СТИВ КЛАБНИК, КЭРОЛ НИКОЛС

ПРОГРАММИРОВАНИЕ на RUST

ОФИЦИАЛЬНЫЙ ГАЙД

ОТ КОМАНДЫ РАЗРАБОТЧИКОВ RUST ИЗ MOZILLA FOUNDATION

ББК 32.973.2-018.1
УДК 004.43
К47

Клабник Стив, Николс Кэрол

К47 Программирование на Rust. — СПб.: Питер, 2021. — 592 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1656-0

Официальный гайд по языку программирования Rust от команды разработчиков Rust из Mozilla Foundation. Добро пожаловать в мир Rust!

Этот официальный гид поможет вам создавать более быстрое и надежное программное обеспечение. Высокоуровневая эргономика и низкоуровневое управление часто противоречат друг другу, но Rust бросает вызов этому конфликту.

Авторы книги входят в команду разработчиков языка, а значит, вы получите всю информацию из первых рук — от установки языка до создания надежных и масштабируемых программ. От создания функций, выбора типов данных и привязки переменных вы перейдете к более сложным концепциям:

- Владение и заимствование, жизненный цикл и типажи.
- Гарантированная безопасность программ.
- Тестирование, обработка ошибок и эффективный рефакторинг.
- Обобщения, умные указатели, многопоточность, типажные объекты и сопоставления.
- Работа со встроенным менеджером пакетов Cargo для создания, тестирования, документирования кода и управления зависимостями.
- Продвинутые средства работы с Unsafe Rust.

Вы найдете множество примеров кода, а также три главы, посвященные созданию полноценных проектов для закрепления знаний: игры-угадайки, создание инструмента командной строки и многопоточного сервера.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с No Starch Press. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1718500440 англ.

ISBN 978-5-4461-1656-0

© 2019 by Mozilla Corporation and the Rust Project Developers.
The Rust Programming Language
ISBN: 978-1-71850-044-0, published by No Starch Press
© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Для профессионалов», 2021

Оглавление

Вступление **19**

Предисловие **20**

Благодарности **21**

Об авторах **22**

Введение **23**

Кому подойдет язык Rust 23

Команды разработчиков 23

Студенты 24

Компании 24

Разработчики открытого исходного кода 24

Люди, ценящие скорость и стабильность 24

Для кого эта книга 25

Как пользоваться этой книгой 25

Ресурсы 27

От издательства 27

Глава 1. Начало работы **28**

Установка 28

Установка инструмента rustup в Linux или macOS 29

Установка инструмента rustup в Windows 30

Обновление и деинсталляция 30

Устранение неисправностей 30

Локальная документация 31

Здравствуй, Мир! 31

Создание каталога проектов 31

Написание и выполнение программы Rust 32

Анатомия программы на языке Rust 33

Компиляция и выполнение являются отдельными шагами 34

Здравствуй, Cargo! 35

Создание проекта с помощью Cargo 35

Построение проекта Cargo и его выполнение 37

Сборка для релиза 39

Cargo как общепринятое средство ... 39
Итоги ... 40

Глава 2. Программирование игры-угадайки ... 41

Настройка нового проекта ... 41
Обработка загаданного числа ... 42
Хранение значений с помощью переменных ... 43
Обработка потенциального сбоя с помощью типа Result ... 45
Печать значений с помощью заполнителей макрокоманды println! ... 47
Тестирование первой части ... 47
Генерирование секретного числа ... 47
Использование упаковки для получения большей функциональности ... 48
Генерирование случайного числа ... 50
Сравнение загаданного числа с секретным числом ... 52
Вывод нескольких загаданных чисел с помощью цикличности ... 56
Выход из игры после правильно угаданного числа ... 57
Обработка ввода недопустимых данных ... 58
Итоги ... 60

Глава 3. Концепции программирования ... 61

Переменные и изменяемость ... 61
Различия между переменными и константами ... 63
Затенение ... 64
Типы данных ... 66
Скалярные типы ... 67
Составные типы ... 71
Функции ... 74
Параметры функций ... 75
Инструкции и выражения в телах функций ... 76
Функции с возвращаемыми значениями ... 78
Комментарии ... 80
Управление потоком ... 81
Выражения if ... 81
Повторение с помощью циклов ... 85
Итоги ... 89

Глава 4. Концепция владения ... 90

Что такое владение? ... 90
Правила владения ... 92

Область видимости переменной ... 92
Строковый тип ... 93
Память и выделение пространства ... 94
Владение и функции ... 100
Возвращаемые значения и область видимости ... 101
Ссылки и заимствование ... 102
Изменяемые ссылки ... 104
Висячие ссылки ... 107
Правила ссылок ... 108
Срезовый тип ... 109
Строковые срезы ... 111
Другие срезы ... 115
Итоги ... 115

Глава 5. Использование структур для связанных данных ... 116

Определение и инстанцирование структур ... 116
Использование краткой инициализации полей: когда у переменных и полей одинаковые имена ... 118
Создание экземпляров из других экземпляров с помощью синтаксиса обновления структуры ... 118
Использование кортежных структур без именованных полей для создания разных типов ... 119
Unit-подобные структуры без полей ... 120
Пример программы с использованием структур ... 121
Рефакторинг с использованием кортежей ... 122
Рефакторинг с использованием структур: добавление большего смысла ... 123
Добавление полезной функциональности с использованием типажей с атрибутом derived ... 124
Синтаксис методов ... 126
Определение методов ... 126
Методы с большим числом параметров ... 128
Связанные функции ... 129
Несколько блоков impl ... 130
Итоги ... 131

Глава 6. Перечисления и сопоставление с паттернами ... 132

Определение перечисления ... 132
Выражение match как оператор управления потоком ... 139
Паттерны, которые привязываются к значениям ... 141

Сопоставление с Option<T> ... 142
Совпадения являются исчерпывающими ... 143
Заполнитель _ ... 144
Сжатое управление потоком с помощью if let ... 145
Итоги ... 146

Глава 7. Управление растущими проектами с помощью пакетов, упаковок и модулей ... 147

Пакеты и упаковки ... 148
Определение модулей для управления областью видимости и конфиденциальностью ... 149
Пути для ссылки на элемент в дереве модулей ... 151
Демонстрация путей с помощью ключевого слова pub ... 154
Начало относительных путей с помощью super ... 156
Обозначение структур и перечислений как публичных ... 157
Введение путей в область видимости с помощью ключевого слова use ... 159
Создание идиоматических путей use ... 160
Предоставление новых имен с помощью ключевого слова as ... 162
Реэкспорт имен с использованием pub ... 162
Использование внешних пакетов ... 163
Использование вложенных путей для очистки больших списков use ... 164
Оператор glob ... 165
Разделение модулей на разные файлы ... 165
Итоги ... 167

Глава 8. Общие коллекции ... 168

Хранение списков значений с помощью векторов ... 168
Создание нового вектора ... 169
Обновление вектора ... 169
Отбрасывание вектора отбрасывает его элементы ... 170
Чтение элементов вектора ... 170
Перебор значений в векторе ... 172
Использование перечисления для хранения нескольких типов ... 173
Хранение текста в кодировке UTF-8 с помощью строк ... 174
Что такое тип String? ... 174
Создание нового экземпляра типа String ... 175
Обновление строки ... 176
Индексирование в строках ... 179
Нарезка строк ... 181

Методы перебора строк ... 182
Строки не так просты ... 182
Хранение ключей со связанными значениями в хеш-отображениях ... 183
Создание нового хеш-отображения ... 183
Хеш-отображения и владение ... 184
Доступ к значениям в хеш-отображении ... 185
Обновление хеш-отображения ... 186
Хеширующие функции ... 188
Итоги ... 188

Глава 9. Обработка ошибок ... 190

Неустранимые ошибки и макрокоманда panic! ... 190
Использование обратной трассировки при вызове panic! ... 192
Устранимые ошибки с помощью Result ... 194
Применение выражения match с разными ошибками ... 197
Краткие формы для паники в случае ошибки: unwrap и expect ... 198
Распространение ошибок ... 200
Паниковать! Или не паниковать! ... 205
Примеры, прототипный код и тесты ... 205
Случаи, когда у вас больше информации, чем у компилятора ... 206
Принципы обработки ошибок ... 206
Создание настраиваемых типов для проверки допустимости ... 208
Итоги ... 210

Глава 10. Обобщенные типы, типажи и жизненный цикл ... 211

Удаление повторов путем извлечения функции ... 212
Обобщенные типы данных ... 214
В определениях функций ... 214
В определениях структуры ... 217
В определениях перечислений ... 219
В определениях методов ... 220
Производительность кода с использованием обобщений ... 222
Типажи: определение совместного поведения ... 223
Определение типажа ... 223
Реализация типажа в типе ... 224
Реализации по умолчанию ... 226
Типажи в качестве параметров ... 228
Возвращение типов, реализующих типажи ... 230

Исправление функции largest с помощью границ типажа 231
Использование границ типажа для условной реализации методов 233
Проверка ссылок с помощью жизненных циклов 235
Предотвращение висячих ссылок с помощью жизненного цикла 235
Контролер заимствования 236
Обобщенные жизненные циклы в функциях 237
Синтаксис аннотаций жизненных циклов..... 239
Аннотации жизненных циклов в сигнатурах функций 240
Мышление в терминах жизненных циклов..... 242
Аннотации жизненных циклов в определениях структур 244
Пропуск жизненного цикла 244
Аннотации жизненных циклов в определениях методов 247
Статический жизненный цикл 248
Параметры обобщенного типа, границы типажа и жизненный цикл вместе 249
Итоги..... 249

Глава 11. Автоматизированные тесты 250

Как писать тесты 251
Анатомия функции тестирования 251
Проверка результатов с помощью макрокоманды assert!..... 255
Проверка равенства с помощью макрокоманд assert_eq! и assert_ne!..... 257
Добавление сообщений об ошибках для пользователя..... 260
Проверка на панику с помощью атрибута should_panic..... 261
Использование типа Result<T, E> в тестах 265
Контроль выполнения тестов..... 265
Параллельное и последовательное выполнение тестов 266
Показ результатов функции 267
Выполнение подмножества тестов по имени 268
Игнорирование нескольких тестов, только если не запрошено иное..... 270
Организация тестов..... 271
Модульные тесты 271
Интеграционные тесты..... 273
Интеграционные тесты для двоичных упаковок..... 277
Итоги..... 277

Глава 12. Проект ввода-вывода: сборка программы командной строки..... 278

Принятие аргументов командной строки..... 279
Чтение значений аргументов..... 279
Сохранение значений аргументов в переменных..... 281

Чтение файла ... 282
Рефакторинг с целью улучшения модульности и обработки ошибок ... 283
Разделение обязанностей в двоичных проектах ... 284
Исправление обработки ошибок ... 289
Извлечение алгоритма из функции main ... 292
Разбивка кода в библиотечную упаковку ... 295
Развитие функциональности библиотеки с помощью методики разработки на основе тестов ... 296
Написание провального теста ... 297
Написание кода для успешного завершения теста ... 299
Работа с переменными среды ... 302
Написание провального теста для функции search, нечувствительной к регистру ... 303
Реализация функции search_case_insensitive ... 304
Запись сообщений об ошибках в стандартный вывод ошибок вместо стандартного вывода данных ... 308
Проверка места, куда записываются ошибки ... 308
Запись сообщения об ошибках в стандартный вывод ошибок ... 309
Итоги ... 310

Глава 13. Функциональные средства языка: итераторы и замыкания ... 311

Замыкание: анонимные функции, которые могут захватывать среду ... 311
Создание абстракции поведения с помощью замыканий ... 312
Логический вывод типа и аннотация замыкания ... 317
Ограничения в реализации структуры Cacher ... 322
Захватывание среды с помощью замыканий ... 323
Обработка серии элементов с помощью итераторов ... 326
Типаж Iterator и метод next ... 327
Методы, которые потребляют итератор ... 328
Методы, которые производят другие итераторы ... 329
Использование замыканий, которые захватывают свою среду ... 330
Создание собственных итераторов с помощью типажа Iterator ... 331
Улучшение проекта ввода-вывода ... 334
Удаление метода clone с помощью Iterator ... 334
Написание более ясного кода с помощью итераторных адаптеров ... 337
Сравнение производительности: циклы против итераторов ... 338
Итоги ... 340

Глава 14. Подробнее о Cargo и Crates.io 341
Собственная настройка сборок с помощью релизных профилей 341
Публикация упаковки для Crates.io 343
Внесение полезных документационных комментариев 343
Экспорт удобного публичного API с использованием pub 347
Настройка учетной записи Crates.io 351
Добавление метаданных в новую упаковку 351
Публикация в Crates.io 353
Публикация новой версии существующей упаковки 353
Удаление версий из Crates.io с помощью команды cargo yank 353
Рабочие пространства Cargo 354
Создание рабочего пространства 354
Создание второй упаковки в рабочем пространстве 355
Установка двоичных файлов из Crates.io с помощью команды cargo install 360
Расширение Cargo с помощью индивидуальных команд 361
Итоги 361

Глава 15. Умные указатели 362
Использование Box<T> для указания на данные в куче 363
Использование Box<T> для хранения данных в куче 364
Применение рекурсивных типов с помощью умных указателей Box 365
Трактовка умных указателей как обыкновенных ссылок с помощью типажа Deref 369
Следование по указателю к значению с помощью оператора разыменования 370
Использование Box<T> в качестве ссылки 371
Определение собственного умного указателя 371
Трактовка типа как ссылки путем реализации типажа Deref 372
Скрытые принудительные приведения типов посредством deref с функциями и методами 374
Как принудительное приведение типа посредством deref взаимодействует с изменяемостью 375
Выполнение кода при очистке с помощью типажа Drop 376
Досрочное отбрасывание значения с помощью std::mem::drop 378
Rc<T> — умный указатель подсчета ссылок 380
Применение Rc<T> для совместного использования данных 380
Клонирование Rc<T> увеличивает число ссылок 383
RefCell<T> и паттерн внутренней изменяемости 384

Соблюдение правил заимствования во время выполнения с помощью RefCell<T> ... 384
Внутренняя изменяемость: изменяемое заимствование неизменяемого значения ... 386
Наличие нескольких владельцев изменяемых данных путем сочетания Rc<T> и RefCell<T> ... 392
Циклы в переходах по ссылкам приводят к утечке памяти ... 394
Создание цикла в переходах по ссылкам ... 394
Предотвращение циклов в переходах по ссылкам: превращение Rc<T> в Weak<T> ... 397
Итоги ... 403

Глава 16. Конкурентность без страха ... 404

Использование потоков исполнения для одновременного выполнения кода ... 405
Создание нового потока с помощью spawn ... 407
Ожидание завершения работы всех потоков с использованием дескрипторов join ... 408
Использование замыкания move с потоками ... 410
Использование передачи сообщений для пересылки данных между потоками ... 413
Каналы и передача владения ... 416
Отправка нескольких значений и ожидание приемника ... 417
Создание нескольких производителей путем клонирования передатчика ... 418
Конкурентность совместного состояния ... 420
Использование мьютексов для обеспечения доступа к данным из одного потока за один раз ... 420
Сходства между RefCell<T>/Rc<T> и Mutex<T>/Arc<T> ... 428
Расширяемая конкурентность с типажами Send и Sync ... 428
Разрешение передавать владение между потоками с помощью Send ... 429
Разрешение доступа из нескольких потоков исполнения с помощью Sync ... 429
Реализовывать Send и Sync вручную небезопасно ... 430
Итоги ... 430

Глава 17. Средства объектно-ориентированного программирования ... 431

Характеристики объектно-ориентированных языков ... 431
Объекты содержат данные и поведение ... 432
Инкапсуляция, которая скрывает детали реализации ... 432
Наследование как система типов и как совместное использование кода ... 434
Использование типажных объектов, допускающих значения разных типов ... 435
Определение типажа для часто встречающегося поведения ... 436

Реализация типажа ... 438
Типажные объекты выполняют динамическую диспетчеризацию ... 441
Объектная безопасность необходима для типажных объектов ... 442
Реализация объектно-ориентированного паттерна проектирования ... 443
Определение поста и создание нового экземпляра в состоянии черновика ... 445
Хранение текста поста ... 446
Делаем пустой черновик ... 446
Запрос на проверку статьи изменяет ее состояние ... 447
Добавление метода approve, который изменяет поведение метода content ... 449
Компромиссы паттерна переходов между состояниями ... 452
Итоги ... 457

Глава 18. Паттерны и сопоставление ... 458

Где могут использоваться паттерны ... 459
Ветви выражения match ... 459
Условные выражения if let ... 459
Условные циклы while let ... 461
Циклы for ... 461
Инструкции let ... 462
Параметры функций ... 463
Опровержимость: возможность несовпадения паттерна ... 464
Синтаксис паттернов ... 466
Сопоставление литералов ... 466
Сопоставление именованных переменных ... 466
Несколько паттернов ... 468
Сопоставление интервалов значений с помощью синтаксиса ... 468
Деструктурирование для выделения значений ... 469
Игнорирование значений в паттерне ... 474
Дополнительные условия с ограничителями совпадений ... 479
Привязки @ ... 481
Итоги ... 482

Глава 19. Продвинутые средства ... 483

Небезопасный Rust ... 483
Небезопасные сверхспособности ... 484
Применение оператора разыменования к сырому указателю ... 485
Вызов небезопасной функции или метода ... 487

Обращение к изменяемой статической переменной или ее модифицирование ... 492
Реализация небезопасного типажа ... 493
Когда использовать небезопасный код ... 494
Продвинутые типажи ... 494
Детализация заполнительных типов в определениях типажей с помощью связанных типов ... 494
Параметры обобщенного типа по умолчанию и перегрузка операторов ... 496
Полный синтаксис для устранения неоднозначности: вызов методов с одинаковым именем ... 498
Использование супертипажей, требующих функциональности одного типажа внутри другого типажа ... 502
Использование паттерна newtype для реализации внешних типажей во внешних типах ... 504
Продвинутые типы ... 505
Использование паттерна newtype для безопасности типов и абстракции ... 506
Создание синонимов типов с помощью псевдонимов типов ... 506
Тип never, который никогда не возвращается ... 508
Динамически изменяемые типы и типаж Sized ... 510
Продвинутые функции и замыкания ... 512
Указатели функций ... 512
Возвращающие замыкания ... 514
Макрокоманды ... 515
Разница между макрокомандами и функциями ... 516
Декларативные макрокоманды с помощью macro_rules! для общего метапрограммирования ... 516
Процедурные макрокоманды для генерирования кода из атрибутов ... 519
Как написать настраиваемую макрокоманду derive ... 520
Макрокоманды, подобные атрибутам ... 525
Макрокоманды, подобные функциям ... 526
Итоги ... 527

Глава 20. Финальный проект: сборка многопоточного сервера ... 528
Сборка однопоточного сервера ... 529
Прослушивание TCP-соединения ... 529
Чтение запроса ... 531
HTTP-запрос ... 533
Написание ответа ... 534
Возвращение реального HTML ... 535

Проверка запроса и выборочный ответ ... 537
Небольшой рефакторинг ... 538
Превращение однопоточного сервера в многопоточный ... 540
Моделирование медленного запроса в текущей реализации сервера ... 540
Повышение пропускной способности с помощью пула потоков исполнения ... 541
Корректное отключение и очистка ... 561
Реализация типажа Drop для ThreadPool ... 562
Подача потокам сигнала об остановке прослушивания заданий ... 564
Итоги ... 569

Приложение А. Ключевые слова ... 570
Ключевые слова, употребляемые в настоящее время ... 570
Ключевые слова, зарезервированные для использования в будущем ... 572
Сырые идентификаторы ... 572

Приложение Б. Операторы и символы ... 574
Операторы ... 574
Неоператорные символы ... 576

Приложение В. Генерируемые типажи ... 581
Debug для вывода рабочей информации ... 582
PartialEq и Eq для сравнений равенств ... 582
PartialOrd и Ord для сравнений порядка ... 583
Clone и Copy для дублирования значений ... 583
Хеш для отображения значения в значение фиксированного размера ... 584
Default для значений по умолчанию ... 585

Приложение Г. Полезные инструменты разработки ... 586
Автоматическое форматирование с помощью rustfmt ... 586
Исправляйте код с помощью rustfix ... 586
Статический анализ кода с помощью Clippy ... 588
Интеграция с IDE с помощью языкового сервера Rust Language Server ... 589

Приложение Д. Редакции ... 590

1 Начало работы

Давайте начнем наше «ржавое»[1] путешествие по языку Rust! Здесь есть чему поучиться, но любое путешествие нужно с чего-то начать. В этой главе мы обсудим следующее:

- Установка языка Rust в Linux, macOS и Windows.
- Написание программы, которая выводит `Hello, World!`.
- Использование `cargo`, пакетного менеджера и системы сборки языка Rust.

Установка

Первый шаг — это установка языка Rust. Мы скачаем Rust через `rustup`, инструмент командной строки для управления версиями языка Rust и связанными инструментами. Для скачивания вам понадобится подключение к интернету.

ПРИМЕЧАНИЕ

Если по какой-либо причине вы предпочитаете не использовать инструмент `rustup`, то, пожалуйста, обратитесь к веб-странице установки языка Rust по адресу https://www.rust-lang.org/tools/install/, чтобы ознакомиться с другими вариантами.

Следующие шаги позволяют установить последнюю стабильную версию компилятора языка Rust. Гарантии стабильности Rust обеспечивают, что все примеры из книги, которые компилируются, будут продолжать компилироваться с более новыми версиями языка Rust. Выходные данные могут немного отличаться от версии к версии, поскольку язык Rust часто улучшает сообщения об ошибках и предупреждения. Другими словами, любая новая стабильная версия языка Rust,

[1] Язык программирования Rust получил свое название от грибов семейства ржавчинные (англ. rust fungi; отсюда и перевод слова rusty как «ржавый»), а также от слова «robust» («надежный»). — *Здесь и далее примеч. пер.*

которую вы устанавливаете с помощью этих инструкций, должна работать с содержимым этой книги.

ОБОЗНАЧЕНИЯ КОМАНДНОЙ СТРОКИ

В этой главе и на протяжении всей книги мы покажем несколько команд, используемых в терминале. Все командные строки, которые вы должны вводить в терминале, начинаются с символа $. Вам не нужно вводить сам символ — он указывает на начало каждой команды. Строки, которые не начинаются с $, обычно показывают выходные данные предыдущей команды. В дополнение к этому, в примерах, относящихся к PowerShell, вместо $ будет использоваться символ >.

Установка инструмента rustup в Linux или macOS

Если вы используете Linux или macOS, то откройте терминал и введите следующую команду:

```
$ curl https://sh.rustup.rs -sSf | sh
```

Указанная команда скачивает скрипт и запускает установку инструмента `rustup`, который устанавливает последнюю стабильную версию языка Rust. Возможно, вам будет предложено ввести пароль. Если установка прошла успешно, то появится следующая строчка («Rust установлен. Отлично!»):

```
Rust is installed now. Great!
```

Если хотите, то можете самостоятельно скачать этот скрипт и проверить его перед запуском.

Скрипт установки автоматически добавит язык Rust в системный путь после следующего входа в систему. Если вместо перезагрузки терминала вы хотите сразу же начать использовать язык Rust, то в командной строке выполните следующую команду, чтобы добавить язык Rust в системный путь вручную:

```
$ source $HOME/.cargo/env
```

Как вариант, вы можете добавить следующую строку в свой профиль ~/.bash_profile:

```
$ export PATH="$HOME/.cargo/bin:$PATH"
```

В дополнение к этому вам понадобится какой-то редактор связей (линкер). Скорее всего, он уже установлен, но когда вы пытаетесь скомпилировать программу Rust и видите ошибки, которые говорят о том, что не получается исполнить редактор связей, это означает, что в вашей системе редактор связей не установлен и вам нужно установить его вручную. Компиляторы C, как правило, поставляются вместе с нужным редактором связей. Сверьтесь с документацией вашей платформы

и выясните, как устанавливать компилятор C. Кроме того, некоторые распространенные пакеты Rust зависят от кода C, и им требуется компилятор C. Поэтому, возможно, стоит установить его сейчас.

Установка инструмента rustup в Windows

В случае с Windows перейдите в раздел https://www.rust-lang.org/tools/install/ и следуйте инструкциям по установке языка Rust. В какой-то момент установки вы получите сообщение, объясняющее, что вам также понадобятся инструменты сборки C++ для Visual Studio 2013 или более поздней версии. Самый простой способ приобрести инструменты сборки — загрузить их для Visual Studio 2019 по адресу https://www.visualstudio.com/downloads/#build-tools-for-visual-studio-2019 в разделе «Другие инструменты и платформы».

В остальной части этой книги используются команды, которые работают как в интерпретаторе командной строки cmd.exe, так и в оболочке PowerShell. Если будут определенные различия, то мы объясним, какой из этих инструментов использовать.

Обновление и деинсталляция

После того как вы инсталлировали язык Rust инструментом `rustup`, его легко можно обновить до последней версии. Из командной строки выполните следующий ниже скрипт обновления:

```
$ rustup update
```

Для того чтобы деинсталлировать язык Rust и инструмент rustup, выполните следующий ниже скрипт деинсталляции из командной строки:

```
$ rustup self uninstall
```

Устранение неисправностей

Для того чтобы проверить правильность установки языка Rust, откройте оболочку и введите следующую команду:

```
$ rustc --version
```

Вы должны увидеть номер версии, хеш фиксации и дату фиксации последней стабильной версии, выпущенной в следующем формате:

```
rustc x.y.z (abcabcabc yyyy-mm-dd)
```

Если вы видите эту информацию, значит, вы успешно инсталлировали язык Rust! Если вы не видите эту информацию и находитесь в Windows, то проверьте, что язык Rust содержится в вашей системной переменной `%PATH%`. Если все

правильно, а Rust по-прежнему не работает, то вот несколько вариантов, где можно получить помощь. Самый простой — это канал `#beginners` на официальном веб-сайте языка Rust в мессенджере Discord по адресу https://discord.gg/rust-lang. Там вы можете пообщаться с другими растианами (растианин, на англ. *rustacean*, произносится как «растейшен», — это наше забавное самоназвание), которые вам непременно помогут. Другие замечательные ресурсы включают форум пользователей Users по адресу https://users.rust-lang.org/ и на веб-сайте Stack Overflow по адресу http://stackoverflow.com/questions/tagged/rust/.

Локальная документация

Установщик также содержит локальную копию документации, благодаря чему вы можете читать ее офлайн. Выполните команду `rustup doc`, и локальная копия документации откроется в браузере.

Всякий раз, когда тип или функция предусмотрены стандартной библиотекой и вы не уверены, что она делает или как ее использовать, то для ответов на эти вопросы используйте документацию об интерфейсе программирования приложений (API).

Здравствуй, Мир!

Теперь, когда вы установили язык Rust, давайте напишем первую программу Rust. Традиционно при изучении нового языка пишется небольшая программа, которая выводит на экране текст `Hello, World!`, и поэтому поступим так же.

ПРИМЕЧАНИЕ

Авторы книги исходят из базового знакомства с командной строкой. Язык Rust не предъявляет особых требований к редактированию или инструментам, а также к тому, где находится ваш код, поэтому, если вместо командной строки вы предпочитаете использовать интегрированную среду разработки (IDE), то, пожалуйста, используйте свою любимую IDE. Многие IDE теперь имеют некоторую степень поддержки языка Rust; для получения подробностей перепроверьте документацию интегрированной среды разработки. В последнее время команда разработчиков языка Rust сосредоточилась на обеспечении отличной поддержки IDE, и в этом направлении был достигнут значительный прогресс!

Создание каталога проектов

Вы начнете с создания каталога для хранения кода Rust. Неважно, где располагается код, но для упражнений и проектов из этой книги мы предлагаем создать каталог проектов projects в вашем домашнем каталоге и хранить там все проекты.

Откройте терминал и введите следующие команды, чтобы создать каталог projects и каталог для проекта Hello, world! в каталоге проектов.

Для Linux, macOS и оболочки PowerShell в Windows введите следующее:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

Для интерпретатора командной строки cmd в Windows введите:

```
> mkdir "%USERPROFILE%\projects"
> cd /d "%USERPROFILE%\projects"
> mkdir hello_world
> cd hello_world
```

Написание и выполнение программы Rust

Далее создайте новый исходный файл и назовите его main.rs. Файлы Rust всегда заканчиваются расширением .rs. Если в имени файла вы используете более одного слова, то используйте нижнее подчеркивание, чтобы их отделить. Например, используйте hello_world.rs вместо helloworld.rs.

Теперь откройте файл main.rs, который вы только что создали, и введите код листинга 1.1.

Листинг 1.1. Программа, которая выводит Hello, World!

main.rs

```
fn main() {
    println!("Hello, World!");
}
```

Сохраните файл и вернитесь в окно вашего терминала. В Linux или macOS введите следующие команды для компиляции и выполнения файла:

```
$ rustc main.rs
$ ./main
Hello, World!
```

В Windows вместо команды ./main введите команду .\main.exe:

```
> rustc main.rs
> .\main.exe
Hello, World!
```

Независимо от вашей операционной системы в терминале должен быть выведен строковый литерал *Hello, World!*. Если вы не видите этих данных, то обратитесь к разделу «Устранение неполадок» для получения справки.

Если `Hello, World!` все-таки напечаталось, то примите наши поздравления! Вы официально написали программу на языке Rust. И значит, вы стали программистом на языке Rust — добро пожаловать!

Анатомия программы на языке Rust

Давайте подробно рассмотрим, что только что произошло в программе *Hello, World!*. Вот первый элемент пазла:

```
fn main() {

}
```

Эти строки кода определяют функцию на языке Rust. Функция `main` является особенной: это всегда первый код, который выполняется в каждой исполняемой программе Rust. Первая строка кода объявляет функцию с именем `main`, которая не имеет параметров и ничего не возвращает. Если бы имелись параметры, то они бы вошли внутрь скобок, `()`.

Также обратите внимание на то, что тело функции заключено в фигурные скобки, `{}`. В Rust они требуются вокруг всех тел функций. Правильно размещать открывающую фигурную скобку на той же строке кода, что и объявление функции, добавив один пробел между ними.

На момент написания этой книги инструмент автоматического форматирования под названием rustfmt находился на стадии разработки. Если в своих проектах Rust вы хотите придерживаться стандартного стиля, то rustfmt отформатирует код в определенном стиле. Коллектив Rust планирует в итоге включить этот инструмент в стандартный дистрибутив Rust, подобно rustc. Поэтому в зависимости от того, когда вы читаете эту книгу, он уже может быть установлен на вашем компьютере. Для получения более подробной информации ознакомьтесь с документацией онлайн.

Внутри функции `main` находится следующий код:

```
println!("Hello, World!");
```

Эта строка кода выполняет всю работу в этой маленькой программе: она выводит текст на экране. Здесь следует отметить четыре важные детали. Во-первых, стиль языка Rust предусматривает не табуляцию, а отступ из четырех пробелов.

Во-вторых, инструкция `println!` вызывает макрокоманду языка Rust. Если бы вместо этого она вызвала функцию, то мы бы ввели ее как `println` (без `!`). Мы обсудим макрокоманды языка Rust подробнее в главе 19. Пока же вам просто нужно знать: использование `!` означает, что вы вызываете макрокоманду вместо обычной функции.

В-третьих, вы видите строковый литерал `"Hello, world!"`. Мы передаем его в качестве аргумента макрокоманды `println!`, и этот строковый литерал выводится на экран.

В-четвертых, мы заканчиваем строку кода точкой с запятой (`;`). Она указывает на то, что это выражение закончено и готово начаться следующее. Большинство строк кода Rust заканчиваются точкой с запятой.

Компиляция и выполнение являются отдельными шагами

Вы выполнили только что созданную программу, поэтому давайте рассмотрим каждый шаг в этом процессе.

Перед выполнением программы Rust вам необходимо ее скомпилировать с помощью компилятора языка Rust, введя команду `rustc` и передав ей имя вашего исходного файла, как показано ниже:

```
$ rustc main.rs
```

Если у вас есть опыт работы с C или C++, то вы заметите, что команда похожа на `gcc` или `clang`. После успешной компиляции Rust выводит двоичный исполняемый файл.

В Linux, macOS и PowerShell в Windows вы увидите исполняемый файл, введя в командной строке команду `ls`. В Linux и macOS вы увидите два файла. С помощью PowerShell в Windows вы увидите те же три файла, что и при использовании интерпретатора командной строки cmd.

```
$ ls
main main.rs
```

С помощью интерпретатора командной строки cmd в Windows можно ввести следующее:

```
> dir /B %= опция /B говорит о том, что нужно показывать только имена файлов =%
main.exe
main.pdb
main.rs
```

Здесь вы видите файл исходного кода с расширением .rs, исполняемый файл (main.exe в Windows, но main на всех других платформах) и при использовании Windows файл, содержащий отладочную информацию с расширением .pdb. Таким образом, вы выполняете файл main, или main.exe, вот так:

```
$ ./main # или .\main.exe в Windows
```

Если бы программа main.rs была вашей программой *Hello, World!*, то в терминале было бы выведено `Hello, World!`.

Если вы лучше знакомы с динамическим языком, таким как Ruby, Python или JavaScript, то вы, возможно, не привыкли к компиляции и выполнению программы как к отдельным шагам. Rust — это язык с предварительным компилирова-

нием (Ahead-of-Time-компилированием), то есть вы можете компилировать программу и передавать исполняемый файл кому-то другому и он может выполнять его даже без установки языка Rust. Если же вы дадите кому-то файл .rb, .py или .js, то этот кто-то должен иметь установленную реализацию соответственно языка Ruby, Python или JavaScript. Но на этих языках для компиляции и выполнения программы вам потребуется только одна команда. В дизайне языков все является компромиссом.

Компиляция только с помощью rustc отлично подходит для простых программ, но по мере развития вашего проекта вам захочется управлять всеми вариантами и упрощать распространение вашего кода. Далее мы познакомим вас с инструментом Cargo, который поможет писать реальные программы языка Rust.

Здравствуй, Cargo!

Cargo — это система сборки и пакетный менеджер языка Rust. Большинство растиан используют этот инструмент для управления проектами Rust, потому что Cargo выполняет много работы за вас, в частности построение кода, скачивание библиотек, от которых зависит код, и построение этих библиотек. (Библиотеки, которые нужны коду, мы называем зависимостями.)

Самые простые программы Rust, как та, которую мы уже написали, не имеют никаких зависимостей. Поэтому, если бы мы построили проект *Hello, World!* с помощью пакетного менеджера Cargo, то он бы задействовал только ту часть Cargo, которая занимается построением вашего кода. При написании более сложных программ Rust вы будете добавлять зависимости, и если вы начнете проект с помощью Cargo, то добавлять зависимости будет намного проще.

Поскольку подавляющее большинство проектов Rust используют пакетный менеджер Cargo, в других главах этой книги мы исходим из того, что вы тоже используете Cargo. Если вы пользовались официальными установщиками, описанными в разделе «Установка», то Cargo устанавливается в комплекте с языком Rust. Если же вы инсталлировали Rust каким-то другим способом, то проверьте наличие установленного Cargo, введя в терминал следующее:

```
$ cargo --version
```

Если вы видите номер версии, то она у вас есть! Если же вы видите ошибку, например `command not found` («команда не найдена»), то обратитесь к документации вашего метода установки, чтобы выяснить, как установить Cargo отдельно.

Создание проекта с помощью Cargo

Давайте создадим новый проект с использованием Cargo и посмотрим, чем он отличается от нашего первоначального проекта *Hello, World!*. Вернитесь в каталог

проектов projects (или туда, где вы решили хранить свой код). Затем в любой операционной системе выполните следующие команды:

```
$ cargo new hello_cargo
$ cd hello_cargo
```

Первая команда создает новый каталог с именем *hello_cargo*. Мы назвали проект hello_cargo, и Cargo создает файлы в каталоге с тем же именем.

Перейдите в каталог *hello_cargo* и выведите список файлов. Вы увидите, что Cargo сгенерировал для нас два файла и одну папку: файл Cargo.toml и каталог src с файлом main.rs внутри. Он также инициализировал новый репозиторий Git вместе с файлом .gitignore.

ПРИМЕЧАНИЕ

Git — это распространенная система управления версиями. С помощью флага `--vcs` вы можете изменить команду `cargo new`, чтобы использовать другую систему управления версиями или вообще не использовать никакой системы. Выполните команду `cargo new --help`, чтобы увидеть имеющиеся варианты.

Откройте файл Cargo.toml в любом текстовом редакторе. Указанный файл должен быть похожим на код в листинге 1.2.

Листинг 1.2. Содержимое Cargo.toml, сгенерированное командой cargo new

Cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Ваше имя <you@example.com>"]
edition = "2018"

[dependencies]
```

Этот файл имеет формат TOML («очевидный минимальный язык Тома», от англ. *Tom's Obvious, Minimal Language*), который является конфигурационным форматом Cargo.

Первая строка файла, `[package]`, является заголовком раздела, который указывает на то, что последующие инструкции настраивают пакет. По мере добавления информации в этот файл мы будем добавлять и другие разделы.

Следующие четыре строки задают информацию о конфигурации, необходимую для компиляции программы: название, версия, имя автора и используемая редакция языка Rust. Cargo получает информацию о вашем имени и электронной почте из вашей среды, поэтому, если эта информация неверная, исправьте ее сейчас и сохраните файл. Мы поговорим о ключе `edition` в приложении Д в конце книги.

Последняя строка, `[dependencies]`, является началом раздела, в котором вы перечисляете все зависимости проекта. В языке Rust пакеты с исходным кодом называются упаковками (crate). Для этого проекта нам не нужны другие упаковки, но они нам понадобятся в первом проекте в главе 2, и поэтому мы будем использовать этот раздел зависимостей.

Теперь откройте файл src/main.rs и посмотрите:

src/main.rs

```
fn main() {
    println!("Hello, World!");
}
```

Cargo сгенерировал для вас программу *Hello, World!*, такую же, как мы написали в листинге 1.1! Пока что различия между нашим предыдущим проектом и проектом, который генерируется Cargo, заключаются в том, что Cargo поместил код в каталог src и что у нас есть конфигурационный файл Cargo.toml, который находится в верхнем каталоге.

Использование Cargo подразумевает, что ваши файлы с исходным кодом будут находиться в каталоге src. Верхнеуровневый каталог проекта предназначен только для файлов README, информации о лицензии, конфигурационных файлов и всего остального, что не связано с вашим кодом. Использование Cargo помогает вам организовывать проекты. Здесь есть место для всего, и все находится на своем месте.

Если вы начали проект, в котором не используется пакетный менеджер Cargo, как было в проекте *Hello, World!*, то вы можете сконвертировать его в проект, в котором будет использоваться пакетный менеджер Cargo. Переместите код проекта в каталог src и создайте соответствующий файл Cargo.toml.

Построение проекта Cargo и его выполнение

Теперь давайте посмотрим, чем отличается ситуация, когда мы создаем и выполняем программу *Hello, World!* с помощью Cargo. Из каталога hello_cargo постройте свой проект, введя следующую команду:

```
$ cargo build
    Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
     Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

Эта команда создает исполняемый файл, правда, не в вашем текущем каталоге, а в target/debug/hello_cargo (или target\debug\hello_cargo.exe в Windows). Вы можете выполнить исполняемый файл с помощью следующей команды:

```
$ ./target/debug/hello_cargo # или .\target\debug\hello_cargo.exe в Windows
Hello, world!
```

Если все сделано правильно, то терминал должен вывести `Hello, World!`. Выполнение команды `cargo build` в первый раз также приводит к тому, что Cargo создает новый файл на верхнем уровне — Cargo.lock. Этот файл отслеживает точные версии зависимостей в проекте. Данный проект не имеет зависимостей, поэтому указанный файл немного разрежен. Вам никогда не придется изменять этот файл вручную, Cargo управляет его содержимым за вас.

Мы только что построили проект с помощью команды `cargo build` и выполнили его с помощью команды `./target/debug/hello_cargo`, но мы также можем использовать команду `cargo run` для компиляции кода и последующего выполнения результирующего исполняемого файла, и все это в одной команде:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, World!
```

Обратите внимание, что на этот раз мы не увидели данных, указывающих на то, что Cargo компилировал `hello_cargo`. Cargo выяснил, что файлы не изменились, поэтому он просто запустил двоичный файл. Если бы вы модифицировали исходный код, Cargo перестроил бы проект перед его выполнением, и тогда на выходе вы бы увидели вот это:

```
$ cargo run
    Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
     Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
      Running `target/debug/hello_cargo`
Hello, World!
```

Cargo также предоставляет команду `cargo check`. Эта команда быстро проверяет ваш код, чтобы убедиться в его компилируемости, но не создает исполняемый файл:

```
$ cargo check
    Checking hello_cargo v0.1.0 (file:///projects/hello_cargo)
     Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

Вы, наверное, захотите иметь исполняемый файл? Часто команда `cargo check` выполняется намного быстрее, чем команда `cargo build`, потому что она пропускает этап порождения исполняемого файла. Если вы непрерывно проверяете свою работу во время написания кода, то использование команды `cargo check` ускорит этот процесс! По этой причине многие растиане периодически выполняют команду `cargo check`, когда пишут программу, чтобы убедиться, что она компилируется. Затем, когда они готовы использовать исполняемый файл, они выполняют команду `cargo build`.

Давайте повторим то, что мы уже узнали о Cargo: