Дата последней сборки документа:
15 февраля 2023 г. в 04:04

Практика использования и наиболее полезные конструкции языка Rust

Содержание

1. Ресуры по языку Rust

https://www.rust-lang.org/tools
https://doc.rust-lang.org/book/
https://doc.rust-lang.org/stable/rust-by-example/

2. Установка Rust

Установить Rust проще всего с помощью утилиты `rustup` – это установщик языка и менеджер версий. Для операционной системы Windows можно скачать `rustup-init.exe` со страницы проекта https://www.rust-lang.org/learn/get-started

Установить Rust на Linux можно так

```
$ curl https://sh.rustup.rs -sSf | bash
...
Current installation options:


default host triple: x86_64-unknown-linux-gnu
default toolchain: stable (default)
profile: default
modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
>1

info: profile set to 'default'
info: default host triple is x86_64-unknown-linux-gnu
info: syncing channel updates for 'stable-x86_64-unknown-linux-gnu'
...
```

Rust часто обновляется и чтобы получить последнюю версию, можно воспользоваться командной `rustup update`.

Собрать проект и обновить его зависимости можно с помощью утилиты `cargo`

```
cargo build  # build your project
cargo run  # cargo run
cargo test  # test project
cargo doc  # build documentation for your project
cargo publish  # publish a libarary to crates.io
```

То есть `cargo` знает, как превратить Rust-код в исполняемый бинарный файл, а также может управлять процессом загрузки и компиляции проектных зависимостей.

3. Вводные замечания

Система владения устанавливает *время жизни* каждого значения, что делает ненужным сборку мусора в ядре языка и обеспечивает надежные, но вместе с тем гибкие интерфейсы для

управления такими ресурсами, как сокеты и описатели файлов. Передеча (move) позволяет передавать значение от одного владельца другому, а заимствование (borrowing) – использовать значение временно, не изменяя владельца.

Rust – типобезопасный язык. Но что понимается под типобезопасностью? Ниже приведено определение «неопределенного поведения» из стандарта языка C 1999 года, известного под названием «C99»: *неопределнное поведение – это поведение, являющееся следствием использования непереносимой или некорректной программной конструкции либо некорректных данных, для которого в настоящем Международном стандарте нет никаких требований.*

Рассмотрим следующую программу на C

```
int main(int argc, char **argv) {
    // объявление одноэлементного массива беззнаковых длинных целых чисел
    unsigned long a[1];
    // обращение к 4-ому элементу массива; индекс, нарушает границу диапазона
    a[3] = 0x7ffff7b36cebUL;
    return 0;
}
```

Эта программа обращается к элементу за концом массива `a`, поэтому согласно C99 ее *поведение не определно*, т.е. она может делать все что угодно. «Неопределенная» операция не просто возвращает неопределнный результат, она дает программе карт-бланш на *произвольное выполнение*(!).

C99 предоставляет компилятору такое право, чтобы он мог генерировать более быстрый код. Чем возлагать на компилятор ответственность за обнаружение и обработку странного поведения вроде выхода за конец массива, стандарт предполагает, что программист должен позаботиться о том, чтобы такие ситуации никогда не возникали.

Если программа написана так, что ни на каком пути выполнения *неопределенное выполнение невозможно*, то будем говорить, что программа *корректна* (well defined).

Если встроенные в язык проверки *гарантируют корректность программы*, то будем называть язык *типобизопасным* (type safe).

Тщательно напсанная программа на C или C++ может оказаться типобезопасной, но ни C, ни C++ не является типобезопасным языком: в приведенном выше примере нет ошибок типизации, и тем не менее она демонстрирует неопределенное поведение. С другой стороны, Python – *типобезопасный* язык, его интерпретатор тратит время на обнаружение выхода за границы массива и обрабатывает его лучше, чем компилятор C.

4. Начало работы

Создать проект на Rust можно командной `cargo new <project_name>`

```
$ cargo new hello  # создать проект hello
$ tree
.
hello/
  Cargo.toml
  src/
    main.rs
$ cd hello
$ cargo run  # запустить проект
   Compiling hello v0.1.0 (/home/kosyachenko/Projects/GARBAGE/rust_projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.42s
```

```
Running `target/debug/hello`
Hello, world!
# Дерево проекта изменилось
$ tree
.
Cargo.lock  # артефакт
Cargo.toml
src/
  main.rs
target/  # артефакт
  CACHEDIR.TAG
  debug/
    build
    deps/
      hello-27...
      hello-27...d
    examples/
    hello
    hello.d
    incremental/
      hello-imy.../
        s-ghim...
```

В основном каталоге имеется файл `Cargo.toml`, содержащий описание метаданных проекта, таких как имя проекта, его версия и его зависимости. Исходный код попадает в директорию `src`.

Выполнение команды `cargo run` привело также к добавлению к проекту новых файлов. Теперь у нас в основном каталоге проекта есть файл `Cargo.lock` и каталог `target`. В `Cargo.lock` указываются конкретные номера версий всех зависимостей, чтобы будущие сборки составлялись точно также, как и эта, пока содержимое `Cargo.toml` не изменится.

4.1. Первая программа на Rust

Нужно как обычно с помощью `cargo new hello` создать новый проект. Перейти в созданную директорию проекта и в файле `main.rs` директории `src` написать следущее

./src/main.rs

```
fn greet_world() {
    println!("Hello, world!");
    let southern_germany = "Germany";
    let japan = "Japan";
    let regions = [southern_germany, japan];

    for region in regions.iter() {
        println!("{}", &region);
    }
}

fn main() {
    greet_world();
}
```

Восклицательный знак свидетельствует об использовании *макроса*. Для операции присваивания в Rust, которую правильнее было бы называть *привязкой переменной*, используется ключевое слово `let`. Поддержка Unicode предоставляется самим языком.

Для *литералов массива* используются *квадратные скобки*. Для возврата итератора метод `iter()` может присутствовать во многих типах. Амперсанд «заимствует» `region` так, чтобы доступ предоставлялся *только для чтения*.

Строки ганатировано получают кодировку UTF-8.

Пример

src/main.rs

```
fn main() {  // (1)
    let penguin_data = "\  // (2)
    common name,length (cm)
    Little penguin,33
    Yellow-eyed penguin,65
    Fiordland penguin,60
    Invalid,data
    ";

    let records = penguin_data.lines();

    for (i, record) in records.enumerate() {
        if i == 0 || record.trim().len() == 0 {  // (3)
            continue;
        }

        let fields: Vec<_> = record  // (4)
            .split(',')  // (5)
            .map(|filed| field.trim())  // (6)
            .collect();  // (7)
            if cfg!(debug_assertations) {  // (8)
                eprintln!("debug: {:?} -> {:?}", record, fields);  // (9)
            }

            let name = fields[0];
            if let Ok(length) = fields[1].parse::<f32>() {  // (10)
                println!("{}, {} cm", name, length);  // (11)
            }
    }
}
```

(1) – исполняемым проектам требуется функция `main()`. (2) – отключение завершающего символа новой строки. (3) – пропуск строки заголовка и строк, состоящих из одних пробелов. (4) – начало со строки текста. (5) – разбиение записи на поля. (6) – обрезка пробелов в каждом поле. (7) – Сборка набора полей. (8) – `cfg!` проверяет конфигурацию в процессе компиляции. (9) – `eprintln!` выводит данные на страндартное устройство сообщений об ошибках (stderr). (10) – попытка выполнения парсинга поля в виде числа с плавающей точкой. (11) – `println!` помещает данные на страндартное устройство вывода (stdout).

Переменная `fields` помечена типом `Vec<_>`. `Vec` – сокращение от `_verctor_`, типа коллекции, способного динамически расширяться. Знак подчеркивания предписывает Rust вывести тип элемента.

На Python решение выглядело бы так

```
#!python
import typing as t

def main():
    penguin_data: str = """
```

```
        common name, length (cm)
        Little penguin, 33
        Yellow-eyed penguin, 65
        Fiordland penguin, 60
        Invalid, data
    """

    records: t.List[str] = penguin_data.split("\n")

    for (i, record) in enumerate(records):
        if i == 0 or len(record.strip()) == 0:
            continue

        fields: t.List[str] = list(map(lambda field: field.strip()), records.split(","))
        # Или с помощью спискового включения
        # fields: t.List[str] = [record.strip() for record in records.split(",")]

        if __debug__:
            print(f"debug: {record} -> {fields}")

        name: str = fields[0]
        # На Rust это блок выглядит изящнее
        try:
            length = float(fields[1])
        except ValueError as err:
            continue
        else:
            print(f"{name}, {length} cm")


if __name__ == "__main__":
    main()
```

Макросы похожи на функции, но вместо возвращения данных они возвращают код. Макросы часто используются для упрощения общеупотребительных шаблонов. Поле заполнения `{}` заставляет Rust воспользоваться методом представления значения в виде строки, который определил программист, а не представлением по умолчанию, доступным при указании поля заполнителя `{:?}`.

`if let Ok(length) = fields[1].parse::<f32>()` читается так «попытаться разобрать `fields[1]` в виде 32-разрядного числа с плавающей точкой, и в случае успеха присвоить число переменной `length`».

Конструкция `if let` – краткий метод обработки данных, предоставляющий также локальную переменную, которой присваиваются эти данные. Метод `parse()` возвращает `Ok(T)` (где `T` означает любой тип), если ему удается провести разбор строки; в противном случае он возвращает `Err(E)` (где `E` означает тип ошибки). Применение `if let Ok(T)` позволяет пропустить любые случаи ошибок, подобные той, что встречаются при обработки строки `Invalid,data`.

Когда Rust не способен вывести тип из окружающего контекста, он запрашивает конкретное указание. В вызов `parse()` включается встроенная аннотация типа в виде `parse::<f32>()`.

Преобразование исходного кода в исполняемый файл называется *компиляцией*.

В Rust-програмах отсуствуют:

1. Висячие указатели – прямые ссылки на данные, ставшие недействительными в ходе выполнения программы,

2. Состояние гонки – неспособность из-за изменения внешних факторов определить, как программа будет вести себя от запуска к запуску,
3. Переполнение буфера – попытка обращения к 12-му элементу массива, состоящего из 6 элементов

В Rust *пустой тип*: `()` (произносится как «юнит»). Когда нет никакого другоо значимого возвращаемого значения, выраженипе возвращает `()`.

Rust предлагает программистам детальный контроль над размещением структур данных в памяти и над схемами доступа к ним. Временами возникает острая потребность в управлении производительностью приложения. При этом важную роль может сыграть хранение данных в *стеке*, а не в *куче*.

Особые возможности Rust:

- Достижение высокой производительности,
- Выполнение одновременных (параллельных) вычислений,
- Достижение эффективной работы с памятью.

Rust позволяет воспользоваться всей доступной производительностью комьютера. Он не использует для обеспечения безопасности памяти сборщик мусора.

В Rust нет никакой глобальной блокировки интерпретатора, ограничивающей скорость потока.

Единицей компиляции программы на Rust является не отдельный файл, а целый пакет (известный как *крейт*). Поскольку крейты могут включать в себя несколько модулей, они могут становится весьма большими объектами компиляции. Это конечно, позволяет оптимизировать весь крейт, но требует также его компиляции.

`let` используется для *привязки переменной*. По умолчанию переменные *неизменяемы*, то есть предназначены только для чтения, а не для чтения-записи.

5. Основы языка

5.1. Числа

Преобразования между типами всегда носят явный характер. В Rust у чисел могут быть методы: например, для округления `24.5` к ближайшему целому числу используется `24.5_f32.round()`, а не `round(24.5)`.

Литералы чисел с плавающей точкой без явно указанной аннотации типа становятся 32- или 64-разрядными в зависимости от контекста.

Имеющиеся в Rust требования к безопасности типов *не позволяют проводить сравнение между типами*. Например, следующий код не пройдет компиляцию:

```
fn main() {
    let a: i32 = 10;
    let b: u16 = 100;

    if a < b {  // error[E0308]: mismatched types
        println!("Ten is less then one hundred.")
    }
}
```

Безопаснее всего привести меньший тип к большему (например, 16-разрядный тип к 32-разрядному): `(b as i32)`. Иногда это называют расширением.

Порой использовать ключевое слово `as` накладывает слишком большие ограничения. В следущем листинге показан Rust-метод, заменяющий ключевое слово `as` в тех случаях, когда приведение может дать сбой

```
use std::convert::TryInto;

fn main() {
    let a: i32 = 10;
    let b: u16 = 100;

    let b_ = b.try_into().unwrap();  // try_into() -> mun Result

    if a < b_ {
        println!("Ten is less than one hundred.");
    }
}
```

Ключевое слово `use` переносит типаж `std::convert::TryInto` в локальную область видимости. В результате этого происходит разблокирование метода `try_into()`, вызываемого в отношении переменной `b`.

Типажи можно рассматривать как *абстрактные классы* или *интерфейсы*. Метод `b.try_into()` возвращает значение типа `i32`, завернутое в значение типа `Result`. Значение успеха может быть обработано методом `unwrap()`, в результате чего здесь будет возвращено значение `b`, имеющее тип `i32`.

Rust включает ряд допуско, позволяющих сравнивать числовые значения с плавающей точкой. Эти допуски определяются как `f32::EPSILON` и `f64::EPSILON`.

Операции, выдающие математически неопределенные результаты, например извлечение квадратного корня из отрицательного числа, создают особые проблемы. Для обработки таких случаев в тип числа с плавающей точкой включены знаения NaN – «not a number».

Чтобы добавить контейнер (крейт) в проект достаточно добавить в раздел `[dependencies]` имя крейта и его версию в файл `Cargo.toml`

Cargo.toml

```
...
[dependencies]
num = "0.4"
...
```

```
use num::complex::Complex;

fn main() {
    let a = Complex {re: 2.1, im: -1.2};  // у каждого типа в Rust имеется литеральный синтаксис
    let b = Complex::new(11.1, 22.2);
    let result = a + b;

    println!("{} + {}i", result.re, result.im);  // доступ к полям через оператор точка
}
```

Ключевое слово `use` помещает тип `Complex` в локальную область видимости. В Rust нет конструкторов, вместо этого у каждого типа есть литеральная форма.

Инициализировать типы можно путем использования имени типа и присвоения его полям значений в фигурных скобках: `Complex { re: 2.1, im: -1.2 }`. Для упрощения программ метод `new()` реализован у многих типов. Но это соглашение не часть языка Rust.

Поддерживаются две форму инициализации неэлементарных типов:

- Литеральный синтаксис: `Complex { re: 2.1, im: -1.2 }`,
- Статическим методом `new()`: `Complex::new(11.1, 22.2)`.

Статический метод – это функция, доступная для *типа*, но не для *экземпляра типа*. В реальном коде предпочтительнее вторая форма.

5.2. Управление ходом выполнения программы

Базовая форма цикла `for` имеет следующий вид

```
for item in container {
    // ...
}
```

Эта базовая форма делает каждый последующий элемент в контейнере `container` доступным в качестве элемента `item`.

Несмотря, на то, что переменная `container` остается в локальной области видимости, теперь ее *время жизни* истекло. Rust считает, что раз блок закончился, то надобности в переменной `container` миновала.

Когда чуть позже в программе возникнет желание воспользоваться переменной `container` еще раз, следует воспользоваться указателем. Когда указатель опущен, Rust полагает, что переменная `container` больше не нужна. Чтобы добавить *указатель* на контейнер, нужно, как показано в следующем примере, поставить перед его именем знак амперсанда (`&`)

```
for item in &container {
    // ...
}
```

Если в ходе циклического перебора элементов нужно внести изменения в каждый элемент, можно воспользоваться *указателем, допускающим изменения*, включив в код ключевое слово `mut`

```
for item in &mut container {
    // ...
}
```

Безымянные циклы. Если в блоке не используется локальная переменная, то по соглашению применяется знак подчеркивания «`_`». Использование этой схемы в сочетании с синтаксисом *исключающего диапазона* (`n..m`) и синтаксисом *включающего диапазона* (`n..=m`) показывает, что целью является выполнение цикла фиксированное количество раз. Например

```
for _ in 0..10 {
    // ...
}
```

Ключевое слово `continue` действует вполне ожидаемым образом

```
// Вывести только нечетные
for item in 0..10 {
    if item % 2 == 0 {
        continue;
    }
}
```

Прерывание цикла выполняется с помощью ключевого слова `break`. При этом Rust работает превычным образом

```
fn main() {
    for (x, y) in (0..).zip(0..) { // zip работает на бесконечной последовательности
        if x + y > 100 {
            break;
        }
        println!("x={}, y={}", x, y);
    }
}
```

В Python пришлось бы организовывать бесконечный цикл `while`, например так

```
def main():
    x, y = (0, 0)
    while True:
        if x + y > 100:
            break
        print(f"x={x}, y={y}")
        x += 1
        y += 1
```

Прерывание во вложенных циклах. Прервать выполнение вложенного цикла можно с помощью *меток циклов*. Метка цикла представляет собой идентификатор с префиксом в виде *апострофа* ’

```
‘outer: for x in 0.. {
    for y in 0.. {
        for z in 0.. {
            if x + y + z > 10 {
                break ‘outer;
            }
            // ...
        }
    }
}
```

Условное ветвление. `if` допускает применение любого выражения, вычисленного в булево значение (`true` или `false`). Когда нужно протестировать несколько значений, можно добавить цепочку блоков `if else`. Блок `else` соответствует всему, чему еще не нашлось соответствие. Например

```
if item == 42 {
    // ...
} else if item == 132 {
    // ...
} else {
    // ...
}
```

В Rust отсутствует концепция «правдивых» или «ложных» типов. В других языках (например, в Python) допускается, чтобы особые значения, например 0 или пустая строка, означали `false`, а другие значения означали `true`, но в Rust это не практикуется. Единственным значением, которое может быть `true`, является `true`, а за `false` может принимать только `false`.

Rust – язык, основанный на выражениях. Для Rust характерно обходится без ключевого слова `return`

```
fn is_even(n: i32) -> bool {
```

```
    n % 2 == 0
}

fn main() {
    let n: i32 = 123456;
    let description = if is_even(n) {
        "even"
    } else {
        "odd"
    };

    prinln!("n={} is {}", n, description);
}
```

Этот прием может распространяться и на другие блоки, включая `match`

```
fn is_even(n: i32) -> bool {
    n % 2 == 0
}

fn main() {
    lst n = 654321;
    let description = match is_even(n) {
        true => "even",
        false => "odd",
    };

    println!("n={} is {}", n, description);
}
```

5.3. Расширенные определения функций

Пример

```
fn add_with_lifetimes<'a, 'b>(i: &'a i32, j: &'b i32) -> i32 {
    *i + *j
}
```

- `fn add_with_lifetimes(...) -> i32` – функция, возвращающая значение типа `i32`,
- `<'a, 'b>` – объявление двух *переменных времени жизни*, `'a` и `'b`, в области видимости функции `add_with_lifetimes()`. Обычно о них говорят как о *времени жизни a* и *времени жизни b*,
- `i: &'a i32` – привязка *переменной времени жизни* `'a` к времени жизни `i`. Этот синтаксис читается так «параметр `i` является *указателем* на `i32` с временем жизни *a*»,
- `j: &'b i32` – привязка *переменной времени жизни* `'b` к времени жизни `j`. Этот синтаксис читается так «параметр `j` является *указателем* на `i32` с времнем жизни *b*».

Основа проводимых в Rust проверок безопасности – система времени жизни, позволяющая убедиться, что все попытки обращения к данным являются допустимыми. Все значения, привязанные к данному времени жизни, должны существовать вплоть до последнего доступа к любому значению, привязанному к этому же времени жизни.

Обычно система времени жизни работает без посторонней помощи. Хотя время жизни есть почти у каждого параметра, проверки в основном проходят скрытно, поскольку компилятор может определить время жизни самостоятельно. Но в сложных случаях компилятору нужна помощь.

При вызове функции аннотации времени жизни не требуются.

```
fn add_with_lifetimes<'a, 'b>(i: &'a i32, j: &'b i32) -> i32 {
    *i + *j  // (1)
}

fn main() {
    let a = 10;
    let b = 20;
    let res = add_with_lifetimes(&a, &b);  // (2)
    println!("{}", res);
}
```

(1) – сложение значений, на которые указывают `i` и `j`, а не сложение непосредственно самих указателей. (2) – `&a` и `&b` означают *указатели* соответственно на 10 и 20.

Использование двух параметров времени жизни (`a` и `b`) показывает, что времени жизни `i` и `j` не связаны друг с другом.

5.3.1. Обобщенные функции

Типовая сигнатура обобщенной функции

```
fn add<T>(i: T, j: T) -> T {
    i + j
}
```

Переменная типа `T` вводится в угловых скобках (`<T>`). Эта функция принимает два аргумента одного и того же типа и возврщает значение такого же типа.

Заглавные буквы вместо типа указывают на *обобщенный тип*. В соответствие с действующим соглашением в качестве заместителей используются произвольно выбираемые переменные `T`, `U` и `V`. А переменная `E` часто применяется для обозначения типа ошибки.

Обобщения позволяют использовать код многократно и могут существенно повысить удобство работы со строго типизированными языками.

Все Rust-операторы, включая сложение, определены в *типажах*. Чтобы выставить требование, что тип `T` должен поддерживать сложение, в определение функции наряду с переменной типа включается *типажное ограничение*

```
// std::ops::Add -- типаж
fn add<T: std::ops::Add<Output = T>>(i: T, j: T) -> T {
    i + j
}
```

Фрагмент `<T: std::ops::Add<Output = T>>` предписывает, что в `T` должна быть реализация операции `std::ops::Add`. Использование одной и той же переменной типа `T` с типажными ограничением гарантирует, что агрументы `i` и `j`, а также возвращемое значение будут одного и того же типа и их типы поддеживают сложение.

Типаж – это что-то вроде *абстактного базового класса*. Все Rust-операции определяются с помощью типажей. Например, оператор сложения (+) определен как типаж `std::ops::Add`.

Все Rust-операторы являются удобным синтаксическим приемом для вызова *методов типажей*. В ходе компиляции выполняется преобразование выражения `a␣+␣b` в `a.add(b)`

```
use std::ops::{Add};
use std::time::{Duration};
```

```
fn add<T: Add<Output = T>>(i: T, j: T) -> {
    j + j
}

fn main() {
    let floats = add(1.2, 3.2);
    let ints = add(10, 20);
    let duratoins = add(
        Duration::new(5, 0),
        Duration::new(10, 0)
    );

    println!("{}", floats);
    println!("{}", ints);
    println!("{:?}", durations);
}
```

5.4. Создание списков с использованием массивов, слайсов и векторов

5.4.1. Массивы

Массивы характеризуются фиксированной шириной и чрезвычайной скромностью в потреблении ресурсов. *Векторы* можно наращивать, но им свойственны издержки времени выполнения из-зи ведения дополнительного учета.

В *массиве* допускается замена элементов, но его *размер менять нельзя*.

Описание типа массива имеет следующий вид: `[T; n]`, где `T` – тип элемента, а `n` – неотрицательное целое число. Например, запись `[f32; 12]` обозначает массив из двенадцати 32-разрядных чисел с плавающей точкой.

Особое внимание в Rust уделяется вопросам безопасности. При этом ведется проверка границ индексации массива. Запрос элемента, выходящего за границы, приводит к сбою (к панике в терминологии Rust), а не к возврату неверных данных.

5.4.2. Слайсы

Слайсы представляют собой похожие на массив объекты с динамическим размером. Понятие «динамический размер» означает, что их размер на момент компиляции *неизвестен*. Но, как и массивы, они не могут расширяться или сокращаться.

Недостаток сведений к моменту компиляции объясняет различие в сигнатуре типа между массивом (`[T;n]`) и слайсом (`[T]`).

Важность слайсов объясняется тем, что реализовать типаже дли них проще, чем для массивов. Поскольку `[T; 1]`, `[T; 2]`, ..., `[T; n]` бывают разных типов, реализация типажей для массивов может стать слишком громоздской. А создание *слайса* из массива дается легко и обходится дешево, поскольку *слайс не нужно привязывать к какому-либо конкретному размеру*.

Слайсы способны действовать как *представление массивов* (и других слайсов). Термин «представление» здесь взят из описания технологии работы с базами данных и означает, что слайсы могут получать быстрый доступ только по чтению данных, что исключает необходимость копирования чего бы то ни было.

5.4.3. Векторы

Векторы (`Vec<T>`) – это наращиваемые списки, состоящие из обобщенных типов `T`. При выполнении программы на них тратится немного больше времени, чем на массивы, из-за дополнительного учета, необходимого для последующего изменения их размера. Но эти издержки на работу с векторами почти всегда компенсируются их дополнительной гибкостью.

`Vec<T>` эффективнее всего работает при возможности указания размера с помощью функции `Vec::with_capacity()`. Предоставление этого показателя сводит к минимуму необходимое количество выделенной памяти операционной системой.

5.5. Чтение данных из файлов

Пример

```
use std::fs::File;
use std::io::BufReader;
use std::io::prelude;

fn main() {
    let f = File::open("readme.md").unwrap();
    let reader = BufReader::new(f);

    for line_ in reader.lines() {
        let line = line_.unwrap();
        println!("{} ({} bytes long)", line, line.len());
    }
}
```

На Python было бы так

```
def main():
    with open("readme.md", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            print(f"{line} ({len(line)} bytes long)")
```

6. Составные типы данных

Чтобы помешать компилятору выдавать предупреждения, в них будут задействованы атрибуты `#![Allow(unused_variables)]`.

Тип, известный как *unit* `()`, формально считается *кортежем нулевой длины*. Он используется для выражения того, что *функция не возвращает никакого значения*.

Функции, которые не имеют возвращаемого типа, возвращают `()`, и выражения, заканчивающиеся точкой с запятой `;`, также возвращают `()`. Например, функция `report()` в следующем блоке кода подразумеваемо возвращает тип *unit*

```
use std::fmt::Debug;

fn report<T: Debug>(item: T) { // item может быть любого типа с реализацией std::fmt::Debug
    println!("{:?}", item);
}
```

А в этом примере возвращение типа unit задается в явном виде

```
fn clear(text: &mut String) -> () {
    *text = String::from(""); // замена строкового значения, на которое указывает text, пустой с
    трокой
}
```

В Python затереть значение переменной в глобальной области видимости можно было бы так

```
>>> text = "global"
>>> def clear():
      global text  # просто расширяем область видимости функции
      text = ""  # привязываем переменную text к пустой строке
>>> text  # 'global'
>>> clear()
>>> text  # ''
```

Последнее выражение в функции не должно заканчиваться точкой с запятой. *Восклицательный знак* `!`, известен как тип «Never». `Never` показывает, что *функция никогда ничего не возвращает*, особенно при гарантированном сбое.

Пример

```
fn dead_end() -> ! { // функция никогда ничего не возвращает
    panic!("you have reached a dead end");
}
```

Макрос `panic!` вызывает сбой программы. То есть функция *гарантировано никогда не вернет управление* вызвавшему ее коду.

Структура `struct` позволяет создавать составной тип, образованный из других типов. Например

```
struct File {
    name: String,
    data: Vec<u8>,
}
```

Чтобы позволить структуре `File` стать выводимой на экран строкой, нужно поместить строку `#[derive(Debug)]` перед определением структуры

```
#[derive(Debug)]  // чтобы можно было вывести на печать структуру
struct File {
  name: String,
  data: Vec<u8>,
}
```

При определении структуры можно явно указывать время жизни каждого поля. Явное указание времени жизни требуется, когда поле является ссылкой на другой объект.

Экземпляр структуры можно создать так

```
fn main() {
    let f1 = File { // экземпляр структуры
        name: String::from("f1.txt"),
        data: Vec::new(),
    };

    let f1_name = &f1.name;
    let f1_length = &f1.data.len();

    println!("{:?}", f1);
}
```

К началу имени добавляется амперсанд (`&f1.name`), свидетельствующий о желании получить *доступ к данным по ссылке*. На языке Rust это означает, что переменные `f1_name` и `f1_length` *заимствуют* данные, на которые они ссылаются.

6.1. Добавление метдов к структуре struct путем использования блока impl

В Rust классы, так сказать, распадаются на структуры `struct` и реализации `impl`

```
struct File {
    // data
}

impl File {
    // methods
}
```

Rust отличается от других языков, поддерживающих методы: в нем нет ключевого слова `class`. Типы, созданные с помощью блока `struct`, иногда кажутся классами, но поскольку они *не поддерживают наследование*, то хорошо, что их называли по-другому.

Для определения методов Rust-программистами используется блок `impl`.

Создание объектов с уместными значениями по умолчанию выполняется с помощью метода `new()`. Каждую структуру можно создать, воспользовавшись литеральным синтаксисом, но это приводит к ненужной многословности.

Использование `new()` – соглашение, принятое в сообществе Rust. В отличие от других языков, `new` не является ключевым словом и не имеет какого-либо особого статуса по сравнению с другими методами

Использование блока impl для добавления методов к структуре

```
#[derive(Debug)]
struct File {
    name: String,
    data: Vec<u8>,
}

impl File {
    fn new(name: &str) -> File {
        File {
            name: String::from(name),
            data: Vec::new(),
        }
    }
}

fn main() {
    let f3 = File::new("f3.txt");

    let f3_name = &f3.name;
    let f3_length = f3.data.len();

    println!("{:?}", f3);
    println!("{} is {} bytes long", f3_name, f3_length);
}
```

В Rust небезопасность означает «тот же уровень безопасности, который всегда обеспечивается языком C».

Небольшие дополнения к языку Rust:

- Изменяемые глобальные переменные обозначаются с помощью `static mut`,
- По соглашению в именах глобальных переменных ВСЕ БУКВЫ ЗАГЛАВНЫЕ,
- Ключевое слово `const` включается для тех значений, которые никогда не изменяются.

Опытным программистам известно, что использование глобальной переменной `errno` во время системных вызовов обычно регулируется операционной системой. Как правило, в Rust такой стиль программирования не приветствуется, поскольку при нем не только нарушается безопасность типов (ошибки кодируются в виде простых целых чисел), но и в «награду» неравдивым программистам, забывающим проверить значение переменной `errno`, может проявиться нестабильность программ.

Разница между `const` и `let` Данные, определяемые с `let`, могут изменяться. Rust позволяет типам обладать явно противоречивым свойством *внутренней изменчивости*.

Некоторые типы, например `std::sync::Arc` и `std::rc::Rc`, представляют собой неизменяемый фасад, но по прошествии времени изменяют свое внутренее состояние. По мере того, как на них делаются ссылки, они увеличивают значение счетчика ссылок и уменьшают его значение, когда срок действия этих ссылок истекает.

На уровне компилятора `let` больше относится к использованию псевдонимов, чем к неизменяемости. Использование псевдонимов в понятиях компилятора означает одновременное наличие нескольких ссылок на одно и то же место в памяти.

Ссылки на переменные, доступные только для чтения (их заимствования), объявленные с помощью `let`, могут указывать на одни и те же данные. Ссылки для чтения-записи (изменяемые заимствования) гарантированно никогда не станут псевдонимами данных.

6.2. Использование возвращаемого типа Result

Подход, принятый в Rust к обработке ошибок, заключается в использовании типа, который соответствует как стандартному случаю, так и случаю ошбики. Этот тип известен как `Result`. У него два состояния: `Ok` и `Err`.

Для вызова функций, возвращающих `Result<File, String>`, требуется дополнительный метод `unwrap()`, позволяющий извлечь значение. Вызов `unwrap()` снимает оболочку с `Ok(File)` для создания `File`. При обнаружении ошибки `Err(String)` прогрмма даст сбой.

`Result` – перечисление `enum`. Перечисление `enum` – это тип, способный представлять несколько известных вариантов, например

```
enum Suit {
    Clubs,
    Spades,
    Diamonds,
    Hearts,
}

enum Card {
    King(Suit),
    Queen(Suit),
    Jack(Suit),
    Ace(Suit),
    Pip(Suit, usize),
}
```

Как и структуры, *перечисления* поддерживают *методы* через блоки `impl`. Перечисления в Rust эффективнее набора констант.

Определение основных характеристик типажа Read для File

```
#![allow(unused_variables)]

// структура
#[derive(Debug)]
struct File;

// типаж для структуры File, задающий протокол
trait Read {
  fn read( // этот метод должен быть реализован в блоке impl
    self: &Self, // псеодотип
    save_to: &Vec<u8>,
  ) -> Result<usize, String>;
}

// имплементация
impl Read for File {
  fn read(
    self: &File,
    save_to: &Vec<u8>,
  ) -> Result<usize, String> {
    Ok(0)
  }
}

fn main() {
  let f = File{};
  let mut buffer = vec![];
  let n_bytes = f.read(&mut buffer).unwrap();

  println!("{} byte(s) read from {:?}", n_bytes, f);
}
```

`Display` требует, чтобы в типах был реализован метод `fmt`, возвращающий `fmt::Result`

```
// типаж (протокол/контракт/интерфейс) Display требует,
// чтобы в типе был реализован метод fmt
impl Display for FileState {
    fn fmt(
        &self,
        f: &mut fmt::Formatter,
    ) -> fmt::Result {
        match *self {
            FileState::Open => write!(f, "OPEN"),
            FileState::Closed => write!(f, "CLOSED"),
        }
    }
}

// типаж (протокол/контракт/интерфейс) Display требует,
// чтобы в типе был реализован метод fmt
impl Display for File {
    fn fmt(
        &self,
        f: &mut fmt::Formatter,
    ) -> fmt::Result {
        write!(f, "<{} ({})>", self.name, self.state)
```

```
    }
}
```

Rust'ие перечисления напоминают Python'ие именованные кортежи

```
# Rust
# enum FileState {
#     Open,
#     Closed
# }
from collection import namedtuple

attrs = ("open", "closed")
FileState = namedtuple("FileState", attrs)(*attrs)
FileState.open # 'open'
FileState.closed # 'closed'
```

Для сборки документации проекта без зависимостей

```
$ cargo doc --no-deps --open
```

Группа символов /// приводит к созданию документов, ссылающихся на элемент, который следует непосредственно за ней

```
/// Represents a "file",
/// which probaly lives on a file system.
#[derive(Debug)]
pub struct File {
    name: String,
    data: Vec<u8>,
}

impl File {
    /// New files are assumed to be empty, but a name is required.
    pub fn new(name: &str) -> File {
         File {
             name: String::from(name),
             data: Vec::new(),
         }
    }
}
```

Можно приемы форматирования текста на Markdown

```
...
impl File {
    /// Creates a new, empty `File`.
    ///
    /// #*Examples*
    /// ```
    /// let f = File::new("f1.txt");
    /// ```
    pub fn new(name: &str) -> File {
        File {
            name: String::from(name),
            data: Vec::new(),
        }
    }
}
```

Строки, начинающиеся с `///` попадут в документацию. То есть это что-то вроде Python'их doc-strings.

С группы символов `//!` начинается описание проекта.

7. Время жизни, владение и заимствование

Контроллер заимствований (borrow checker) – проверяет *законность* любого *доступа к данным*, что позволяет Rust избежать проблем с безопасностью.

Проверка заимствований основана на трех взаимосвязанных понятиях:

1. Время жизни,
2. Владение,
3. Заимствование.

Владение в Rust связано *с избавлением от значений*, в которых больше нет надобности. Например, функция возвращает управление, необходимо освободить память, содержащую ее локальные переменные.

Время жизни значения – это период, в течение которого доступ к этому значению – допустимое поведение. Локальные переменные функции живут до тех пор, пока функция не вернет управление, а глобальные переменные могут жить в течение всего времени жизни программы.

Позаимствовать значение означает *получить к нему доступ*. Суть этого термина призвана подчеркнуть возможность общего доступа к значениям из многих частей программы при наличии у них одного владельца.

Термин *перемещение* (move) в Rust означает нечто специфическое. Движение внутри кода Rust относится к *переходу владения*, а не к перемещению данных.

Владение – это понятие, используемое в сообществе Rust для обозначения процесса времени компиляции, который проверяет, что каждое использование значения допустимо и что каждое значение полностью уничтожено. Каждое значение внутри Rust – это владение.

Попытка перезаписи значения, которое все еще доступно в другом месте программы, приводит к тому, что компилятор отказывается компилировать программу.

```
fn main() {
  // Владение возникает здесь при создании объекта CubeSat
    let sat_a = CubeSat { id: 0 };
    // ...
    // Владение объектом переходит к check_status(), но не возвращается к main()
    let a_status = check_status(sat_a);
    // ...
    // sat_a больше не владелец объекта, что делает доступ недействительным
    let a_status = check_status(sat_a);
}
```

В ходе вызова `check_status(sat_a)` владение переходит к функции `check_status()`. Когда `check_status()` возвращает сообщение, она удаляет значение `sat_a`. Здесь время жизни `sat_a` заканчивается. И все же после первого вызова `check_status()` переменная `sat_a` остается в локальной области видимости функции `main()`. Попытка получения доступа к этой переменной вызовет возмущение контролера зависимостей.

В Rust у *элементарных типов* особое поведение. В них реализован типаж `Copy`. Формально *элементарные типы* обладают *семантикой копирования*, а все другие типы имеют *семантику перемещения*.

При использовании значений в качестве аргумента той функции, которая становится их владельцем, получить к этим значениям новый доступ из внешней области видимости уже невозможно.

Семантика копирования элементарных типов Rust

```
fn use_value(_val: i32) {}

fn main() {
    let a = 123;
    use_value(a);

    println!("{}", a);  // + получение доступа к `a` после вызова use_value() вполне нормально
}
```

Семантика перемещения для типов, не реализующих Copy

```
fn use_value(_val: Demo) {}

struct Demo {
    a: i32,
}

fn main() {
    let demo = Demo {a: 123};
    use_value(demo);

    println!("{}", demo.a);  // - доступ к demo.a невозможен даже после возвращения из use_value
    ()
}
```

В Rust *передача владения* от одной переменной к другой осуществляется двумя способами:

1. по привязке переменной

   ```
   fn main() {
       let sat_a = CubeSat { id: 0 }; // передача владения по привязке переменной
   }
   ```

2. через функциональный барьер либо в качестве аргумента, либо в качестве возвращаемого значения

   ```
   fn main() {
       let sat_a = CubeSat { id: 0 };
       // ...
       let new_sat_a = check_status(sat_a); // передача владения через функциональный барьер
       // ...
   }
   ```

7.1. Решение проблем, связанных с владением

Изюминка Rust – система владения. Ею обеспечивается безопасность памяти без использования сборщика мусора.

7.1.1. Если полное владение не требуется, используйте ссылки

Чаще всего в код вносится уменьшение необходимого уровня доступа. Вместо запроса владения в определениях функций можно воспользоваться «заимствованием».

Для доступа *только по чтению* следует использовать `&T`, а для доступа *по чтению-записи* – `&mut T`.

Использование владения

```
fn send(to: CubeSat, msg: Message) {
    to.mailbox.messages.push(msg); // владение значением переменной to переходит функции send
}
```

Владение значением переменной `to` переходит к функции `send()`. При возвращении из `send()` значение переменной `to` *удаляется*.

Использование *ссылки* на изменяемое значение

```
fn send(to: &mut CubeSat, msg: Message) {
    to.mailbox.messages.push(msg);
}
```

Добавление префикса `&mut` к типу `CubeSat` позволяет *внешней области видимости сохранять владение данными*, на которые *указывает* переменная `to`.

```
impl GrounStation {
    fn send(
        &self,
        to: &mut CubeSat,
        msg: Message,
    ) {
        to.mailbox.messages.push(msg);
    }
}

impl CubeSat {
    fn recv(&mut self) -> Option<Message> {
        self.mailbox.messages.pop()
    }
}
```

Здесь `&self` указывает, что `GroundStation.send()` требуется ссылка на `self` *с доступом только на чтение*. Получатель берет *изменяемое заимствование* (`&mut`) экземпляра `CubeSat`, а `msg` становится полноправным владельцем его экземпляра `Message`.

Владение экземпялром сообщения `Message` переходит от `msg` к локальной переменной функции `message.push()`.

7.1.2. Сократите количество долгоживующих значений

Если есть крупный долгоживующий объект, например глобальная переменная, то хранить его для каждого компонента программы, который в нем нуждается, весьма неудобно.

Вместо использования долгоживующих объектов стоит подумать о создании недолговечных отдельных объектов. Иногда проблемы, связанные с владением, можно решить за счет пересмотра конструкций всей программы.

```
impl GroundStation {
    fn send(
        &self,
        mailbox: &mut Mailbox,
        to: &CubeSat,
        msg: Message,
```

```
    ) {
        mailbox.post(to, msg);
    }
}

impl CubeSat {
    fn recv(
        &self,
        mailbox: &mut Mailbox,
    ) -> Option<Message> {
        mailbox.deliver(&self)
    }
}

impl Mailbox {
    fn post(
        &mut self,  // изменяемый доступ к самому себе
        msg: Message  // владение сообщением
    ) {
        self.message.push(msg);
    }

    fn deliver(
        &mut self,
        recipient: &CubeSat
    ) -> Option<Message> {
        for i in 0..self.message.len() {
            if self.messages[i].to == recipient.id {
                let msg = self.messages.remove(i);
                return Some(msg);
            }
        }
        None
    }
}
```

Реализации стратегии недолговечных переменных

```
#![allow(unused_variables)]

#[derive(Debug)]
struct CubeSat {
    id: u64
}

#[derive(Debug)]
struct Mailbox {
    messages: Vec<Message>,
}

#[derive(Debug)]
struct Message {
    to: u64,
    content: String,
}

struct GroundStation {}

impl Mailbox {
    fn post(
```

```
        &mut self,  // метод будет изменять экземпляр Mailbox
        msg: Message
    ) {
        self.messages.push(msg);
    }

    fn deliver(
        &mut self,  // метод будет изменят экземпляр Mailbox
        recipient: &CubeSat
    ) -> Option<Message> {
        for i in 0..self.messages.len() {
            if self.messages[i].to == recipient.to {
                let msg = self.messages.remove(i);
                return Some(msg);
            }
        }
        None
    }
}

impl GroundStation {
    fn connect(&self, sat_id: u64) -> CubeSat {
        CubeSat {
            id: sat_id,
        }
    }

    fn send(
        &self, // доступ на чтение GroundStation
        mailbox: &mut Mailbox,  // экземпляр Mailbox будет изменяться
        msg: Message
    ) {
        mailbox.post(msg);
    }
}

impl CubeSat {
    fn recv(
        &self,  // доступ только на чтение CubeSat
        mailbox: &mut Mailbox  // экземпляр Mailbox будет изменяться
    ) -> Option<Message> {
        mailbox.deliver(&self)
    }
}

fn fetch_sat_ids() -> Vec<u64> {
    vec![1, 2, 3]
}

fn main() {
    let mut mail = Mailbox { messages: vec![] };

    let base = GroundStation {};

    let sat_ids = fetch_sat_ids();

    for sat_id in sat_ids {
        let sat = base.connect(sat_id);
        let msg = Message { to: sat_id, content: String::from("hellow")};
        base.send(&mut mail, msg);
```

```
    }

    let sat_ids = fetch_sat_ids();

    for sat_id in sat_ids {
        let sat = base.connect(sat_id);

        let msg = sat.recv(&mut mail);
        println!("{:?}: {:?}", sat, msg);
    }
}
```

На Python код выглядел бы так

```
import typing as t
from dataclasses import dataclass, field

# В поисках типов код просматривается сверху вниз, поэтому
# приходится вводить служебные типы
_CubeSat = t.NewType("_CubeSat", type)
_Mailbox = t.NewType("_Mailbox", type)

# В Rust это `struct Message`
class Message(t.NamedTuple):
  to: int  # поле
  content: str  # поле

# В Rust это `struct Mailbox` и `impl Mailbox`
@dataclass(frozen=False)
class Mailbox:
  # В Rust это блок struct
  messages: t.List[_Message] = field(default_factory=list)  # NB!

  # В Rust это блок impl
  def post(self, msg: _Message) -> t.NoReturn:
    # Доступ к полю messages экземпляра через self
    self.messages.append(msg)  # эта инструкция изменяет экземпляр дата-класса

  def delivier(self, recipient: _CubeSat) -> t.Optional[Message]:
    for i in range(len(self.messages)):
      if self.messages[i].to == recipient.id:
        msg = self.messages.pop(i)  # эта инстукция изменяет экземпляр дата-класса
        return msg

    return None

# В Rust это `struct CubeSat` и `impl CubeSat`
@dataclass(frozen=False)
class CubeSat:
  id: int  # поле

  def recv(self, mailbox: _Mailbox) -> t.Optional[Message]:
    return mailbox.delivier(self)

def fetch_sat_ids() -> t.List[int]:
  return [1, 2, 3]

# В Rust это `struct GroundStation` и `impl GroundStation`
@dataclass(frozen=False)
class GroundStation:
```

```
  # В Rust это блок impl
  def connect(self, sat_id: int) -> CubeSat:
    return CubeSat(id=sat_id)

  def send(self, mailbox: _Mailbox, msg: Message):
    return mailbox.post(msg)


def main():
  mail = Mailbox()
  base = GroundStation()
  sat_ids = fetch_sat_ids()

  for sat_id in sat_ids:
    sat = base.connect(sat_id)
    msg = Message(to=sat_id, content="hello")
    base.send(mail, msg)

  sat_ids = fetch_sat_ids()
  for sat_id in sat_ids:
    sat = base.connect(sat_id)
    msg = sat.recv(mail)
    print(f"{sat}: {msg}")


if __name__ == "__main__":
  main()
```

Экземпляр сообщения `Message` не изменяется после создания, поэтому его можно представить простым именованным кортежем, а не дата-классом. `Mailbox` приходится представлять дата-классом, потому что поле `messages` изменяемое. Причем это поле нужно создавать обязательно как `default_factory=list`, чтобы безопасно инициализировать поле экземпляра пустым списком

```
messages: t.List[Message] = field(default_factory=list)
```

7.1.3. Продублируйте значение

Наличие одного владельца для каждого объекта может свидетельствовать о серьезной предварительной проработке замысла.

Одной из самых простых альтернатив реструктуризации может стать простое копирование значения. Зачастую копирование не приветствуется, но в крайнем случае оно может оказаться полезным. Хорошим примером могут послужить элементарные типы, такие как целые числа. Их дублирование обходится центральному процессору настолько дешево, что Rust-компилятор, чтобы не заниматься переходом владения, всегда именно так и делает.

Типы могут выбрать один из двух режимов дублирования:

- клонирование `std::clone::Clone`,
- копирование `std::marker::Copy`.

У каждого режима имеется свой типаж. Копирование выполняется подразумеваемым образом. Если владение переходит во внутреннюю область видимости, то значение просто дублируется. Клонирование выполняется явным образом.

Клонирование (`std::clone::Clone`):

- Может быть медленным и затратным,

- Никогда не бывает подразумеваемым. Всегда требует вызова метода `.clone()`,
- Могут быть отличия от оригинала. Что именно означают клонирования, для их типов определяется автором контейнера.

Копирование (`std::marker::Copy`):

- Всегда бывает быстрым и дешевым,
- Всегда бывает подразумеваемым,
- Всегда создает идентичную копию. Копии являются побитными дубликатами оригинального значения.

Но иногда стоит отдать предпочтнеие клонированию:

1. Предполагается, что типаж `Copy` практически не снижает производительность. С числами – да, но только не с типами произвольных размеров, такими как `String`,
2. Поскольку `Copy` создает абсолютно точные копии, он не способен корректно интерпретировать ссылки. Простое копирование ссылки на `T` приведет к попытке создания второго владельца `T`. Впоследствии будут проблемы с несколькими попытками удаления `T` по мере удаления каждой ссылки,
3. Некоторые типы перегружают типаж `Clone` с целью предоставления чего-то похожего, но отличного от создания дубликатов.

При работе с Rust типажи `std::clone::Clone` и `std::marker::Copy` фигурируют обычно просто как `Clone` и `Copy`. Они включены в область видимости каджого контейнера через стандартную прелюдию.

8. Углубленное изучение данных

Инструмент `unsafe` сообщает компилятору Rust следующее: «Не трогай, я сам обо всем позабочусь. Все под контролем». Это сигнал компилятору, что специфика кода выходит за рамки проверки корректности программы.

Использование ключевого слова `unsafe` не означает, что код по своей сути опасен. К примеру, его указание не позволяет обойти выполняемую в Rust проверку заимствований. Это означает, что компилятор не может автоматически гарантировать безопасность памяти программы. Использование `unsafe` означает, что ответственность за целостность программы полностью возлагается на программиста.

Использование блоков `unsafe` без крайеней нужды воспринимается Rust-сообществом весьма неодобрительно. Безопасность программы может быть поставлена под удар за счет появления в ней серьезных уязвимостей.

Основная цель использования таких блоков – позволить программе на языке Rust взаимодействовать с внешним кодом, например с библиотеками, написанными на других языках, и с интерфейсом операционной системы.

16-разрядное целое число может представлять числа от 0 до 65 535 включительно. А что произойдет, если нужно будет сосчитать до 65 536?

Технический термин для класса исследуемой проблемы – *целочисленное переполнение*. Одним из самых безвредных способов переполнения целого числа является бесконечное увеличение.

Запаниковавшая программа – мертвая программа. Паника означает, что программист попросил сделать что-то невозможное. Она не знает, что нужно сделать, чтобы продолжить выполнение, и отключается.

Программисты с опытом работы исключительно с динамическими языками программирования вряд ли когда-либо столкнуться с целочисленным переполнением.

Динамические языки обычно проверяют, умещаются ли результаты целочисленных выражений в используемый диапазон. Если нет, то переменная, получающая результат, *переводится в более широкий целочисленный тип.*

Некоторые процессоры упорядочивают многобайтовые последовательности слева направо, а другие – справа налево. Эта особенность известна как присущий центральному процессору порядок следования байтов. В ней кроется одна из причин, по которой копирование исполняемого файла с одного компьютера на другой может привести его в нерабочее состояние.

Блок `impl From<T> for U` предписывает языку Rust порядок преобразования типа `T` в тип `U`. При этом требуется, чтобы в типе `U` была реализована функция `from()`, принимающая в качестве своего единственного аргумента значение типа `T`. Например

```
impl From<f64> for Q7 {
    fn from(n: f64) -> Self {
        if n >= 1.0 {
            Q7(127)
        } else if n <= -1.0 {
            Q7(-128)
        } else {
            Q7((n * 128.0) as i8)
        }
    }
}
```

Для модульного тестирования используется инструментальное средство `cargo test`. Реализация формата `Q7`

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Q7(i8);

impl From<f64> for Q7 { // f64 -> Q7
    fn from(n: f64) -> Self {
        if n >= 1.0 {
            Q7(127)
        } else if n <= -1.0 {
            Q7(-128)
        } else {
            Q7((n * 128.0) as i8)
        }
    }
}

impl From<Q7> for f64 {  // Q7 -> f64
    fn from(n: Q7) -> f64 {
        (n.0 as f64) * 2_f64.powf(-7.0)
    }
}

impl From<f32> for Q7 {  // f32 -> Q7
    fn from(n: f32) -> Self {
        Q7::from(n as f64)
    }
}

impl From<Q7> for f32 {  // Q7 -> f32
    fn from(n: Q7) -> f32 {
```

```
        f64::from(n) as f32
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn out_of_bounds() {
        assert_eq!(Q7::from(10.0), Q7::from(1.0));
        assert_eq!(Q7::from(-10.0), Q7::from(-1.0));
    }

    #[test]
    fn f32_to_q7() {
        let n1: f32 = 0.7;
        let q1 = Q7::from(n1);

        let n2 = -0.4;
        let q2 = Q7::from(n1);

        let n3 = 123.0;
        let q3 = Q7::from(n3);

        assert_eq!(q1, Q7(89));
        assert_eq!(q2, Q7(-51));
        assert!eq!(q3, Q7(127));
    }

    #[test]
    fn q7_to_f32() {
        let q1 = Q7::from(0.7);
        let n1 = f32::from(q1);
        assert_eq!(n1, 0.6953125);

        let q2 = Q7::from(n1);
        let n2 = f32::from(q2);
        assert_eq!(n1, n2);
    }
}
```

8.1. Краткий обзор модульной системы в Rust

Модульная симстема основана на следующих положениях:

- Модули объединяются в контейнеры.
- Модули могут быть определены структурой каталогов проекта. Если каталог `src/` содержит файл `mod.rs`, то его подкаталоги становятся модулями.
- Модули также могут определены в файле с помощью ключевого слова `mod`.
- Модули могут иметь произвольные вложения.
- Все элементы модуля, включая его подмодули, по умолчанию *закрытые*. Досуп к закрытым элементам можно получить как в самом модуле, так и в любых его потомках.
- К тому, что нужно сделать доступным, следует добавить в качестве префикса ключевое слово `pub`. У этого ключевого слова имеется ряд особенностей:
  - `pub(crate)` предосталяет доступ к элементу другим модулями внутри контейнера (крейта).

- `pub(super)` предоставляет доступ к элементу со стороны родительского модуля.
- `pub(in path)` предоставляет доступ к элементу в пределах указанного пути.
- `pub(self)` явным образом сохраняет открытый доступ к элементу в его модуле.

- Элементы из других модулей переносятся в локальную область видимости с помощью ключевого слова `use`.

8.2. Память

8.2.1. Указатели

Указатели – это просто числа, ссылающиеся на что-либо иное. Внутри компьютера *указатели* кодируются в виде *целого числа* (эквивалентного `usize`), являющегося *адресом памяти* объекта ссылки (данных, на которые ссылается указатель).

В Rust указатели чаще всего встречаются в виде `&T` и `&mut T`, где `T` – это тип.

Адреса памяти – абстракции, предоставляемые языками Ассемблера. Указатель представляет собой адрес памяти, указывающий на значение какого-либо типа. Указатели, по сути, – абстракции, предоставляемые языками более более высокого уровня. Ссылки – абстракции, предоставляемые языком Rust.

У Rust-ссылок имеются существенные преимущества перед указателями:

- Ссылки всегда указыают на реально существующие данные.
- Ссылки корректно выравнены по кратным `usize`. По техническим причинам центральные процессоры крайне негативно реагируют на требование извлечь данные без выравнивания памяти. В типы Rust включаются байты заполнения, чтобы создание ссылок на них не замедляло работу программ.
- Ссылки гарантируют производительную работу с типами, имеющими динамически изменяемый размер.

8.2.2. Обычные указатели, используемые в Rust

Обычный указатель – адрес памяти. Стандартные гарантии Rust на него не распространяются, что делает его небезопасным. Например, в отличие от ссылок (`&T`), обычные указатели могут иметь значение `null`.

Обычные неизменяемые указатели станем обозначать как `*const T`, а *изменяемые* – как `*mut T`. Их тип `T` в качестве обычного указателя на `String` выглядит как `*const String`. Обычный указатель на `i32` выглядит как `*mut i32`.

Важно:

- Разница между `a *mut T` и `a *const T` минимальна. Они могут свободно приводится друг к другу и, как правило, обладают взаимозаменяемостью, действуя в исходном коде в качестве документации.
- Rust-ссылки (`&mut T` и `&T`) *при компиляции* превращаются в *обычные указатели*. То есть для достижения высокой производительности, присущей обычным указателям, можно вполне обойтись и без риска использования небезопасных `unsafe`-блоков.

Пример приведения *ссылки на переменную* (`&a`) к *неизменяемому обычному указателю* (`*const i64`)

```
fn main() {
    let a: i64 = 42;
```

```
    let a_ptr = &a as *const i64;

    println!("a={} ({:p})", a, a_ptr);
}
```

Иногда термины «указатель» и «адрес памяти» используются как синонимы. Это целые числа, представляющие собой место в виртуальной памяти. Но с позиции компилятора имеется одно важное отличие. Типы *Rust-указателей* `*const T` и `*mut T` *всегда нацелены на начальный байт* `T`, и им также известна ширина типа `T` в байтах. А *адрес памяти* может относится к любому месту в памяти.

Тип `i64` имеет ширину 8 байт (64 бита при 8 битах на байт). Следовательно, если `i64` храниться по адресу `0x7fffd`, то для воссоздания целочисленного значения из оперативной памяти должет быть извлечен каждый из байтов диапазона `0x7ffd...0x8004`. Процесс выборки данных *из оперативной памяти* называется *разыменованием указателя*.

Закулисно *ссылки* (`&T` и `&mut T`) реализуются в виде простых указателей. Им сопутствуют дополнительные гарантии, и предпочтение следует неизменно отдавать только им.

Обычные указатели небезопасны!!! Им присущи некоторые свойства, определющие крайнюю нежелательность их повседневного использования в Rust-коде:

- Обычные указатели не владельцы своих значений. При обращении к ним *компилятор не проверяет доступность данных*, на которые они указывают.
- Допускается использование нескольких обычных указателей на одни и те же данные. Каждый обычный указатель может иметь доступ к записи или к чтению и записи данных. Это означает, что Rust не может гарантировать действительность совместно используемых данных.

Обычные указатели небезопасны. Альтернативой может послужить использование *интеллектуальных указателей*. Как правило, типы интеллектуальных указателей Rust служат оболочкой для обычных указателей и наделяют их дополнительной семантикой.

8.2.3. Предоставление программам памяти для размещения их данных

Стек работает быстро, а *куча* – медленно.

Стек С записями в стеке обращаются по принципу «последней пришла – первой ушла» (LIFO). Записи называются *кадрами стека*. Они создаются по мере выполнения вызовов функций.

В отличие от обеденных тарелок, каждый кадр стека имеет разный размер. В нем имеется пространство для аргументов его функции, указатель на исходное место вызова и значения локальных переменных (за исключением тех данных, что размещены в куче).

Основная роль стека – предоставить место для локальных переменных. Все переменные функции находятся в памяти рядом друг с другом. Это ускоряет доступ.

У `&str` и `String` разные представления в памяти: `&str` память выделяется в стеке, а `String` – в куче.

В тех случаях, когда требуется *доступ только по чтению*, следует использовать функции с сигнатурой типа `fn x<T: AsRef<str>>(a: T)`, а не `fn x(a: String)`. Читается так: «Будучи функцией, `x` получает аргумент пароля типа `T`, где в `T` реализуется `AsRef<str>`». Средства реализации `AsRef<str>` ведут себя как ссылки на `str`, даже если это и не соответствует действительности.

```
fn is_strong<T: AsRef<str>>(
    password: T  // либо String, либо &str
) -> bool {
    password.as_ref().len() > 5
}
```

Когда к аргументу требуется доступ по чтению и записи, в большинстве случаев можно воспользоваться родственником `AsRef<T>` типажом `AsMut<T>`.

Куча *Куча* – область программной памяти для тех *типов*, *размер* которых в ходе компиляции *еще не известен*. Некоторые типы по мере надобности меняются в размере в обе стороны. Очевидные примеры – `String` и `Vec<T>`. Есть и другие типы, неспособные сообщить Rust-компилятору, сколько памяти под них выделять, несмотря на то что их размер в ходе выполнения программы не меняется. Их называют типами с динамически определяемым размером. У слайсов на момент компиляции отсутствует длина. Слайс по сути – указатель на какую-то часть массива. Но фактически слайсы представляют некоторое количество элементов этого массива.

С позиции пользователя главной отиличительной чертой *кучи* является то, что обращение к находящимся в ней переменным должно осуществляться *через указатель*, чего не требуется переменным, доступным в стеке.

Простой пример

```
let a: i32 = 40;  // находится в стеке
let b: Box<i32> = Box::new(60);  // находится в куче
```

Упакованное значение, присвоенное `b`, доступно *только через указатель*. Чтобы получить доступ к этому значению, нам нужно его *разыменовать*. Унарным оператором разыменования служит символ `*`, помещаемый перед именем переменной:

```
...
let result = a + *b;  // разыменование указателя
println!("{} + {} = {}", a, b, result);
```

Использование синтаксиса `Box::new(T)` приводит к размещению `T` в *куче*. Что-то, что было упаковано, размещено в *куче* с *указателем* на него, помещенным в *стек*.

Стек и куча – это всего лишь концептуальные *абстракции*. Это *не физические разделы памяти* вашего компьютера.

В *стеке* скоростной доступ к данным обусловлен тем, что размещенные в нем *локальные переменные функций располагаются в оперативной памяти рядом* друг с другом. Иногда это называют *сплошной раскладкой*. Сплошная раскладка хорошо подходит для кеширования.

В куче значения переменных вряд ли будут располагаться рядом друг с другом. Более того, доступ к данным *в куче невозможен без разыменования указателя*.

Список литературы

1. *Кольцов Д.М.* Си на примерах. Практика, практика и только практика. – СПб.: Наука и Техника, 2019. – 288 с.

Листинги