Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу *dmkpress@gmail.com*, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Благодарности

Эта книга не состоялась бы без терпения, поддержки и любви моей жены, Мел Волверсон.

Маргарет Элдридж заслуживает особой благодарности за то, что предложила мне написать эту книгу. Я уже наполовину написал «Hands-on Rust», когда она подкатила ко мне с этим названием. После минутного ошеломления от такой инициативы со стороны издателя я согласился. Я также благодарен всему коллективу издательства Pragmatic Programmers, в особенности Дэйву Рэнкину и Мики Тебека, которые позволили мне небольшие вольности с жанром серии «Хорошо ли вы знаете…».

Спасибо редактору Тэмми Корону за помощь и неравнодушное отношение к этому проекту. Тэмми – поразительный редактор, который далеко вышел за пределы своих обязанностей и протестировал трудный код на сайте *Rust Playground*, не давая мне при этом отвлекаться и сопровождая на всем трудном пути к публикации книги. Без него я бы эту книгу не закончил.

Мои родители – Роберт Волверсон и Дон Макларен – заслуживают безмерной благодарности. Они пробудили во мне любовь к компьютерам в юном возрасте и побуждали экспериментировать и учиться. Сейчас оба они педагоги на пенсии, и именно им я обязан пожизненным стремлением учиться и учить.

Отдельное спасибо Кенту Фрёшле и Стивену Тэрнеру, моим коллегам по *iZones*, за постоянную поддержку, предоставление гибкого рабочего дня и терпеливое отношение как к глобальной пандемии, так и к моему писательскому графику.

Эта книга не стала бы тем, чем стала, без терпеливой и кропотливой помощи со стороны технических рецензентов: Юргиса Балчиунаса, Фореста Андерсона, Владислава Батыренко, База Залмстра, Ремко Куйпера, Энди Лестера и бесчисленных читателей бета-версий, которые сообщали об опечатках и задавали вопросы. Спасибо также Стиву Коттерилу, который был моим слушателем на протяжении всего творческого процесса.

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

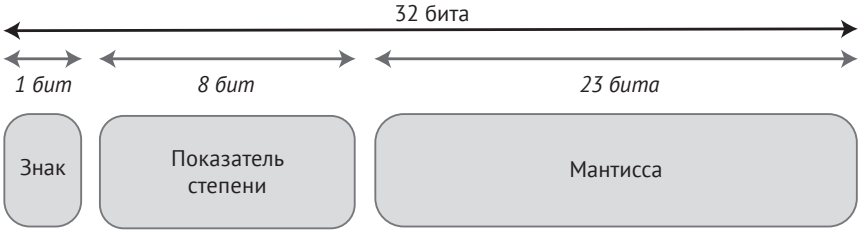
Попробуйте угадать результат, не переворачивая страницу.

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:

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

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!");
  }
}
```