**УДК 004.438Rust**
**ББК 32.973.2**
**В67**

**Герберт Волверсон**

**В67** Хорошо ли вы знаете Rust? / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2025. – 140 с.: ил.

**ISBN 978-5-93700-361-4**

В этой книге на примере подборки из 24 нестандартных задач говорится о причудах языка Rust, а иногда и программирования вообще. Но не следует рассматривать причуды языка как его недостатки. Как правило, после углубленного знакомства с языком программистам удается с пользой применять его особенности на практике. Предполагается, что читатель имеет опыт создания и выполнения программ на языке Rust, и на компьютере установлена необходимая среда разработки.

Задачи в данной книге ориентированы на разработчиков программ Rust начального и среднего уровня.

УДК 004.438Rust
ББК 32.973.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (анг.) 978-1-68050-917-5
ISBN (рус.) 978-5-93700-361-4

Copyright © 2022 The Pragmatic Programmers, LLC.
© Оформление, издание, перевод, ДМК Пресс, 2025

# Оглавление

**Предисловие от издательства** ..................... 7

**Благодарности** ..................... 8

**Предисловие** ..................... 9

**Об авторе** ..................... 10

**О коде** ..................... 11

**О вас** ..................... 12

**Будьте открыты к новому** ..................... 13

**Задача 1.** Три с небольшим ..................... 15

**Задача 2.** Нестандартный ввод ..................... 19

**Задача 3.** Преобразование типа ..................... 23

**Задача 4.** Элементы размером с байт ..................... 29

**Задача 5.** Какова длина строки? ..................... 33

**Задача 6.** Перезагрузите Вселенную ..................... 37

**Задача 7.** Туда и обратно ..................... 41

**Задача 8.** И ходит как утка, и крякает как утка ..................... 45

**Задача 9.** Не по порядку ..................... 51

**Задача 10.** Обманчивый X ..................... 55

**Задача 11.** Стопка боксов ........ 59

**Задача 12.** Амнезия ........ 67

**Задача 13.** Измените полярность потока нейтронов ........ 73

**Задача 14.** Измерение структур ........ 79

**Задача 15.** И так до бесконечности ........ 83

**Задача 16.** Удвой или отстань ........ 89

**Задача 17.** Какой длины вектор? ........ 93

**Задача 18.** Изменить неизменяемое ........ 97

**Задача 19.** Бессонница в Токио ........ 101

**Задача 20.** Хэлло, бонжур ........ 111

**Задача 21.** Завязать гордиев узел ........ 117

**Задача 22.** В ожидании Годо ........ 123

**Задача 23.** Константные циклы ........ 127

**Задача 24.** Дом на ранчо ........ 131

**Литература** ........ 134

**Предметный указатель** ........ 135

# Предисловие

Rust – очень последовательный язык. Команда Rust Core немало потрудилась над тем, чтобы Rust делал именно то, что вы хотите, и не преподносил сюрпризов, выполняя какие-то дополнительные действия у вас за спиной. Инструментарий Rust – в особенности Clippy и гарантии безопасности – проверяет программу на наличие типичных ошибок и зачастую предлагает улучшения. Программирующие на Rust знают, что написание программы на нем занимает немного больше времени, но, будучи запущена, она работает именно так, как ожидалось.

В языке Rust есть свои причуды. Иногда они прячутся в зазорах между системами, а иногда являются осознанным проектным решением, призванным избежать худшего. В этой книге мы рассмотрим ряд замкнутых программ на Rust для исследования этих причуд. Каждая программа заставляет вас напрячь мозги и таким образом выучить какой-то аспект Rust, призванный удивить вас. Прочитав код задачи, попробуйте угадать, что она выведет. Возможных ответов три:

- программа не откомпилируется;
- программа порождает неожиданный вывод (например, «Арифметика все еще работает!»);
- программа паникует и завершается с сообщением об ошибке.

После каждой задачи объясняется, почему программа ведет себя именно так и каким образом подобные проблемы могут повлиять на код ваших программ. Чтобы извлечь максимум пользы из этой книги, старайтесь выполнить код, *прежде* чем перевернете страницу и начнете читать ответ и обсуждение. Так вы сможете лучше запомнить пройденное. Понимая причуды Rust, вы сможете лучше писать программы на этом языке и, надеюсь, избежите ловушек в своих проектах.

# Об авторе

Гербер Волверсон – автор книг «Hands-on Rust»[1] и «Rust Roguelike Tutorial»[2]. Он разработал и сопровождает библиотеку с открытым исходным кодом `bracket-lib` (вошедшую в состав Amethyst Foundation) и на протяжении ряда лет принимал участие во многих проектах с открытым исходным кодом[3]. Герберт – единоличный владелец компании Bracket Productions.

1 *https://pragprog.com/titles/hwrust/hands-on-rust/*.

2 *http://bfnightly.bracketproductions.com/rustbook/*.

3 *https://github.com/amethyst/bracket-lib*.

# О коде

Проекты и код максимально краткие и преследуют цель представить минимальный пример для каждой задачи. Примеры являются частью *рабочего пространства* Rust. Для выполнения каждой программы зайдите в каталог примера в своем терминале и наберите `cargo run`.

Для некоторых задач необходимы дополнительные библиотеки. В таких случаях рядом с исходным кодом примера отображается файл `Cargo.toml`.

# О вас

Предполагается, что на вашем компьютере установлен Rust и что вы знакомы с созданием и выполнением приложений на этом языке. Таким образом, задачи ориентированы на разработчиков начального и среднего уровня. (Если вы член команды Rust Core, то, наверное, знаете обо всех этих причудах больше меня.)

Эта книга не учебник; если вы никогда не работали с Rust прежде, то начните с книги «The Rust Programming Language» [KN19] или «Hands-on Rust [Wol21][1].

[1] *https://doc.rust-lang.org/book/*.

# Будьте открыты к новому

В этой книге речь идет о причудах Rust, а иногда и программирования вообще. Rust – фантастический язык, несмотря на все свои странности, и слово «причуды» не следует рассматривать как его критику. Напротив, во многих случаях, когда вы узнаете, *почему* все устроено так, а не иначе, причуды будут выглядеть не столько *причудливыми*, сколько *осознанными* чертами языка.

По мере работы над книгой будьте внимательны и подходите к каждой задаче как сыщик к месту преступления. Все ключи присутствуют, и, поняв обсуждение, вы будете лучше понимать, почему все работает как работает, и как самому не попасть в эту конкретную западню. Возможно, вы даже пополните свой арсенал какими-то новыми приемами.

Если захотите узнать больше, не стесняйтесь обратиться к Герберту по адресу `@herberticus` в Twitter или `u/thebracket` в Reddit.

## Задача 1

# Три с небольшим

three_and_a_bit/src/main.rs

```
fn main() {
  const THREE_AND_A_BIT : f32 = 3.4028236;
  println!("{}", THREE_AND_A_BIT);
}
```

**Угадайте результат**

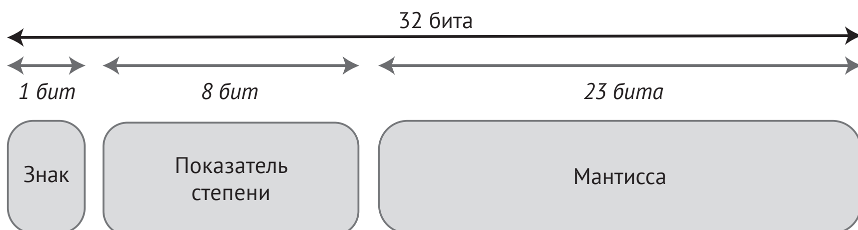
Попробуйте угадать результат, не переворачивая страницу.

Программа печатает следующий результат:

```
3.4028237
```

## Обсуждение

Вы, наверное, ожидали, что программа напечатает `3.4028236`. Но, как ни удивительно, результат отличается на 0.0000001 – вы присвоили значение `3.4028236`, а результат оказался `3.4028237`. Эта разница объясняется тем, как в Rust представляются 32-разрядные числа с плавающей точкой (типа `f32`). В Rust – как и во многих других языках – числа с плавающей точкой представляются согласно стандарту *IEEE-754*, который определяет размещение числа в памяти следующим образом:




В стандарте также имеется формула для извлечения данных из переменной с плавающей точкой в памяти:

$$f32 = \text{знак}\ (-1\ \text{или}\ 1) \times 2^{\text{показатель степени}-127} \times 1.\text{мантисса}.$$

Rust вычисляет, что самый эффективный способ представить число 3.4028236 – использовать показатель степени 2 и мантиссу 1.7014118432998657. Это очень точное приближение: 1.7014118, умноженное на 2, дает 3.4028236 – правильный ответ.
Но, как выясняется, 7014118 не может быть точно представлено с помощью 32 бит. Отметим, что стандарт IEEE-754 предполагает, что начало числа (1.) существует, но фактически оно не хранится. Ближайшее представление равно 7014118432998657, что привносит следующую ошибку:

$$3.4028237 = 1 \times 2^{(128-127)} \times 1.7014118432998657,$$
$$3.4028237 = 1 \times 2^{(128-127)} \times 3.4028236865997314.$$

Цифра, следующая за 6, приводит к округлению с избытком.

**Нам необходимо более широкое число с плавающей точкой**

Иногда проблему точности представления чисел с плавающей точкой можно решить, взяв более широкий тип. Число 3.4028237 можно представить типом `f64`. Если и 64 бит недостаточно, то имеется крейт `f128`, предоставляющий 128-разрядные числа с плавающей точкой (ценой снижения производительности). Это не панацея – некоторые числа упрямо не хотят быть представленными в виде с плавающей точкой. Некоторые константы, например π, вообще нельзя представить точно, без аппроксимации не обойтись. Другие числа представить можно, но не так, как определяет стандарт IEEE-754.

Если вам *действительно* необходимо идеальное представление, то Cargo предлагает математические библиотеки (например, `rug`) с произвольной точностью[1]. Зачастую их использование сопровождается значительной потерей производительности, так что подумайте, какая точность вам действительно необходима.

## Какая точность необходима?

Разным программам нужна разная точность представления чисел с плавающей точкой. Например, в видеоиграх небольшие отклонения в расположении графических объектов обычно остаются незамеченными. Если вы работаете с реальными деньгами, то ошибки округления могут оказаться катастрофическими (обычно используют целый тип, в котором копейки занимают последние два знака, или библиотеку вычислений с фиксированной точкой).

Полностью избежать связанных с точностью проблем можно благодаря изобретательному дизайну. Предположим, что вы проектируете игровой космический конструктор и хотите моделировать конструкции на Земле и Плутоне. В таком случае вряд ли стоит помещать обе планеты в одну систему координат. Вместо этого можно использовать «локальную планетную» систему, которая:

- позволяет использовать координаты с гораздо меньшей точностью;

[1] *https://lib.rs/crates/rug*.

- упрощает учет неприятной привычки небесных тел все время двигаться;
- не тратит ценные диапазоны координат на пустынные области космоса.

Как всегда в информатике, имеется компромисс между производительностью и точностью. Поэтому подумайте, какую задачу вы хотите решить с помощью своей программы, и выберите точность представления, соблюдая баланс между тем, что вам необходимо, и требуемой скоростью работы программы. Числа с плавающей точкой непосредственно поддерживаются процессором – и работают очень быстро. Но даже при встроенном представлении тип `f32` может оказаться быстрее, чем `f64`, потому что 32-разрядные числа занимают меньше памяти и, значит, в кеше их помещается больше. Библиотеки с фиксированной точкой и с произвольной точностью могут работать быстро, но все равно это будет медленнее по сравнению со встроенной поддержкой плавающей точки. Вам решать, какое соотношение между точностью и производительностью приемлемо в вашей программе.

## Для дополнительного чтения

*Стандарт IEEE-754*
*https://en.wikipedia.org/wiki/IEEE_754.*

*RUG – крейт для работы с числами произвольной точности*
*https://lib.rs/crates/rug.*

*Крейт f128*
*https://lib.rs/crates/f128.*

*Крейт fixed*
*https://docs.rs/fixed/1.10.0/fixed/.*

Задача 2

# Нестандартный ввод

standard_input/src/main.rs

```
use std::io::stdin;

fn main() {
    println!("What is 3+2? Type your answer and press enter.");
    let mut input = String::new();
    stdin()
      .read_line(&mut input)
      .expect("Unable to read standard input");
    if input == "5" {
      println!("Correct!");
    } else {
      println!("Incorrect!");
    }
}
```

**Угадайте результат**

Попробуйте угадать результат, не переворачивая страницу.

Взаимодействие с программой будет выглядеть следующим образом:

```
<= What is 3+2? Type your answer and press enter.
=> 5
<= Incorrect!
```

## Обсуждение

Обычно 3 + 2 равно 5, но не тогда, когда за обработку строк берется Rust. Чтобы понять почему, добавим следующую строку в конец программы:

```
println!("{:#?}", input);
```

После этого вы сможете увидеть полную строку, прочитанную Rust из `stdin`:

```
<= What is 3+2? Type your answer and press enter.
=> 5
<= Incorrect!
"5\r\n"
```

В системах на основе UNIX вы увидите `5\n`.

Система стандартного ввода Rust включает *управляющие последовательности*, представляющие клавишу `Enter`. `\r` обозначает возврат каретки, а `\n` – перевод строки. Убрать непечатаемые символы позволяет функция `trim()`.

В следующей программе решение арифметической задачи правильно:

```
use std::io::stdin;

fn main() {
  println!("What is 3+2? Type your answer and press <enter>");
  let mut input = String::new();
  stdin()
    .read_line(&mut input)
   .expect("Unable to read standard input");
  if input.trim() == "5" {
    println!("Correct!");
  } else {
    println!("Incorrect!");
  }
}
```

## Не доверяйте вводу

Есть хорошее правило – никогда не доверять введенной информации. Однако кое-какие действия позволяют свести к минимуму проблемы в случае, если ввод необходим:

- при работе со строками пользуйтесь функцией `trim()` для удаления пробельных символов;
- при сравнении строк пользуйтесь функциями `to_lowercase()` или `to_uppercase()`, чтобы гарантированно сравнивать строки в одном регистре. Эти функции знают о регистре в Юникоде[1];
- при разборе сложных строк пользуйтесь регулярными выражениями для извлечения частей строки.

**Внедрение SQL-кода**

Особенно настороженно относитесь к введенным данным, которые передаются базе данных SQL или другим системам, принимающим текстовые команды. Злонамеренный пользователь мог бы ввести свое имя в виде `10; DROP TABLE members; /*`. Если ваша программа просто конкатенирует строки, то дело может закончиться выполнением в базе данных команды

```
SELECT * FROM members WHERE id=10; DROP TABLE members; /*
```

И это было бы печально, потому что вы только что удалили из базы данных таблицу `members`. Чтобы избежать подобной проблемы, можно использовать параметризованные запросы, которые база данных должна поддерживать.

## Для дополнительного чтения

*Строки*
*https://doc.rust-lang.org/std/string/struct.String.html.*

*Функция trim()*
*https://doc.rust-lang.org/std/string/struct.String.html#method.trim.*

*Крейт Regex*
*https://crates.io/crates/regex/.*

*Шпаргалка по внедрению SQL-кода*
*https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/.*

---

[1] *https://github.com/rust-lang/rust/issues/9363.*

Задача 3

## Преобразование типа

type_conversion/src/main.rs

```
fn main() {
  let x : u64 = 4_294_967_296;
  let y = x as u32;
  if x == y as u64 {
    println!("x equals y.");
  } else {
    println!("x does not equal y.");
  }
}
```

**Угадайте результат**

Попробуйте угадать результат, не переворачивая страницу.

Программа печатает следующий результат:

x does not equal y.

## Обсуждение

Ключевое слово `as` в Rust ведет к *потере информации*. А когда вы используете его для преобразования типов, то есть риск потерять точность без предупреждения.

В этом примере переменной `y` присваивается значение `4_294_967_296`, но результат обрезается, потому что число больше максимального значения 32-разрядного целого без знака. Сюрприз в том, что ни компилятор Rust, ни Clippy, ни среда выполнения не выдают ни ошибки, ни предупреждения о потере данных.

Если все-таки планируете использовать ключевое слово `as` для преобразования типов, – поскольку чаще всего Rust производит это преобразование без проблем, – то имейте в виду следующее:

- преобразование меньшего типа в больший (например, `u32` в `u64`) никогда не приводит к потере точности, так что вы в безопасности;
- при работе с числами, которые гарантированно умещаются в оба типа, потери данных не произойдет. Но будьте осторожны с данными, полученными от пользователей или в результате вычислений, – если вы не контролируете данные, то не можете быть *уверены* в том, что они окажутся в допустимых диапазонах;
- будьте осторожны с преобразованиями чисел с плавающей точкой в целые, потому что Rust всегда округляет с недостатком. А раз так, то лучше явно указать желаемое поведение, вызвав функцию `my_float.floor()` для округления с недостатком, `my_float.ceil()` для округления с избытком и `my_float.round()` для обычного численного округления. Если вам нужно округление, то производите его *до* использования `as`.

По счастью, Rust предлагает кое-какую помощь и другие способы справиться с преобразованием типов.

## Литеральные и нелитеральные значения

Если вы имеете дело с *литеральными* значениями (например, определенными непосредственно в исходном коде), то компилятор Rust

обладает даром определять, что значение не поместится в тип. Например, взгляните на этот код:

```
let x: u32 = 18_446_744_073_709_551_615;
```

Он не компилируется с сообщением об ошибке «the literal 18_446_744_073_709_551_615 does not fit into the type u32 whose range is 0..=4294967295» (литерал 18_446_744_073_709_551_615 не помещается в тип u32, имеющий диапазон 0..=4294967295).

Rust может также защитить вас от ошибок вследствие арифметических операций над литералами. Например, предложение let x = 4_294_967_295 * 2; не откомпилируется.

При работе с нелитеральными переменными (например, полученными от пользователя или вычисленными) компилятор Rust не может увидеть значение заранее. Используя as, вы говорите Rust «я знаю, что делаю».

Еще один вариант – не использовать as, тогда Rust будет проще защитить вас от неожиданного поведения.

## Как защититься от потери точности

Rust предлагает характеристику (trait) Into для безопасных преобразований типов на этапе компиляции. Например, можно следующим образом преобразовать тип u32 в u64:

```
let y = u32::max_value();
let z: u64 = y.into();
```

Характеристика Into решает проблему потенциально невозможных преобразований тем, что не реализует их.

Так, выполнить обратное преобразование – u64 в u32 – с помощью Into не получится. Если вы попробуете написать let z : u32 = (12_u64).into(), то вызов функции into() не откомпилируется.

Для теоретически возможных преобразований Rust предлагает другую характеристику: TryInto. В следующем коде try_into() пытается преобразовать u64 в u32:

```
use std::convert::TryInto;
let z: u32 = (5000_u64).try_into().expect("Conversion error");
```

Функция try_into() возвращает тип Result. К его содержимому можно обратиться так же, как к другим значениям типа Result. Например, вы можете:

- распаковать содержимое функцией `unwrap` и «грохнуться», если при преобразовании возникла ошибка;
- распаковать содержимое функцией `unwrap_or` и подставить значение по умолчанию в случае ошибки;
- применить к `Result` функцию `match`, чтобы обработать ошибку явно;
- использовать `expect`.

В примере выше использована `expect`. Если заменить `5000` числом, не помещающимся в 32-разрядное целое без знака, то программа в панике завершится при попытке выполнить преобразование.

**Избыточные преобразования типа**

Очень многие преобразования типа можно квалифицировать как «запашок в коде», т. е. указание на какой-то порок в ваших рассуждениях. Если все функции, в которых используется переменная `x`, ожидают, что ее значение имеет тип `u32`, то так и объявите ее с самого начала. Если впоследствии появятся функции, ожидающие значения типа `usize`, то вы сможете сделать свой код гораздо чище, если выполните преобразование один раз, а не при каждом вызове.

## Нахождение ошибок преобразования типа с помощью Clippy

Rust включает инструмент *Clippy*, который помогает находить ошибки в коде. Для его вызова наберите в терминале команду `cargo clippy` – и в ответ получите список найденных Clippy ошибок. Параметры Clippy по умолчанию заданы так, что в примере выше не будут найдены все ошибки, но более строгий режим `pedantic` может обнаружить потенциальные проблемы. Чтобы включить режим `pedantic`, добавьте одну строку в начало своего файла `main.rs`:

```
#[warn(clippy::pedantic)]
```

Теперь Clippy выдает следующие предупреждения при выполнении `cargo clippy`:

```
warning: casting `u64` to `u32` may truncate the value
warning: casting `u32` to `u64` may become silently lossy if you later
change the type
```

В педантичном режиме Clippy радостно сообщает обо всех замеченных потенциальных ошибках – даже если они не приводят к проблемам. Многие разработчики считают такой уровень информирования назойливым. Кроме того, педантичная проверка может замедлять работу в больших проектах. Приемлемый компромисс – периодически запускать Clippy в педантичном режиме, а после обработки результатов закомментировать эту строку.

## Для дополнительного чтения

*«as»*
*https://doc.rust-lang.org/std/keyword.as.html.*

*Считать ли «as» вредным?*
*https://users.rust-lang.org/t/as-considered-harmful/35338.*

*into*
*https://doc.rust-lang.org/std/convert/trait.Into.html.*

*try_into*
*https://doc.rust-lang.org/std/convert/trait.TryInto.html.*

*округление f32*
*https://doc.rust-lang.org/std/primitive.f32.html#method.round.*