значением.
- **panic:**
Невосстановимая ошибка в Rust, приводящая к завершению работы программы.
- **parameter:**
Значение, передаваемое в функцию или метод при их вызове.
- **pattern:**
Комбинация значений, литералов или структур, которая может быть сопоставлена с выражением в Rust.
- **payload:**
Данные или информация, передаваемые сообщением, событием или структурой данных.
- **program:**
Набор инструкций, которые компьютер может выполнить для решения конкретной задачи или определённой проблемы.
- **programming language:**
Формальная система, используемая для передачи инструкций компьютеру, такая как Rust.
- **получатель:**
Первый параметр в методе Rust, представляющий экземпляр, для которого вызывается метод.
- **подсчёт ссылок:**
Метод управления памятью, при котором отслеживается количество ссылок на объект, и объект освобождается, когда счётчик достигает нуля.
- **return:**
Ключевое слово в Rust, используемое для указания значения, возвращаемого из функции.
- **Rust:**
Системный язык программирования, ориентированный на безопасность, производительность и конкурентность.
- **Основы Rust:**
Дни с 1 по 4 этого курса.
- **Rust в Android:**
См. Rust в Android.
- **Rust в Chromium:**
См. Rust в Chromium.
- **безопасный:**
Относится к коду, который соблюдает правила владения и заимствования в Rust, предотвращая ошибки, связанные с памятью.
- **область видимости:**
Область программы, в пределах которой переменная действительна и может использоваться.
- **стандартная библиотека:**
Набор модулей, предоставляющих базовую функциональность в Rust.
- **static:**
Ключевое слово в Rust, используемое для определения статических переменных или элементов с 'static' временем жизни .
- **string:**

Тип данных, хранящий текстовую информацию. Подробнее см. в разделе Strings.

- **struct:**
Составной тип данных в Rust, объединяющий переменные разных типов под одним именем.
- **test:**
Модуль Rust, содержащий функции для проверки корректности других функций.
- **thread:**
Отдельная последовательность выполнения в программе, позволяющая выполнять задачи параллельно.
- **thread saflety:**
Свойство программы, обеспечивающее корректное поведение в многопоточной среде.
- **trait:**
Набор методов, определённых для неизвестного типа, обеспечивающий полиморфизм в Rust.
- **trait bound:**
Абстракция, позволяющая требовать от типов реализации определённых трейтов, представляющих ваш интерес.
- **кортеж:**
Составной тип данных, содержащий переменные различных типов. Поля кортежа не имеют имён и доступны по их порядковым номерам.
- **тип:**
Классификация, определяющая, какие операции могут выполняться над значениями определённого вида в Rust.
- **вывод типа:**
Способность компилятора Rust выводить тип переменной или выражения.
- **неопределённое поведение:**
Действия или условия в Rust, не имеющие определённого результата, часто приводящие к непредсказуемому поведению программы.
- **объединение:**
Тип данных, который может содержать значения разных типов, но только одно за раз.
- **модульный тест:**
Rust включает встроенную поддержку для запуска небольших модульных тестов и более крупных интеграционных тестов. См. Модульные тесты.
- **тип unit:**
Тип, не содержащий данных, записываемый как кортеж без элементов.
- **unsafle:**
Подмножество Rust, позволяющее вызывать *неопределённое поведение*. См. Unsafle Rust.
- **переменная:**
Ячейка памяти, в которой хранятся данные. Переменные действительны в области видимости.

Глава 72

Другие ресурсы по Rust

Сообщество Rust создало множество качественных и бесплатных ресурсов в интернете.

Официальная документация

Проект Rust содержит множество ресурсов. Они охватывают Rust в целом:

- [The Rust Programming Language](#): каноническая бесплатная книга о Rust. Подробно рассматривает язык и включает несколько проектов для практики.
- [Rust By Example](#): изучение синтаксиса Rust через серию примеров, демонстрирующих различные конструкции. Иногда включает небольшие упражнения, в которых предлагается расширить код из примеров.
- [Rust Standard Library](#): полная документация стандартной библиотеки Rust.
- [The Rust Reference](#): неполная книга, описывающая грамматику Rust и модель памяти.
- [Rust API Guidelines](#): рекомендации по проектированию API.

Более специализированные руководства, размещённые на официальном сайте Rust:

- [The Rustonomicon](#): охватывает unsafe Rust, включая работу с сырьими указателями и взаимодействие с другими языками (FFI).
- [Асинхронное программирование в Rust](#): охватывает новую модель асинхронного программирования, введённую после написания Книги по Rust.
- [Книга по Embedded Rust](#): введение в использование Rust на встроенных устройствах без операционной системы.

Неофициальные учебные материалы

Небольшой подбор других руководств и учебников по Rust:

- [Изучение Rust опасным способом](#): рассматривает Rust с точки зрения программистов низкоуровневого языка C.
- [Rust для Embedded C-программистов](#): рассматривает Rust с точки зрения разработчиков, пишущих прошивки на C.
- [Rust для профессионалов](#): охватывает синтаксис Rust с помощью сравнений с другими языками, такими как C, C++, Java, JavaScript и Python.

- Rust на Exercism: более 100 упражнений для изучения Rust.
- Учебные материалы Ferrous: серия небольших презентаций, охватывающих как базовые, так и продвинутые аспекты языка Rust. Также рассматриваются такие темы, как WebAssembly и async/await.
- Продвинутое тестирование приложений на Rust: самостоятельный курс, выходящий за рамки встроенного тестового фреймворка Rust. Он охватывает `googletest`, тестирование снимков, мокирование, а также объясняет, как написать собственный пользовательский тестовый каркас.
- Серия для начинающих по Rust и Сделайте первые шаги с Rust: два руководства по Rust, ориентированных на новых разработчиков. Первое состоит из 35 видео, а второе — из 11 модулей, охватывающих синтаксис Rust и базовые конструкции.
- Изучайте Rust с помощью книги Entirely Too Many Linked Lists: глубокое исследование правил управления памятью в Rust через реализацию нескольких различных типов структур списков.

Пожалуйста, ознакомьтесь с Little Book of Rust Books для получения ещё большего количества книг по Rust.

Глава 73

Благодарности

Материал здесь основан на многих отличных источниках документации по Rust. См. страницу с другими ресурсами для полного списка полезных материалов.

Материал Comprehensive Rust лицензирован на условиях лицензии Apache 2.0, подробностисмотрите в [LICENSE](#).

Rust by Example

Некоторые примеры и упражнения были скопированы и адаптированы из Rust by Example. Пожалуйста,смотрите каталог `third_party/rust-by-example/` для подробностей, включая условия лицензии.

Rust на Exercism

Некоторые упражнения были скопированы и адаптированы из Rust на Exercism. Пожалуйста,смотрите каталог `third_party/rust-on-exercism/` для подробной информации, включая условия лицензии.

CXX

Раздел «Взаимодействие с C++» использует изображение из CXX. Пожалуйста,смотрите каталог `third_party/cxx/` для подробной информации, включая условия лицензии.