Дата последней сборки документа:
5 марта 2023 г. в 13:24

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. Troubleshooting

2.1. Unable to find libclang: could't find any valid shared libraies ...

Больше полезной информации можно получить по ссылке https://esp-rs.github.io/book/installation/troubleshooting.html

Ошибка

```
thread 'main' panicked at 'Unable to find libclang: "couldn't find any valid shared libraries
    matching: ['libclang.so', 'libclang-*.so', 'libclang.so.*', 'libclang-*.so.*'], set the `
    LIBCLANG_PATH` environment variable to a path where one of these files can be found (invalid
    : [])"', /home/esp/.cargo/registry/src/github.com-1ecc6299db9ec823/bindgen-0.60.1/src/lib.rs
    :2172:31
```

Нам нужно LIBCLANG для bindgen, чтобы сгенерировать привязки Rust к заголовкам ESP-IDF C. Нужно убедиться, что переменная окружения `LIBCLANG_PATH` установлена и указывает на пользовательский LLVM

LLVM (Low Level Virtual Machine) – проект программной инфраструктуры для создания компиляторов и сопутсвующих им утилит.

Clang (произносится «кленг») – транслятор для C-подобных языков, созданный специально для работы на базе LLVM. Комбинация Clang и LLVM представляет собой полноценный компилятор и предоставляет набор инструментов, позволяющих полностью заменить GCC.

Установить LLVM на Linux Centos 7 https://nanomode.ru/how/kak-ustanovit-llvm-na-centos7/ можно так. Более старая версия LLVM доступна в официальном репозитории дополнительных компонентов на Centos 7. Но при желании можно загрузить и установить последнюю версию LLVM по адресу http://llvm.org.

Самый простой способ установить библиотеки C и C++ для LLVM Clang – это установить `gcc` и `g++` на Centos 7

```
$ sudo yum makecache
$ sudo yum install gcc gcc-c++
```

LLVM Clang версии 3.4.2 доступен в операционной системе Centos 7 в репозитории `extras`

```
sudo yum info clang
```

Вывести список всех включенных репозиториев Centos 7

```
sudo yum repolist
```

Если репозиторий `extras` не включен в Centos 7, то нужно сделать следущее

```
sudo yum install yum-utils
sudo yum-config-manager --enable extras
```

Обновить кэш репозитория пакетов `yum`

```
sudo yum makecache
```

Установить LLVM Clang

```
sudo yum install clang
```

Проверка

```
clang --version
# clang version 3.4.2 (tags/RELEASE_34/dot2-final)
# Target: x86_64-redhat-linux-gnu
# Thread model: posix
```

`bindgen` требует Clang 5.0 или старше. Если менеджер пакетов не предлагает Clang 5.0, то придется собирать из исходников https://clang.llvm.org/get_started.html

3. 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.

4. 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.

5. 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.

5.1. 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.

6. Основы языка

6.1. 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.

6.2. 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);
}
```

6.3. 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.

6.3.1. 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);
}
```

6.4. Создание списков с использованием массивов, слайсов и векторов

6.4.1. 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.

6.4.2. 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.

6.4.3. 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.

6.5. Чтение данных из файлов

Пример

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

```rust
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

```python
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)")
```

7. 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*

```rust
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

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

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

```python
>>> 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.

7.1. 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.

7.2. 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, // пседотип
    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.

8. 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
    // ...
}
```

8.1. Решение проблем, связанных с владением

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

8.1.1. 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()`.

8.1.2. 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
```

```python
# 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)
```

8.1.3. 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.

9. 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);
    }
}
```

9.1. 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`.

9.2. Память

9.2.1. 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.

9.2.2. 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.

9.2.3. 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*.

10. Файлы и хранилища

```
use std::fs::File;
use std::io::prelude::*;
use std::env;

const BYTES_PER_LINE: usize = 16;

fn main() {
  // ввод не проверяется!!!
    let arg1 = env::args().nth(1);

    let fname = arg1.expect("usage: fview FILENAME");

    let mut f = File::open(&fname).expect("Unable to open file");
    let mut pos = 0;
    let mut buffer = [0; BYTES_PER_LINE];

    while let Ok(_) = f.read_exact(&mut buffer) {  // f --data--> buffer
```

```
        print!("[0x{:08x}] ", pos);
        for byte in &buffer {
            match *byte {
                0x00 => print!(". "),
                0xff => print!("## "),
                _ => print!("{:02x} ", byte),
            }
        }

        println!("");
        pos += BYTES_PER_LINE;
    }
}
```

`while let Ok(_) {...}` – с помощью этой структуры управление ходом выполнения программы цикл продолжается до тех пор, пока `f.read_exact()` не вернет `Err`, что случится, когда закончатся байты для чтения.

`f.read_exact()` – метод из типажа `Read`, передающий данные из источника (в данном случае `f`) в буфер, предоставленный в качестве аргумента. Его работа завершиться при заполнении буфера.

Каждый метод итератора `nth()` возвращает `Option`. Когда n превышает длину итератора, возвращается `None`. Для обработки значений `Option` используются вызовы метода `expect()`.

Метод `expect()` считается более удобной версией метода `unwrap()`. Метод `expect()` получает в качестве аргумента сообщение об ошибке, а метод `unwrap()` просто внезапно впадает в панику.

10.1. Файловые операции, проводимые в Rust

Основной тип для работы с файловой системой – `std::fs::File`. Для создания файла доступны два метода: `open()` и `create()`. Если известно, что файл уже существует, то используется `open()`.

Если нужен более жесткий контроль, то используетвся функция `std::fs::OpenOptions`.

10.2. Безопасное взаимодействие с файловой системой

В стандартной библиотеке Rust существуют типобезопасные варианты `str` и `String`: `std::path::Path` и `std::path::PathBuf`

```
let hello = PathBuf::from("/tmp/hello.txt");
hello.extension();
```

Если разбираться с тонкостями реализации `std::fs::Path` и `std::fs::PathBuf`, то выясниться, что они, соответственно, являются надстройками над `std::ffi::OsStr` и `std::ffi::OsString`. То есть, `Path` и `PathBuf` не гарантируют совместимости с UTF-8.

10.3. Реализация хранилища «ключ-значение» с архитектурой, структурированной по записям и доступом только для добавления

Шаблон библиотеки можно создать так

```
cargo new --lib actionkv
```

Cargo.toml

```
[package]
name = "actionkv"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
betyorder = "1.2"
crc = "1.7"

[lib]  # в контейнере может быть только одна библиотка
name = "libactionkv" # имя создаваемой библиотеки
path = "src/lib.rs"

[[bin]] # [[...]] -- раздел можно повторять
name = "akv_mem"
path = "src/akv_mem.rs"
```

Раздел `[[bin]]`, которых может быть много, определяет *исполняемый файл*, созданный из этого контейнера. Синтаксис двойной скобки необходим, поскольку он четко описывает `bin` как раздел, имеющий один или несколько элементов.

Открытый API-интерфейс actionkv состоит из четырех операций: получения, удаления, вставки и обновления.

10.3.1. Настройка продукта условной компиляции

В Rust предоставляются широкие возможности изменнения *продукта компиляции* в зависимости от заданной компилятору *целевой архитектуры*. Как правило, речь идет о *целевой операционной системе*, но можно воспользоваться и возможностями, предоставляемыми целевым процессором. Изменение продукта компиляции в зависимости от заданных условий самого процесса компиляции называют *условной компиляцией*

```
#[cfg(target_os = "windows")]
const USAGE: &str = "
Usage:
    akv_mem.exe FILE get KEY
    akv_mem.exe FILE delete KEY
    akv_mem.exe FILE insert KEY VALUE
    akv_mem.exe FILE update KEY VALUE
";

#[cfg(not(target_os = "windows"))]
const USAGE: &str = "
Usage:
    ...
";
...
```

Для добавления в проект условной компиляции нужно аннотировать исходный код атрибутами `cfg`. Атрибут `cfg` работает вместе с целевым параметром, предоставляемым `rustc` в ходе компиляции [2, стр. 296].

Для предоставления в коде двух определений `const USAGE` используется условная компиляция. Когда проект создается под Windows, строка использования содержит расширение файла

`.exe`. В получаемые на выходе двоичные файлы включаются только те данные, которые имеют отношение к их целевому назначению.

Для отрицания следует использовать выражение `#cfg(not(...))`. Для сопоставления с элементами списка доступны также выражения `#[cfg(all(...))]` и `#[cfg(any(...))]`. Кроме всего этого, атрибуты `cfg` можно настроить при вызове `cargo` или `rustc` с помощью аргумента командной строки `--cfg ATTRIBUTE` [2, стр. 297].

В Rust при проведении операций с файлами может возвращаться ошибки типа `std::io::ErrorKind::Unex`
EOF – это нулевой байт (`0u8`).

При чтении данных из файла операционная система сообщает приложению о количестве байтов, успешно считаных из хранилища. Если успешного считывания байтов с диска не произошло и при этом не возникла никакая ошибочная ситуация, то операционная система, а стало быть, и приложение, предполагают, что достигнут конец файла – EOF.

Хеш-функция – отображение значений переменной длины на значения фиксированной длины. На практике значение, возвращаемое хеш-функцией. является целым числом.

Базовая хеш-функция для `&str`, которая просто интерпретирует первый символ строки как целое число без знака. То есть первый символ строки используется этой функцией в качестве хеш-значения

```
fn basic_hash(
    key: &str  // любое строковое значение
) -> u32 {
    let first = key.chars()
                   .next()  // -> значение типа Option: Some(char) | None
                   .unwrap_or('\0');
    unsafe {
        std::mem::transmute::<char, u32>(first)
    }
}
```

Итератор `.chars()` преобразует строку в серию символьных значений, каждое длиной 4 байта. `.next()` возвращает значение типа `Option` с распаковкой либо в `Some(char)`, либо, для пустых строк, в `None`.

Функция `unwrap_or()` ведет себя как `unwrap()`, но при встрече с `None` не паникует, а предоставляет значение.

Если несколько входных параметров начинаются с одного и того же символа, на выходе будет одинаковый результат. Такое происходит всякий раз, когда бесконечное пространство входных параметров отображается на конечное пространство, но в данном случае это имеет крайне негативные последствия.

Хеш-таблицы, включая имеющуюся в Rust карту `HashMap`, справляются с этой особенностью, которую называют хеш-коллизией. Для ключей с одинаковым хеш-значением в этих таблицах предоставляется место для резервных копий. Обычно это резервное хранилище относится к типу `Vec<T>`, и называется хранилищем коллизий. При возникновении коллизий выполняется обращение к хранилищу коллизий и происходит его сквозное сканирование. По мере увеличения хранилища это линейное сканирование занимает все больше и больше времени.

Буквальный синтаксис в стандартной библиотеке Rust для `HashMap` не предоставляется.

```
use std::collections::HashMap;

fn main() {
    let mut capitals = HashMap::new();  // создаем экземпляр 'словаря' {}
```

```
    capiatls.insert("Cook Islands", "Avarua");
    capitals.insert("Fiji", "Suva");
    capitals.insert("Kiribati", "South Tarawa");
    capitals.insert("Niue", "Alofi");
    capitals.insert("Tonga", "Nuku'alofa");

    let tonga_capital = capitals["Tonga"];
    // Или так
    // let tonga_capital = capitals.get("Tonga").expect("Oops");
    println!("Capital of Tonga is: {}", tongan_capital);
}
```

В Python можно было бы сделать так

```
capitals: t.Dict[str, str] = {}

capitals.update(**{"Cook Islands", "Avarua"})
capitlas["Fiji"] = "Suva"
...
tonga_capital = captials["Tonga"]
tonga_capital = capitals.get("Tonga", None)
```

При поддержке расширенной экосистемы Rust имеется возможность вставки JSON-строк в код Rust

```
#[macro_use]  // использование макросов контейнера serde_json
extern crate serde_json;

fn main() {
    let capitals = json!({
        "Cook Islnads": "Avarua",
        "Fiji": "Suva",
    });

    println!("Capital of Tonga is: {}", capitals["Tonga"]);
}
```

`capitals["Tonga"]` возвращает ссылку на значение, предназначенную *только для чтения*.

NB: используйте `HashMap`, пока не будет веских оснований для использования `BTreeMap`. Структура `BTreeMap` работает быстрее, если ключи обладают естественной упорядоченностью и ваше приложение пользуется таким их расположением.

- `std::collections::HashMap` (с хеш-функцией `SipHash`): вариант, обладающий криптографической безопасностью и устойчивостью к атакам типа «отказ в обслуживании», но работающий медленее других хеш-функций.
- `std::collections::BTreeMap`: вариант, более подходящий *для ключей с естественной упорядоченностью*, где согласованность кеша может обеспечить ускорение работы.

11. Работа в сети

Типажные объекты являются посредниками конкретных типов. Синтаксис `Box<dyn std::error::Error>` означает `Box` (указатель) на *любой тип* с реализацией `std::error::Error`.

Типажные объекты добавляют в Rust форму *полиморфизма*, то есть допускают посредством *динамической диспетчеризации* совместное использование интерфейса сразу несколькими типами. А *обобщения* допускают *полиморфизм* посредством *статической диспетчеризации*. Выбор

между обобщением и типажными объектами обычно основывается на компромиссах между дисковым пространством и временем:

- Обобщения используют больше дискового пространства и характеризуются более высоким темпом выполнения программы.
- Типажные объекты занимают меньше дискового пространства, но из-за косвенности указателя влекут за собой незначительные издержки времени выполнения.

Типажные объекты существуют в трех формах:

- `&dyn Trait`: заимствуется,
- `&mut dyn Trait`: заимствуется,
- `Box<dyn Trait>`: находится в чьем-то владении.

Использование типажного объекта `&dyn Enchanter`

```
use rand;
use rand::seq::SliceRandom;
use rand::Rng;

#[derive(Debug)]
struct Dwarf {};

#[derive(Debug)]
struct Elf {};

#[derive(Debug)]
struct Human {};

#[derive(Debug)]
enum Thing {
    Sword,
    Trinket,
}

trait Enchanter: std::fmt::Debug {
  // self -- это структура, к которой будет подмешан этот типаж
    fn competency(&self) -> f64; // абстрактный метод

    fn enchant(&self, thing: &mut Thing) {
        let probability_of_success = self.competency();
        let spell_is_successful = rand::thread_rng().gen_bool(probability_of_success);

        print!("{:?} mutters incoherently. ", self); // self можно вывести на печать благодаря
    std::fmt::Debug
        if spell_is_successful {
            println!("The {:?} glows brightly.", thing);
        } else {
            println!("The {:?} fizzes, \
                then turns into a worthless trinket.", thing);
        }
    }
}

// реализация методов для структуры Dwarf по протоколу Enchanter
impl Enchanter for Dwarf {
    fn competency(&self) -> f64 {
        0.5
    }
}
```

```
// реализация методов для структуры Elf по протоколу Enchanter
impl Enchanter for Elf {
    fn competency(&self) -> f64 {
         0.95
    }
}

// реализация методов для структуры Human по протоколу Enchanter
impl Enchanter for Human {
    fn competency(&self) -> f64 {
        0.8
    }
}

fn main() {
    let mut it = Thing::Sword;

    let d = Dwarf {};
    let e = Elf {};
    let h = Human {};

    let party: Vec<&dyn Enchanter> = vec![&d, &e, &h];  // типажный объект
    // метод choose() берется из типажа rand::seq::SliceRandom
    let spellcaster = party.choose(&mut rand::thread_rng()).unwrap();

    spellcaster.enchant(&mut it);
}
```

`&dyn Rng` – это ссылка на что-то, имеющее реализацию типажа `Rng`, `&ThreadRng` – ссылка на значение `ThreadRng`.

Приведем несколько типичных случаев использования типажных объектов:

- Создание коллекции гетерогенных объектов,
- Возвращение значения. Типажные объекты позволяют функциям возвращать несколько конкретных типов/
- Поддержка динамической диспетчеризации, при этом вызываемая функция определяется в ходе выполнения программы, а не в ходе компиляции ее кода.

Типажные объекты ближе к *миксинам*. Типажные объекты не существуют самы по себе, они – агенты какого-то другого типа.

Номера портов – это чисто виртуальные понятия. Это просто значения типа `u16`. Номера портов позволяют по одному и тому же IP-адресу размещать сразу несколько служб.

Утилита командной строки, предназначенная для разрешения IP-адресов из имен хостов

```
use std::net::{SocketAddr, UdpSocket};
use std::time::Duration;

use clap::{App, Arg};
use rand;
use trust_dns::op::{Message, MessageType, OpCode, Query};
use trust_dns::rr::domain::Name;
use trust_dns::rr::record_type::RecordType;
use trust_dns::serialize::binary::*;

fn main() {
    let app = App::new("resolve")
        .about("A simple to use DNS resolver")
```

```
        .arg(Arg::with_name("dns-server").short("s").default_value("1.1.1.1"))
        .arg(Arg::with_name("domain-name").required(true))
        .get_matches();

    let domain_name_raw = app
        .value_of("domain-name").unwrap();
    let domain_name = Name::from_ascii(&domain_name_raw).unwrap();

    let dns_server_raw = app.value_of("dns-server").unwrap();
    let dsn_server: SockerAddr = format!("{}:53", dns_server_raw)
        .parse()
        .expect("invalid address");

    let mut request_as_bytes: Vec<u8> = Vec::with_capacity(512);
    let mut response_as_bytes: Vec<u8> = vec![0; 512];

    let mut msg = Message::new();
    msg
        .set_id(rand::random::<u16>())
        .set_message_type(MessageType::Query)
        .add_query(Query::query(domain_name, RecordType::A))
        .set_op_code(OpCode::Query)
        .set_recursion_desired(true);

    let mut encoder =
        BinEncoder::new(&mut request_as_bytes);
        msg.emit(&mut encoder).unwrap();

    let localhost = UdpSocket::bind("0.0.0.0:0")
        .expect("cannot bind to local socket");
    let timeout = Duration::from_secs(3);
    localhost.set_read_timeout(Some(timeout)).unwrap();
    localhost.set_nonblocking(false).unwrap();

    let _amt = localhost
        .send_to(&request_as_bytes, dns_server)
        .expect("socket misconfigured");

    let (_amt, _remote) = localhost
        .recv_from(&mut response_as_bytes)
        .expect("timeout reached");

    let dns_message = Message::from_vec(&response_as_bytes)
        .expect("unable to parse response");

    for answer in dns_message.answers() {
         if answer.record_type() == RecordType::A {
              let resource = answer.rdata();
              let ip = resource
                  .to_ip_addr()
                  .expect("invalid IP address received");
              println!("{}", ip.to_string());
         }
    }
}
```

"0.0.0.0:0" – прослушивание всех адресов на произвольном порту. Конкретный порт выбирается операционной системой.

`Vec::with_capacity(512)` создает `Vec<T>` с длиной 0 и емкостью 512, `vec![0; 512]` создает `Vec<T>` с длиной 512 и емкостью 512.

11.1. Способы обработки ошибок, наиболее удобные для помещения в библиотеки

Возвращение `Result<T, E>` успешно работает только при наличии одного типа ошибок `E`. Но, как только возникает потребность возвращения нескольких типов ошибок, ситуация резко усложняется.

При работе с отдельными файлами код лучше компилировать с помощью команды `rustc <file_name>`, отказавшись от использования `cargo build`. Например, если файл называется `io-error.rs`, то в командной строке оболочки следует набрать `rustc io-error.rs && ./io-error[.exe]`.

Оператор `?` это удобная синтаксическая замена макросу `try!`, выполняющему две функции:

- При обнаружении `Ok(value)` это выражение вычисляется в значение `value`,
- при обнаружении `Err(err)`, `try!` или `?` выполняет возвращение сразу же после попытки преобразования `err` в тип `error`, определение которого находится в вызывающей функции.

В Rust-подобном псевдокоде макрос `try!` можно определить следующим образом

```
macro try {
    match expression {
        Result::Ok(val) => val,
        Result::Err(err) => {
           let converted = convert::From::from(err);
           return Result::Err(converted);
        }
    }
}
```

Использование типажного объекта в возвращаемом значении

```
use std::fs::File;
use std::error::Error;
use std::net::Ipv6Addr;

// типажный объект "обобщает" типы Error и AddrParseError
fn main() -> Result<(), Box<dyn Error>> {
    let _f = File::open("invisible.txt")?; // тип ошибки std::io::Error

    let _localhost = "::1"
        .parse::<Ipv6Addr>()? // тип ошибки std::net::AddrParseError

    Ok(())
}
```

Типажный объект, `Box<dyn Error>`, является представителем *любого типа*, реализующего тип `Error`.

Необходимость заключения *типажных объектов* в `Box` обуславливается тем, что их *размер* (в байтах в стеке) *на момент компиляции не известен*. У `Box` есть известный размер в стеке. И смысл его применения заключается в том, чтобы *указывать* на то, для чего этот *размер неизвестен*, в том числе и на типажные объекты.

Использование типажных объектов известно также как затирание типов. При этом Rust теряет сведения о том, что ошибка берет свое начало в вышестоящих контейнерах.

Использование `Box<dyn Error>` в качестве варианта ошибки, закладываемого в `Result`, означает, что *вышестоящие типы ошибок* в некотором смысле *теряются*. Исходные ошибки теперь перобразуются в один и тот же тип.

Функция `map_err()` отображает ошибку на функцию. В качестве функции могут использоваться варианты перечисления. Оператор `?` ставится в самом конце. Иначе функция может возвратить управление еще до того, как у кода будет возможность преобразовать ошибку.

Заключение вышестоящих ошибок в оболочку нашего собственного типа

```
use std::io;
use std::fmt;
use std::net;
use std::fs::File;
use std::net::Ipv6Addr;

#[derive(Debug)]
enum UpstreamError {
    IO(io::Error),
    Parsing(net::AddrParseError),
}

// реализация метода fmt для перечисления UpstreamError
// в соответствии с типажом Display
impl fmt::Display for UpstreamError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{:?}", self)
    }
}

// реализация всех нужных для работы методов
// в соответствии с типажом Error
impl error::Error for UpstreamError { }

fn main() -> Result<(), UpstreamError> {
    let _f = File::open("invisible.txt")
        .map_err(UpstreamError::IO)?;

    let _localhost = "::1"
        .parse::<Ipv6Addr>()
        .map_err(UpstreamError::Parsing)?;

    Ok(())
}
```

Можно также избавиться от вызова `map_err()`. Но для этого нужна реализация типажа `From`. Типаж `std::convert::From` располагает единственным необходимым метдом `from()`. Чтобы придать нашим двум вышестоящим типам ошибок возможность подвергаться преобразованиям, нужны два блока `impl`

```
impl From<io::Error> for UpstreamError {
    fn from(error: io::Error) -> Self {
         UpstreamError::IO(error) // io::Error -> UpstreamError
    }
}

impl From<net::AddrParseError> for UpstreamError {
    fn from(error: net::AddrParseError) -> Self {
        UpstreamError::Parsing(error) // net::AddrParseError -> UpstreamError
```

```
    }
}
```

Теперь функция `main()` возвращается в присущую ей простую форму

```
fn main() -> Result<(), UpstreamError> {
    let _f = File::open("invisible.txt")?;
    let _localhost = "::1".parse::<Ipv6Addr>()?;

    Ok(())
}
```

12. Время и хронометраж

Структуры без полей `struct Clock;` называют *типами с нулевым размером* (zero-sized type) или ZST.

12.1. Предоставление полноценного интерфейса командной строки

Конфигурация проекта

```
[package]
name = "clock"
version = "0.1.1"
authors = ["Tim ..."]
edition = "2018"

[dependencies]
chrono = "0.4"
clap = "2"
```

Приложение

```
use chrono::DateTime;
use chrono::Local;
use clap::{App, Arg};

struct Clock;

impl Clock {
    fn get() -> DateTime<Local> {
        Local::now()
    }

    fn set() -> ! {
        unimplemented!()
    }
}

fn main() {
   let app = App::new("clock")
        .version("0.1")
        .about("Gets and ...")
        .arg(
            Arg::with_name("action")
                .takes_value(true)
                .possible_values(&["get", "set"])
                .default_value("get"),
```

```
        )
        .arg(
            Arg::with_name("std")
                .short("s")
                .long("use-standard")
                .takes_value(true)
                .possible_values(&[
                    "rfc2822",
                    "rfc3339",
                    "timestamp",
                ])
                .default_value("rfc3339"),
        )
        .arg(Arg::with_name("datetime").help(
            "When <action> is 'set' ..."
        ));

    let args = app.get_matches();

    let action = args.value_of("action").unwrap();
    let std = args.value_of("std").unwrap();

    if action == "set" {
        unimplemented!()
    }

     let now = Clock::get();
     match std {
          "timestamp" => println!("{}", now.timestamp()),
          "rfc28222" => println!("{}", now.to_rfc2822()),
          "rfc3339" => println!("{}", now.to_rfc3339()),
          _ => unreachable!(),
     }
}
```

12.2. tСоглашения о наименовании типов, действующие в libc

В libc при обозначении типов предпочтение отдается именам в символах нижнего регистра, а стиль PascalCase не используется. То есть, там, где в Rust использовалось бы название `TimeVal`, в libc ему соответствовало бы имя `timeval`. При работе с псевдонимами типов соглашение немного меняется. К имнам псевдонимов типов в libc добавляется знак нижнего подчеркивания, за которым следует буква `t` (то есть `_t`)

```
libc::{timeval, time_t, suseconds_t};
```

В Rust-синтаксисе они определяются следующим образом

```
#![allow(non_camel_case_types)] // разрешает давать имена типам не в camel-case

type time_t = i64;  // псевдоним типа
type suseconds_t = i64; // псевдоним типа

pub struct timeval {
    pub tv_sec: time_t, // секунды с начала эпохи
    pub tv_usec: suseconds_t, // дробная составляющая текущей секунды
}
```

Чтобы поместить в контейнер привязку к libc, необходимую для платформ, отличных от Windows

Cargo.toml

```
...
[target.'cfg(not(windows))'.dependencies]
libc = "0.2"
```

12.3. Листинг полного кода clock v0.1.2

Конфигурация проекта

Cargo.toml

```
[package]
name = "clock"
version = "0.1.2"
authors = ["Tim ..."]
edition = "2018"

[dependencies]
chrono = "0.4"
clap = "2"

[target.'cfg(windows)'.dependencies]
winapi = "0.2"
kernel32-sys = "0.2"

[target.'cfg(not(windows))'.dependencies]
libc = "0.2"
```

Кроссплатформенный код установки системного времени

```
#[cfg(windows)]
use kernel32;
#[cfg(not(windows))]
use libc;
#[cfg(windows)]
use winapi;

use chrono::{DateTime, Local, TimeZone};
use clap::{App, Arg};
use std::mem::zeroed;

struct Clock;

impl Clock {
    fn get() -> DateTime<Local> {
        Local::now()
    }

    #[cfg(windows)]
    fn set<Tz: TimeZone>(t: DateTime<Tz>) -> () {
        use chrono::Weekday;
        use kernel32::SetSystemTime;
        use winapi::{SYSTEMTIME, WORD};

        let t = t.with_timezone(&Local);
```

```
        let mut systime: SYSTEMTIME = unsafe { zeroed() };

        let dow = match t.weekday() {
            Weekday::Mon => 1,
            Weekday::Tue => 2,
            Weekday::Wed => 3,
            Weekday::Thu => 4,
            Weekday::Fri => 5,
            Weekday::Sat => 6,
            Weekday::Sun => 0,
        };

        let mut ns = t.nanosecond();
        let is_leap_second = ns > 1_000_000_000;

        if is_leap_second {
            ns -= 1_000_000_000;
        }

        systime.wYear = t.year() as WORD;
        systime.wMonth = t.month() as WORD;
        systime.wDayOfWeek = dow as WORD;
        systime.wDay = t.day() as WORD;
        systime.wHour = t.hour() as WORD;
        systime.wMinute = t.minute() as WORD;
        systime.wSecond = t.second() as WORD;
        systime.wMilliseconds = (ns / 1_000_000) as WORD;

        let systime_ptr = &systime as *const SYSTEMTIME;

        unsafe {
             SetSystemTime(systime_ptr);
        }
    }

    #[cfg(not(windows))]
    fn set<Tz: TimeZone>(t: DateTime<Tz>) -> () {
        use libc::{timeval, time_t, susecond_t};
        use libc::{settimeofday, timezone};

        let t = t.with_timezone(&Local);
        let mut u: timeval = unsafe { zeroed() };

        u.tv_sec = t.timestamp() as time_t;
        u.tv_usec = t.timestamp_subsec_micros() as suseconds_t;

        unsafe {
            let mock_tz: *const timezone = std::ptr::mull();
            settimeofday(&u as *const timeval, mock_tz);
        }
    }
}

fn main() {
    let app = App::new("clock")
        .version("0.1.2")
        .about("Gets ...")
        .after_help(
            "Note: UNIX timestamps are parsed \
            seconds ..."
```

```
        )
        .arg(
            Arg::with_name("action")
                .takes_value(true)
                .possible_values(&["get", "set"])
                .default_value("get"),
        )
        .arg(
            Arg::with_name("std")
                .short("s")
                .long("use-standard")
                .takes_value(true)
                .possible_values(&[
                    "rfc2822",
                    "rfc3339",
                    "timestamp",
                ])
                .default_value("rfc3339"),
        )
        .arg(
            Arg::with_name("datetime").help(
            "When ..."
            ));

        let args = app.get_matches();

        let action = args.value_of("action").unwrap();
        let std = args.value_of("std").unwrap();

        if action == "set" {
            let t_ = args.value_of("datetime").unwrap();

            let parser = match std {
                "rfc2822" => DateTime::parse_from_rfc2822,
                "rfc3339" => DateTime::parse_from_rfc3339,
                _ => unimplemented!(),
            };

        let err_msg = format!(
            "Unable to parse {} according to {}",
            t_, std
        );
        let t = parser(t_).expect(&err_msg);

        Clock::set(t)
    }

    let now = Clock::get();

    match std {
        "timestamp" => println!("{}", now.timestamp()),
        "rfc2822" => println!("{}", now.to_rfc2822()),
        "rfc3339" => println!("{}", now.to_rfc3339()),
        _ => unreachable!(),
    }

}
```

13. Процессы, потоки и контейнеры

Система конкурентных вычислений может выстраиваться и на одном ядре центрального процессора.

13.1. Безымянные функции

Определение функции выглядит так

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

Примерный эквивалент в форме лямбда-функции (безымянной функции) имеет следующий вид

```
let add = |a: i32, b: i32| {a + b};
```

В Python было бы так

```
add = lambda a, b: a + b
```

В Rust лямбда-функции могут читать значения переменных *из своей области видимости.*

В отличие от обычных функций, лямбда-функции не могут определяться в глобальной области видимости. Безымянные функции можно определять в точках входа `main()`.

В Rust для порождения потока безымянная функция передается в функцию `std::thread::spawn()`.

Когда порожденному потоку требуется доступ к переменным, определенным в родительской области видимости, называемой *захватом*, Rust зачастую настаивает на *перемещении захватов в замыкание*. Чтобы обозначить намерение передачи владения, в безымянной функции применяется ключевое слово `move`

```
// move расширяет область видимости лямбда-функции для потока
thread::spawn(move || {
    // ...
});
```

Ключевое слово `move` позволяет безымянной функции получать доступ к переменным *из более широкой области видимости* (для потока!). Поскольку Rust всегда гарантирует действительность доступа к данным, ему нужно, чтобы *владение перешло к самому замыканию.*

При *порождении потоков* говорят, что они *ответвились* от своего родительского потока. Срастить потоки (с помощью функции `join`) означает снова их слить.

На практике слияние (`join`) означает ожидание завершения другого потока.

Каждому потоку нужна своя память, следовательно, память системы в конечном счете может быть исчерпана. Но даже не доходя до предела, создание потока начинает вызывать замедление в других областях. По мере увеличения числа потоков растет объем работы по их диспетчеризации со стороны операционной системы. Когда приходится регулировать работу множества потоков, решение о том, какой поток запускать следующим, занимает больше времени.

Порождение потоков не обходится без издержек. Оно потребляет память и время центрального процессора. Переключение между потоками также обесценивает данные, хранящиеся в кеш-памяти.

Как только *подпоток* сливается с *основным потоком*, он перестает существовать. А Rust не позволит нам сохранить ссылку на то, чего не существует. Следовательно, чтобы вызвать `join()`

в отношении описателя потока внутри вектора описателей `handlers`, описатель потока должен быть удален из `handlers`

```
while let Some(handler) = handlers.pop() {
    handler.join();
}
```

Можно было бы сделать и так

```
for handler in handlers { // без амперсанда!!!
    handler.join();
}
```

Непосредственный последовательный перебор описателей сохраняет право владения. Тем самым уходят в сторону любые опасения по поводу совместного доступа и можно выполнить задуманное.

`std::thread::yield_now()` – это сигнал операционной системе о *снятии текущего потока с диспетчеризации*. Тем самым позволяется продолжить выполнение других потоков, пока текущий будет ждать истечения 20 мс. Недостатком уступки управления является отсутсвтие информации о возможности возобвновления выполнения потока ровно через 20 мс.

Альтернатива уступке управления – функция `std::sync::atomic::spin_loop_hint()`. Функция `spin_loop_hint()` не обращаться к операционной системе, отправляя сигнал непосредственно центральному процессору, который может воспользоваться подсказкой для прекращения функционирования, экономя тем самым потребление энергии.

13.2. Отличие безымянных функций от функций

Безымянные функции (`|| {}`) отличаются от функций (`fn`). Поэтому они не взаимозаменяемы.

Безымянные функции и функции имеют разные внутренние представления. Замыкания по сути являются безымянными структурами, реализующими типаж `std::ops::FnOnce`, а также, возможно, типажи `std::ops::Fn` и `std::ops::FnMut`. В исходном коде эти структуры не видны, но в них содержаться любые переменные из окружения замыкания, использующиеся внутри него.

А вот функции реализованы *как указатели на функции*, указывающие на код, а не на данные. Код в этом смысле, по сути, является памятью компьютера, помеченной как исполняемый фрагмент. Ситуация усугубляется еще и тем, что замыкания, не содержащие никаких переменных из своего окружения, также являются указателями на функции.

Макрос `format!` в Rust работает как f-строки в Python (или метод `format`)

```
let file_name = "test";
let default: String = format!("{}.svg", file_name);
```

В Python можно было бы написать

```
file_name = "test"
default = f"{file_name}.svg"
# или
default = "{}.svg".format(file_name)
```

Первый шаг к добавлению параллелизма – замена цикла `for`.

Было

Процедурный стиль

```
fn parse(input: &str) -> Vec<Operation> {
    let mut steps = Vec::<Operation>::new();

    for byte in input.bytes() {
        let step = match byte {
            b'0' => Home,
            b'1'..=b'9' => {
                let distance = (byte - 0x30) as isize;
                Forward(distance * (HEIGHT / 10))
            }
            b'a' | b'b' | b'c' => TurnLeft,
            b'd' | b'e' | b'f' => TurnRight,
            _ => Noop(byte),
        };
        steps.push(step);
    }
    steps
}
```

Стало

Функциональный стиль

```
fn parse(input: &str) -> Vec<Operation> {
    input.bytes().map(| byte | { // input.bytes() -- итератор
      match byte {
          b'0' => Home,
          b'1'..=b'9' => {
              let distance = (byte - 0x30) as isize;
              Forward(distance * (HEIGHT / 10))
          },
          b'a' | b'b' | b'c' => TurnLeft,
          b'd' | b'e' | b'f' => TurnRight,
          _ => Noop(byte),
      }}).collect()
}
```

`map()` применяет безымянную функцию к каждому элементу итератора. Метод `map()` возвращает *итератор*. Это позволяет связать множество преобразований вместе. Несмотря на возможность появления `map()` сразу в нескольких местах вашего кода, Rust зачастую оптимизирует эти вызовы функций в скомпилированном двоичном файле. Итераторы дают возможность делегировать компилятору больше работы.

Использование параллельного итератора Контейнер от Rust-сообщества под названием rayon разработан специально для добавления в код *параллелизма данных*, заключающегося в применении одной и той же функции (или безымянной функции) к разным данным (таким как `Vec<T>`).

Метод `par_bytes()` применяется к строковым слайсам, а метод `par_iter()` – к байтовым слайсам. Эти методы допускают совместную обработку данных сразу несколькими потоками.

```
use rayon::prelude::*;

fn parse(input: &str) -> Vec<Operation> {
    input
        .as_bytes() // слайс входной строки -> байтовый слайс
        .par_iter() // байтовый слайс -> параллельный итератор
```

```
        .map(|byte| match byte {
          b'0' => Home,
          b'1'..=b'9' => {
              let distance = (byte - 0x30) as isize;
              Forward(distance * (HEIGHT / 10))
            },
            b'a' | b'b' | b'c' => TurnLeft,
            b'd' | b'e' | b'f' => TurnRight,
            _ => Noop(*byte),
    }).collect()
}
```

Функция `par_iter()` гарантирует абсолютное исключение состояния гонки.

Порой у нас просто нет хорошего итератора. Тогда приходится вспоминать еще об одном шаблоне – *очереди задач*. В этих условиях группа рабочих потоков может выбирать себе задачу сразу же по завершении выполнения своих текущих задач.

Можно создать `Vec<Task>` и `Vec<Result>` и организовать совместное использование потоками ссылок на задачи и результаты. А чтобы потоки не переписывали данные друг друга, нужна стратегия защиты данных.

Самым распространенным *инструментом защиты данных*, совместно используемых потоками, является `Arc<Mutex<T>>`. Мьютекс (Mutex) – это взаимоисключающая блокировка. В данном контексте понятие взаимного исключения означает, что особых прав нет ни у кого. Блокировка, удерживаемая любым потоком, предотвращает доступ к данным со стороны всех других потоков. Как ни странно, но от других потоков должен быть защищен и сам `Mutex`. Поэтому мы обращаемся за дополнительной поддержкой к `Arc`, обеспечивающей безопасный многопоточный доступ к `Mutex`.

`Mutex` может понадобиться только для одного поля, но в `Arc` можно заключить всю структуру.

Односторонняя связь Контейнер `crossbeam` включает как *ограниченные*, так и не *ограниченные очереди*. Ограниченные очереди имеют предопределенное максимальное использование памяти, но они заставляют производителей очередей ждать, пока не освободиться достаточное место. Это может выразиться в непригодности ограниченных очередей для не терпящих ожидания асинхронных сообщений.

Создание канала для приема сообщений формата i32

```
#[macro_use]
extern crate crossbeam; // предоставляет макрос select!

use std::thread;
use crossbeam::channel::unbounded;

fn main() {
    let (tx, rx) = unbounded(); // канал tx -> rx

    thread::spawn(move | | {
        tx.send(42)
            .unwrap();
    });

    select!{
        recv(rx) -> msg => println!("{:?}", msg),
    }
```

```
}
```

В макросах могут определяться свои собственные правила синтаксиса. Именно поэтому в макросе `select!` используется синтаксис `recv(rx) ->` , не относящийся к допустимому синтаксису Rust.

Двунаправленная связь Двунаправленную (дуплексную) связь моделировать с одним каналом неудобно. Проще обратиться к созданию двух наборов отправителей и получателей, по одному для каждого направления.

Отправка сообщений в созданный поток и из него

```
#[macro_use]
extern crate crossbeam;

use crossbeam::channel::unbounded;
use std::thread;

// Чтобы к элементам нашего перечисления можно было обращаться без префикса ConnectivityCheck::
use crate::ConnectivityCheck::*;

#[derive(Debug)]
enum ConnectivityCheck {
    Ping,
    Pong,
    Pang,
}

fn main() {
    let n_messages = 3;
    let (requests_tx, requests_rx) = unbounded();  // канал
    let (responses_tx, responses_rx) = unbounded();  // канал

    thread::spawn(move | | loop {
        match requests_rx.recv().unwrap() {
            Pong => eprintln!("unexpected pong response"),
            Ping => responses_tx.send(Pong).unwrap(),
            Pang => return,
        }
    });

    for _ in 0..n_messages {
        requests_tx.send(Ping).unwrap();
    }

    requests_tx.send(Pang).unwrap();

    for _ in 0..n_messages {
        select! {
            recv(responses_rx) -> msg = println!("{:?}", msg),
        }
    }
}
```

Запущенные программы выполняются в виде процессов. У процесса как минимум один поток. Важно различать две теоретические концепции:

- Конкурентность, являющуюся одновременным выполнением *нескольких задач* любого уровня абстракции.

- Параллелизм, являющийся одновременным выполнением *нескольких потоков* на *нескольких процессорах*.

Для переключения с одного потока на другой требуется очистка регистров центрального процессора, очистка кеш-памяти центрального процессора и сброс значений переменных в операционной системе. По мере увеличения изоляции растут и издержки переключения контекста.

Центальные процессоры (CPU) могут выполнять инструкции *только в последовательном* режиме.

Потоки сущствуют внутри процесса. Отличительной особенностью процесса является то, что его память не зависит от других процессов. Операционная система вместе с центральным процессором *защищает память* одного *процесса* от всех остальных.

В чем разница между `const` и `static`:

- Значения `static` появляется в памяти только в одном месте,
- Значения `const` могут дублироваться в тех местах, где они доступны.

Дублирование `const`-значений может быть оптимизацией, удобной для процессора. *Статические значения* находятся вне пространства стека!, в области, где хранятся строковые литералы, ближе к нижней части адресного пространства. Это означает, что доступ к статической переменной практически наверняка подразумевает *разыменование указателя*.

Константа в `const`-значениях относится к самому значению. При доступе из кода, если компилятор посчитает, что это приведет к более быстрому доступу, данные могут дублироваться во все необходимые места.

13.3. Общие сведения об указателях на функции и их синтаксисе

Если

```
fn handle_sigterm() {
    ...
}
```

то выражение `handle_sigterm as usize` означает, что *адрес памяти*, по которому хранится функция, преобразуется в *целое число*.

Используемое в Rust ключевое слово `fn` создает *указатель на функцию*. Иными словами функция – последовательность байтов, имеющая смысл для центрального процессора, а указатель на функцию – указатель на начало такой последовательности.

Внутренее представление указателей – это целое число типа `usize`.

```
fn noop() {}

fn main() {
  // адрес памяти, по которому хранится функция -> в целое число
    let fn_ptr = noop as usize;

    println!("noop as usize: 0x{:x}", fn_ptr);  // noop as usize 0x10ad46700
}
```

А какой тип у *указателя на функцию*, созданного из выражения `fn noop()`? В случае выражения `fn noop()` типом является `*const fn() -> ()`. Его можно прочитать так «константный указатель на функцию, которая не принимает аргументов и возвращает значение типа `unit`».

```
fn noop() {}
```

```
fn main() {
    let fn_ptr = noop as usize;
    let typed_fn_ptr = noop as *const fn() -> ();

    println!("noop as usize: 0x{:x},\ntyped_fn_ptr as *const T: {:p}", fn_ptr, typed_fn_ptr);
    // noop as usize: 0x10824b6d0,
    // typed_fn_ptr as *const fn() -> (): 0x10824b6d0
}
```

13.4. Настройка встроенных функций

Порядок сообщения Rust о функциях LLVM

```
extern "C" {
    #[link_name = "llvm.eh.sjlj.setjmp"] // где искать определения функций
    pub fn setjmp(_: *mut i8) -> i32;

    #[link_name = "llvm.eh.sjlj.longjmp"]  // где искать определения функций
    pub fn longjmp(_: *mut i8);
}
```

`extern "C"` означает «этот блок кода должен подчиняться соглашениям языка C, а не Rust». Атрибут `link_name` сообщает компоновщику, где найти две объявляемые нами функции.

`*mut i8` – указатель на байт со знаком.

Встроенные функции (intrinsics) не часть языка и доступны только через компилятор.

С позиции Rust-программиство LLVM можно рассматривать как подкомпонент Rust-компилятора rustc. LLVM – это внешний инструмент, связанный с rustc. Один из наборов, предоставляемых LLVM – встроенные функции. LLVM сам по себе является компилятором.

LLVM преобразует код, созданный rustc в виде кода на LLVM IR (промежуточном языке), в машиночитаемый язык ассемблера.

13.5. Приведение указателя к другому типу

```
const JMP_BUF_WIDTH: usize = mem::size_of::<usize>() * 8;
type jmp_buf = [i8; JMP_BUF_WIDHT];  // псевдоним типа для массива
```

Способ инициирования `jmp_buf`

```
static mut RETURN_HERE: jmp_buf = [0; JMP_BUF_WIDTH];
```

В Rust-коде `RETURN_HERE` имеет вид *ссылки* (`&RETURN_HERE`). А LLVM ожидает, что эти байты будут переданы как `*mut i8`. Для преобразования применяются четыре шага

```
unsafe {
    &RETURN_HERE as *const i8 as *mut i8
}
```

В чем суть этих шагов:

- Все начинается с `&RETURN_HERE` – доступной только для чтения ссылки на глобальную статическую переменную типа `[i8; 8]` для 64-разрядных машинах или `[i8; 4]` на 32-разрядных машинах.
- В Rust приведение одних типов указателей к другим их типам считается безопасным, но для определения этого указателя требуется `unsafe`-блок.

- После этого выполняется преобразование `*const i8` в `*mut i8`. То есть *место в памяти* объявляется *изменяемым* (доступным по чтению и записи).
- Преобразование заключается в `unsafe`-блок, поскольку в нем ведется работа с доступом к *глобальной переменной*.

Список литературы

1. *Кольцов Д.М.* Си на примерах. Практика, практика и только практика. – СПб.: Наука и Техника, 2019. – 288 с.
2. *Макнамара Т.* Rust в действии. – СПб.: БХВ-Петербург, 2023. – 528 с.

Листинги