Растономикон

Команда Rust. Перевод сообщества rustycrate.ru

27.01.2016

Оглавление

1

Введение

Перевод “Rustonomicon”

> Растономикон - это игра слов Rust и Некрономикон[1] - вымышленная книга, придуманная Говардом Лавкрафтом и часто упоминаемая в литературных произведениях, основанных на мифах Ктулху, ее название можно перевести как “Книга законов Мёртвых”

Оригинал

https://github.com/rust-lang/rust/tree/master/src/doc/nomicon

Статус

Переведено всё, что есть в оригинале. В данный момент книга не публикуется в виде веб-страниц, но её можно читать прямо на GitHub.

Читать

Содержание[2]

[1] https://ru.wikipedia.org/wiki/Некрономикон
[2] https://github.com/ruRust/rustonomicon/blob/master/src/SUMMARY.md

Темное искусство продвинутого и небезопасного программирования на Rust

Внимание: Это сырой документ, поэтому может содержать серьезные ошибки.

> Вместо писания программ, что было моей мечтой, мной овладела тоска и невыразимое одиночество; и наконец я увидел страшную правду, в сторону которой остальные боялись даже дышать — непроизносимая тайна тайн открылась мне: этот язык камня и режущих звуков не способен сохранить в себе черт старого языка как Лондон старого Лондона, а Париж старого Парижа, и что на самом деле он довольно небезопасный, а его распростертые внутренности дурно набальзамированы и заселены странными существами, в действительности не имеющими ничего общего с тем, как все выглядит на этапе компиляции.

Эта книга зарывается в ужасающие подробности, которые нужно понять для правильного написания небезопасных программ на Rust. В связи с характером проблем, все может привести к высвобождению невыразимого ужаса, который разорвет вашу душу на миллиард бесконечно малых фрагментов отчаяния.

Хотите иметь долгую и счастливую карьеру программиста на Rust? Закройте и никогда не вспоминайте, что видели эту книгу. Она вам не нужна. Но если вы все же захотите написать небезопасный код - или просто покопаться во внутренностях языка - здесь содержится бесценная информация.

В отличие от книги[3] мы предполагаем, что у вас есть обширные базовые знания. По крайней мере вы должны быть знакомы с системным программированием и Rust. Если это не так, то сначала почитайте книгу[4]. Хотя мы не предполагаем, что вы знаете все вообще, и будем время от времени освежать основы там, где это нужно. Вы можете пропустить чтение той книги, если хотите; просто знайте, что мы не будем объяснять все, начиная с основ.

Для ясности, эта книга погрузит вас в глубокие детали. Мы нырнем в безопасность во время исключительных ситуаций, псевдонимы указателей, модели памяти и даже теорию типов. Мы будем много рассказывать о различных типах безопасности и гарантий.

[3] https://github.com/ruRust/rust_book_ru
[4] https://github.com/ruRust/rust_book_ru

2

Встреча безопасного и небезопасного

Программисты безопасных “высокоуровневых” языков встречаются с фундаментальной дилеммой. С одной стороны, было бы *очень* здорово сказать, что ты хочешь и не переживать, как это сделается. С другой стороны, это может привести к неприемлемо низкой производительности. Придется перейти к менее четким или менее идиоматичным практикам, чтобы добиться лучшей производительности. Или может вы брезгливо отдерните руки от высокоуровневого языка и решите раскошелиться на реализацию на менее приторно-сладком *небезопасном* языке.

Все еще хуже, если вы захотите иметь дело напрямую с операционной системой и вам *придется* разговаривать на небезопасном языке: *Си*. Си вездесущ и неизбежен. Это лингва-франка мира программирования. Все безопасные языки на самом деле используют интерфейсы Си для общения с внешним миром! Как бы вы ни хотели, как только ваша программа начинает взаимодействовать с Си, она перестает быть безопасной.

С учетом сказанного Rust *абсолютно* безопасный язык программирования.

Ну, вообще-то, в Rust *есть* безопасный язык программирования. Отступим немного назад.

Rust можно считать состоящим из двух языков программирования: *Безопасный Rust* и *Небезопасный Rust*. Безопасный Rust ваще, полностью безопасен. Неудивительно, что небезопасный Rust ваще, полностью *не* безопасен. На самом деле Небезопасный Rust позволяет вам делать действительно сумасшедшие небезопасные вещи.

Безопасный Rust это и *есть* язык программирования Rust. Если вы пишете все на Безопасном Rust, вам никогда не придется волноваться о безопасности типов или памяти. Вы никогда не испытаете проблем с нулевым или висячим указателями или всей этой чушью с неопределенным поведением.

Это офигенно круто.

Стандартная библиотека дает вам достаточно готовых утилит, позволяющих писать высокопроизводительные приложения или библиотеки на чистом идиоматичном Безопасном Rust.

Но возможно вам хочется говорить на другом языке. Может вы пишите низкоуровневые абстракции, не включенные в стандартную библиотеку. Может вы просто *пишете* стандартную

библиотеку (которая написана полностью на Rust). Может вам нужно сделать то, что не понимает система типов, и просто *программку из чертовских бит*. Может вам нужен Небезопасный Rust.

Небезопасный Rust это тот же Безопасный Rust, с теми же правилами и семантикой. Но он позволяет вам делать некоторые штуки, которые Точно Не Безопасны.

Единственные отличия Небезопасного Rust в том, что вы можете:

- Разыменовывать сырые указатели
- Вызывать `unsafe` функции (включая функции на Си, встроенные функции, сырое распределение)
- Реализовывать `unsafe` типажи
- Изменять статические переменные

Вот и все. Эти операции сделаны Небезопасными, потому что злоупотребление ими вызывает абсолютно Неопределенное Поведение. При вызове неопределенного поведения компилятор может самовольно делать плохие вещи с вашей программой. Вы совершенно точно *не* должны вызывать Неопределенное поведение!

В отличие от Си, вызвать Неопределенное поведение в Rust могут только несколько операций. Базовый язык заботится о предотвращении следующих действий:

- Разыменование нулевых или висячих указателей
- Чтение [неинициализированной памяти]
- Нарушение [правил совпадения указателей]
- Создание неверных примитивных значений:
  - висячих/нулевых ссылок
  - `bool`, который не 0 и не 1
  - неопределенных вариантов `enum`
  - `char` вне границ [0x0, 0xD7FF] и [0xE000, 0x10FFFF]
  - Не-utf8 `str`
- Размотка в другой язык (unwinding)
- Вызов гонок данных

Вот и все. Только эти операции вызывают неопределенное поведение в Rust. Конечно, небезопасные функции и типажи могут спокойно сами определять ограничения, которые должны соблюдаться в программе для избежания Неопределенного поведения. Однако смысл их все равно сведется к одной из проблем выше. Некоторые дополнительные ограничения могут появиться благодаря встроенным функциям компилятора, который делает предположения об оптимизации кода.

С другой стороны Rust довольно либерально относится к некоторым сомнительным операциям. Rust считает “безопасными”:

- Дедлоки
- Состояние гонки
- Утечка памяти
- Невызванные деструкторы

- Переполнение целых чисел
- Аварийное завершение программы
- Удаление рабочей базы данных

Однако любая программа, которая выполняет такие операции, наверняка, неправильна. Rust предоставляет много инструментов для избавления от этих штук, но считаетстя, что эти проблемы нельзя полностью предотвратить.

Как относятся Безопасный и Небезопасный Rust

Итак, каковы отношения между Безопасным и Небезопасным Rust? Как они взаимодействуют между собой?

Rust разделяет Безопасный и Небезопасный Rust с помощью ключевого слова `unsafe`, которое можно трактовать как *интерфейс внешних функций (foreign function interface)* (FFI) для взаимодействия Безопасного и Небезопасного Rust. Это магия, благодаря которой можно сказать, что Безопасный Rust - действительно безопасен: вся работа со страшными небезопасными частями языка, *как и в других безопасных языках*, отводится исключительно FFI.

Но из-за того, что один язык, получается, входит в другой, их можно спокойно смешивать, обозначая границы между ними ключевым словом `unsafe`. Не надо писать заголовочные файлы, инициализировать среду исполнения или делать какие-либо другие рутинные операции по обработке FFI.

На данный момент `unsafe` может появиться в Rust только в определенных местах, которые грубо можно разделить на две категории:

- Перед объявлением непроверенных контрактов. Я, как автор контракта, требую писать `unsafe`, чтобы убедиться, что вы, как пользователь, поняли следующее:

- Перед функциями `unsafe` показывает, что ее вызывать небезопасно. Вы должны посмотреть документацию, чтобы определить, в чем конкретно это выражается, и написать `unsafe`, чтобы подтвердить, что вы поняли опасность вызова такой функции.

- Перед объявлением типажей `unsafe` показывает, что *реализация* типажа является небезопасной операцией, потому что у типажа есть контракты, которым другой небезопасный код имеет право слепо доверять. (Больше об этом ниже.)

- Перед использованием непроверенных контрактов. Я в меру своих знаний заявляю, что придерживаюсь следующей логики использования непроверенных контрактов:

- В реализации типажей `unsafe` показывает, что соблюдается `unsafe` контракт типажа.

- В блоке `unsafe` показывает, что любая работа с небезопасными операциями должна обрабатываться внутри, и, следовательно, родительская функция безопасна.

В языке есть особый случай, флаг `#[unsafe_no_drop_flag]`, присутствующий по историческим причинам и находящийся на пути к выпиливанию. Смотрите для уточнения раздел флаги удаления.

Примеры небезопасных функций:

- `slice::get_unchecked` выполняет непроверенное индексирование, позволяющее свободно нарушить безопасность памяти.
- любой сырой указатель на тип фиксированного размера обладает внутренним методом `offset`, который вызывает Неопределенное Поведение, если находится “вне границ”, определенных LLVM.
- `mem::transmute` интерпретирует значение полученного типа как другого типа, самовольно обходя безопасность типов. (смотрите для уточнения [преобразования типов])
- Все функции FFI являются `unsafe`, потому что могут выполнять произвольные сценарии. Часто очевидным виновником этого является Си, но вообще-то любой язык может сделать что-то, от чего Rust не будет в восторге.

В Rust 1.0 есть ровно два небезопасных типажа:

- `Send` - это маркерный типаж (у него нет своего API), который обещает, что типы, реализующие его, можно можно безопасно посылать (перемещать) в другой поток.
- `Sync` - это маркерный типаж, который обещает, что потоки могут безопасно делить между собой типы, реализующие его, используя общую ссылку на них.

Необходимость в небезопасных типажах кроется в основных свойствах безопасного кода:

Каким бы убогим ни был Безопасный код, он не сможет вызвать неопределенное поведение.

Это означает, что Небезопасный Rust, **как передовой отряд неопределенного поведения**, должен *очень подозрительно* относиться к обобщенному безопасному коду. Для ясности, Небезопасный Rust доверяет конкретному безопасному коду абсолютно. Другое поведение выродилось бы для него в *бесконечные спирали параноидального отчаяния*. В частности, доверять корректности стандартной библиотеки абсолютно нормально. `std` - это, по сути, расширение языка, и вам, действительно, следует доверять ему. Если `std` нарушает свои гарантии, тогда это точно ошибка в языке.

Тем не менее, лучше минимизировать напрасные надежды на железобетонность безопасного кода. Ошибки случаются! Я ещё раз подчеркну: беспокоиться надо только за Небезопасный код. Безопасный код может слепо верить всему, что не нарушает безопасность памяти.

С другой стороны безопасные типажи могут объявлять произвольные контракты, но, небезопасный код не может считать, что они будут на самом деле соблюдаться из-за того, что их реализация считается безопасной,. *Кто угодно* может реализовать интерфейс так, как он хочет. В этом и состоит фундаментальная разница - доверять ли правильности конкретного участка кода или доверять правильности *любого кода, который будет когда либо написан*.

Например, в Rust есть типажи `PartialOrd` и `Ord`, нужные для того, чтобы можно было отличить типы, которые можно “только” сравнивать, от тех, значения которых находятся в отношении строгого порядка. В большинстве своем, каждое API, которое хочет работать с упорядоченными данными, хочет иметь `Ord`. Например, упорядоченный словарь BTreeMap *не имеет никакого смысла* создавать для частично упорядоченных данных. Если вы объявите, что тип реализует `Ord`, но не предоставите значения, которые действительно упорядочены, BTreeMap *попадет в просак*, ему станет очень плохо. Вставленные данные будет уже невозможно найти!

Но это еще нормально. BTreeMap безопасен, поэтому он гарантирует, что даже если вы дадите ему абсолютно бредовую реализацию `Ord`, он будет все равно делать что-то *безопасное*. Он не начнет читать неинициализированную или невыделенную память. На самом деле BTreeMap даже не потеряет ваши данные. После его удаления, все деструкторы будут вызваны успешно! Ура!

Только надо помнить, что BTreeMap реализован с использованием маленькой щепотки Небезопасного Rust (как и большинство коллекций). Поэтому не всегда можно утверждать, что плохая реализация `Ord` не приведёт к небезопасному поведению BTreeMap. BTreeMap не должен полагаться на `Ord`, *ставя под угрозу безопасность*. `Ord` предоставляется безопасным кодом, а безопасный код считает всё безопасным.

Но правда было бы здорово, если бы небезопасный код мог бы в *каких-нибудь местах* доверять контрактам какого-либо типажа? Эта проблема, которой занимаются небезопасные типажи: помечая небезопасность реализации *самого типажа*, другой небезопасный код может доверять реализациям контракта такого типажа. Хотя она может быть неправильна во всех произвольных случаях.

Например, имея гипотетический типаж UnsafeOrd, технически такая реализация будет правильна:

```
unsafe impl UnsafeOrd for MyType {
    fn cmp(&self, other: &Self) -> Ordering {
        Ordering::Equal
    }
}
```

Но, наверное, это совсем не та реализация, которую вам бы хотелось иметь.

Rust традиционно не делает типажи небезопасными по умолчанию, потому что это сделало бы небезопасность повсеместной, что абсолютно нежелательно. Send и Sync небезопасны, потому что потокобезопасность - это *фундаментальное свойство*, от которого небезопасный код даже не может попытаться защититься таким же образом, как он защитился бы от плохой реализации Ord. Единственный способ защититься от потоконебезопасности - *не использовать потоки вообще*. Сделать каждую загрузку и сохранение атомарными недостаточно, потому что могут существовать сложные варианты, задействующие отдельные области памяти. Например, указатель и размер у Vec должны быть синхронизированы.

Даже такая парадигма параллельности как обмен сообщениями, которая традиционно считается Абсолютно Безопасной, неявно опирается на потокобезопасность - действительно ли вы используете обмен сообщениями, если передаете указатель? Для Send и Sync, таким образом, требуется базовый уровень доверия, который Безопасный код не может предоставить, поэтому их реализацию необходимо сделать небезопасной. Чтобы избежать небезопасности, проникающей везде, возникнувшей вследствие этого, Send и Sync автоматически выводятся для всех типов, состоящих из значений типов, реализующих Send и Sync. 99% типов реализуют Send и Sync, и 99% из них никогда не сообщают об этом (оставшийся 1% - это по большей части примитивы синхронизации).

Работа с небезопасным кодом

В общем случае инструменты языка Rust достаточно ограничены и сводят все ситуации к двум вариантам - безопасному и небезопасному. К сожалению, реальность оказывается невообразимо сложнее этого. Например, у нас есть такая игрушечная функция:

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx < arr.len() {
        unsafe {
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

Ясно, что эта функция безопасна. Мы проверяем, что индекс находится внутри границ, и если это так, обращаемся по нему к массиву в небезопасном виде. Но даже в такой тривиальной функции, область действия небезопасного блока вызывает вопросы. Поменяем < на <=:

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx <= arr.len() {
        unsafe {
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

Программа сломалась, а мы *только поменяли безопасный код*. Это фундаментальная проблема безопасности: она не локальна. Устойчивость нашей небезопасной операции обязательно зависит от состояния, полученного из другой, “безопасной”, операции.

Безопасность является модульной, в том смысле, что использование небезопасного кода не приведет к отбрасыванию других произвольных видов проблем. Например, выполняя непроверенное индексирование в срезе не означает, что вам вдруг надо беспокоиться о том, что срез станет нулем или будет содержать неинициализированную память. Ничего фундаментально не меняется. Однако безопасность *не* модульна в том смысле, что у программы есть состояние, и ваша небезопасная операция может зависеть от другого произвольного состояния.

Всё ещё хитрее, когда мы имеем дело с настоящим контролем за состоянием. Представьте простую реализацию `Vec`:

```
use std::ptr;

// Помните, что это определение недостаточное. Смотрите секцию о реализации Vec.
pub struct Vec<T> {
    ptr: *mut T,
```

```
    len: usize,
    cap: usize,
}

// Обратите внимание, что эта реализация некорректно обрабатывает типы нулевого
// размера. Здесь мы живем в прекрасном выдуманном мире положительных типов
// фиксированного размера.
impl<T> Vec<T> {
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap {
            // неважно для этого примера
            self.reallocate();
        }
        unsafe {
            ptr::write(self.ptr.offset(self.len as isize), elem);
            self.len += 1;
        }
    }

    # fn reallocate(&mut self) { }
}
```

Этот код достаточно прост для разумного аудита и проверки. Добавим следующий метод:

```
fn make_room(&mut self) {
    // наращиваем размерность
    self.cap += 1;
}
```

Это 100% Безопасный Rust, но он абсолютно неустойчив. Изменение размера нарушает инварианты Vec (то, как `cap` отражает распределение памяти в Vec). И нет ничего в остальном Vec, что могло бы защитить от этого. *Придется* доверять полю размера, потому что проверить его никак нельзя.

`unsafe` не просто загрязняет всю функцию: он загрязняет весь *модуль*. В общем случае, ограничить небезопасность кода можно на границах модуля, с помощью контроля за видимостью (приватностью/публичностью) членов модуля. Указание публичности членов модуля (через ключевое слово pub) — это инструмент ограничения небезопасности, который действует только на границе модуля (это особенность языка).

Однако работает это *идеально*. Существование `make_room` - это *не* проблема устойчивости Vec, потому что мы не пометили его как публичный. Эту функцию можно вызывать только внутри модуля, в котором она определена. Также и `make_room` получает доступ напрямую к приватным полям Vec, поэтому она может быть написана только в том же модуле, что и Vec.

Таким образом мы можем написать полностью безопасную абстракцию, которая опирается на сложные инварианты. Это *очень важно* для связи между Безопасным и Небезопасным Rust. Мы уже видели, что небезопасный код должен доверять *некоторому* безопасному, но он не может доверять *произвольному* безопасному коду. Он не может доверять тому, что произвольная реализация типажа или любой функции, которые передаются ему, будут правильно себя вести в тех случаях, о которых не заботится безопасный код.

В то же время безопасность будет потеряна, если небезопасный код не сможет помешать безопасному клиентскому коду портить его состояние в произвольных случаях. Но благодаря приватности, он хотя бы *может* предотвратить то, что произвольный код будет портить его критическое состояние.

Живи безопасно!

3

Представление данных в Rust

Низкоуровневое программирование уделяет много внимания тому, как представляются данные. Это сложный вопрос. Он влияет на весь язык, поэтому начнем копать в направлении представления данных в Rust.

repr(Rust)

Во-первых, все типы имеют выравнивание, указываемое в байтах. Выравнивание типа определяет в каких адресах разрешается хранить значения. Значение, имеющее выравнивание `n`, должно храниться по адресу кратному `n`. То есть выравнивание 2 означает, что его можно хранить только по четным адресам, а 1 означает, что по любым. Выравнивание всегда больше или равно 1, и всегда является степенью 2. Большинство примитивов выровнены по своему размеру, хотя это очень платформно- специфичное поведение. В частности, на x86 `u64` и `f64` могут быть выровнены только по 32 бита.

Размер типа всегда должен быть кратным его выравниванию. Это гарантирует, что массив типов может всегда быть проиндексирован посредством смещения на величину, кратную размера типа. Имейте ввиду, что размер и выравнивание типа могут быть неизвестны статически в случае типов динамического размера.

Rust дает вам следующие способы для размещения составных данные:

- struct (именованные типы-произведения)
- tuple (анонимные типы-произведения)
- array (гомогенные типы-произведения)
- enum (именованные типы-суммы)

Enum называется *Си-подобным* если ни у одного из его вариантов нет связанных данных.

Составные структуры будут иметь выравнивание, равное максимуму из выравниваний их полей. Вследствие этого Rust добавит вставки (padding), где это необходимо, чтобы гарантиро-

вать, что все поля правильно выровнены и общий размер типа соответствует этому выравниванию. Например:

```
struct A {
    a: u8,
    b: u32,
    c: u16,
}
```

будет выровнено по 32-бита в архитектуре, которая выравнивает эти примитивы по соответствующему размеру. Таким образом полная struct будет иметь размер, кратный 32-бит. Вероятно, все станет таким:

```
struct A {
    a: u8,
    _pad1: [u8; 3], // для выравнивания `b`
    b: u32,
    c: u16,
    _pad2: [u8; 2], // для того чтобы сделать размер равным 4
}
```

Для этих типов *не используется косвенная адресация*; все данные хранятся внутри структуры, также как было бы и в Си. За исключением массивов (которые плотно упакованы и имеют порядок), положение данных, по умолчанию, не определено в Rust. Возьмем два определения struct:

```
struct A {
    a: i32,
    b: u64,
}

struct B {
    a: i32,
    b: u64,
}
```

Rust *гарантирует*, что два экземпляра A будут иметь одинаковое расположение данных. Но Rust *не гарантирует* на данный момент, что экземпляр A будет иметь такой же порядок или паддинг, как и экземпляр B, хотя на практике нет никакого основания предполагать, почему он должен отличаться.

Возьмем A и B как здесь написано, кажется, что все ясно, но другие особенности Rust используют более сложные игры с расположением данных внутри языка.

Например, считаем, что есть такая struct:

```
struct Foo<T, U> {
    count: u16,
    data1: T,
    data2: U,
}
```

Рассмотрим мономорфизации `Foo<u32, u16>` и `Foo<u16, u32>`. Если Rust расположит поля в указанном порядке, мы ожидаем, что он набьет вставками (padding) значения в struct для того, чтобы удовлетворить требования выравнивания. Поэтому, если Rust не переопределит порядок полей, мы ожидаем, что он выработает следующее:

```
struct Foo<u16, u32> {
    count: u16,
    data1: u16,
    data2: u32,
}

struct Foo<u32, u16> {
    count: u16,
    _pad1: u16,
    data1: u32,
    data2: u16,
    _pad2: u16,
}
```

В последнем случае место просто напрасно расходуется. Оптимальное использование места, таким образом, требует, чтобы у разных мономорфизаций *менялся порядок полей*.

Внимание: это гипотетическая оптимизация, которая еще не реализована в Rust 1.0

Перечисления делают этот анализ даже еще сложнее. Наивно предполагаем, что перечисление:

```
enum Foo {
    A(u32),
    B(u64),
    C(u8),
}
```

будет расположен так:

```
struct FooRepr {
    data: u64, // здесь или u64, u32 или u8 - зависит от `tag`
    tag: u8,   // 0 = A, 1 = B, 2 = C
}
```

И на самом деле почти так и будет все расположено в памяти в общем случае (размер данных и позиция `tag`).

Но есть случаи, в которых такое представление является неэффективным. Классический вариант - “оптимизация нулевого указателя” Rust: перечисления, состоящие из одного варианта (например, `None`) и (возможно вложенного) варианта с ненулевым указателем (например, `&T`) делают тэг необязательным, потому что значение нулевого указателя может безопасно означать, что выбран первый вариант (в данном примере, `None`). Конечный результат - например, будет таким `size_of::<Option<&T>>() == size_of::<&T>()`.

Много типов в Rust, которые являются или содержат ненулевые указатели, такие как `Box<T>`, `Vec<T>`, `String`, `&T` и `&mut T`. По аналогии, можно представить, как вложенные перечисления объединяют свои тэги в одно значение, так как по определению известно, что у них ограниченный набор правильных значений. В принципе перечисления могут использовать довольно сложные алгоритмы для размещения битов всех вложенных типов в специальном ограничивающем их представлении. Поэтому сейчас нам кажется, что оставить нетронутым расположение перечислений в памяти, будет *особенно* целесообразно.

Типы экзотического размера

Большую часть времени мы думаем в терминах типов фиксированного положительного размера. Однако, это не всегда так.

Типы динамического размера

Rust на самом деле поддерживает типы динамического размера (ТДР; англ. dynamically sized type, DST): типы без статически известного размера или выравнивания. На поверхности, все это кажется бессмысленным: Rust *должен* знать размер и выравнивание чего-либо, чтобы корректно с этим работать! В этом отношении, ТДР - это не нормальные типы. Из-за отсутствия статически известного размера, они могут скрываться только за указателем особого типа. Любой указатель на ТДР вследствие этого становится *толстым*, состоящим из указателя и информации, которая “дополняет” его (подробнее об этом ниже).

Язык предлагает два главных ТДР: типажи-объекты и срезы.

Типаж-объект представляет тип, реализующий указанный типаж. Точный исходный тип *затирается* и заменяется таблицей виртуальных методов (vtable), содержащей всю информацию, необходимую для использования типа. Вот эта информация и дополняет типаж-объект: указатель на его таблицу виртуальных методов.

Срез - это просто отображение в какое-то подходящее хранилище - обычно в массив или `Vec`. Информация, которая дополняет срез - это просто количество элементов, на которые он указывает.

Вообще, структуры могут хранить один ТДР прямо в своем последнем поле, но это делает их также ТДР:

```
// Невозможно хранить напрямую в стеке
struct Foo {
    info: u32,
    data: [u8],
}
```

Внимание: В Rust 1.0 ТДР, являющиеся структурами, работают неправильно, если последнее поле имеет переменную позицию, зависящую от выравнивания.

Типы с нулевым размером

Rust на самом деле позволяет объявлять типы, которые не занимают места:

```
struct Foo; // Нет полей = нет размера

// У каждого поля нет размера  = нет размера
struct Baz {
    foo: Foo,
    qux: (),      // у пустого кортежа нет размера
    baz: [u8; 0], // у пустого массива нет размера
}
```

Сами по себе типы с нулевым размером (ТНР; англ. zero-sized types, ZST), по очевидным причинам, довольно бесполезны. Но, как и с другими любопытными решениями по выделению памяти в Rust, их потенциал выражается в контексте обобщений: Rust очень хорошо понимает, что любые операции, которые создают или хранят ТНР, могут быть заменены на пустую операцию. Прежде всего хранение их вообще не имеет смысла - они не занимают никакого места. Также, есть только одно значение этого типа, поэтому все, что загружает их, может создать их из эфира - что тоже является пустой операцией из-за того, что они опять же не занимают никакого места.

Одним из самых важных примеров являются множества (sets) и словари (maps). Имея `Map<Key, Value>`, часто реализуют `Set<Key>` в качестве тонкой обёртки вокруг `Map<Key, БесполезныйМусор>`. Многие языки заставляют выделять место под БесполезныйМусор и вынуждают обрабатывать его хранение и загрузку только для того, чтобы потом выкинуть его. Их компиляторам сложно доказать, что эти действия не нужны.

Однако в Rust мы можем просто сказать `Set<Key> = Map<Key, ()>`. Так Rust понимает, что любая загрузка и хранение бесполезны, и выделение памяти не нужно. В результате получаем, что мономорфный код - это обычная частая реализация HashSet без каких бы накладных расходов по поддержке значений.

Безопасному коду не надо волноваться о ТНР, но *небезопасный* должен быть очень аккуратен с последствиями, которые влекут типы без размера. В частности, смещение указателей это no-op, и стандартный распределитель памяти (включая jemalloc, который использует по умолчанию Rust) может вернуть `nullptr` если запрашивается выделение памяти под тип с нулевым размером, и это будет неотличимо от ситуации нехватки памяти.

Пустые типы

Rust также позволяет объявлять типы, *экземпляр которых нельзя создать*. О них можно говорить только на уровне типов, но никогда на уровне значений. Пустые типы можно объявить, указав перечисление без вариантов:

```
enum Void {} // Вариантов нет = Пусто
```

Пустые типы еще более маргинальны чем ТНР. Единственное назначение типа Void из примера выше - недостижимость на уровне типов. Предположим, API в общем случае должно

возвращать Result, но какой-то частный случай никогда не возвращает ошибку. Здесь можно сообщить об этом на уровне типа, возвращая `Result<T, Void>`. Пользователи этого API могут спокойно делать unwrap такого Result, зная что *статически невозможно* получить `Err` в значении, так как пришлось бы предоставлять значение типа `Void`.

В принципе, Rust мог бы выполнять некоторый интересный анализ и оптимизацию, зная все это. Например, `Result<T, Void>` мог бы быть представлен просто как `T`, потому что случая `Err` на самом деле не существует. Код ниже *мог* бы компилироваться:

```
enum Void {}

let res: Result<u32, Void> = Ok(0);

// Err не существует, поэтому Ok полностью безошибочен.
let Ok(num) = res;
```

Но на данный момент ни один из этих трюков не работает, поэтому все, что дает вам тип void - это возможность быть уверенным в том, что некоторые ситуации *статически невозможны*.

И последний тонкий момент о пустых типах - сырые указатели на них создавать можно и это считается правильным, но разыменование таких указателей приведет к Неопределенному поведению, потому что никакого смысла в этом нет. Таким образом, вы можете смоделировать `void *` из Си с помощью `*const Void`, но это не обязательно даст выигрыш по сравнению с использованием, например, `*const ()`, который *безопасен* по отношению к случайному разыменованию.

Альтернативные представления

Rust позволяет вам определять альтернативные стратегии расположения данных в памяти, отличающиеся от стратегии по умолчанию.

repr(C)

Это самая важная `repr`. У нее простая цель: делать то, что делает Си. Порядок, размер и выравнивание полей полностью соответствуют тому, что вы ожидаете от Си или C++. Все типы, проходящие через границы FFI, должны иметь `repr(C)`. Связано это с тем, что Си - это лингва-франка в мире программирования. Она также необходима, чтобы правильно делать некоторые сложные трюки с размещением данных в памяти вроде интерпретации чисел, как значений других типов.

Но надо держать в голове особенности взаимодействия с более экзотическим размещением данных Rust в памяти. Из-за своего двойственного назначения - “для FFI” и “для контроля размещения в памяти”, `repr(C)` можно применять к типам, которые нечувствительны или, наоборот, проблемны при прохождении границ FFI.

- ТНР не занимают места, даже учитывая, что это не стандартное поведение в Си, и что это явно противоречит поведению пустых типов в C++, которые занимают байт свободного места.

- ТДР, кортежи и типы-суммы не существуют в Си и, следовательно, всегда являются FFI небезопасными.
- Кортежные структуры похожи на структуры, если смотреть относительно `repr(C)`, с единственным отличием, заключающимся в отсутствии имен у полей.
- **Если у типа есть [drop flags], они все равно будут добавлены**
- Для перечислений это эквивалентно одному из `repr(u*)` (смотри следующий раздел). Выбранный размер является размером по умолчанию перечислений для C ABI целевой платформы. Помните, что представление перечислений в Си зависит от реализации, поэтому все это на самом деле только догадка. В частности, это может быть не так, если интересующий код на Си компилировать с определенными флагами.

repr(u8), repr(u16), repr(u32), repr(u64)

Для создания Си-подобных перечислений нужно указать размер. Если дискриминант переполняет целое, в который его нужно уместить, возникнет ошибка компиляции. Вы можете вручную попросить Rust явно заменять переполняющийся элемент на 0. Но Rust не разрешит вам создать перечисление, в котором два варианта будут иметь одинаковый дискриминант.

Для не-Си-подобных перечислений, это запретит выполнять определенные оптимизации, такие как оптимизации нулевого указателя.

Эти представления никак не влияют на структуры.

repr(packed)

`repr(packed)` заставит Rust убрать любой паддинг и выравнять тип по байту. Это улучшит использование памяти, но появятся негативные побочные последствия.

В частности, большинство архитектур *строго* требуют выравнивать значения. Это означает, что обращение к памяти по невыровненному адресу выполняется дольше (x86), или прервется с ошибкой (на некоторых чипах ARM). Для простых случаев, как прямая загрузка или сохранение упакованного поля, компилятору удастся сгладить проблемы выравнивания сдвигами и масками. Но если вы создадите ссылку на упакованное поле, очень маловероятно, что компилятору удастся избежать невыровненной загрузки.

Для Rust 1.0 это может вызвать неопределенное поведение.[1]

`repr(packed)` использовать непросто. Если у вас нет строгих требований, лучше ее не использовать.

Это - модификатор для `repr(C)` и `repr(rust)`.

[1] https://github.com/rust-lang/rust/issues/27060

4

Владение и время жизни

Владение - это пробивная особенность Rust. Оно позволяет Rust быть полностью безопасным по памяти и эффективным, избегая сборки мусора. Перед тем как детально разобрать систему владения, мы объясним предпосылки такого дизайна.

Мы подразумеваем, что вы принимаете, что сборщик мусора - не всегда самое оптимальное решение, и что желательно в некоторых случаях управлять памятью вручную. Если вы это не принимаете, возможно, Rust не заинтересует вас.

Несмотря на ваши чувства к GC, он является *огромным* благом, позволяющим делать код безопасным. Вам никогда не нужно волноваться, что некоторые вещи удалятся *слишком рано* (хотя нужно ли вам по-прежнему иметь ссылку на них в этом случае - это другой вопрос...). Это распространенная проблема, с которой приходится иметь дело программам на C и C++. Посмотрите на эту простую ошибку, которую каждый из нас когда-то делал в языке без GC:

```
fn as_str(data: &u32) -> &str {
    // compute the string
    let s = format!("{}", data);

    // О НЕТ! Мы возвращаем ссылку на что-то,
    // что существует только в этой функции!
    // Висячий указатель! Используется после освобождения! Увы и ах!
    // (Rust не компилирует этот код)
    &s
}
```

Вот что именно должна решать система владения в Rust. Rust знает область видимости, в которой живет `&s`, и поэтому не даст выйти из нее. Однако это простой случай, который, скорее всего, сможет поймать даже компилятор Си. Все становится сложнее, когда код растет и указатели передаются в различные функции. В конечном счете, компилятор Си не сможет выполнить анализ областей видимости, чтобы доказать, что ваш код сломан. Ему придется заставить себя принять вашу программу, предполагая, что она правильна.

Этого никогда не произойдет в Rust. Программист должен доказать компилятору, что ничего не сломается.

Конечно, рассказ о владении в Rust гораздо сложнее, чем просто проверка, что ссылка не выходит за область видимости того, на что она ссылается. Потому что доказать, что указатели всегда правильны, гораздо сложнее. Например, в этом коде

```
let mut data = vec![1, 2, 3];
// получаем внутреннюю ссылку
let x = &data[0];

// О НЕТ! `push` заставляет пересчитать занимаемое место `data`.
// Висячий указатель! Используется после освобождения! Увы и ах!
// (Rust не компилирует этот код)
data.push(4);

println!("{}", x);
```

обычный анализ областей видимости не сможет поймать ошибку, потому что `data` на самом деле живет столько, сколько надо. Но она *поменялась* в то время, когда у нас есть ссылка на нее. Вот поэтому Rust требует, чтобы любые ссылки замораживали объекты, на которые ссылаются, и владельцев этих объектов.

Ссылки

В этом разделе дается высокоуровневый взгляд на модель памяти, которой должны соответствовать *все* программы на Rust. Безопасный код статически проверяется на соответствие этой модели анализатором заимствований. Небезопасный код может выходить за рамки анализатора заимствований, если он соответствует этой модели. Анализатор заимствований можно расширить, позволив большему количеству программ компилироваться, пока они удовлетворяют самой основной модели.

Существует два типа ссылок:

- Общая ссылка: `&`
- Изменяемая ссылка: `&mut`

которые подчиняются следующим правилам:

- Ссылки не могут пережить то, на что ссылаются
- Изменяемые ссылки не могут совпадать

Вот и все. Это вся модель. Конечно, нам надо определить, что означает *совпадение указателей*. Чтобы это сделать, мы должны определить значения *путей* и *живучести*.

Внимание: Приведённая ниже модель считается сомнительной и имеет проблемы. Она нормальна для объяснения, но не в состоянии охватить всю семантику. Оставим ее так для объяснения понятий, дающихся дальше в этой главе. Она существенно поменяется в будущем. TODO: поменять ее.

Пути

Если бы у Rust были только значения (без указателей), тогда каждым значением владела бы одна переменная или составная структура. Из этого мы получаем *дерево* владения. Сам стек является корнем дерева, а каждая его переменная является прямым наследником. Прямыми наследниками каждой переменной будут её поля (если они есть) и так далее.

С этой точки зрения, каждому значению в Rust соответствует уникальный *путь* по дереву владения. Особый интерес представляют *предки* и *потомки*: если x владеет y, то x является предком y, а y потомком x. Заметьте, что это включительное отношение: x является предком и потомком самого себя.

Мы можем определить ссылки как просто *названия* путей. Когда вы создаете ссылку, вы объявляете, что есть владеющий путь к этому адресу в памяти.

К несчастью, множество данных живут не на стеке, а мы должны это учитывать. Глобальные переменные и переменные, локальные для потока (т.е. находящиеся в TLS), достаточно просты, и их можно разместить на дне стека в модели (хотя мы должны быть осторожны с изменяемыми глобальными переменными). Данные же в куче обнажают другие проблемы.

Если бы в Rust в куче могли размещаться только данные, которыми уникально владеет указатель на стеке, то мы могли бы просто трактовать такой указатель как структуру, владеющую значением в куче. Box, Vec, String, и HashMap являются примерами типов, уникально владеющими данными в куче.

К сожалению, у данных в куче не *всегда* есть один уникальный владелец. Rc, например, представляет собой вариант *общего* владения. Общее владение значением означает, что есть больше одного пути к нему. Значение, у которого больше одного пути к нему, ограничивает то, что можно с ним сделать.

Итак, только общие ссылки могут быть созданы к неуникальным путям. Однако механизмы, гарантирующие взаимоисключение, могут временно обозначить Одного Настоящего Владельца, определив уникальный путь к этому значению (и, таким образом, к его детям). Если так получится, значение можно будет изменять. В частности, можно создать изменяемую ссылку на него.

Наиболее распространенным способом является создание такого пути через *внутреннюю изменяемость*, которая отличается от *наследуемой изменяемости*, используемой обычно везде в Rust. Cell, RefCell, Mutex и RWLock - это все примеры типов с внутренней изменяемостью. Эти типы предоставляют эксклюзивный доступ с помощью проверок во время исполнения.

Интересным случаем является Rc сам по себе: если у Rc счетчик ссылок равен 1, то Rc можно безопасно изменять и даже перемещать его внутренние значения. Помните, однако, что счетчик ссылок сам по себе использует внутреннюю изменяемость.

Чтобы правильно взаимодействовать с системой типов, которая позволяет переменным или полям структуры иметь внутреннюю изменяемость, необходимо обернуть все в UnsafeCell. Что само по себе не делает безопасным выполнение операции по внутренней изменяемости значений. Вы сами должны гарантировать, что обеспечите взаимоисключение изменений (например, из разных потоков).

Живучесть

Внимание: Живучесть - это не то же самое, что и *время жизни*, которое объясняется детально в следующем разделе этой главы.

Грубо говоря, ссылка *жива* в какой-то момент в программе, если ее можно разыменовать. Общие ссылки всегда живы, даже если они буквально недостижимы (например, они живут в освобожденной или утекшей памяти). Изменяемые ссылки могут быть достижимы, но не быть *живыми* во время процесса *передачи заимствования*.

Изменяемая ссылка может передать заимствование в общую или в изменяемую ссылку одному из своих потомков. Ссылка с переданным заимствованием оживет заново, после того, как у всех производных от нее ссылок истечет время жизни. Например, изменяемая ссылка может передать заимствование полю объекта, на который она указывает:

```
let x = &mut (1, 2);
{
    // передача заимствования под-полем x в y
    let y = &mut x.0;
    // y теперь жива, а x нет
    *y = 3;
}
// y выходит из области видимости, поэтому x опять жива
*x = (5, 7);
```

Разрешается также передавать заимствование *несколькими* изменяемыми ссылками, если они *не пересекаются*: каждая ссылка не является предком другой. Rust позволяет явно делать это с помощью непересекающихся полей структур, потому что их разделение может быть статически доказано:

```
let x = &mut (1, 2);
{
    // передача заимствования x двум непересекающимся под-полям
    let y = &mut x.0;
    let z = &mut x.1;

    // y и z живы, но x нет
    *y = 3;
    *z = 4;
}
// y и z выходят из области видимости, поэтому x опять жива
*x = (5, 7);
```

Однако, часто случается, что Rust недостаточно умен, чтобы доказать, что множественное заимствование не пересекается. *Это не означает, что фундаментально неправильно делать такое заимствование*, просто Rust не настолько умен, как вам бы хотелось.

Для упрощения, мы можем представлять переменные, как ссылки несуществующего типа: *обладаемые* ссылки. Обладаемые ссылки похожи семантикой на изменяемые ссылки: они могут передавать заимствование также как и изменяемые и общие ссылки, заканчивая жить после

этого. Живые обладаемые ссылки обладают уникальным свойством того, что из них можно перемещать значение (хотя значение из изменяемых ссылок *можно* заменить другим). Эта сила дается только *живым* обладаемым ссылкам, потому что перемещение того, на что они указывают, преждевременно сделало бы все внешние ссылки недействительными.

Благодаря локальному статическому анализу на правильность изменяемости, только переменные, помеченные `mut` могут быть заимствованы изменяемыми.

Интересно отметить, что Box ведет себя также как обладаемая ссылка. То, на что он указывает, можно переместить, и Rust достаточно умен, чтобы рассуждать о пути к нему, как об обычной переменной.

Совпадение указателей

Определив живучесть и путь, можем перейти к определению *совпадения указателей*:

У изменяемой ссылки совпадает указатель с другой ссылкой, если существует хотя бы одна другая живая ссылка на один из ее предков или потомков.

(Если хотите, можете сказать, что у двух живых ссылок совпадают указатели *друг с другом*. На семантике это не сказывается, но, вероятно, так будет понятнее для проверки корректности кода.)

Вот и все. Очень просто, правда? За исключением того, что нам пришлось на двух страницах определять все термины для этого. Ну, знаете: это Очень Просто.

На самом деле все немного сложнее. Помимо ссылок в Rust есть *сырые указатели*: `*const T` и `*mut T`. У них нет наследуемого владения или семантики совпадения указателей. В результате, Rust не делает абсолютно никаких попыток отследить, что они правильно используются, и они дико небезопасны.

Это еще открытый вопрос, под каким углом зрения сырые указатели определяются относительно семантики совпадения указателей. Но важно, чтобы в этом определении было обоснованно, что существование сырого указателя не подразумевает некоего живого пути

Время жизни

Rust проверяет соблюдение правил с помощью *времён жизни*. Время жизни - это просто название областей видимости где-то в программе. Каждая ссылка, или то, что содержит ссылки, помечается временем жизни, определяя область видимости, в которой она правильна.

Внутри тела функции Rust обычно не позволяет вам явно назвать время жизни. Потому что в общем случае в локальном контексте нет смысла говорить о времени жизни; Rust обладает всей информацией и может обработать все достаточно оптимально. Иначе вам пришлось бы создавать много анонимных областей и временных переменных, просто чтобы заставить ваш код работать.

Но, когда вы переходите границы тела функции, вы должны начать говорить о времени жизни. Время жизни обозначаются апострофами: `'a`, `'static`. Чтобы окунуться во время жизни, мы сделаем вид, что нам разрешено помечать области видимости временем жизни, и разберем пример из начала этой главы.

Изначально, наши примеры созданы с использованием *сильного* синтаксического сахара – с кукурузным сиропом с высоким содержанием фруктозы – это касается областей видимости и времен жизни, потому что написание всего этого в явной форме *сильно замусоривает* код. Весь код Rust опирается на агрессивный вывод и опускание “очевидных” вещей.

Первым особо интересным кусочком сахара является то, что каждое утверждение `let` неявно объявляет область видимости. Для большинства случаев это абсолютно не важно. Но это очень важно, когда переменные ссылаются друг на друга. В качестве простого примера, давайте полностью избавимся от сахара в этом куске кода на Rust:

```
let x = 0;
let y = &x;
let z = &y;
```

Анализатор заимствований всегда пытается минимизировать величину времени жизни, поэтому скорее всего после избавления от сахара пример будет выглядеть так:

```
// Внимание: `'a: {` и `&'b x` это неправильный синтаксис!
'a: {
    let x: i32 = 0;
    'b: {
        // используемое время жизни 'b потому что его достаточно.
        let y: &'b i32 = &'b x;
        'c: {
            // то же самое с 'c
            let z: &'c &'b i32 = &'c y;
        }
    }
}
```

Ух ты! Это... ужасно. Давайте здесь остановимся и поблагодарим Rust за то, что делает это проще.

На самом деле передача ссылок наружу из области видимости заставит Rust вывести большее время жизни:

```
let x = 0;
let z;
let y = &x;
z = y;
```

```
'a: {
    let x: i32 = 0;
    'b: {
        let z: &'b i32;
        'c: {
            // Нужно использовать 'b здесь, потому что ссылка
            // передается снаружи в эту область видимости.
            let y: &'b i32 = &'b x;
```

```
            z = y;
        }
    }
}
```

Пример: ссылки, которые переживают то, на что ссылаются

Отлично, давайте теперь посмотрим на некоторые примеры из тех, что были раньше:

```
fn as_str(data: &u32) -> &str {
    let s = format!("{}", data);
    &s
}
```

убрав сахар, получаем:

```
fn as_str<'a>(data: &'a u32) -> &'a str {
    'b: {
        let s = format!("{}", data);
        return &'a s;
    }
}
```

Сигнатура `as_str` берет ссылку на u32 с *каким-то* временем жизни, и обещает создать ссылку на str, которая сможет жить *столько, сколько надо*. Мы уже видели, почему такая сигнатура вызывает проблемы. Подразумевается, в принципе, что мы собираемся использовать str в области видимости, в которой возникла ссылка на u32 или *в еще большей области видимости*. Это уже задача потруднее.

Мы вычисляем строку `s`, и возвращаем ссылку на нее. По контракту нашей функции ссылка должна пережить `'a`, поэтому это время жизни мы и подставим для нее. К сожалению, `s` была определена в области видимости `'b`, поэтому единственный вариант этого, если `'b` содержит `'a` – что, очевидно, быть не может, ведь во время жизни `'a` должен входить сам вызов функции. Таким образом мы создали ссылку, чье время жизни превосходит время жизни того, на что она ссылается, а это *буквально* первое, что мы сказали, что ссылки делать не могут. Компилятор по праву взрывается перед нашими глазами.

Для лучшего объяснения расширим пример:

```
fn as_str<'a>(data: &'a u32) -> &'a str {
    'b: {
        let s = format!("{}", data);
        return &'a s
    }
}

fn main() {
    'c: {
```

```
        let x: u32 = 0;
        'd: {
            // Анонимная область видимости создается, потому что заимствование
            // не обязано продолжаться на всю область видимости x.
            // Возврат as_str должен найти str где-то раньше вызова
            // этой функции. Очевидно, что этого не произойдет.
            println!("{}", as_str::<'d>(&'d x));
        }
    }
}
```

Выстрел!

Конечно, правильно будет написать эту функцию так:

```
fn to_string(data: &u32) -> String {
    format!("{}", data)
}
```

Мы можем создать обладаемое значение внутри функции, чтобы вернуть его! Единственный способ, которым мы могли бы вернуть `&'a str`, это если бы оно было полем структуры `&'a u32`, но это явно не наш случай.

(Вообще, мы могли бы просто вернуть литеральную строку, которая, являясь глобальным объектом, живет на дне стека; хотя это ограничит нашу реализацию *немного*.)

Пример: совпадение указателей у изменяемой ссылки

Как насчет другого примера:

```
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);
```

```
'a: {
    let mut data: Vec<i32> = vec![1, 2, 3];
    'b: {
        // 'b длится столько, сколько нам нужно, чтобы длилось заимствование
        // (достаточно, чтобы добраться до  `println!`)
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {
            // Временная область нужна, поскольку мы не хотим чтобы
            // &mut жила дольше.
            Vec::push(&'c mut data, 4);
        }
        println!("{}", x);
    }
}
```

Проблема здесь чуть тоньше и интересней. Мы хотим, чтобы Rust не принял эту программу по следующей причине: у нас есть живая общая ссылка `x` на потомок `data` в то время, когда мы пытаемся взять `data` как изменяемую ссылку и выполнить `push`. Это приведет к созданию совпадающего указателя на изменяемую ссылку, что нарушает *второе* правило ссылок.

Однако, *совсем не по этой причине* Rust считает, что программа неверна. Rust не понимает, что `x` - это ссылка на под-путь к `data`. Он абсолютно не понимает Vec. Все, что он *видит*, это то, что `x` должен жить время `'b`, чтобы его можно было напечатать. Дальше сигнатура `Index::index` требует, чтобы ссылка по которой мы берем `data`, была живой в течение времени `'b`. Когда мы пытаемся вызвать `push`, компилятор видит, что мы пытаемся сделать `&'c mut data`. Rust знает, что `'c` содержится внутри `'b`, и отказывается принимать нашу программу, потому что `&'b data` все еще жива к этому моменту!

Мы видим, что система времен жизни гораздо грубее, чем ссылочная семантика, в сохранении которой мы на самом деле заинтересованы. Для большинства случаев, *так совершенно нормально*, потому что позволяет не проводить целый день, объясняя нашу программу компилятору. Но это означает, что некоторые абсолютно правильные по семантике программы будут отвергнуты, из-за того что система времен жизни слишком глупа.

Границы времени жизни

У нас есть следующий код:

```
struct Foo;

impl Foo {
    fn mutate_and_share(&mut self) -> &Self { &*self }
    fn share(&self) {}
}

fn main() {
    let mut foo = Foo;
    let loan = foo.mutate_and_share();
    foo.share();
}
```

Ожидаем, что он компилируется. Мы вызываем `mutate_and_share`, который временно заимствует `foo` как изменяемую ссылку, но затем возвращает только как общую ссылку. Поэтому мы ожидаем, что `foo.share()` выполнится успешно, ведь `foo` уже не должна быть заимствована как изменяемая ссылка.

Однако, когда мы попытаемся выполнить компиляцию:

```
<anon>:11:5: 11:8 error: cannot borrow `foo` as immutable because it is also borrowed
↳ as mutable
<anon>:11     foo.share();
              ^~~
<anon>:10:16: 10:19 note: previous borrow of `foo` occurs here; the mutable borrow pre
↳ vents subsequent moves, borrows, or modification of `foo` until the borrow ends
```

```
<anon>:10     let loan = foo.mutate_and_share();
                         ^~~
<anon>:12:2: 12:2 note: previous borrow ends here
<anon>:8 fn main() {
<anon>:9     let mut foo = Foo;
<anon>:10     let loan = foo.mutate_and_share();
<anon>:11     foo.share();
<anon>:12 }
          ^
```

Что произошло? Ну, причина все та же, что и в примере 2 из предыдущей секции. Уберем синтаксический сахар из программы и получим следующее:

```
struct Foo;

impl Foo {
    fn mutate_and_share<'a>(&'a mut self) -> &'a Self { &'a *self }
    fn share<'a>(&'a self) {}
}

fn main() {
    'b: {
        let mut foo: Foo = Foo;
        'c: {
            let loan: &'c Foo = Foo::mutate_and_share::<'c>(&'c mut foo);
            'd: {
                Foo::share::<'d>(&'d foo);
            }
        }
    }
}
```

Система времени жизни вынуждена продлить время жизни `&mut foo` до времени `'c` из-за времени жизни `loan` и сигнатуры mutate_and_share. Дальше, когда мы пытаемся вызвать `share`, и она видит, что мы пытаемся взять ту же ссылку, что и `&'c mut foo`, все взрывается у нас на глазах!

Программа абсолютно корректна в части семантики ссылок, о которой мы на самом деле заботимся, но система времени жизни слишком крупнозерниста, чтобы понять это.

TODO: другие общие проблемы? SEME regions stuff, mostly?

Опускание времени жизни

Для того, чтобы распространённый код было удобнее писать, Rust позволяет *опускать* время жизни в сигнатурах функций.

Положение времени жизни находится в:

```
&'a T
&'a mut T
T<'a>
```

Время жизни может появляться как “на входе”, так и “на выходе”:

- В описании `fn` на входе время жизни применяется к аргументам `fn`, в то время как на выходе - к возвращаемому результату. Поэтому в `fn foo(s: &str) -> (&str, &str)` опускается одно время жизни на входе и два на выходе. Помните, что в описании метода на входе не опускается время жизни из заголовка `impl` этого метода (также не опускаются времена жизни из заголовка типажа, для метода по умолчанию).
- В будущем, можно будет опускать время жизни также в заголовках `impl`.

Правила опускания:

- Каждое опущенное время жизни на входе становится отдельным параметром.
- Если есть только одно время жизни на входе (опущенное или нет), это время жизни присваивается *всем* опущенным временам жизни на выходе.
- Если есть несколько времен жизни на входе, но одно из них `&self` или `&mut self`, время жизни `self` присваивается *всем* опущенным временам жизни на выходе.
- Все другие случаи считаются ошибочными, если время жизни на выходе опущено.

Примеры:

```
fn print(s: &str);                                    // опущено
fn print<'a>(s: &'a str);                             // указано явно

fn debug(lvl: uint, s: &str);                         // опущено
fn debug<'a>(lvl: uint, s: &'a str);                  // указано явно

fn substr(s: &str, until: uint) -> &str;              // опущено
fn substr<'a>(s: &'a str, until: uint) -> &'a str;    // указано явно

fn get_str() -> &str;                                 // НЕПРАВИЛЬНО

fn frob(s: &str, t: &str) -> &str;                    // НЕПРАВИЛЬНО

fn get_mut(&mut self) -> &mut T;                      // опущено
fn get_mut<'a>(&'a mut self) -> &'a mut T;            // указано явно

fn args<T: ToCStr>(&mut self, args: &[T]) -> &mut Command                  // опущено
fn args<'a, 'b, T: ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command // указано
↳ явно

fn new(buf: &mut [u8]) -> BufWriter;                  // опущено
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>        // указано явно
```

Безграничные времена жизни

Небезопасный код часто может создавать ссылки или времена жизни из воздуха. Такие времена рождаются *безграничными*. Основным источником этого является разыменование сырого указателя, создающее ссылку с безграничным временем жизни. Такое время устанавливается соответствующим контексту. На самом деле оно даже мощнее, чем просто `'static`, потому что, например, `&'static &'a T` не пройдет проверку типов, а безграничное время жизни спокойно превратится в `&'a &'a T`, если это необходимо. Однако, в большинстве случаев, безграничное время жизни можно рассматривать как `'static`.

У нас почти нет ссылок с временем жизни `'static`, поэтому, возможно, последнее утверждение и неверно. `transmute` и`transmute_copy` являются двумя другими основными нарушителями этого. Следует прилагать все усилия, чтобы как можно быстрее ограничить такие безграничные времена жизни, в особенности это касается пересечения границ функций.

В описании функции любые времена жизни на выходе, которые нельзя вывести из времен на входе, будут безграничными. Например:

```
fn get_str<'a>() -> &'a str;
```

создаст `&str` с безграничным временем жизни. Самым простым способом избежать этого будет использовать опускание времени жизни на границе функции. Если время жизни на выходе опущено, то оно *должно* быть ограничено временем жизни на входе. Конечно, возможно, оно будет ограничено *неправильно*, но это скорее вызовет просто ошибку компиляции, но не нарушит безопасность памяти.

Внутри функции ограниченные времена жизни больше подвержены ошибкам. Самым безопасным и простым способом ограничить время жизни будет вернуть его из функции с ограниченным временем жизни. Но, если это невозможно, то ссылку можно разместить в позиции, у которого указано конкретное время жизни. К сожалению, невозможно именовать все времена жизни используемые внутри функции.

Ограничения типажей высшего порядка

Типажи `Fn` в Rust - это уличная магия. Например, мы можем написать следующий код:

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    where F: Fn(&(u8, u16)) -> &u8,
{
    fn call(&self) -> &u8 {
        (self.func)(&self.data)
    }
}
```

```
fn do_it(data: &(u8, u16)) -> &u8 { &data.0 }

fn main() {
    let clo = Closure { data: (0, 1), func: do_it };
    println!("{}", clo.call());
}
```

Если мы попытаемся убрать синтаксический сахар так же, как мы делали в предыдущих секциях, у нас возникнут проблемы:

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    // where F: Fn(&'??? (u8, u16)) -> &'??? u8,
{
    fn call<'a>(&'a self) -> &'a u8 {
        (self.func)(&self.data)
    }
}

fn do_it<'b>(data: &'b (u8, u16)) -> &'b u8 { &'b data.0 }

fn main() {
    'x: {
        let clo = Closure { data: (0, 1), func: do_it };
        println!("{}", clo.call());
    }
}
```

Каким же образом нам выразить границы времени жизни типажа `F`? Мы должны предложить какое-нибудь время жизни, однако, оно не будет известно до тех пор пока мы не войдем в тело `call`! К тому же, это не какое-то фиксированное время; `call` работает с *любым* временем жизни, которое будет у `&self` в этот момент.

Такая работа требует магии ограничения типажей высшего порядка (ОТВП). Убрать синтаксический сахар можно так:

```
where for<'a> F: Fn(&'a (u8, u16)) -> &'a u8,
```

(где `Fn(a, b, c) -> d` - это сам по себе сахар для нестабильного *настоящего* типажа `Fn`)

`for<'a>` можно прочитать как “для всех возможных `'a`”, и в общем случае это создаст *бесконечный список* границ типажа, которым должен соответствовать F. Сильно. Помимо типажей `Fn` есть не так уж много мест, где мы можем встретить ОТВП. И даже в этих случаях чаще всего нам поможет синтаксический сахар.

Подтипы и вариантность

Хотя в Rust не входят никакие средства структурного наследования, он *включает* в себя выделение подтипов. В Rust оно полностью вытекает из времен жизни. Благодаря временам жизни и областям видимости мы можем частично упорядочить выделение подтипов в виде отношений *содержания* (`'a` переживает `'b`). Мы даже можем выразить это в виде обобщенной границы.

Выделение подтипов по временам жизни в терминах таких отношений будет следующим: если `'a: 'b` (“a содержит b” или “a живет дольше, чем b”), то `'a` является подтипом `'b`. Здесь существует огромная вероятность ошибиться, потому что интуитивно кажется, что должно быть наоборот: большая область видимости является *подтипом* меньшей.

Хотя на самом деле в этом есть смысл. Интуитивная причина этого - если вы ожидаете `&'a u8`, то вполне нормально передать вам `&'static u8`, так же как если бы вы ожидали Животного в Java, то было бы нормально передать вам Кошку. Кошка - это Животное *и кое-что еще*, также как `'static` это просто `'a` *и кое- что еще*.

(Помните, что отношения подтипов и типизация времен жизни - это довольно произвольная конструкция, с которой некоторые не соглашаются. Однако она прилично упрощает нам жизнь в части анализа времен жизни.)

Высокоуровневые времена жизни - это тоже подтипы каждого конкретного времени жизни. Так происходит, потому что произвольное время жизни, строго говоря, больше, чем какое-либо конкретное время жизни.

Вариантность

Вот где вещи становятся немного сложнее.

Вариантность - это то, как *конструкторы типа* относятся к своим аргументам. Конструктор типа в Rust - это обобщённый тип без ограничений на аргументы. Например `Vec` - это конструктор типа, который принимает на входе `T` и возвращает `Vec<T>`. `&` и `&mut` - это конструкторы типа, которые принимают 2 аргумента на входе: время жизни и тип, на который указывать.

Вариантность конструктора типа - это то, как выделение подтипов из аргументов на входе влияет на выделение подтипов на выходе. В Rust присутствуют два типа вариантности:

- Если из того, что `T` является подтипом `U`, следует, что `F<T>` является подтипом `F<U>`, то F *вариантна* над `T`. (выделение подтипов “проходит насквозь”)
- F *инвариантна* над `T` в противном случае (нельзя выделить отношений подтипов)

(Для тех, кто сталкивался с вариантностью в других языках - то, что мы относим к “просто” вариантности, на самом деле является *ковариантностью*. У Rust есть *контрвариантность* для функций. Будущее контрвариантности еще не определено, и она может быть удалена. На данный момент `fn(T)` контрвариантна над `T`, и используется при поиске реализации, подходящей под определение типажа. У типажей нельзя вывести вариантность, поэтому `Fn(T)` инвариантна к `T`).

Некоторые важные вариантности:

- `&'a T` вариантна над `'a` и `T` (по аналогии, `*const T` ведёт себя также)

- `&'a mut T` вариантна над `'a`, но инвариантна над `T`
- `Fn(T) -> U` инвариантна над `T`, но вариантна над `U`
- `Box`, `Vec` и другие коллекции вариантны над типами их содержимого
- `UnsafeCell<T>`, `Cell<T>`, `RefCell<T>`, `Mutex<T>` и другие типы с внутренней изменяемостью инвариантны над `T` (по аналогии, `*mut T` ведёт себя также)

Чтобы понять, почему эти вариантности правильны и важны, рассмотрим несколько примеров.

Мы уже рассматривали, почему `&'a T` должна быть вариантна над `'a`, когда представляли выделение подтипов: желательно иметь возможность передавать что-то с большим временем жизни туда, где ожидается что-то с более коротким временем жизни.

По похожей причине `&'a T` должна быть вариантна над `T`. Разумно иметь возможность передавать `&&'static str` где ожидается `&&'a str`. Дополнительный уровень косвенности не влияет на передачу чего-то с большим временем жизни туда, где ожидается что-то с более коротким временем жизни.

Однако, эта логика не применима к `&mut`. Для того, чтобы понять почему `&mut` должна быть инвариантна над `T`, возьмем следующий код:

```
fn overwrite<T: Copy>(input: &mut T, new: &mut T) {
    *input = *new;
}

fn main() {
    let mut forever_str: &'static str = "hello";
    {
        let string = String::from("world");
        overwrite(&mut forever_str, &mut &*string);
    }
    // Упс, вывод освобожденной памяти
    println!("{}", forever_str);
}
```

Сигнатура `overwrite` абсолютна правильна: она берет изменяемую ссылку на два значения одного типа и переписывает одно в другое. Если `&mut T` была бы вариантна над `T`, то `&mut &'static str` была бы подтипом `&mut &'a str` из-за того, что `&'static str` является подтипом `&'a str`. Таким образом, время жизни `forever_str` успешно “усохло” бы до более короткого времени жизни `string` и `overwrite` бы успешно вызвалось. `string` впоследствии бы уничтожилось и `forever_str` указывало бы на освобожденную память, когда мы вызываем печать! Данный пример показывает, почему `&mut` должна быть инвариантна.

Это основная тема в вариантности против инвариантности: если вариантность позволяет хранить коротко живущее значение в долго живущей ячейке памяти, то надо использовать инвариантность.

В то же время `&'a mut T` вариантна над `'a`. Основное отличие между `'a` и `T` - `'a` является свойством самой ссылки, в то время как `T` - это что-то, что ссылка захватила. Если вы поменяете тип `T`, то источник будет все еще помнить оригинальный тип. Но если вы поменяете время жизни типа, никто кроме ссылки не помнит эту информацию, поэтому все в порядке. Говоря по другому: `&'a mut T` владеет `'a`, но только *заимствует* `T`.

Интересными случаями являются `Box` и `Vec`, потому что они вариантны, но вы можете хранить значения в них! Вот именно тут Rust становится особенно умным: они вариантны, потому что вы можете хранить значения в них только *посредством изменяемой ссылки*! Изменяемая ссылка делает весь тип инвариантным и поэтому не позволяет перевезти коротко-живущие значения контрабандой в них.

Вариантность позволяет `Box` и `Vec` ослаблять условия общей изменяемости. Поэтому вы можете передать `&Box<&'static str>` туда , где ожидается `&Box<&'a str>`.

Если идет передача *по значению*, то все гораздо менее очевидно. Оказывается, да, вы можете выделять подтипы при передаче по значению. Вот как это работает:

```
fn get_box<'a>(str: &'a str) -> Box<&'a str> {
    // строковые литералы являются `&'static str`
    Box::new("hello")
}
```

Ослабление при передаче по значению нормально проходит, потому что нет никого, кто бы “помнил” старое время жизни в Box. Вариантность `&mut` была проблемой, потому что всегда был настоящий владелец, который помнил оригинальный под-тип.

Инвариантность типов ячеек можно объяснить так: `&` похожа на `&mut` для ячеек, потому что вы можете менять значение в них по средствам `&`. Поэтому ячейки должны быть инвариантны, чтобы избежать незаконного ввоза времени жизни.

`Fn` - это самый тонкий случай, потому что у него смешанная вариантность. Чтобы понять почему `Fn(T) -> U` должна быть инвариантна над T, создадим следующую сигнатуру функции:

```
// 'a получается из родительской области видимости
fn foo(&'a str) -> usize;
```

Эта сигнатура утверждает, что может принять любую `&str`, которая живет по крайней мере `'a`. Теперь если бы эта сигнатура была вариантна над `&'a str`, это означало бы, что

```
fn foo(&'static str) -> usize;
```

можно подставить в это место, так как она является под-типом. Но у этой функции требования строже: она утверждает, что может принимать только `&'static str` и ничего кроме. Невозможно было бы дать ей `&'a str`, потому что она свободно могла бы предположить, что ей дали то, что должно жить вечно. Поэтому функции инвариантны над своими аргументами.

Чтобы понять, почему `Fn(T) -> U` должна быть вариантна над U, создадим следующую сигнатуру функции:

```
// 'a получается из родительской области видимости
fn foo(usize) -> &'a str;
```

Эта сигнатура утверждает, что вернет что-то, что будет жить дольше, чем `'a`. Поэтому абсолютно разумно подставить

```
fn foo(usize) -> &'static str;
```

на ее место. Поэтому функции вариантны над своим возвращаемым типом.

У `*const` такая же семантика как и у `&`, а, следовательно, и вариантность. С другой стороны `*mut` можно разыменовать в `&mut`, поэтому она инвариантна, также как типы ячеек.

Все это хорошо для типов из стандартной библиотеки, но как вариантность вычисляется для типов, которые определили *вы*? Структуры, говоря неформально, наследуют вариантность своих полей. Если у структуры `Foo` есть обобщенный аргумент `A`, который используется в поле `a`, то вариантность `Foo` над `A` будет совпадать с вариантностью `a`. Это усложняется, если `A` используется в нескольких полях.

- Если все использования A вариантны, то Foo вариантна над A
- Иначе, Foo инвариантна над A

```
use std::cell::Cell;

struct Foo<'a, 'b, A: 'a, B: 'b, C, D, E, F, G, H> {
    a: &'a A,     // вариантна над 'a и A
    b: &'b mut B, // вариантна над 'b и инвариантна над B
    c: *const C,  // вариантна над C
    d: *mut D,    // инвариантна над D
    e: Vec<E>,    // вариантна над E
    f: Cell<F>,   // инвариантна над F
    g: G,         // вариантна над G
    h1: H,        // была бы вариантна над H если бы не...
    h2: Cell<H>,  // была бы инвариантна над H, потому что инвариантность побеждает
}
```

Проверка сброса

Мы уже посмотрели, как времена жизни предоставляют нам простые правила, гарантирующие, что мы никогда не прочитаем висячий указатель. В то же время до этого мы имели дело только с отношениями вида, `'a` *переживает* `'b`, которые сформулированы включительно. То есть, когда мы говорим о `'a: 'b`, то в порядке вещей для `'a` жить *ровно* столько же сколько `'b`. На первый взгляд это кажется бессмысленным. Ничего же не может удалится одновременно с другим, правда? Вот почему утверждение `let` без синтаксического сахара выглядит следующим образом:

```
let x;
let y;
```

```
{
    let x;
    {
        let y;
```

```
    }
}
```

Каждое `let` создает свою область видимости, ясно утверждая, что одно удаляется после другого. А что, если мы сделаем так?

```
let (x, y) = (vec![], vec![]);
```

Живет ли одно значение дольше другого? Ответ, *нет*, ни одно значение не живет дольше другого. Конечно, или x, или y удалится одно после другого, но сам порядок не определен. Кортежи не одиноки в этом вопросе; начиная с Rust 1.0, составные структуры тоже не гарантируют порядок своего удаления.

Мы *могли бы* указать порядок для внутренних полей составных объектов, таких как кортежи или структуры. Но как насчет Vec или чего-то подобного? Vec приходится вручную удалять элементы посредством вызова кода из библиотеки. В общем случае все, что реализует Drop, имеет возможность повозиться со своими внутренностями, когда уже звучит похоронный звон. Поэтому компилятор не может достаточно точно определить порядок удаления полей типа, реализующего Drop.

Ну, а нам то какая разница? На самом деле это важно, потому что если система типов будет неаккуратна, она может случайно создать висячий указатель. Предположим, что у нас есть следующая программа:

```
struct Inspector<'a>(&'a u8);

fn main() {
    let (inspector, days);
    days = Box::new(1);
    inspector = Inspector(&days);
}
```

Программа абсолютно правильна и успешно компилируется. Тот факт, что `days` не живет *строго дольше* `inspector` не важен. Пока жив `inspector`, живы и days.

Но если мы добавим деструктор, программа больше не будет компилироваться!

```
struct Inspector<'a>(&'a u8);

impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("Я был всего в {} днях от пенсии!", self.0);
    }
}

fn main() {
    let (inspector, days);
    days = Box::new(1);
    inspector = Inspector(&days);
```

```
    // Предположим, что `days` удалилась первой.
    // Когда Inspector будет удаляться, он попытается прочитать освобожденную
    // память!
}
```

```
<anon>:12:28: 12:32 error: `days` does not live long enough
<anon>:12     inspector = Inspector(&days);
                                    ^~~~
<anon>:9:11: 15:2 note: reference must be valid for the block at 9:10...
<anon>:9 fn main() {
<anon>:10     let (inspector, days);
<anon>:11     days = Box::new(1);
<anon>:12     inspector = Inspector(&days);
<anon>:13     // Предположим, что `days` удалилась первой.
<anon>:14     // Когда Inspector будет удаляться, он попытается прочитать освобожденну
↳ ю память!
          ...
<anon>:10:27: 15:2 note: ...but borrowed value is only valid for the block suffix foll
↳ owing statement 0 at 10:26
<anon>:10     let (inspector, days);
<anon>:11     days = Box::new(1);
<anon>:12     inspector = Inspector(&days);
<anon>:13     // Предположим, что `days` удалилась первой.
<anon>:14     // Когда Inspector будет удаляться, он попытается прочитать освобожденну
↳ ю память!
<anon>:15 }
```

Добавление Drop позволяет Inspector выполнить на смертном одре произвольный код. Это означает, что он, теоретически, может заметить, что типы, которые должны были бы жить столько же, сколько и он, на самом деле уже уничтожены.

Самое интересное, что только обобщённые типы должны об этом волноваться. Для не обобщённых типов время жизни может быть только `'static`, а значит они могут жить *вечно*. Именно поэтому проблема называется проблемой *правильного удаления обобщённых типов*. Правильное удаление обобщённых типов гарантируется *анализатором удалений*. Во время написания этой главы некоторые детали, связанные с тем, как анализатор удалений проверяет типы, еще находились в подвешенном состоянии. Однако, Главным Правилом, на котором мы фокусируемся в этом разделе, будет следующее:

Чтобы обобщённый тип правильно реализовывал Drop, его обобщённые аргументы должно жить строго дольше него самого

Для того, чтобы подчиниться этому правилу, (обычно) необходимо удовлетворить требования анализатора заимствований; достаточно подчинятся этому правилу, хотя это и не обязательно. Таким образом, если ваш тип подчиняется этому правилу, то он точно правильно удалится.

Причиной, по которой не всегда обязательно подчиняться правилу выше, является то, что некоторые реализации Drop не обладают доступом к заимствованным данным, даже при том, что их тип обладает возможностью такого доступа.

Например, у этого варианта `Inspector` никогда не будет доступа к заимствованным данным:

```
struct Inspector<'a>(&'a u8, &'static str);

impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) знает, когда *не* проверять.", self.1);
    }
}

fn main() {
    let (inspector, days);
    days = Box::new(1);
    inspector = Inspector(&days, "gadget");
    // Предположим, что `days` удалилась первой.
    // Даже когда Inspector будет удаляться, у его деструктора не будет доступа
    // к заимствованным `days`.
}
```

У следующего варианта `Inspector` тоже никогда не будет доступа к заимствованным данным:

```
use std::fmt;

struct Inspector<T: fmt::Display>(T, &'static str);

impl<T: fmt::Display> Drop for Inspector<T> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) знает, когда *не* проверять.", self.1);
    }
}

fn main() {
    let (inspector, days): (Inspector<&u8>, Box<u8>);
    days = Box::new(1);
    inspector = Inspector(&days, "gadget");
    // Предположим, что `days` удалилась первой.
    // Даже когда Inspector будет удаляться, у его деструктора не будет доступа
    // к заимствованным `days`.
}
```

Однако *оба* этих варианта будут отвергнуты анализатором заимствований во время проверки `fn main`, говоря, что `days` не живет достаточно долго.

Анализатор заимствований не знает о внутренностях каждой реализации Drop для Inspector при проверке `main`. Проверяя `main`, он понимает, что тело деструктора Inspector может получить доступ к заимствованным данным.

По этой причине анализатор удалений требует, чтобы все заимствованные данные в значении типа жили строго дольше значения этого типа.

Аварийный люк

Четкие правила, управляющие проверкой удалений, возможно, в будущем будут менее строгими.

Текущая проверка нарочито консервативна и тривиальна; она требует, чтобы все заимствованные данные в значении типа жили строго дольше значения этого типа, что несомненно правильно.

Будущие версии языка, возможно, сделают проверки более точными, чтобы уменьшить количество случаев, когда правильный код отвергается как небезопасный. Это помогло бы решить случаи, как с двумя Inspector выше, которые знают, что не надо проверять их во время удаления.

В настоящее время существует нестабильный атрибут, который можно использовать, чтобы указать (небезопасно), что деструктор обобщенного типа *гарантированно* не будет иметь доступ к просроченным данным, даже при том, что у такого типа есть возможность сделать это.

Аттрибут называется `unsafe_destructor_blind_to_params`. Чтобы применить его к Inspector из примера выше, напишем:

```
struct Inspector<'a>(&'a u8, &'static str);

impl<'a> Drop for Inspector<'a> {
    #[unsafe_destructor_blind_to_params]
    fn drop(&mut self) {
        println!("Inspector(_, {})  знает, когда *не* стоит проверять.", self.1);
    }
}
```

Внутри аттрибута есть слово `unsafe`, потому что компилятор не проверяет, что осуществляется доступ к потенциально просроченным данным (здесь, например, к `self.0`).

Иногда очевидно, что такой доступ не происходит, как в примере выше. Однако, когда мы имеем дело с параметром обобщенного типа, такой доступ может произойти не напрямую. Примерами такого косвенного доступа являются:

- выполнение обратного вызова,
- вызов метода типажа.

(Будущие изменения в языке, такие как специализация impl, могут добавить другие возможности такого косвенного доступа.)

Вот пример с выполнением обратного вызова:

```
struct Inspector<T>(T, &'static str, Box<for <'r> fn(&'r T) -> String>);

impl<T> Drop for Inspector<T> {
    fn drop(&mut self) {
        // Вызов `self.2` может получить доступ к заимствованию, например, если `T` эт
↳ о `&'a _`.
```

```
        println!("Inspector({}, {}) нечаянно проверяет просроченные данные.",
                 (self.2)(&self.0), self.1);
    }
}
```

Вот пример с вызовом метода типажа:

```
use std::fmt;

struct Inspector<T: fmt::Display>(T, &'static str);

impl<T: fmt::Display> Drop for Inspector<T> {
    fn drop(&mut self) {
        // Ниже есть невидимый вызов `<T as Display>::fmt`, который
        // может получить доступ к заимствованию, например, если `T` это `&'a _`
        println!("Inspector({}, {}) нечаянно проверяет просроченные данные.",
                 self.0, self.1);
    }
}
```

И конечно, все эти получения доступа могут быть дополнительно скрыты внутри других методов, вызываемых деструктором, а не написанных непосредственно внутри него.

Во всех случаях выше, где доступ к `&'a u8` получается в деструкторе благодаря добавлению аттрибута `#[unsafe_destructor_blind_to_params]`, становится возможным неправильно использовать тип, это не поймает анализатор заимствований, и возникнет хаос. Лучше избежать добавление этого аттрибута.

Это все о проверке удалений?

Мы выяснили, что при написании небезопасного кода, нам обычно не надо волноваться о том, как правильно пройти проверки удалений. Но есть один особый случай, в котором надо волноваться об этом. Рассмотрим его в следующем разделе.

Призрачные данные

При работе с небезопасным кодом мы часто можем попасть в ситуацию, когда типы или времена жизни логически ассоциируются со структурой, но не являются на самом деле частью конкретного поля. Особенно часто это происходит с временами жизни. Например, `Iter` для `&'a [T]` описывается (примерно) так:

```
struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
}
```

Но, в связи с тем, что `'a` не используется внутри тела структуры, оно *безгранично*. Из-за проблем, которые исторически возникли, безграничные времена жизни и типы *запрещено* использовать в описании структур. Поэтому необходимо как-то перестроить эти поля внутри тела. Чтобы проверки вариантности и удаления выполнились корректно, это важно сделать правильно.

Используем `PhantomData`, который является специальным маркерным типом. `PhantomData` не занимает места в памяти, а симулирует поле необходимого типа для целей статического анализа. Считается, что так код менее подвержен ошибкам, чем при явном указании необходимой вариантности системе типов, а заодно предоставляется другая полезная информация, необходимая для проверки удаления.

Iter по логике содержит кучу `&'a T`, поэтому именно ее мы и будем симулировать PhantomData:

```
use std::marker;

struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
    _marker: marker::PhantomData<&'a T>,
}
```

И все. Время жизни будет ограничено, и ваш итератор будет вариантен над `'a` и `T`. Все Просто Работает.

Важным примером также является Vec, который описывается (примерно) так:

```
struct Vec<T> {
    data: *const T, // *const для вариантности!
    len: usize,
    cap: usize,
}
```

В отличие от предыдущего примера *кажется*, что все именно так, как мы хотим. Один обобщенный аргумент в Vec проявляется по крайней мере в одном поле. Идем дальше!

Не-а.

Анализатор удалений великодушно определит, что Vec не владеет ни одним значением типа T. Это в свою очередь заставит его сделать вывод, что ему не надо волноваться при выполнении проверки правильности удаления любых T в деструкторе Vec. Что в свою очередь позволит людям создавать дыры в системе типов с помощью деструктора Vec.

Для того, чтобы объяснить анализатору удалений, что мы *на самом деле* владеем значениями типа T, и, следовательно, можем удалять некоторые T во время удаления *всего Vec*, мы должны добавить дополнительное поле PhantomData, явно утверждая:

```
use std::marker;

struct Vec<T> {
    data: *const T, // *const для ковариантности!
    len: usize,
```

```
    cap: usize,
    _marker: marker::PhantomData<T>,
}
```

Сырые указатели, владеющие пространством в памяти - это настолько повсеместная картина, что в стандартной библиотеке создали утилиту для этого, называемую `Unique<T>`, которая:

- оборачивает `*const T` для вариантности
- включает в себя `PhantomData<T>`,
- автоматически выводит Send/Sync, как будто T их реализует,
- помечает указатель NonZero для оптимизации нулевого указателя.

Деление заимствований

Свойство эксклюзивного изменения у изменяемых ссылок может сильно ограничивать работу с составными структурами. Анализатор заимствований понимает какие-то базовые вещи, но может поломаться довольно просто. Он достаточно понимает структуры, чтобы знать, что можно одновременно заимствовать непересекающиеся поля структуры. Вот так работать будет:

```
struct Foo {
    a: i32,
    b: i32,
    c: i32,
}

let mut x = Foo {a: 0, b: 0, c: 0};
let a = &mut x.a;
let b = &mut x.b;
let c = &x.c;
*b += 1;
let c2 = &x.c;
*a += 10;
println!("{} {} {} {}", a, b, c, c2);
```

Однако анализатор заимствований вообще не понимает массивы или срезы, поэтому так работать не будет:

```
let mut x = [1, 2, 3];
let a = &mut x[0];
let b = &mut x[1];
println!("{} {}", a, b);
```

```
<anon>:4:14: 4:18 error: cannot borrow `x[..]` as mutable more than once at a time
<anon>:4 let b = &mut x[1];
                      ^~~~
<anon>:3:14: 3:18 note: previous borrow of `x[..]` occurs here; the mutable borrow pre
```

```
↳ vents subsequent moves, borrows, or modification of `x[..]` until the borrow ends
<anon>:3 let a = &mut x[0];
                     ^~~~
<anon>:6:2: 6:2 note: previous borrow ends here
<anon>:1 fn main() {
<anon>:2 let mut x = [1, 2, 3];
<anon>:3 let a = &mut x[0];
<anon>:4 let b = &mut x[1];
<anon>:5 println!("{} {}", a, b);
<anon>:6 }
         ^
error: aborting due to 2 previous errors
```

Хоть и кажется, что анализатор заимствований мог бы понимать этот простой случай, очевидно, безнадежным кажется, что он мог бы понимать различия в общих типах контейнеров, таких как деревья, особенно, если разные ключи *скрывают* одно и то же значения.

Для того чтобы “объяснить” анализатору заимствований, что мы делаем правильно, нам нужно перейти к небезопасному коду. Например, изменяемые срезы подвергаются действию функции `split_at_mut`, которая берет срез и возвращает два изменяемых среза. В один попадает то, что находится слева от индекса, в другой - справа. Интуитивно мы понимаем, что это безопасно, потому что срезы не пересекаются и, следовательно, не совпадают ссылки на них. Однако для реализации потребуется щепотка небезопасного кода:

```
fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]) {
    let len = self.len();
    let ptr = self.as_mut_ptr();
    assert!(mid <= len);
    unsafe {
        (from_raw_parts_mut(ptr, mid),
         from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

Все здесь тонко. Поэтому, чтобы избежать создания двух `&mut` к одному значению, мы явно конструируем два абсолютно новых среза через сырые указатели.

Еще сложнее работают итераторы, перебирающие изменяемые ссылки. Типаж итератора описывается так:

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

Глядя на определение, видим, что у Self::Item *нет* связи с `self`. Это означает, что мы можем вызвать `next` несколько раз подряд, и получим все результаты *одновременно*. Для итераторов

по-значению, у которых именно такая семантика, все по-другому. Для общих ссылок все нормально, потому что они допускают сколь угодно много ссылок на одно и то же (хотя итератор и общий объект должны быть разными объектами).

Но с изменяемыми ссылками все превращается в кашу. На первый взгляд кажется, что они полностью несовместимы с этим API, из-за того, что создадут много изменяемых ссылок на один и тот же объект!

Однако, *на самом деле* все работает, именно, потому что итераторы - это одноразовые объекты. Все, по чему пройдется IterMut, будет пройдено только один раз, поэтому на самом деле мы никогда не создадим много изменяемых ссылок на один кусок данных.

Возможно, это удивительно, но изменяемым итератором не нужно использовать небезопасный код для реализации разных типов!

Например, вот пример однонаправленного списка:

```
type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

pub struct LinkedList<T> {
    head: Link<T>,
}

pub struct IterMut<'a, T: 'a>(Option<&'a mut Node<T>>);

impl<T> LinkedList<T> {
    fn iter_mut(&mut self) -> IterMut<T> {
        IterMut(self.head.as_mut().map(|node| &mut **node))
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        self.0.take().map(|node| {
            self.0 = node.next.as_mut().map(|node| &mut **node);
            &mut node.elem
        })
    }
}
```

Вот изменяемый срез:

```rust
use std::mem;

pub struct IterMut<'a, T: 'a>(&'a mut[T]);

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        let slice = mem::replace(&mut self.0, &mut []);
        if slice.is_empty() { return None; }

        let (l, r) = slice.split_at_mut(1);
        self.0 = r;
        l.get_mut(0)
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        let slice = mem::replace(&mut self.0, &mut []);
        if slice.is_empty() { return None; }

        let new_len = slice.len() - 1;
        let (l, r) = slice.split_at_mut(new_len);
        self.0 = l;
        r.get_mut(0)
    }
}
```

Двоичное дерево:

```rust
use std::collections::VecDeque;

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    left: Link<T>,
    right: Link<T>,
}

pub struct Tree<T> {
    root: Link<T>,
}

struct NodeIterMut<'a, T: 'a> {
    elem: Option<&'a mut T>,
    left: Option<&'a mut Node<T>>,
    right: Option<&'a mut Node<T>>,
```

```
}

enum State<'a, T: 'a> {
    Elem(&'a mut T),
    Node(&'a mut Node<T>),
}

pub struct IterMut<'a, T: 'a>(VecDeque<NodeIterMut<'a, T>>);

impl<T> Tree<T> {
    pub fn iter_mut(&mut self) -> IterMut<T> {
        let mut deque = VecDeque::new();
        self.root.as_mut().map(|root| deque.push_front(root.iter_mut()));
        IterMut(deque)
    }
}

impl<T> Node<T> {
    pub fn iter_mut(&mut self) -> NodeIterMut<T> {
        NodeIterMut {
            elem: Some(&mut self.elem),
            left: self.left.as_mut().map(|node| &mut **node),
            right: self.right.as_mut().map(|node| &mut **node),
        }
    }
}


impl<'a, T> Iterator for NodeIterMut<'a, T> {
    type Item = State<'a, T>;

    fn next(&mut self) -> Option<Self::Item> {
        match self.left.take() {
            Some(node) => Some(State::Node(node)),
            None => match self.elem.take() {
                Some(elem) => Some(State::Elem(elem)),
                None => match self.right.take() {
                    Some(node) => Some(State::Node(node)),
                    None => None,
                }
            }
        }
    }
}

impl<'a, T> DoubleEndedIterator for NodeIterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        match self.right.take() {
```

```
            Some(node) => Some(State::Node(node)),
            None => match self.elem.take() {
                Some(elem) => Some(State::Elem(elem)),
                None => match self.left.take() {
                    Some(node) => Some(State::Node(node)),
                    None => None,
                }
            }
        }
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
    fn next(&mut self) -> Option<Self::Item> {
        loop {
            match self.0.front_mut().and_then(|node_it| node_it.next()) {
                Some(State::Elem(elem)) => return Some(elem),
                Some(State::Node(node)) => self.0.push_front(node.iter_mut()),
                None => if let None = self.0.pop_front() { return None },
            }
        }
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        loop {
            match self.0.back_mut().and_then(|node_it| node_it.next_back()) {
                Some(State::Elem(elem)) => return Some(elem),
                Some(State::Node(node)) => self.0.push_back(node.iter_mut()),
                None => if let None = self.0.pop_back() { return None },
            }
        }
    }
}
```

Все это абсолютно безопасно и работает на стабильном Rust! На самом деле, это вытекает из случая с простой структурой, который мы видели выше: Rust понимает, что вы можете делить изменяемую ссылку на ссылки на под-поля. Мы можем возвращать постоянно употребляемую ссылку по средствам Options (или в случае срезов, заменить на пустой срез).

5

Преобразование типов

В конечном счете, все является кучей бит, расположенных где-то, а система типов нужна, чтобы понять, что эти биты правильны. Существует две основных проблемы при типизации битов: необходимость интерпретировать эти биты в качестве другого типа и необходимость менять биты, чтобы получить эквивалентное значение другого типа. Из-за того, что Rust поддерживает кодировку важных свойств в системе типов, эти проблемы невероятно распространены. В связи с этим Rust дает вам несколько способов их решения.

Сначала посмотрим на то, как Безопасный Rust может по-разному интерпретировать значения. Самый простой способ - разбить значение на части и потом из них собрать новое значение, например:

```
struct Foo {
    x: u32,
    y: u16,
}

struct Bar {
    a: u32,
    b: u16,
}

fn reinterpret(foo: Foo) -> Bar {
    let Foo { x, y } = foo;
    Bar { a: x, b: y }
}
```

Но это, в лучшем случае, утомительно. Для обычных преобразований Rust предоставляет более удобные альтернативы.

Неявные приведения типов

Типы могут неявно приводиться к другим типам в определенных контекстах. Эти изменения в основном - просто *ослабление* типов, сильно сфокусированное на указателях и временах жизни. Их главная задача - заставить Rust “просто работать” в большинстве случаев, они в основном безвредны.

Неявное приведение разрешено между следующими типами:

- Транзитивность: `T_1` к `T_3`, где `T_1` неявно приводится к `T_2` и `T_2` неявно приводится к `T_3`
- Ослабление Указателей:
  - `&mut T` к `&T`
  - `*mut T` к `*const T`
  - `&T` к `*const T`
  - `&mut T` к `*mut T`
- Неявное приведение к безразмерному типу (ТДР): `T` к `U` если `T` реализует `CoerceUnsized<U>`

`CoerceUnsized<Pointer<U>> for Pointer<T> where T: Unsize<U>` реализовано для всех типов указателей (включая умные указатели, такие как Box и Rc). Неявное приведение к безразмерному типу реализуется только автоматически и разрешает следующие преобразования:

- `[T; n] => [T]`
- `T => Trait`, где `T: Trait`
- `Foo<..., T, ...> => Foo<..., U, ...>`, где:
  - `T: Unsize<U>`
  - `Foo` - это структура
  - Только у последнего поля `Foo` тип `T`
  - `T` не является частью типа любых других полей

Неявное приведение происходит в *месте неявного приведения*. Любая явно типизированная область памяти выполняет неявное приведение к ее типу. Неявное приведение производится не будет, если нужно производить вывод типов. Все места неявного приведения `e` к типу `U` это:

- Утверждения let, статические переменные и константы: `let x: U = e`
- Аргументы функций: `takes_a_U(e)`
- Любые возвращаемые выражения: `fn foo() -> U { e }`
- Литералы структур: `Foo { some_u: e }`
- Литералы массивов: `let x: [U; 10] = [e, ..]`
- Литералы кортежей: `let x: (U, ..) = (e, ..)`
- Последние выражения в блоке: `let x: U = { ..; e }`

Заметьте, что мы не выполняем неявное приведение при поиске совпадения типажей (кроме получателей, смотри ниже). Если есть impl для типа `U`, а `T` приводится к `U`, это не означает, что эта реализация подойдет для `T`. Например, следующее выражение не пройдет проверку типов, даже при том, что приводить `t` к `&T` можно и есть impl для `&T`:

```
trait Trait {}

fn foo<X: Trait>(t: X) {}

impl<'a> Trait for &'a i32 {}


fn main() {
    let t: &mut i32 = &mut 0;
    foo(t);
}
```

```
<anon>:10:5: 10:8 error: the trait `Trait` is not implemented for the type `&mut i32`
↳ [E0277]
<anon>:10     foo(t);
              ^~~
```

Оператор Точка

Оператор точка выполняет много волшебных действий для преобразований типов. Он выполняет автоматическое создание ссылок, автоматическое разыменование ссылок и неявные приведения до тех пор, пока типы не совпадут.

TODO: своровать информацию из http://stackoverflow.com/questions/28519997/what-are-rusts-exact-auto-dereferencing-rules/28552082#28552082

Явные приведения типов

Явное приведение - это надстройка над неявным приведением: каждое неявное приведение можно вызвать посредством явного приведения. Некоторые преобразования типов требуют явного приведения. В то время как неявные приведения распространены и в основном безвредны, эти “настоящие явные приведения типов” редки и потенциально опасны. Поэтому явные приведения должны явно вызываться с помощью ключевого слова `as`: `expr as Type`.

Настоящие явные приведения обычно крутятся вокруг сырых указателей и примитивных числовых типов. Даже при том, что они опасны, эти приведения не могут сломаться во время выполнения. Если явное приведение вызовет какую-то проблему, это невозможно будет обнаружить. Явное приведение просто выполнится. Несмотря на это, явные приведения должны быть правильны на уровне типов, или они будут предотвращены статически. Например, `7u8 as bool` не компилируется.

При этом явные приведения не `unsafe`, потому что они вообще не могут нарушить безопасность памяти *сами по себе*. Например, преобразование целого в сырой указатель может легко привести к ужасным вещам. Но само действие по созданию указателя безопасно, потому что на самом деле использование сырого указателя уже помечено `unsafe`.

Вот полный список всех правильных явных приведений. Для краткости используем *, чтобы обозначить `*const` или `*mut`, и `integer` - для любого целого примитива:

- `*T as *U`, где `T, U: Sized`
- `*T as *U` TODO: объяснить ситуацию с безразмерными типами
- `*T as integer`
- `integer as *T`
- `number as number`
- `C-like-enum as integer`
- `bool as integer`
- `char as integer`
- `u8 as char`
- `&[T; n] as *const T`
- `fn as *T`, где `T: Sized`
- `fn as integer`

Заметьте, что длины не корректируются при явном приведении сырых срезов - `*const [u16] as *const [u8]` создаст срез, который состоит из половины памяти оригинального среза.

Явное приведение не транзитивно, это означает, что, даже если `e as U1 as U2` правильное выражение, `e as U2` не обязано быть таким же.

В случае с числами нужно пояснить несколько моментов:

- явное приведение между двумя целыми одного размера (e.g. i32 -> u32) это пустая операция
- явное приведение большего целого к меньшему (e.g. u32 -> u8) обрежет большее
- явное приведение меньшего целого к большего (e.g. u8 -> u32) будет
  - дополнять нулями если источник беззнаковый
  - дополнять знаком если источник знаковый
- явное приведение дробного к целому осуществляется округлением дробного к нулю
  - **[ВНИМАНИЕ: на данный момент может вызвать Неопределенное Поведение, если округленное значение не может быть представлено целевым целым типом][float-int]**. Включает в себя Inf и NaN. Это ошибка и она будет исправлена.
- явное приведение целого к дробному осуществляется созданием дробного числа, округленного при необходимости (стратегия округления не указана)
- явное приведение f32 к f64 выполняется отлично и без потерь
- явное приведение f64 к f32 создаст ближайшее возможное значение (стратегия округления не указана)
  - **ВНИМАНИЕ: на данный момент может вызвать Неопределенное Поведение, если значение конечно, но больше или меньше самого маленького или самого большого конечного числа, представляемым f32**[1]. Это ошибка и она будет исправлена.

Трансмутации

Уберись с нашей дороги, система типов! Мы будем интерпретировать эти биты по- своему или умрем пытаясь! Хотя эта книга и про создание небезопасных вещей, я действительно не могу

[1] https://github.com/rust-lang/rust/issues/15536

не подчеркнуть, что вы должны глубоко задуматься над поиском Другого способа, кроме того, что рассматривается в этом разделе. Это действительно, по-настоящему, самая ужасно небезопасная вещь, которую вы можете сделать в Rust. Это как если бы железнодорожное полотно охраняла зубная нить.

`mem::transmute<T, U>` берет значение типа `T` и интерпретирует его, как тип `U`. Единственное ограничение - размер `T` и `U` должен совпадать. Варианты, которые вызывают Неопределенное Поведение этим, сводят с ума.

- Первое и самое главное, создание экземпляра *любого* типа с неправильным состоянием вызовет огромный хаос, который невозможно будет предсказать.
- Трансмутация имеет перегруженный тип возврата. Если вы не укажете тип возврата она создаст какой-то неизвестный тип, удовлетворяющий выводу типов.
- Создание примитива с неправильным значением - это неопределённое поведение (НП)
- Трансмутации между non-repr(C) типами - это НП
- Трансмутации & в &mut - это НП
  - Трансмутации & в &mut - это *всегда* НП!
  - Нет, ты не можешь так делать!
  - Нет, ты не особенный!
- Трансмутации в ссылку без явного указания времени жизни создает [unbounded-lifetimes]

`mem::transmute_copy<T, U>` каким-то образом стал *даже более дико* небезопасным чем это. Он копирует `size_of<U>` байтов из `&T` и интерпретирует их как `U`. Проверка длины, которая была у `mem::transmute` пропала (потому что копировать префикс может оказаться допустимым), хотя если `U` длиннее, чем `T` - это Неопределенное Поведение.

Вы можете получить большую часть этой функциональности, используя явное приведение указателей.

6

Работа с неинициализированной памятью

Вся используемая во время выполнения память в программах на Rust начинает свою жизнь *неинициализированной*. В таком состоянии значения в памяти представляют собой кучку неопределённых бит, о которых даже неизвестно, смогут ли они правильно представить состояние значения того типа, который разместится в этой памяти. Попытка интерпретировать эту память в качестве значения *любого* типа вызовет Неопределенное Поведение. Не Делайте Так.

Rust предоставляет механизмы для работы с неинициализированной памятью в проверяемом (безопасном) и непроверяемом (небезопасном) ключе.

Проверяемая неинициализированная память

Как и в Си, в Rust все переменные на стеке не инициализированы до тех пор пока им явно не присвоено значение. В отличие от Си Rust статически ограничивает их чтение, пока вы не сделаете это:

```
fn main() {
    let x: i32;
    println!("{}", x);
}
```

```
src/main.rs:3:20: 3:21 error: use of possibly uninitialized variable: `x`
src/main.rs:3     println!("{}", x);
                                 ^
```

Все основывается на базовом анализе веток: каждая ветка должна присвоить x значение до его первого использования. Интересно, что Rust не требует, чтобы переменная была изменяемой, чтобы выполнить отложенную инициализацию, если каждая ветка присваивает значение лишь однажды. В то же время такой анализ не использует анализ констант или что-либо подобное. Поэтому это компилируется:

```
fn main() {
    let x: i32;

    if true {
        x = 1;
    } else {
        x = 2;
    }

    println!("{}", x);
}
```

а это нет:

```
fn main() {
    let x: i32;
    if true {
        x = 1;
    }
    println!("{}", x);
}
```

```
src/main.rs:6:17: 6:18 error: use of possibly uninitialized variable: `x`
src/main.rs:6   println!("{}", x);
```

хотя это тоже компилируется:

```
fn main() {
    let x: i32;
    if true {
        x = 1;
        println!("{}", x);
    }
    // Не обращайте внимания на то, что есть еще ветки, в которых x не
    // инициализирована, ведь мы не используем в этих ветках ее значение
}
```

Несмотря на то, что в анализе не участвуют настоящие значения, компилятор довольно хорошо понимает зависимости и поток выполнения. Например, это работает:

```
let x: i32;

loop {
    // Rust не понимает, что эта ветка безоговорочно выполнится,
    // потому что это зависит от настоящих значений.
    if true {
        // Но он понимает, что попадет сюда лишь один раз, потому что
```

```
        // мы однозначно выходим отсюда. Поэтому `x` не надо помечать
        // изменяемым.
        x = 0;
        break;
    }
}
// Он также понимает, что невозможно добраться сюда, не достигнув break.
// И, следовательно, `x` должна быть уже инициализирована здесь!
println!("{}", x);
```

Если переменная перестает владеть значением, эта переменная становится логически неинициализированной, если только тип значения не реализует Copy. Это означает:

```
fn main() {
    let x = 0;
    let y = Box::new(0);
    let z1 = x; // x остается в силе из-за того, что i32 реализует Copy
    let z2 = y; // y теперь логически не инициализирована, потому что Box не
                // реализует Copy
}
```

Но переприсваивание y в этом примере *потребует*, чтобы y была помечена изменяемой, дабы программа на Безопасном Rust могла заметить, что значение y поменялось:

```
fn main() {
    let mut y = Box::new(0);
    let z = y; // y теперь логически не инициализирована, потому что Box не
               // реализует Copy
    y = Box::new(1); // переинициализация y
}
```

Иначе y станет абсолютно новой переменной.

Флаги удаления

Пример в предыдущем разделе показал интересную проблему для Rust. Мы увидели, что можно использовать условную инициализацию, деинициализацию и переинициализацию участков памяти абсолютно безопасно. Для типов, реализующих Copy, это не особо важно, потому что они являются просто случайной кучкой бит. Но для типов с деструкторами - это совсем другая история: Rust нужно знать, вызывать ли деструктор, когда переменная присваивается или выходит из области видимости. Откуда это можно узнать в случае условной инициализации?

Заметьте, что не все присваивания должны волноваться об этой проблеме. В частности, присваивание через разыменование безусловно вызывает деструктор, а присваивание в `let` безусловно не вызывает его:

```
let mut x = Box::new(0); // let создает новую переменную, поэтому деструктор не
                         // нужно вызывать
let y = &mut x;
*y = Box::new(1); // Deref подразумевает, что референт инициализирован, поэтому
                  // деструктор вызывается всегда
```

Это проблема возникает только при перезаписывании ранее инициализированных переменных или их под-полей.

Выясняется, что Rust на самом деле следит за тем, нужно ли *во время исполнения* вызывать деструктор или нет. Когда переменная становится инициализированной или неинициализированной, ее *флаг удаления* переключается. Когда необходимо удалить переменную, по этому флагу оценивается нужно ли вызывать у нее деструктор.

Конечно, часто можно статически определить в любом месте программы состояние инициализации у значения. В этом случае компилятор, теоретически, может создать более эффективный код! Например, прямолинейный код обладает такой *семантикой статических удалений*:

```
let mut x = Box::new(0); // x была не инициализирована; просто перезаписать.
let mut y = x;           // y была не инициализирована; просто перезаписать и
                         // сделать x неинициализированной.

x = Box::new(0);         // x была не инициализирована; просто перезаписать.

y = x;                   // y была инициализирована; Вызвать деструктор y,
                         // перезаписать ее, и сделать x неинициализированной!

                         // y выходит из области видимости; y была
                         // инициализирована; Вызвать деструктор y!

                         // x выходит из области видимости; x была не
                         // инициализирована; ничего не делать.
```

Код с условным ветвлением, где внутри веток наблюдается похожее поведение, обладает такой же семантикой статических удалений:

```
let mut x = Box::new(0);    // x была не инициализирована; просто перезаписать.
if condition {
    drop(x)                 // y x забирается владение ; сделать x
                            // неинициализированной.
} else {
    println!("{}", x);
    drop(x)                 // y x забирается владение ; сделать x
                            // неинициализированной.
}
x = Box::new(0);            // x была не инициализирована; просто перезаписать.
                            // x выходит из области видимости; x была
                            // инициализирована; Вызвать деструктор x!
```

Однако такому коду *требуется* информация из времени исполнения для правильного удаления:

```
let x;
if condition {
    x = Box::new(0);        // x была не инициализирована; просто перезаписать.
    println!("{}", x);
}
                            // x выходит из области видимости; x возможно была
                            // не инициализирована; проверить флаг!
```

Конечно, в данном случае легко можно получить семантику статических удалений:

```
if condition {
    let x = Box::new(0);
    println!("{}", x);
}
```

Что касается Rust 1.0, флаги удаления на самом деле не-так-уж-секретно спрятаны в невидимом поле любого типа, реализующего Drop. Rust устанавливает флаг, переписывая старое значение новым набором бит. Очевидно, что это Не Самый Быстрый способ, вызывающий несколько проблем при оптимизации. Он пришел еще с того времени, когда вы могли выполнять гораздо более сложную условную инициализацию.

Сейчас идет работа по переносу флагов в кадр стека, которому они по-настоящему и принадлежат. К сожалению, эта работа потребует немало времени, потому что требуется внести довольно существенные изменения в компилятор.

Независимо от этого, программы на Rust не должны волноваться о корректности неинициализированных значений в стеке. Хотя они могут волноваться о производительности. К счастью, Rust позволяет легко контролировать её! Неинициализированные значения существуют, и вы никогда не попадете в беду при работе с ними в Безопасном Rust.

Непроверяемая неинициализированная память

Особо интересным исключением из этого правила является работа с массивами. Безопасный Rust не разрешит вам частично инициализировать массив. При инициализации массива вы должны или установить всем одно и то же значение `let x = [val; N]`, или установить каждому члену отдельно `let x = [val1, val2, val3]`. К сожалению, это довольно негибко, особенно если вам нужно инициализировать массив более инкрементальным или динамичным способом.

Небезопасный Rust дает вам мощный инструмент для решения этой проблемы: `mem::uninitialized`. Эта функция делает вид, что возвращает значение, когда в действительности она вообще ничего не делает. Используя ее, мы можем убедить Rust в том, что переменная у нас инициализирована, и это позволяет делать хитрые вещи с условной или инкрементальной инициализацией.

К сожалению, это может привести к возникновению кучи проблем. Присваивание имеет разный смысл для Rust, если он считает, что переменная инициализирована и наоборот. Если считается, что переменная не инициализирована, то Rust просто семантически сделает memcopy новых бит на место неинициализированных и ничего больше. Однако, если Rust считает, что значение инициализировано, он попытается выполнить `Drop` старого значения! Из-за того, что

мы обманули Rust в части того, что значение инициализировано, мы больше не можем безопасно использовать обычное присваивание.

Эта же проблема возникает с системным распределителем памяти, который возвращает сырой указатель на неинициализированную память.

Для решения этого мы должны использовать модуль `ptr`. В частности, он предоставляет три функции, которые позволяют присваивать байты определенному месту в памяти, не удаляя старое значение: `write`, `copy` и `copy_nonoverlapping`.

- `ptr::write(ptr, val)` берет `val` и заносит его по адресу `ptr`.
- `ptr::copy(src, dest, count)` копирует из src в dest столько памяти, сколько занимают `count` экземпляров типа T. (это эквивалент memmove - заметьте, что порядок аргументов перевернут!)
- `ptr::copy_nonoverlapping(src, dest, count)` делает то же, что и `copy`, но немного быстрее, основываясь на предположении, что две области памяти не пересекаются. (это эквивалент memcpy – заметьте, что порядок аргументов перевернут!)

Надеюсь не надо говорить, что эти функции в случае неправильного использования приведут к серьезному хаосу или прямиком к Неопределенному Поведению. *Единственным* требованием этих функций является то, что используемые области должны находится в памяти. В то же время невозможно перечислить все случаи, когда запись произвольных бит в произвольное место в памяти может все сломать!

Объединяя все, получаем:

```
use std::mem;
use std::ptr;

// длина массива жестко закодирована, но это легко поменять. Это означает, что мы
// не можем использовать синтаксис [a, b, c] для инициализации массива!
const SIZE: usize = 10;

let mut x: [Box<u32>; SIZE];

unsafe {
    // убеждаем Rust, что x Абсолютно Инициализирована
    x = mem::uninitialized();
    for i in 0..SIZE {
        // очень аккуратно переписываем каждый индекс, не читая его
        // Внимание: безопасность исключений не важна; Box не может вызвать панику
        ptr::write(&mut x[i], Box::new(i as u32));
    }
}

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

Стоит отметить, что вам не нужно волноваться о махинациях в стиле `ptr::write` с типами, которые не реализуют `Drop` сами и не содержат типы, реализующие его, потому что Rust знает, что для них не надо вызывать деструктор. Аналогично, можно выполнять присваивания полям

частично инициализированных структур напрямую, если эти поля не содержат типы, реализующие `Drop`.

В то же время, работая с неинициализированной памятью, вам надо постоянно следить, чтобы Rust не попытался вызвать деструктор значений, которые вы создали, до их полной инициализации. Каждый путь выполнения, содержащий область видимости этой переменной, должен инициализировать ее до своего конца, если у нее есть деструктор. *Это включает в себя поведение кода в случае паники*[1].

Вот и все по работе с неинициализированной памятью! Обычно неинициализированная память нигде не обрабатывается, поэтому вам следует быть *очень* осторожным, если вы собираетесь передать ее куда-то.

[1] unwinding.md

7

Управление ресурсами на основе владения

Управление ресурсами на основе владения (англ. OBRM, Ownership Based Resource Management) - RAII: Resource Acquisition Is Initialization - это то, с чем вы будете много сталкиваться в Rust. Особенно если будете использовать стандартную библиотеку.

Грубо говоря, правило следующее: для получения ресурса, вы создаете объект, управляющий им. Для освобождения ресурса просто уничтожаете объект, а он сам чистит ресурсы за вас. Самым частым “ресурсом”, который управляется этим правилом, является просто *память*. `Box`, `Rc` и почти все в `std::collections` - это удобство, созданное для правильного управления памятью. Это особенно важно в Rust, потому что у нас нет всепроникающего GC, на который можно было бы возложить управление памятью. Важно понимать: Rust - это контроль. В то же время мы не ограничены только памятью. Почти каждый ресурс системы - поток, файл или сокет - проходит через этот API.

Конструкторы

Есть только один способ создать экземпляр пользовательского типа: дать ему имя, и инициализировать сразу все его поля:

```
struct Foo {
    a: u8,
    b: u32,
    c: bool,
}

enum Bar {
    X(u32),
    Y(bool),
}
```

```
struct Unit;

let foo = Foo { a: 0, b: 1, c: false };
let bar = Bar::X(0);
let empty = Unit;
```

Вот и все. Любой другой способ создания экземпляра типа - это только вызов совершенно обычной функции, которая делает какие-то вещи и в конечном итоге вызывает Один Единственный Конструктор.

В отличие от C++, Rust не поставляется с убийственным набором конструкторов. В нем нет Copy, Default, Assignment, Move или еще каких-либо конструкторов. Этому множество причин, но основной является философия Rust - *быть явным*.

Конструкторы перемещения бессмысленны в Rust, потому что мы не позволяем типам “заботиться” о своем расположении в памяти. Каждый тип должен быть готов быть скопированным в другое место в памяти. Это означает, что чистые на-стеке-но-все- еще-перемещаемые встроенные связные списки (intrusive linked lists) просто невозможно встретить в Rust (безопасном).

Конструкторы присваивания и копирования не существуют, потому что семантика перемещения - это единственная семантика в Rust. В большинстве случаев `x = y` просто перемещает биты y в переменную x. Rust дает две возможности для предоставления copy-ориентированной семантики C++: `Copy` и `Clone`. `Clone` - это наш духовный эквивалент конструктора `Copy`, но он никогда не вызовется неявно. Вам нужно явно вызвать `clone` у элемента, который вы хотите клонировать. Copy - особый случай Clone, у которого реализацией является просто “скопируй биты”. Типы Copy *неявно* клонируются во время перемещения, но, исходя из определения Copy, это означает просто не считать старую копию неинициализированной - то есть, это пустая операция.

Хоть Rust и предоставляет типаж `Default` для определения духовного эквивалента конструктора по умолчанию, его очень редко используют. Все из-за того, что переменные не инициализируются неявно. Default в основном полезен только для обобщенного программирования. В конкретном контексте, тип предоставит статический метод `new` для любого типа конструкторов “по умолчанию”. Здесь нет связи с `new` из других языков, и особого смысла это слово тоже не несет. Это просто соглашение именования.

TODO: рассказать о “размещающем new” (placement new)?

Деструкторы

Что язык *действительно* дает, так это полноценные автоматические деструкторы в виде типажа `Drop`, который предоставляет следующий метод:

```
fn drop(&mut self);
```

Этот метод дает типу время каким-то образом завершить то, что он делал.

После запуска `drop`, Rust рекурсивно попытается удалить все поля `self`.

Это удобная возможность, позволяющая не писать *рутинный деструктор* для удаления всех полей. Если у структуры нет никакой дополнительной логики при удалении кроме удаления всех своих полей, то `Drop` не надо реализовывать вовсе!

Нельзя запретить это поведение в Rust 1.0.

Заметьте, обладание `&mut self` означает, что даже если вы сможете отменить рекурсивный вызов деструктора, Rust запретит вам, например, передавать владение полями из self. Для большинства типов это абсолютно нормально.

Например, в собственной реализации `Box` можно написать `Drop` так:

```
#![feature(alloc, heap_api, drop_in_place, unique)]

extern crate alloc;

use std::ptr::{drop_in_place, Unique};
use std::mem;

use alloc::heap;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(*self.ptr);
            heap::deallocate((*self.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}
```

и это нормально работает, потому что, когда Rust собирается удалить поле `ptr`, он видит [Unique], у которого нет реализации `Drop` в данном примере. Аналогично, ничто не сможет использовать-после-освобождения `ptr`, потому что это поле станет недостижимым, когда закончится выполнение деструктора.

В то же вермя так работать не будет:

```
#![feature(alloc, heap_api, drop_in_place, unique)]

extern crate alloc;

use std::ptr::{drop_in_place, Unique};
use std::mem;

use alloc::heap;

struct Box<T>{ ptr: Unique<T> }
```

```
impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(*self.ptr);
            heap::deallocate((*self.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}

struct SuperBox<T> { my_box: Box<T> }

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // Гипер-оптимизация: освобождаем содержимое box вместо него,
            // не вызывая `drop` у его содержимого
            heap::deallocate((*self.my_box.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}
```

После освобождения ptr из box в деструкторе SuperBox Rust с радостью приступит к вызову деструктора у самого box, и все сломается из-за использования-после-освобождения и двойного-освобождения.

Заметьте, что поведение рекурсивного вызова деструктора применяется ко всем структурам и перечислениям, независимо от того, реализуют они Drop или нет. Поэтому что-то вроде

```
struct Boxy<T> {
    data1: Box<T>,
    data2: Box<T>,
    info: u32,
}
```

будет вызывать деструкторы полей data1 и data2 всякий раз, когда они “должны” быть удалены, даже при том, что они сами не реализуют Drop. Мы говорим, что такому типу *нужен Drop*, хотя он сам не реализует Drop.

Аналогично,

```
enum Link {
    Next(Box<Link>),
    None,
}
```

удалит внутреннее поле Box тогда и только тогда, когда экземпляр будет содержать вариант Next.

В большинстве случаев это отлично работает, потому что вам не нужно волноваться о добавлении/удалении деструкторов во время рефакторинга ваших данных. Тем не менее, есть, конечно, еще много случаев, в которых действительно нужно делать с деструкторами вещи посложнее.

Классическим безопасным решением для того, чтобы отменить рекурсивный вызов деструктора и позволить передать владение из Self во время `drop`, является использование Option:

```
#![feature(alloc, heap_api, drop_in_place, unique)]

extern crate alloc;

use std::ptr::{drop_in_place, Unique};
use std::mem;

use alloc::heap;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(*self.ptr);
            heap::deallocate((*self.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}

struct SuperBox<T> { my_box: Option<Box<T>> }

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // Гипер-оптимизация: освобождаем содержимое box вместо него,
            // не вызывая `drop` у содержимого. Необходимо установить поля `box`
            // как `None` для того, чтобы Rust не пытался вызвать Drop у них.
            let my_box = self.my_box.take().unwrap();
            heap::deallocate((*my_box.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
            mem::forget(my_box);
        }
    }
}
```

В то же время здесь довольно странная семантика: вы говорите, что поле, которое всегда *долж-*

но быть Some, *может* быть None, только потому что это произошло в деструкторе. Конечно, в этом, наоборот, большой смысл: вы можете вызывать произвольные методы у self во время вызова деструктора, а после деинициализации поля это будет делать запрещено. Хотя это и не запретит вам создавать любые другие произвольные недопустимые состояния.

В конечном счете это нормальное решение. Безусловно, так вы добьетесь отмены вызова деструктора. В то же время мы надеемся, что найдется в будущем первоклассный способ сообщить, что у поля не должен автоматически вызываться деструктор.

Утечка

Управление ресурсами на основе владения предназначено для упрощения композиции. Вы получаете ресурсы, создавая объект, и отпускаете ресурсы, удаляя его. Из-за того, что удаление производится за вас, вы не можете забыть отпустить ресурсы, и это происходит настолько быстро, насколько это возможно! Конечно, все прекрасно, у вас нет никаких проблем.

На самом деле все ужасно и мы должны попытаться решить появившиеся более экзотические проблемы.

Многие люди полагают, что Rust устраняет утечку ресурсов. На практике, это в основном правда. Вы бы очень удивились, увидев, что у программы на Безопасном Rust утекают ресурсы в неконтролируемом направлении.

Однако с теоретической точки зрения все абсолютно не так, независимо от того, как вы смотрите на это. В самом строгом смысле, “утечка” настолько абстрактна, насколько и неизбежна. Довольно просто инициализировать коллекцию вначале программы, наполнить ее кучей объектов с деструкторами и затем войти в бесконечный цикл, который никогда не обращается к ней. Коллекция будет бесполезно храниться в памяти, удерживая свои драгоценные ресурсы до окончания программы (в этот момент все эти ресурсы все равно будут собраны сборщиком ОС).

Можем ограничить определение утечки: невозможность вызова деструктора у значения, которое уже недоступно. Rust не борется с ней. На самом деле у Rust даже есть функция *для осуществления такой утечки*: `mem::forget`. Эта функция съедает полученное значение *и не вызывает его деструктор*.

Раньше `mem::forget` помечалась unsafe в качестве статической индикации того, что ошибка при вызове деструктора это чаще всего неправильный подход (хотя он и полезен в некотором особом случае в небезопасном коде). В то же время в целом это считали непригодной ситуацией: есть много способов получить ошибки при вызове деструктора в безопасном коде. Самым известным примером является создание цикла из указателей подсчета-ссылок (RC), использующих внутреннюю изменяемость.

В безопасном коде разумно предполагать, что утечка самого деструктора не происходит, потому что любая программа с такой утечкой неправильна. Однако *небезопасный* код не может полагаться на то, что вызов деструктора является безопасным. Для большинства типов это не играет роли: если сам деструктор утек, то тип по определению недоступен, поэтому это и не важно, не так ли? Например, если утекает `Box<u8>`, то вы тратите память впустую, но это вряд ли нарушит безопасность памяти.

Мы должны быть очень осторожны с утечкой деструкторов в *прокси* типах. Это типы, управляющие доступом к определенному объекту, но на самом деле не владеющие им. Прокси объекты

встречаются редко. Прокси объекты, о которых надо волноваться, встречаются еще реже. И все же рассмотрим три интересных примера из стандартной библиотеки:

- `vec::Drain`
- `Rc`
- `thread::scoped::JoinGuard`

Опустошение (Drain)

`drain` - это API коллекций, который передает владение данными из контейнера, не уничтожая сам контейнер. Это позволяет нам заново использовать место расположения `Vec` после передачи владения всего содержимого. Он создает итератор (Drain), который возвращает содержимое Vec по значению.

Теперь представьте Drain в середине итерации: некоторые значения уже перемещены, некоторые еще нет. Это означает, что часть Vec - это абсолютно неинициализированные данные! Каждый раз перед удалением значения мы могли бы сдвигать назад все элементы Vec, но это сильно скажется на производительности.

Вместо этого можно сделать так, чтобы Drain восстанавливал хранилище данных Vec, когда удаляется. Он должен закончить итерирование, переместить оставшиеся в векторе элементы ближе к началу хранилища и изменить `len` у Vec. Он даже будет безопасен при размотке! Элементарно!

Теперь представим следующее:

```
let mut vec = vec![Box::new(0); 4];

{
    // начало опустошения, vec больше не доступен
    let mut drainer = vec.drain(..);

    // вытаскиваем два элемента и тут же их уничтожаем
    drainer.next();
    drainer.next();

    // избавляемся от drainer, но не вызываем его деструктор
    mem::forget(drainer);
}

// Ой, vec[0] удален, мы читаем указатель на освобожденную память!
println!("{}", vec[0]);
```

Это точно Не Хорошо. К сожалению, мы застряли между молотом и наковальней: поддержка согласованного состояния имеет неподъемную цену (и обесценит любые преимущества API). Несогласованное состояние дает нам Неопределенное Поведение в безопасном коде (делает API несостоятельным).

Так что же нам делать? Можем выбрать тривиальное согласованное состояние: установить длину Vec в 0 вначале итерации, и поменять ее при необходимости в деструкторе. Таким образом,

если все выполняется нормально, мы получим предсказуемое поведение с небольшими накладными расходами. Но если у кто-то *наберется наглости* и он выполнит mem::forget в середине итерации, все *утечет еще сильнее* (и возможно оставит Vec в неожиданном, но при этом согласованном состоянии). Из-за того, что mem::forget безопасен, все остальное тоже абсолютно безопасно. Мы называем утечки, вызывающие еще большие утечки, *усилением утечек*.

Rc

Rc нам интересен, потому что, на первый взгляд, он вообще не является прокси значением. В конце концов он управляет данными, на которые указывает, и удаляет их после удаления всех Rc. Утечка в Rc не кажется особо опасной. Она оставит счетчик ссылок в постоянном значении, что не даст данным удалиться или освободиться, но это же очень похоже на Box, не правда ли?

Неа.

Представим упрощенную реализацию Rc:

```
struct Rc<T> {
    ptr: *mut RcBox<T>,
}

struct RcBox<T> {
    data: T,
    ref_count: usize,
}

impl<T> Rc<T> {
    fn new(data: T) -> Self {
        unsafe {
            // Правда было бы здорово, если бы heap::allocate так работал?
            let ptr = heap::allocate::<RcBox<T>>();
            ptr::write(ptr, RcBox {
                data: data,
                ref_count: 1,
            });
            Rc { ptr: ptr }
        }
    }

    fn clone(&self) -> Self {
        unsafe {
            (*self.ptr).ref_count += 1;
        }
        Rc { ptr: self.ptr }
    }
}

impl<T> Drop for Rc<T> {
    fn drop(&mut self) {
```

```
        unsafe {
            (*self.ptr).ref_count -= 1;
            if (*self.ptr).ref_count == 0 {
                // удаляем данные и освобождаем их
                ptr::read(self.ptr);
                heap::deallocate(self.ptr);
            }
        }
    }
}
```

В коде содержится неявное и неуловимое предположение: `ref_count` подходит по размеру к `usize`, потому что количество RC в памяти не может быть больше, чем `usize::MAX`. Но это само по себе подразумевает, что `ref_count` точно отражает количество Rc в памяти, что, как мы знаем, не всегда правда с `mem::forget`. Используя `mem::forget`, мы можем переполнить `ref_count`, и затем опустить его до 0 оставшимися Rc. Дальше можем счастливо использовать-после-освобождения внутренние данные. Плохо Плохо Не Хорошо.

Можно исправить это, просто проверяя `ref_count` и выполняя *что-то*. Позиция стандартной библиотеки - просто вызвать `abort`, потому что программа ужасно ухудшается в таком случае. К тому же, *бог ты мой*, это все настолько нелепо.

thread::scoped::JoinGuard

API thread::scoped разрешает порождать потоки, ссылающиеся на данные из родительского стека, без какой-либо синхронизации этих данных, гарантируя, что родитель завершит поток до того, как любые из этих общих данных выйдут из области видимости.

```
pub fn scoped<'a, F>(f: F) -> JoinGuard<'a>
    where F: FnOnce() + Send + 'a
```

Здесь `f` - это замыкание, выполняемое в другом потоке. Выражение `F: Send +'a` означает, что F замыкается на данных, которые живут `'a`, и либо он владеет данными, либо данные реализуют Sync (подразумевая, что `&data` реализует Send).

Из-за того, что у JoinGuard есть время жизни, он держит все данные замыкания заимствованными в потоке родителе. Это означает, что JoinGuard не может жить дольше, чем данные, с которыми работает другой поток. Когда JoinGuard *в действительности* удаляется, он блокирует родительский поток, гарантируя, что дочерний поток удалится до того, как данные замыкания выйдут из области видимости родительского потока.

Использование выглядит так:

```
let mut data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
{
    let guards = vec![];
    for x in &mut data {
        // Перемещаем изменяемую ссылку в замыкание, и выполняем его в
        // другом потоке. У замыкания границы времени жизни совпадают с
```

```
        // временем жизни изменяемой ссылки `x`, которую мы храним в нем.
        // Возвращаемому сторожевому значению (guard) в свою очередь присвоено
        // время жизни замыкания, и он также изменяемо заимствует `data`, как
        // сделал `x`. Это означает, что у нас нет доступа к `data`, пока
        // сторожевое значение не уйдет.
        let guard = thread::scoped(move || {
            *x *= 2;
        });
        // сохраняем сторожевое значение потока на будущее.
        guards.push(guard);
    }
    // Все сторожевые значения удаляются здесь, заставляя завершаться потоки
    // (текущий поток блокируется здесь пока другие потоки не завершатся).
    // Когда потоки завершились, заимствование заканчивается и данные становятся
    // опять доступными в текущем потоке.
}
// данные определенно будут изменены здесь.
```

В принципе все нормально работает! Система владения Rust отлично гарантирует это! ...кроме одного - она ожидает, что вызываемый деструктор должен быть безопасным.

```
let mut data = Box::new(0);
{
    let guard = thread::scoped(|| {
        // Это в самом лучшем случае гонка данных. В худшем -
        // использование-после-освобождения.
        *data += 1;
    });
    // Из-за того что guard забыт, заимствование заканчивается без
    // блокировки текущего потока.
    mem::forget(guard);
}
// Итак, Box удаляется здесь, в то время как поток из области видимости выше
// может попытаться получить доступ к нему.
```

Бум. Здесь выполнение деструктора было базовой штукой в API, но в итоге было сделано совсем по-другому.

8

Размотка

У Rust *многоуровневая* схема перехвата ошибок:

- Если чего-то разумно не может быть, используется Option.
- Если что-то идет не так и может быть разумно перехвачено, используется Result.
- Если что-то идет не так и не может быть разумно перехвачено, поток паникует.
- Если что-то катастрофическое случается, программа закрывается с ошибкой.

Option и Result предпочитаемы в подавляющем большинстве ситуаций, особенно из-за того, что они могут быть преобразованы в панику или прерывание с ошибкой по усмотрению пользовательского API. Паника заставляет поток прервать нормальное выполнение и размотать свой стек, вызывая деструкторы так, как-будто каждая функция в стеке нормально завершилась.

По состоянию на 1.0, у Rust есть два мнения, когда дело доходит до паники. Давным давно Rust был очень похож на Erlang. Как и у Erlang, у Rust были легковесные потоки, и они должны были убивать себя с паникой, когда переходили в неприемлемое состояние. В отличие от исключений в Java или C++, панику нельзя было поймать в любое время. Ее мог поймать только владелец потока в определенном месте перехвата или *и этот* поток начинал паниковать.

Размотка была очень важна в этом рассказе, потому что не вызов деструкторов позволял утекать памяти и другим ресурсам системы. Из-за того, что ожидалось, что потоки будут умирать во время нормального выполнения, Rust становился очень слабым при работе с долгоживущими системами!

Rust, каким мы его знаем сейчас, вырос из стиля программирования в виде создания все меньших-и-меньших абстракций. Легковесные потоки были убиты и заменены на тяжеловесные потоки ОС. Однако, на стабильном Rust 1.0 паники могут перехватываться только родительским потоком. Это означает, что поимка паники требует размотки целого потока ОС! Это, к сожалению, идет в разрез с философией Rust - использование абстракций нулевой стоимости.

Существует нестабильное API, называемое `catch_panic`, которое позволяет ловить панику, не порождая поток. По-прежнему, мы просим вас пользоваться им умеренно. В частности, теку-

щая реализация размотки в Rust сильно оптимизирована под “невыполняющие размотку” случаи. Если программа не выполняет размотку, цена *ожидания* размотки является нулевой во время исполнения. Как следствие, текущая версия размотки является более дорогостоящей, чем, например, в Java. Не стройте программы, использующие размотку в обычных ситуациях. В идеале вы должны вызывать панику только в случае программных ошибок или *огромных* проблем.

Стратегия размотки в Rust не заточена под полную совместимость с размоткой в других языках. Поэтому размотка в Rust из других языков или наоборот является Неопределенным Поведением. Вы должны ловить *абсолютно* все паники на границе FFI! Что конкретно вы будете делать, зависит от вас, но ч*то-то* делать надо обязательно. Если вы ошибетесь с этим, лучшее, что произойдет, ваше приложение сломается и сгорит. Худшее - оно *не* сломается и *не* сгорит, а продолжит работать в полностью расколошмаченном состоянии.

Безопасность исключений

Хотя программы и должны использовать размотку редко, есть много кода, который *может* запаниковать. Если вы делаете unwrap у None, индекс вне границ массива или делите на 0, ваша программа вызовет панику. В режиме отладки каждая арифметическая операция при переполнении может вызвать панику. Если не быть очень аккуратным и не контролировать строго, какой код исполняется, нужно быть к этому готовым.

Готовность к размотке часто называется *безопасностью исключений* в остальном мире программирования. В Rust присутствуют два уровня безопасности исключений, с которыми можно столкнуться:

- В небезопасном коде *необходимо* соблюдать безопасность исключений в том смысле, что не позволять нарушать безопасность памяти. Назовем это *минимальной* безопасностью исключений.
- В безопасном коде *хорошо бы* соблюдать безопасность исключений до тех пор, пока программа работает правильно. Назовем это *максимальной* безопасностью исключений.

Как и в многих других ситуациях в Rust, небезопасный код должен быть готов к работе с плохим безопасным кодом в случае размотки. Код, который временно создает неправильные состояния, должен заботиться, чтобы не вызвалась паника в этом состоянии. В общем смысле это означает, что необходимо гарантировать, что только код, не вызывающий панику, выполняется пока все находится в неправильном состоянии или необходимо создать охранное значение, которое почистит это состояние в случае паники. Это не обязательно означает, что состояние во время паники должно быть полностью вразумительным. Мы должны только гарантировать, что это *безопасное* состояние.

Большая часть небезопасного кода является листовидной, и поэтому легко делается безопасной от исключений. Она контролирует весь запускаемый код, и большинство этого кода не вызовет панику. Однако работа с массивами временно неинициализированных данных путём вызова обработчика, предоставленного вызывающей стороной, не является чем-то необычным для небезопасного кода. Такой код должен быть аккуратным и подразумевать безопасность исключений.

Vec::push_all

`Vec::push_all` - это временный хак, позволяющий очень эффективно увеличить Vec, используя срез обобщенных данных. Вот простая реализация:

```
impl<T: Clone> Vec<T> {
    fn push_all(&mut self, to_push: &[T]) {
        self.reserve(to_push.len());
        unsafe {
            // не может переполниться, потому что мы только что зарезервировали его
            self.set_len(self.len() + to_push.len());

            for (i, x) in to_push.iter().enumerate() {
                self.ptr().offset(i as isize).write(x.clone());
            }
        }
    }
}
```

Мы обходим `push` для избежания избыточных проверок размера и `len` Vec, которые мы точно знаем. Логика абсолютно правильна, кроме маленькой проблемы: код не безопасен от исключений! `set_len`, `offset` и `write` - надежны; `clone` - бомба, которую мы просмотрели.

Clone абсолютно не контролируется нами и свободно может вызвать панику. Если так случится, наша функция выйдет раньше времени и длина Vec будет слишком большой. Если к нему обратятся или удалят его, произойдет чтение неинициализированной памяти!

Исправление тут очень простое. Если мы хотим гарантировать, что значения, которые мы *на самом деле* клонировали, удаляются, мы можем устанавливать `len` на каждом цикле итераций. Если мы просто хотим гарантировать, что не будет прочтена неинициализированная память, мы можем установить `len` после цикла.

BinaryHeap::sift_up

Поднять наверх элемент в куче чуть сложнее, чем расширить Vec. Псевдокод будет следующим:

```
bubble_up(heap, index):
    while index != 0 && heap[index] < heap[parent(index)]:
        heap.swap(index, parent(index))
        index = parent(index)
```

Буквальное переписывание этого кода на Rust абсолютно нормально, но характеристики производительности раздражают: исходный элемент постоянно меняется местами с соседним. Исправим на следующее:

```
bubble_up(heap, index):
    let elem = heap[index]
```

```
    while index != 0 && element < heap[parent(index)]:
        heap[index] = heap[parent(index)]
        index = parent(index)
    heap[index] = elem
```

Код гарантирует, что каждый элемент копируется максимально малое количество раз (на самом деле `elem` скопируется дважды в общем случае). Но теперь появились проблемы с безопасностью исключений! Все время существуют две копии одного значения. Если вызовется паника в функции, что-то будет дважды удалено. К сожалению, у нас также нет полного контроля над кодом: сравнение определяется пользователем!

В отличие от Vec исправление здесь не такое простое. Первым вариантом будет разбить пользовательский код и небезопасный код на две отдельные фазы:

```
bubble_up(heap, index):
    let end_index = index;
    while end_index != 0 && heap[end_index] < heap[parent(end_index)]:
        end_index = parent(end_index)

    let elem = heap[index]
    while index != end_index:
        heap[index] = heap[parent(index)]
        index = parent(index)
    heap[index] = elem
```

Поломка пользовательского кода теперь больше не проблема, потому что мы еще не меняли состояние кучи. Во время обращения к куче мы работаем только с данными и функциями, которым доверяем, поэтому не надо волноваться о возникновении паники.

Возможно, вы недовольны такой конструкцией. Конечно, это обман! Нам приходится выполнять сложное прохождение кучи *дважды*! Ладно, давайте стиснем зубы. Давайте *по-настоящему* смешаем ненадежный и небезопасный код вместе.

Если б у Rust были `try` и `finally` как в Java, мы могли бы сделать следующее:

```
bubble_up(heap, index):
    let elem = heap[index]
    try:
        while index != 0 && element < heap[parent(index)]:
            heap[index] = heap[parent(index)]
            index = parent(index)
    finally:
        heap[index] = elem
```

Базовая идея проста: если сравнение вызывает панику, мы просто присваиваем потерянный элемент по логически неинициализированному индексу в куче и катапультируемся. Каждый, кто проходит кучу, видит ее потенциально *несогласованной*, но по крайней мере мы избавились от двойного удаления! Если алгоритм нормально завершится, то независимо ни от чего в конце выполнится операция присвоения элемента по индексу.

Жалко, что у Rust нет такой конструкции, придется накатать свою! Сделаем ее одним из возможных способов - будем хранить состояние алгоритма в отдельной структуре и создадим для логики “finally” деструктор. Независимо от того, вызовется паника или нет, он выполнится и почистит все за нами.

```
struct Hole<'a, T: 'a> {
    data: &'a mut [T],
    /// `elt` всегда `Some` от создания до удаления.
    elt: Option<T>,
    pos: usize,
}

impl<'a, T> Hole<'a, T> {
    fn new(data: &'a mut [T], pos: usize) -> Self {
        unsafe {
            let elt = ptr::read(&data[pos]);
            Hole {
                data: data,
                elt: Some(elt),
                pos: pos,
            }
        }
    }

    fn pos(&self) -> usize { self.pos }

    fn removed(&self) -> &T { self.elt.as_ref().unwrap() }

    unsafe fn get(&self, index: usize) -> &T { &self.data[index] }

    unsafe fn move_to(&mut self, index: usize) {
        let index_ptr: *const _ = &self.data[index];
        let hole_ptr = &mut self.data[self.pos];
        ptr::copy_nonoverlapping(index_ptr, hole_ptr, 1);
        self.pos = index;
    }
}

impl<'a, T> Drop for Hole<'a, T> {
    fn drop(&mut self) {
        // заполним заново hole
        unsafe {
            let pos = self.pos;
            ptr::write(&mut self.data[pos], self.elt.take().unwrap());
        }
    }
}

impl<T: Ord> BinaryHeap<T> {
```

```
    fn sift_up(&mut self, pos: usize) {
        unsafe {
            // Вытащим значение по `pos` и создадим hole.
            let mut hole = Hole::new(&mut self.data, pos);

            while hole.pos() != 0 {
                let parent = parent(hole.pos());
                if hole.removed() <= hole.get(parent) { break }
                hole.move_to(parent);
            }
            // Hole будет безусловно заполнена здесь; с паникой или нет!
        }
    }
}
```

Отравление

Несмотря на то что весь небезопасный код *должен* гарантировать минимальную безопасность исключений, не все типы гарантируют *максимальную* безопасность исключений. Даже если тип гарантирует ее, ваш код может приписать ему дополнительное значение. Например, integer точно безопасен от исключений, но никакой семантики этого у него самого нет. Возможно, код, вызывающий панику, не сможет корректно изменить integer, создав несогласованное состояние программы.

Обычно это нормально, потому что все, что наблюдает возникновение исключения должно быть уничтожено. Например, если вы посылаете Vec в другой поток и тот поток вызывает панику, неважно в каком состоянии находится Vec. Он будет уничтожен и отброшен навсегда. Но есть некоторые типы, которые особенно хороши в контрабанде своих значений через границы паники.

Эти типы могут явно *отравить* себя, если становятся свидетелями паники. Отравление ничего не влечет за собой. Обычно оно просто означает препятствие нормальной работе. Самым заметным примером этого является Mutex из стандартной библиотеки. Mutex отравится, если один из его MutexGuards (то, что он возвращает когда получает захват) удалится во время паники. Все будущие попытки захватить Mutex вернут `Err` или панику.

Mutex отравляется не для настоящей безопасности в том смысле, как обычно это понимает Rust. Он отравляется как охранник безопасности слепого использования пришедших во время паники данных пока Mutex был захвачен. Данные в таком Mutex были в каком-то промежуточном состоянии, и поэтому, возможно, являются несогласованными или незаконченными. Необходимо отметить, что нельзя нарушить безопасность памяти таким типом, если он корректно написан. В конце концов он должен быть в минимальной безопасности исключений!

Однако, если Mutex содержит, скажем, BinaryHeap, у которого на самом деле нет свойств кучи, не похоже, что любой код, использующий его, делает то, что задумывал автор. Поэтому программа не будет работать правильно. Итак, если вы дважды убедились, что вы можете сделать *что-то* со значением, Mutex предоставляет метод для получения захвата в любом случае. Это *безопасно* в конце концов. Просто может получиться чушь.

9

Многопоточность и параллелизм

У Rust, как у языка, *на самом деле* нет мнения о том, как быть с многопоточностью или параллелизмом. Стандартная библиотека предлагает потоки ОС и блокирующие системные вызовы, потому что они есть у всех, и они достаточно универсальны для построения абстракций над ними относительно однозначным способом. Передача сообщений, зеленые потоки и асинхронные API достаточно разнообразны, чтобы построение любой абстракции над ними как правило заставляло бы искать компромисс, на который мы не хотели идти в версии 1.0.

Однако то, как Rust моделирует параллелизм, позволяет вам относительно просто разработать собственную парадигму параллелизма похожую на библиотечную и заставит любой другой код Просто Работать с вашим. Только необходимо проставить правильные времена жизни и Send и Sync, в соответствующих случаях, и вы готовы начать гонку. Или наоборот… не… начинать… гонки.

Гонки данных и их условия

Безопасный Rust гарантирует отсутствие гонок данных, определяемых так:

- два или больше потока одновременно получают доступ к участку памяти
- один из них пишет
- один из них не синхронизирован

У гонок данных Неопределенное Поведение, и, соответственно, их невозможно выполнить в безопасном Rust. Гонки данных *в большинстве своем* предотвращаются системой владения Rust: изменяемые ссылки не могут совпадать, поэтому и невозможно получить гонки данных. Внутренняя изменяемость делает все сложней, именно поэтому у нас есть типажи Send и Sync (смотри ниже).

Однако Rust не предотвращает общие условия для гонок.

Это принципиально невозможно, и, честно говоря, нежелательно. Ваше железо может вызывать гонки, ваша ОС может вызывать гонки, другие программы на вашем компьютере могут вызывать гонки и мир, в котором все это выполняется, тоже этому подвержен. Всеми системами, искренне утверждающими, что предотвращают *все* состояния гонок, будет довольно ужасно пользоваться, если просто не невозможно.

Поэтому программы на безопасном Rust абсолютно “спокойно” могут зайти в тупик или сделать что-нибудь невероятно глупое в случае некорректной синхронизации. Очевидно, такие программы не очень хороши сами по себе, но Rust не может предотвратить всё. Итак, состояния гонок сами по себе не могут нарушить безопасность памяти в Rust. Они могут это сделать только вместе с другим небезопасным кодом. Например:

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
// Arc нужен затем, чтобы память, в которой хранится AtomicUsize, всё ещё существовала
↳
// на момент, когда другой поток попытается увеличить этот AtomicUsize, даже если
// исполнение полностью завершится к этому моменту. Rust не компилирует программу без
↳ этого,
// из-за требований времен жизни для thread::spawn!
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

// `move` захватывает other_idx по-значению, передавая его в этот поток
thread::spawn(move || {
    // Нормально изменять idx, потому что это значение атомарно,
    // тем самым оно не может вызвать гонку данных.
    other_idx.fetch_add(10, Ordering::SeqCst);
});

// Индексируем значением, полученным из атомарного. Это безопасно, потому что мы
// читаем атомарную память только один раз, и затем передаем копию этого
// значения в реализацию индексирования Vec. Это индексирование проверит
// корректность границ и шанс, что значение поменяется в середине выполнения,
// равен нулю. Но наша программа может вызвать панику если поток, который мы
// создали выполнит инкремент перед этим запуском. Условия для гонки во время
// корректного выполнения программы (паника очень редко правильна) зависит от
// порядка вызова потоков.
println!("{}", data[idx.load(Ordering::SeqCst)]);
```

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];

let idx = Arc::new(AtomicUsize::new(0));
```

```
let other_idx = idx.clone();

// `move` захватывает other_idx по-значению, передавая его в этот поток
thread::spawn(move || {
    // Нормально изменять idx, потому что это значение атомарно,
    // тем самым оно не может вызвать гонку данных.
    other_idx.fetch_add(10, Ordering::SeqCst);
});

if idx.load(Ordering::SeqCst) < data.len() {
    unsafe {
        // Некорректная загрузка idx после выполнения проверки границ.
        // Оно может поменяться. Это условие для гонки, *и это опасно*,
        // потому что мы решили сделать `get_unchecked`, который `unsafe`.
        println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst)));
    }
}
```

Send и Sync

Не все подчиняется наследуемой изменяемости, однако. Некоторые типы позволяют размножать совпадающие ссылки на место в памяти, и при этом изменять его. Если эти типы не используют синхронизацию, чтобы управлять доступом к этой памяти, они абсолютно не потокобезопасны. Rust отражает это через типажи `Send` и `Sync`.

- Тип является Send если его безопасно передавать другому потоку.
- Тип является Sync если его безопасно разделять между потоками (&T является Send).

Send и Sync - это основа многопоточности в Rust. Из-за этого существует большое количество специальных инструментов для того, чтобы заставить их работать правильно. Первый и самый главный - это то, что они являются небезопасными типажами. Это означает, что их реализация небезопасна, а другой небезопасный код подразумевает, что они реализованы корректно. Из-за того, что они являются *маркерными типажами* (у них нет объектов, связанных с ними, таких как методы), корректная реализация просто означает, что они обладают внутренними свойствами, которыми должна обладать реализация. Некорректная реализация Send или Sync может вызвать Неопределенное Поведение.

Send и Sync это к тому же автоматически наследуемые типажи. Это означает, что в отличие от любого другого типажа, если тип состоит только из типов Send или Sync, то он тоже Send или Sync. Почти все примитивные типы являются Send и Sync, и, как следствие, большинство типов, с которыми вы столкнетесь, тоже будут Send и Sync.

Главные исключения:

- сырые указатели не являются ни Send ни Sync (из-за того, что у них нет охранников безопасности).
- `UnsafeCell` не является Sync (и, следовательно, `Cell` и `RefCell` тоже).

- `Rc` не является Send или Sync (потому что количество ссылок является общим и несинхронизированным).

`Rc` и `UnsafeCell` фундаментально непотокобезопасны: они разрешают несинхронизированное общее изменяемое состояние. Однако сырые указатели, строго говоря, помечены непотокобезопасными в большинстве своем для *статического анализа*. Для выполнения чего-нибудь полезного с сырыми указателями их необходимо разыменовать, что уже небезопасно. В этом смысле, можно было бы сказать, что их можно пометить потокобезопасными.

Но все же важно пометить их непотокобезопасными, чтобы запретить автоматическую маркировку типов, их содержащих, потокобезопасными. У таких типов нетривиальная передача владения, и, вряд ли, их автор обязательно сильно задумывался о потокобезопасности. В случае с Rc, мы имеем замечательный пример типа, содержащего `*mut`, который точно непотокобезопасен.

Типы, для которых `Send` и `Sync` не выведены автоматически, могут легко реализовать их:

```
struct MyBox(*mut u8);

unsafe impl Send for MyBox {}
unsafe impl Sync for MyBox {}
```

В *чрезвычайно редких* случаях у типов, которым не нужно автоматическое выведение Send или Sync, необходимо убрать реализацию Send и Sync:

```
#![feature(optin_builtin_traits)]

// Я обладаю особой магической семантикой для некоторых примитивов синхронизации!
struct SpecialThreadToken(u8);

impl !Send for SpecialThreadToken {}
impl !Sync for SpecialThreadToken {}
```

Заметьте, что, *само по себе*, невозможно некорректно вывести Send и Sync. Только типы, которым другой небезопасный код приписывает особое значение, могут, вероятно, доставить проблемы при некорректном Send или Sync.

Большинство типов, использующих сырые указатели, должны быть инкапсулированы в абстракцию, к которой можно вывести Send и Sync. Например, все стандартные коллекции Rust являются Send и Sync (когда содержат типы Send и Sync), несмотря на постоянное использование ими сырых указателей для управления размещением в памяти и сложным владением. Аналогично, большинство итераторов по этим коллекциям являются Send и Sync, потому что они во многом ведут себя как `&` или `&mut` в коллекции.

TODO: лучше объяснить, что могут и что не могут Send или Sync. Достаточно ли обращений к гонкам данных?

Атомарные операции

Rust нагло наследует модель памяти C11 для атомарных операций. Не потому что эта модель великолепна или легка для понимания. Даже наоборот, эта модель довольно сложна и имеет

несколько дефектов[1]. Просто мы принимаем во внимание, что все модели атомарных операций имеют недостатки. По крайней мере, мы можем извлечь выгоду из существующих инструментов и изученности Си.

Попытаться полностью объяснить модель в этой книге довольно безнадежно. Она описывается в терминах графов безумных зависимостей, для объяснения которых понадобится отдельная книга. Если хотите изучить все эти штучки-дрючки, смотрите в спецификацию Си (Секция 7.17)[2]. Итак, мы попытаемся объяснить основы и некоторые проблемы, с которыми столкнулись разработчики Rust.

Модель памяти C11 нацелена на преодоление разрыва между той семантикой, которую мы хотим, теми оптимизациями, которые хочет компилятор, и тому хаосу из противоречий, который хочет наше железо. *Нам* же просто хочется писать программы и чтобы они делали то, что мы написали, да побыстрее. Разве это так много?

Изменение порядка компилятором

Компилятор нацелен на то, чтобы иметь возможность делать разные виды сумасшедших преобразований для уменьшения зависимости между данными и удаления мертвого кода. В частности, он может полностью поменять текущий порядок событий или даже сделать так, что некоторые события никогда не произойдут! Если мы напишем, что-нибудь такое

```
x = 1;
y = 3;
x = 2;
```

Компилятор может решить, что лучше бы ваша программа делала так

```
x = 2;
y = 3;
```

Порядок событий поменялся, и одно из них просто удалилось. С точки зрения одного потока, это не играет роли: после выполнения все окажется в том же состоянии. Но если наша программа многопоточна, мы рассчитываем, что x присвоится 1 до присвоения y. Мы бы хотели, чтобы компилятор мог выполнять оптимизации для ускорения производительности. С другой стороны мы бы также хотели, чтобы наша программа *делала то, что мы написали*.

Изменение порядка железом

С другой стороны даже если компилятор полностью понял, что мы хотим и уважительно отнесся к нашим желаниям, железо может доставить нам неприятности. Проблемы придут от ЦП из-за иерархий памяти. Есть единственное глобальное общее пространство памяти где-то в вашей аппаратной части, но с точки зрения каждого ядра ЦП оно *так далеко* и *так медленно*. Каждый ЦП лучше будет работать со своим локальным кэшем данных и будет преодолевать все страдания обмена с общей памятью, только если этих данных в кэше нет.

[1] http://plv.mpi-sws.org/c11comp/popl15.pdf

[2] http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899

В конце концов это ведь и есть задача кэша, так? Если бы каждое чтение из кэша сопровождалось чтением из общей памяти для двойной проверки, что данные не изменились, какой был бы в нем смысл? В результате получаем, что железо не гарантирует, что порядок событий в *одном* потоке, совпадает с порядком в *другом*. Чтобы гарантировать его, мы должны использовать специальные инструкции CPU, указывая ему быть чуть менее умным.

Например, заставим компилятор имитировать следующую логику:

```
initial state: x = 0, y = 1

THREAD 1        THREAD2
y = 3;          if x == 1 {
x = 1;              y *= 2;
                }
```

В идеале, у этой программы 2 вероятных конечных состояния:

- `y = 3`: (поток 2 выполнил проверки перед тем, как завершился поток 1)
- `y = 6`: (поток 2 выполнил проверки после того, как завершился поток 1)

Однако потенциально у этой программы есть третье состояние, возможное благодаря железу:

- `y = 2`: (поток 2 видит `x = 1`, но не `y = 3`, и затем переписывает `y = 3`)

Стоит отметить, что разные типы CPU предоставляют разные гарантии. Обычно разделяют железо на две категории: со строгим и нестрогим порядком выполнения. Надо заметить, что x86/64 гарантируют строгий порядок, а ARM, наоборот, нестрогий. Из этого можно сделать два вывода для параллельного программирования:

- Запрос гарантий строгого порядка у железа со строгим порядком может стоить очень дешево, или даже не стоить ничего, потому что они и так предоставляют такие гарантии безусловно. Гарантии нестрого порядка могут дать выигрыш в производительности только на железе с нестрогим порядком выполнения.
- Запрос гарантий, которые слишком слабы, на железе со строгим порядком *скорее всего* сработает, даже если ваша программа не является строго правильной. Если это возможно, параллельные алгоритмы лучше всего проверять на железе с нестрогим порядком.

Обращения к данным

Модель памяти C11 пытается уменьшить разрыв между миром программ и железа, позволяя говорить нам о *причинно-следственной связи* нашей программы. Как правило, между частями программы и потоками, исполняющими их, устанавливаются отношения *“выполняется прежде”*. Это дает железу и компилятору место для более агрессивной оптимизации программы, когда строгие отношения *“выполняется прежде”* еще не установлены, но заставляет их быть очень аккуратными, когда они уже установлены. Мы устанавливаем эти отношения путём *обращения к данным* и *атомарных доступов*.

Обращения к данным - это хлеб насущный в мире программирования. Они принципиально не синхронизированы, и компиляторы могут агрессивно оптимизировать их. В частности, компилятор может спокойно поменять порядок обращения к данным, если программа однопоточна. Железо также может спокойно вносить изменения в обращения к данным из других потоков ленивым и непоследовательным образом, когда захочет. Важно, что доступ к данным - это способ осуществить гонку данных. Обращения к данным очень дружелюбны к железу и компилятору, но, как мы видим, предлагают *ужасную* семантику для написания синхронизированного кода. На самом деле, они слишком нестроги.

Написать правильно синхронизированный код, используя только обращения к данным, буквально невозможно.

Атомарные доступы нужны, чтобы сказать железу и компилятору, что наша программа многопоточна. Каждый атомарный доступ может быть помечен *порядком*, указывающим тип отношения, устанавливаемого с другим обращением. На практике, это выражается в объяснении железу и компилятору тех вещей, которые они *не понимают*. Для компилятора все в большинстве своем вращается вокруг переупорядочивания инструкций. Для железа все в большинстве своем вращается вокруг того, как запись передается другому потоку. Rust предоставляет несколько порядков обращения:

- Последовательный порядок (SeqCst)
- Освобождение (Release)
- Получение (Acquire)
- Расслабленный порядок (Relaxed)

(Заметьте: Мы явно не выделяем *потребляющий* порядок C11)

TODO: плюсы и минусы? TODO: “не забыть синхронизацию”

Последовательный порядок

Последовательный порядок является самым мощным из всех, включающим в себя все ограничения других порядков. Интуитивно понятно, что последовательные операции не могут быть переупорядочены: все обращения в одном потоке, происходящие до и после доступа SeqCst, остаются до и после него. Программа без гонок данных, использующая только последовательный порядок атомарных обращений и обращений к данным, обладает замечательным свойством, есть только одно единственное выполнение программных инструкций, по которому согласуются все потоки. Это выполнение также можно легко объяснить: это просто чередование выполнений всех потоков. Это не работает, если вы используете нестрогий порядок атомарных операций.

Относительная дружелюбность разработчику не достается бесплатно. Даже на платформах со строгим порядком выполнения последовательный порядок обращений выстраивает барьеры в памяти.

На практике, последовательный порядок редко необходим для правильности программ. Однако он будет точно правильным выбором, если вы не осведомлены о других порядках обращения к памяти. Чуть более медленная программа - это гораздо лучше, чем неправильная программа! Также в будущем легко будет перейти к более слабым ограничениям атомарных операций. Просто измените `SeqCst` на `Relaxed` и все! Конечно, это еще большой вопрос, будет ли это изменение правильно работать.

Получение-Освобождение

Получение и Освобождение в основном работают в паре. Их имена объясняют их использование: они отлично подходят для получения и освобождения блокировки, гарантируя что критические секции не пересекутся.

Интуитивно понятно, что обращение после получения блокировки остается после него. Но операции, происходящие до получения блокировки, могут спокойно быть переупорядочены и могут произойти после него. Аналогично, освобождение блокировки гарантирует, что каждое обращение до него останется до него. Но операции, происходящие после освобождения блокировки, могут спокойно быть переупорядочены и произойти до него.

Когда поток A освобождает место в памяти, и после этого поток B последовательно получает *то же* место в памяти, устанавливается причинно-следственная связь. Каждая запись, происходящая до освобождения блокировки A, будет замечена B после освобождения блокировки. Однако никакая другая причинно-следственная связь не устанавливается с другими потоками. Аналогично, она не устанавливается между A и B, если они обращаются к *разным* местам в памяти.

Базовое использование получения-освобождения, поэтому, простое: вы получаете блокировку места в памяти в начале критической секции, и освобождаете его блокировку в конце. Например, простая блокировка может быть такой:

```
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
    let lock = Arc::new(AtomicBool::new(false)); // значение отвечает на вопрос "Я зан
↳ ят?"

    // ... распределяем блокировку по потокам каким-то образом ...

    // Пытаемся получить блокировку, устанавливая ее в true
    while lock.compare_and_swap(false, true, Ordering::Acquire) { }
    // выход из цикла говорит, что мы успешно получили блокировку!

    // ... ужасные обращения к данным ...

    // Ок, закончили, освобождаем блокировку
    lock.store(false, Ordering::Release);
}
```

На платформах со строгим порядком выполнения большинство обращений к данным обладают семантикой получения-освобождения, делая ее абсолютно бесплатной. Но это не относится к платформам с нестрогим порядком.

Расслабленный порядок

Расслабленные обращения являются самыми слабыми. Их можно спокойно переупорядочивать, не устанавливая ранние связи. Хотя, расслабленные операции все еще атомарны. Это означает, они не считают, что обращение к данным и любые операции чтения-изменения-записи, выполняемые с ними, происходят атомарно. Расслабленные операции подходят для вещей, которые вы хотите чтобы точно произошли, но о которых вы не особо заботитесь. Например, увеличение счетчика может быть безопасно выполнено несколькими потоками расслабленным `fetch_add`, если вы не используете этот счетчик для синхронизаций других обращений.

Редко можно получить выгоду, выполняя расслабленные операции на платформах со строгим порядком выполнения, из-за того, что обращения к данным в них обладают семантикой получения-освобождения. Однако расслабленные операции могут быть дешевы на платформах с нестрогим порядком выполнения.

10

Пример: Реализация Vec

Объединив все вместе, напишем `std::Vec` с самого начала. Данный проект будет работать только на ночной сборке (по крайней мере для Rust 1.2.0), из-за того что все самые лучшие инструменты для написания небезопасного кода нестабильны. Большинство используемого нестабильного кода будет тем или иным образом стабилизировано со временем, за исключением API аллокатора.

Однако, мы все же будем стараться избегать нестабильного кода там, где это только возможно. В частности, мы не будем использовать никакие встроенные функции, которые делают код немножко лучше или немножко эффективней, потому что они постоянно нестабильны. Хотя многие встроенные функции в других местах действительно *стали* стабильными (`std::ptr` и `str::mem` состоят из множества встроенных функций).

В общем случае это означает, что наша реализация не будет обладать преимуществами всех возможных оптимизаций, хотя и без сомнений не будет *наивной*. Мы погрузимся во все самые мелкие детали, даже если вопросы не будут стоить выеденного яйца.

Вы хотели продвинутого программирования. Будет вам продвинутое.

Содержимое структуры

Для начала необходимо разобраться с компоновкой структуры. Vec состоит из трех частей: указатель на место в памяти, размер места в памяти и количество инициализированных элементов.

Наивно полагаем, что нам нужен такой дизайн:

```
pub struct Vec<T> {
    ptr: *mut T,
    cap: usize,
    len: usize,
}
```

И в самом деле это скомпилируется. К сожалению, дизайн неправильный. Во- первых, компилятор даст нам слишком строгую вариантность. Поэтому `&Vec<&'static str>` нельзя будет использовать, где ожидается `&Vec<&'a str>`. Что более важно, он даст некорректную информацию о владении анализатору сброса, потому что компилятор будет думать, что мы не хотим владеть никакими значениями типа `T`. Смотри главу по владениям и временам жизни для всех деталей вариантности и проверки сброса.

Как мы уже видели в главе о владении, следует использовать `Unique<T>` вместо `*mut T`, если у нас есть сырой указатель на место в памяти, которым мы владеем. Хотя Unique нестабилен, поэтому нам бы не следовало его использовать.

Поясним, Unique - это обертка вокруг сырого указателя, которая:

- вариантна над `T`
- может владеть значением типа `T` (для проверки сброса)
- является Send/Sync, если `T` - Send/Sync
- разыменуется в `*mut T` (поэтому она действует как `*mut` в нашем коде)
- Наш указатель никогда не будет нулевым (поэтому `Option<Vec<T>>` оптимизирован по нулевому указателю)

Мы можем реализовать все эти требования кроме последнего в стабильном Rust:

```
use std::marker::PhantomData;
use std::ops::Deref;
use std::mem;

struct Unique<T> {
    ptr: *const T,              // *const для вариантности
    _marker: PhantomData<T>,    // Для анализатора сброса
}

// Выведение Send и Sync безопасно, потому что мы уникальные владельцы
// данных. В этом случае Unique<T> это "просто" T.
unsafe impl<T: Send> Send for Unique<T> {}
unsafe impl<T: Sync> Sync for Unique<T> {}

impl<T> Unique<T> {
    pub fn new(ptr: *mut T) -> Self {
        Unique { ptr: ptr, _marker: PhantomData }
    }
}

impl<T> Deref for Unique<T> {
    type Target = *mut T;
    fn deref(&self) -> &*mut T {
        // Нет явного приведения *const к *mut, с одновременным получением
        // ссылки. Поэтому мы просто используем
        // трансмутацию, ведь это все "просто указатели".
        unsafe { mem::transmute(&self.ptr) }
```

```
    }
}
```

К сожалению, механизмы, позволяющие утверждать, что ваше значение отличается от нуля, нестабильны и вряд ли будут стабилизированы в ближайшее время. Поэтому, ладно, примем удар и используем Unique из стандартной библиотеки:

```
#![feature(unique)]

use std::ptr::{Unique, self};

pub struct Vec<T> {
    ptr: Unique<T>,
    cap: usize,
    len: usize,
}
```

Если вы не волнуетесь за оптимизацию нулевого указателя, можете использовать стабильный код. Однако мы будет проектировать остальной код, обладая этой оптимизацией. В частности, `Unique::new` вызывать небезопасно, потому что, если передавать `null`, то получишь Неопределенное Поведение. Нашему стабильному Unique не нужен небезопасный `new`, потому что он не дает никаких интересных гарантий своего содержимого.

Выделение памяти

Использование Unique портит важную характеристику Vec (и на самом деле все стандартные коллекции): пустой Vec на самом деле вообще никак не размещается в памяти. Итак, если мы не размещаем ничего в памяти, а также не можем подсунуть нулевой указатель в `ptr`, что нам делать в `Vec::new`? Ну, просто подсунем какой-нить другой мусор туда!

Это отлично подходит, потому что мы уже ставили `cap == 0` как проверку того, что выделение памяти не нужно. Нам даже не нужно специально перехватывать это в коде, потому что обычно требуется в любом случае проверить `cap > len` или `len >0`. По традиции в такой ситуации в Rust используют значение `0x01`. Стандартная библиотека действительно экспортирует его в виде `alloc::heap::EMPTY`. Мы будем использовать `heap::EMPTY` довольно часто, потому что мы не выделяем память, а `null` сломало бы компилятор.

Всё API `heap` абсолютно нестабильно и скрыто под отключаемой нестабильной возможностью `heap_api`. Мы могли бы сами определить `heap::EMPTY`, но нам в любом случае нужна остальная часть API `heap`, поэтому давайте просто добавим зависимость от нее.

Итак:

```
#![feature(alloc, heap_api)]

use std::mem;

use alloc::heap::EMPTY;
```

```
impl<T> Vec<T> {
    fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "Мы не готовы работать с THP");
        unsafe {
            // необходимо явно привести EMPTY к настоящему типу ptr, позволим
            // выводу типов справиться с этим.
            Vec { ptr: Unique::new(heap::EMPTY as *mut _), len: 0, cap: 0 }
        }
    }
}
```

Я намеренно поставил assert, потому что THP потребуют особой обработки в нашем коде, а я хочу отложить эту проблему на потом. Без этого assert некоторые из наших ранних проектов делали бы Очень Плохие Вещи.

Следующее, что нам нужно, это определить, что делать, когда нам *на самом деле* нужно место в памяти. Для этого воспользуемся остальным `heap_api`. Он позволяет напрямую обращаться к аллокатору Rust (по умолчанию, jemalloc).

Также нам нужно обрабатывать условия нехватки памяти (англ. out-of-memory, OOM). Стандартная библиотека вызывает внутреннюю функцию `abort`, которая выполняет неправильную инструкцию, приводящую к краху всей программы. Причиной, по которой мы используем `abort`, а не панику, является то, что размотка может заставить выделить еще память, а это кажется не очень правильным, если выделение вернется с “эй, у меня же нет больше памяти”.

Конечно, это немного глупо, ведь на большинстве платформ обычно не заканчивается память. Ваша ОС, наверняка, убьет приложение по другой причине, если оно станет использовать всю память. Самый вероятный случай, когда сработает OOM, является запрос смехотворного количества памяти за один раз (например, половина теоретического адресного пространства). В данном случае, *вероятно*, будет нормально вызвать панику, и ничего плохого не случится. Но мы пытаемся действовать как стандартная библиотека, поэтому просто убьем всю программу.

Мы сказали, что не будем использовать внутренние функции (intrinsics), поэтому сделаем то, что делает `std` - выйдем из программы. Вызовем `std::process::exit` со случайным числом.

```
fn oom() {
    ::std::process::exit(-9999);
}
```

Ок, теперь можем описать увеличение размера. Грубо говоря, нам нужна следующая логика:

```
if cap == 0:
    allocate()
    cap = 1
else:
    reallocate()
    cap *= 2
```

Но единственное поддерживаемое в Rust API аллокатора настолько низкоуровневое, что придется выполнить дополнительную работу. Нам также надо защититься от особых условий, которые могут возникнуть, таких как, действительно большое или пустое распределение памяти.

В частности, `ptr::offset` приносит немало неприятностей, потому что обладает семантикой ограниченных инструкций GEP LLVM (LLVM GEP inbounds instructions). Если вам везло раньше не сталкиваться с этими инструкциями, так вот, что делает GEP: анализ совпадения ссылок, анализ совпадения ссылок, анализ совпадения ссылок. Для оптимизирующего компилятора очень важно, чтобы он понимал зависимости данных и совпадения ссылок.

В качестве простого примера возьмем следующий фрагмент кода:

```
*x *= 7;
*y *= 3;
```

Если компилятор сможет доказать, что `x` и `y` указывают на разные места в памяти, то, теоретически, эти две операции могут выполниться параллельно (загрузятся, например, в различные регистры и выполнятся независимо). Но если ему это не удастся, и он решит, что `x` и `y` указывают на одно место в памяти, операции придется делать с одним значением и их нельзя будет объединить потом.

Если вы используете ограниченные инструкции GEP, вы особым образом говорите LLVM, что нужные вам смещения находятся внутри границ одной “размещенной” сущности. Основной выигрыш состоит в том, что LLVM может предположить, что если два указателя ссылаются на два непересекающихся объекта, смещения этих указателей *также* не совпадают (потому что вы не окажетесь в случайном месте в памяти). LLVM сильно оптимизирован на работу со смещениями GEP, а особенно с ограниченными инструкциями, поэтому важно использовать их по полной.

Итак, если это все, что делает GEP, то как же это может принести неприятности?

Первой проблемой является индексация массивов с помощью беззнаковых целых, а GEP (и, как следствие, `ptr::offset`) принимает целые со знаком. Это означает, что половина кажущихся правильными индексов вызовут переполнение GEP и, на самом деле, поведут в обратном направлении! Таким образом, мы должны ограничить все выделения памяти значением `isize::MAX`. Это означает, что мы должны волноваться только за объекты размером в 1 байт, потому что, например, выделение памяти под массив `u16`, длиной больше чем `isize::MAX`, просто исчерпает всю системную память. Однако для того, чтобы избежать проблем, когда кто-то будет интерпретировать массив длиной менее `isize::MAX` как байты, стандартная библиотека ограничивает выделение памяти `isize::MAX` байтами.

На всех 64-битных платформах, которые на данный момент поддерживает Rust, мы искусственно ограничены адресным пространством, гораздо меньшим чем все 64 бита (современные платформы x64 предлагают только 48-битную адресацию), поэтому, для начала, можем положиться на нехватку памяти. Однако на 32-битных платформах, особенно с расширенным адресным пространством (PAE x86 или x32), теоретически, возможно выделить больше чем `isize::MAX` байт в памяти.

Но, ведь это учебник, не будем особо оптимизировать это место, и просто безоговорочно выполним проверку, вместо того чтобы использовать умные платформо- зависимые `cfg`.

Другой проблемой встают выделения нулевого размера. Тут два выделения, о которых надо волноваться: `cap = 0` для всех T и `cap > 0` для THP.

Эти случаи сложны, потому что придется обратиться к тому, что LLVM подразумевает под “распределением”. Смысл распределения в LLVM существенно более абстрактный, чем то, как мы

обычно понимаем его. Из-за того, что LLVM должен работать с разными семантиками языка и пользовательскими аллокаторами, она не может действительно глубоко понимать смысл распределения. Вместо этого, главной идеей, стоящей за распределением, является “не пересекаться с другими вещами”. Вот поэтому выделенная память в куче, на стеке и глобальные переменные не пересекаются случайным образом. Да, это всё касается проверок совпадения ссылок. Таким образом, Rust может технически играть немного быстрее и свободнее с понятием распределения до тех пор, пока оно *согласуется* с LLVM.

Возвращаясь к случаю с выделением нулевого размера, есть пара мест, в которых мы хотим смещаться на 0 как следствие обобщенного кода. Встает тогда вопрос: согласовано ли так делать? Для THP мы пришли к выводу, что делать ограниченное GEP смещение на произвольное число элементов действительно согласовано. Это пустая операция во время исполнения, потому что каждый элемент не занимает места, и, нормальным считается предполагать, что бесконечное число THP расположены по адресу `0x01`. Ни один аллокатор не выделит этот адрес, потому что они не будут выделять адрес `0x00` и, как правило, будут выделять память больше одного байта. Также, как правило, вся первая страница памяти в любом случае защищена от выделения в ней памяти (все 4k на многих платформах).

Однако, что насчет типов положительного размера? Они чуть посложнее. В принципе, вы можете поспорить, что смещение на 0 не дает информации LLVM: есть ли элемент до или после адреса, невозможно его распознать. Согласившись, что он может делать плохие вещи, защитимся от этого в явной форме.

Фух

Ок, отбросим всю эту чепуху, давайте наконец распределим память:

```
fn grow(&mut self) {
    // все здесь довольно деликатно, поэтому давайте скажем, что все небезопасно
    unsafe {
        // текущее API требует указания размера и выравнивания вручную.
        let align = mem::align_of::<T>();
        let elem_size = mem::size_of::<T>();

        let (new_cap, ptr) = if self.cap == 0 {
            let ptr = heap::allocate(elem_size, align);
            (1, ptr)
        } else {
            // подразумеваем, что `self.cap < isize::MAX`,
            // поэтому это не надо проверять.
            let new_cap = self.cap * 2;
            // аналогично, здесь не будет переполнения того, что мы раньше уже выделял
↳ и память такого размера
            let old_num_bytes = self.cap * elem_size;

            // проверяем, что новое выделение не превышает `isize::MAX`
            // вообще, независимо от текущей ёмкости. Это объединяет проверки
            // `new_cap <= isize::MAX` и `new_num_bytes <= usize::MAX`,
            // которые нам нужны. Хоть мы и теряем возможность выделить,
            // например, 2/3 из адресного пространства одному Vec из i16 на 32-битной
↳ платформе.
```

```
            // Увы, бедный Йорик - Я знал его, Горацио.
            assert!(old_num_bytes <= (::std::isize::MAX as usize) / 2,
                    "capacity overflow");

            let new_num_bytes = old_num_bytes * 2;
            let ptr = heap::reallocate(*self.ptr as *mut _,
                                       old_num_bytes,
                                       new_num_bytes,
                                       align);
            (new_cap, ptr)
        };

        // Если выделение или перераспределение возвращается с ошибкой, получим `null`
        if ptr.is_null() { oom(); }

        self.ptr = Unique::new(ptr as *mut _);
        self.cap = new_cap;
    }
}
```

Ничего особо сложного. Просто вычисления размеров и выравниваний и выполнения некоторых осторожных проверок умножения.

Push и Pop

Отлично. Мы можем инициализировать. Мы можем выделять память. Давайте теперь реализуем немного функциональности! Начнем с `push`. Все что ему нужно, это проверить можем ли мы расти, безусловно записать по следующему индексу элемент и увеличить длину.

Для записи нам надо быть осторожными с обращениями к памяти, в которую мы собираемся делать запись. В худшем случае - это настоящая неинициализированная память, полученная от аллокатора. В лучшем - это биты, оставшиеся от старых значений, которые уже удалились. Так или иначе, мы не можем проиндексировать ее и разыменовать, потому что это приведет к обращению к ней, как к правильному экземпляру типа T. Что хуже, `foo[idx] = x` попытается вызвать `drop` у старого значения `foo[idx]`!

Правильным путем будет сделать `ptr::write`, который вслепую заменяет значение по заданному адресу значением, которое мы предоставим. Никаких обращений не происходит.

Для `push` если старая длина (до его вызова) была 0, то записываем по 0-ому индексу. Итак, нам нужно сместиться на старое значение длины.

```
pub fn push(&mut self, elem: T) {
    if self.len == self.cap { self.grow(); }

    unsafe {
        ptr::write(self.ptr.offset(self.len as isize), elem);
    }
```

```
    // Не вызовет ошибку, потому что мы вызовем сначала OOM.
    self.len += 1;
}
```

Легко! Что насчет pop? Хоть индекс и инициализирован во время доступа, Rust не позволит нам разыменовать место в памяти для вытаскивания значения, потому что это оставит память неинициализированной! Для этого нам нужно `ptr::read`, которая просто копирует байты из целевого адреса и интерпретирует их как значение типа T. Это оставит память по этому адресу логически неинициализированной, хотя на самом деле там отличный экземпляр типа T.

Для pop если старое значение длины 1, то мы хотим прочитать 0-й индекс. Итак, нам нужно сместиться на новое значение длины.

```
pub fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None
    } else {
        self.len -= 1;
        unsafe {
            Some(ptr::read(self.ptr.offset(self.len as isize)))
        }
    }
}
```

Освобождение

Следующее, что мы должны сделать - это реализовать Drop так, чтобы у нас не происходила широкомасштабная утечка кучи ресурсов. Самым простым способом будет вызывать pop до тех пор пока он не вернет None, и затем освободить наш буфер. Помните, что вызов pop не нужен, если `T: !Drop`. В теории мы можем спросить у Rust, является ли T `needs_drop` и избежать вызова pop. Однако, на практике, LLVM *действительно* хорош в удалении такого простого независимого кода без побочных эффектов, поэтому я бы не стал беспокоится, если только вы не заметите, что он не удален (в этом случае он будет удален).

Мы не должны вызывать `heap::deallocate`, если `self.cap == 0`, так как в этом случае мы еще на самом деле не выделили память.

```
impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            while let Some(_) = self.pop() { }

            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes, align);
```

```
            }
        }
    }
}
```

Разыменование

Отлично! Мы реализовали приличный минимальный стек. Можем делать push, можем делать pop и можем освобождать ресурсы за собой. Однако, нам нужен еще приличный объем функциональности. В частности, у нас есть функциональность массива, но никакой функциональности среза. Это довольно просто решить: можем реализовать `Deref<Target=[T]>`. Он магическим образом заставит наш Vec неявно приводиться и вести себя как срез в любых условиях.

Все, что нам нужно - `slice::from_raw_parts`. Он будет корректно обрабатывать пустые срезы для нас. А также, раз мы добавили поддержку THP, он будет Просто Работать и для них.

```
use std::ops::Deref;

impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe {
            ::std::slice::from_raw_parts(*self.ptr, self.len)
        }
    }
}
```

И сделаем DerefMut тоже:

```
use std::ops::DerefMut;

impl<T> DerefMut for Vec<T> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe {
            ::std::slice::from_raw_parts_mut(*self.ptr, self.len)
        }
    }
}
```

Теперь у нас есть `len`, `first`, `last`, индексирование, нарезка, сортировка, `iter`, `iter_mut` и все другие примочки среза. Мило!

Insert и Remove

Вот, что срез *не* предоставляет, так это `insert` и `remove`, поэтому давайте сделаем их следующими.

Insert нужно сдвинуть все элементы от целевого направо на единицу. Для этого используем `ptr::copy`, являющийся нашей версией `memmove` в Си. Он копирует кусок памяти из одного места в другое, корректно обрабатывая случаи, если источник и назначение пересекаются (что точно произойдет здесь).

Если мы вставим по индексу `i`, мы должны сдвинуть `[i .. len]` в `[i+1 .. len+1]`, используя старую длину.

```
pub fn insert(&mut self, index: usize, elem: T) {
    // Внимание: `<=` потому что считается привильным вставлять после любого элемента
    // что было бы эквивалентно push.
    assert!(index <= self.len, "index out of bounds");
    if self.cap == self.len { self.grow(); }

    unsafe {
        if index < self.len {
            // ptr::copy(src, dest, len): "копировать из источника в назначение len эл
↳ ементов"
            ptr::copy(self.ptr.offset(index as isize),
                      self.ptr.offset(index as isize + 1),
                      self.len - index);
        }
        ptr::write(self.ptr.offset(index as isize), elem);
        self.len += 1;
    }
}
```

Remove ведет себя наоборот. Нужно сдвинуть все элементы из `[i+1 .. len + 1]` в `[i .. len]`, используя *новую* длину.

```
pub fn remove(&mut self, index: usize) -> T {
    // Внимание: `<` потому что *не* правильно удалять после всего
    assert!(index < self.len, "index out of bounds");
    unsafe {
        self.len -= 1;
        let result = ptr::read(self.ptr.offset(index as isize));
        ptr::copy(self.ptr.offset(index as isize + 1),
                  self.ptr.offset(index as isize),
                  self.len - index);
        result
    }
}
```

IntoIter

Продвинемся к написанию итераторов. `iter` и `iter_mut` уже написаны для нас, спасибо Магии Deref. Однако Vec предоставляет ещё два интересных итератора, которые не могут предоставить срезы: `into_iter` и `drain`.

IntoIter потребляет Vec по значению, и, следовательно, может пройтись по его элементам по значению. Для этого IntoIter должен взять контроль над размещением в памяти Vec.

IntoIter к тому же должен быть двусторонним, чтобы уметь читать с обоих концов. Чтение с конца можно реализовать вызовом `pop`, чтение с начала гораздо труднее. Мы могли бы вызывать `remove(0)`, но это было бы чрезвычайно дорого. Вместо этого мы просто используем `ptr::read`, чтобы скопировать значение с любого конца Vec, вообще не изменяя буфер.

Для этого используем очень популярную идиому Си по итерации массива. Сделаем два указателя; один, указывающий на начало массива, и один, указывающий на элемент после конца массива. Если нам нужен элемент с одной стороны, мы читаем значение указателя и сдвигаем указатель на единицу. Когда два указателя эквивалентны, мы знаем, что закончили.

Заметьте, что порядок чтения и сдвига противоположны для `next` и `next_back`. Для `next_back` указатель всегда указывает на элемент после того, который ему нужно прочитать, а для `next` - всегда на элемент, который он хочет следующим прочитать. Чтобы понять почему это так, предположим случай, в котором каждый элемент кроме одного был пройден.

Массив выглядит так:

```
         S  E
[X, X, X, O, X, X, X]
```

Если E указывал бы напрямую на элемент, который надо пройти следующим, то этот случай был бы не отличим от случая, когда элементов больше нет.

Несмотря на то, что мы на самом деле не волнуемся о расположении Vec в памяти во время итерации, нам также надо владеть информацией об этом, чтобы освободить память во время освобождения IntoIter.

Итак, используем следующую структуру:

```
struct IntoIter<T> {
    buf: Unique<T>,
    cap: usize,
    start: *const T,
    end: *const T,
}
```

И вот с чем мы заканчиваем инициализацию:

```
impl<T> Vec<T> {
    fn into_iter(self) -> IntoIter<T> {
        // Нельзя деструктурировать Vec из-за того, что он Drop
        let ptr = self.ptr;
        let cap = self.cap;
        let len = self.len;

        // Убеждаемся, что не освобождаем Vec, из-за того что он освободит буфер
        mem::forget(self);
```

```
        unsafe {
            IntoIter {
                buf: ptr,
                cap: cap,
                start: *ptr,
                end: if cap == 0 {
                    // нельзя сместить этот указатель, он не расположен в памяти!
                    *ptr
                } else {
                    ptr.offset(len as isize)
                }
            }
        }
    }
}
```

Вот итератор с начала:

```
impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = self.start.offset(1);
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let len = (self.end as usize - self.start as usize)
                  / mem::size_of::<T>();
        (len, Some(len))
    }
}
```

А вот с конца.

```
impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                self.end = self.end.offset(-1);
```

```
                Some(ptr::read(self.end))
            }
        }
    }
}
```

Из-за того, что IntoIter забирает владение своего места расположения, необходимо реализовать Drop для его освобождения. Но также надо реализовать Drop, чтобы освободить те элементы, которые еще не были пройдены.

```
impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            // освобождаем все оставшиеся элементы
            for _ in &mut *self {}

            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.buf as *mut _, num_bytes, align);
            }
        }
    }
}
```

RawVec

Мы дошли до интересного момента: у нас дублируется логика по размещению буфера и освобождения его памяти в Vec и IntoIter. Теперь после реализации и обнаружения *действительного* дублирования логики пришло время выполнить ее сжатие.

Мы абстрагируем пару `(ptr, cap)` и опишем для нее логику размещения в памяти, возрастания и освобождения:

```
struct RawVec<T> {
    ptr: Unique<T>,
    cap: usize,
}

impl<T> RawVec<T> {
    fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "TODO: реализовать поддержку THP");
        unsafe {
            RawVec { ptr: Unique::new(heap::EMPTY as *mut T), cap: 0 }
        }
    }
```

```
    // взято из Vec без изменений
    fn grow(&mut self) {
        unsafe {
            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();

            let (new_cap, ptr) = if self.cap == 0 {
                let ptr = heap::allocate(elem_size, align);
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let ptr = heap::reallocate(*self.ptr as *mut _,
                                           self.cap * elem_size,
                                           new_cap * elem_size,
                                           align);
                (new_cap, ptr)
            };

            // Если выделение или перераспределение памяти возвращается с ошибкой, пол
↳ учим `null`
            if ptr.is_null() { oom() }

            self.ptr = Unique::new(ptr as *mut _);
            self.cap = new_cap;
        }
    }
}


impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes, align);
            }
        }
    }
}
```

И изменим Vec:

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}
```

```rust
impl<T> Vec<T> {
    fn ptr(&self) -> *mut T { *self.buf.ptr }

    fn cap(&self) -> usize { self.buf.cap }

    pub fn new() -> Self {
        Vec { buf: RawVec::new(), len: 0 }
    }

    // push/pop/insert/remove в основном без изменений:
    // * `self.ptr -> self.ptr()`
    // * `self.cap -> self.cap()`
    // * `self.grow -> self.buf.grow()`
}

impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() {}
        // освобождение обрабатывается RawVec
    }
}
```

И наконец можем сильно упростить IntoIter:

```rust
struct IntoIter<T> {
    _buf: RawVec<T>, // нам не нужно волноваться об этом. Просто нужно, чтобы это жило
↳ .
    start: *const T,
    end: *const T,
}

// next и next_back, буквально, не поменялись, потому что не ссылаются на buf

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        // только убедимся что все элементы прочтены;
        // буфер сам почистит себя после этого.
        for _ in &mut *self {}
    }
}

impl<T> Vec<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        unsafe {
            // нужно использовать ptr::read чтобы небезопасно передать buf, потому что
            // он не Copy, а Vec реализует Drop (поэтому мы не можем деструктурировать
↳  ero).
            let buf = ptr::read(&self.buf);
```

```
            let len = self.len;
            mem::forget(self);

            IntoIter {
                start: *buf.ptr,
                end: buf.ptr.offset(len as isize),
                _buf: buf,
            }
        }
    }
}
```

Гораздо лучше.

Drain

Перейдем к Drain. Drain очень похож на IntoIter, за исключением того, что он не потребляет Vec, а заимствует Vec и оставляет его расположение в памяти нетронутым. Для начала реализуем только “базовую” полноразмерную версию.

```
use std::marker::PhantomData;

struct Drain<'a, T: 'a> {
    // Нужно ограничить время жизни, поэтому делаем это с помощью `&'a mut Vec<T>`
    // потому что семантически именно это и содержится. Мы "просто" вызываем
    // `pop()` и `remove(0)`.
    vec: PhantomData<&'a mut Vec<T>>
    start: *const T,
    end: *const T,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
```

– постойте, кажется это уже было. Выполним еще сжатие логики. И IntoIter и Drain имеют одну и ту же структуру, просто вынесем ее.

```
struct RawValIter<T> {
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    // небезопасно создавать, потому что нет связанных времен жизни.
```

```
    // Важно хранить RawValIter в той же структуре, что и ее настоящее
    // размещение. Это допустимо, поскольку это скрытые детали нашей реализации.
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if slice.len() == 0 {
                // если `len = 0`, то это не настоящее место размещения.
                // Нужно избежать сдвига, потому что это даст неправильную
                // информацию LLVM через GEP.
                slice.as_ptr()
            } else {
                slice.as_ptr().offset(slice.len() as isize)
            }
        }
    }
}

// Iterator и DoubleEndedIterator реализуются аналогично IntoIter.
```

А IntoIter станет следующим:

```
pub struct IntoIter<T> {
    _buf: RawVec<T>, // нам не нужно волноваться об этом. Просто нужно, чтобы это жило
↳ .
    iter: RawValIter<T>,
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        for _ in &mut self.iter {}
    }
}

impl<T> Vec<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            let buf = ptr::read(&self.buf);
```

```
            mem::forget(self);

            IntoIter {
                iter: iter,
                _buf: buf,
            }
        }
    }
}
```

Заметьте, что я оставил несколько причудливых мест в проекте, чтобы сделать модернизацию Drain по работе с произвольными поддиапазонами немного проще. В частности мы *могли бы* сделать, чтобы RawValIter выполнял опустошение самого себя при освобождении, но это не будет работать правильно для более сложного Drain. Также возьмем срез для упрощения инициализации Drain.

Итак, теперь Drain по-настоящему прост:

```
use std::marker::PhantomData;

pub struct Drain<'a, T: 'a> {
    vec: PhantomData<&'a mut Vec<T>>,
    iter: RawValIter<T>,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<'a, T> Drop for Drain<'a, T> {
    fn drop(&mut self) {
        for _ in &mut self.iter {}
    }
}

impl<T> Vec<T> {
    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            // это безопасный mem::forget. Если Drain забыт, у нас просто утечет
            // все содержимое Vec. К тому же нам нужно сделать это со *временем*
            // в любом случае, так почему не сделать это сейчас?
```

```
            self.len = 0;

            Drain {
                iter: iter,
                vec: PhantomData,
            }
        }
    }
}
```

Для деталей по проблеме `mem::forget`, смотри раздел по утечкам.

Обработка типов нулевого размера

Пришло время. Начнем бороться с чудищем, называемым типами нулевого размера. Безопасному Rust *никогда* не нужно волноваться об этом, а вот Vec очень интенсивно использует сырые указатели и сырое выделение места, именно которым и надо заботиться о THP. Надо быть осторожным в двух вещах:

- API сырого распределения места вызовет неопределенное поведение при передаче 0 в качестве размера выделяемого места.
- Сдвиги сырых указателей являются пустыми операциями для THP, что сломает наш Си-подобный итератор указателей.

К счастью, мы абстрагировали наши итераторы по указателям и обработку распределения места в RawValIter и RawVec заранее. Как неожиданно удобно получилось.

Размещение типов нулевого размера

Итак, если API аллокатора не поддерживает выделение памяти нулевого размера, что же нам хранить в нашей выделенной памяти? Ну что ж, `heap::EMPTY`, конечно! Почти любая операция с THP является пустой операцией из-за того, что у THP только одно значение, и, следовательно, не нужно учитывать никакое состояние ни при чтении, ни при записи значений таких типов. Это на самом деле распространяется на `ptr::read` и `ptr::write`: они вообще не смотрят на указатель. По сути, нам никогда не придётся менять указатель.

Заметим, однако, что мы больше не можем надеяться на возникновение нехватки памяти до переполнения в случае THP. Мы должны явно защититься от переполнения емкости для THP.

В нашей текущей архитектуре это означает написание 3 охраняющих условий, по одному в каждый метод RawVec.

```
impl<T> RawVec<T> {
    fn new() -> Self {
        unsafe {
            // !0 это usize::MAX. Эта ветка удалится во время компиляции.
```

```
            let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

            // heap::EMPTY служит как для "невыделения", так и для "выделения нулевого
↳  размера"
            RawVec { ptr: Unique::new(heap::EMPTY as *mut T), cap: cap }
        }
    }

    fn grow(&mut self) {
        unsafe {
            let elem_size = mem::size_of::<T>();

            // из-за того, что мы установили емкость в usize::MAX если elem_size равен
            // 0, то попадание сюда обозначает, что Vec переполнен.
            assert!(elem_size != 0, "capacity overflow");

            let align = mem::align_of::<T>();

            let (new_cap, ptr) = if self.cap == 0 {
                let ptr = heap::allocate(elem_size, align);
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let ptr = heap::reallocate(*self.ptr as *mut _,
                                           self.cap * elem_size,
                                           new_cap * elem_size,
                                           align);
                (new_cap, ptr)
            };

            // Если выделение или перераспределение памяти возвращается с ошибкой, пол
↳ учим `null`
            if ptr.is_null() { oom() }

            self.ptr = Unique::new(ptr as *mut _);
            self.cap = new_cap;
        }
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        let elem_size = mem::size_of::<T>();

        // не освобождаем выделения нулевого размера, потому что выделение никогда не
↳ происходило.
        if self.cap != 0 && elem_size != 0 {
            let align = mem::align_of::<T>();
```

```
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes, align);
            }
        }
    }
}
```

Вот и все. Теперь мы добавили поддержку push и pop для THP. Хотя наши итераторы (не предоствляемые срезом Deref) все еще не работают.

Итерирование по типам нулевого размера

Смещения нулевого размера являются пустыми операциями. Это означает, что в нашей текущей архитектуре мы всегда будем инициализировать `start` и `end` одним и тем же значением, и наши итераторы ничего не вернут. Хорошим решением будет явно привести указатели к целым, увеличивать их, и затем явно приводить их обратно:

```
impl<T> RawValIter<T> {
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if mem::size_of::<T>() == 0 {
                ((slice.as_ptr() as usize) + slice.len()) as *const _
            } else if slice.len() == 0 {
                slice.as_ptr()
            } else {
                slice.as_ptr().offset(slice.len() as isize)
            }
        }
    }
}
```

Теперь у нас другая ошибка. Раньше наши итераторы вообще не запускались, теперь они выполняются *вечно*. Необходимо сделать тот же трюк в реализации итераторов. Также, наш код вычисления size_hint будет вызывать деление на 0 в случае THP. Мы считаем, что два указателя ссылаются на байты, поэтому просто подставим деление на 1 в случае нулевого размера.

```
impl<T> Iterator for RawValIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = if mem::size_of::<T>() == 0 {
```

```
                    (self.start as usize + 1) as *const _
                } else {
                    self.start.offset(1)
                };
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let elem_size = mem::size_of::<T>();
        let len = (self.end as usize - self.start as usize)
                  / if elem_size == 0 { 1 } else { elem_size };
        (len, Some(len))
    }
}

impl<T> DoubleEndedIterator for RawValIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                self.end = if mem::size_of::<T>() == 0 {
                    (self.end as usize - 1) as *const _
                } else {
                    self.end.offset(-1)
                };
                Some(ptr::read(self.end))
            }
        }
    }
}
```

И все. Итерация работает!

Получившийся код

```
#![feature(unique)]
#![feature(alloc, heap_api)]

extern crate alloc;

use std::ptr::{Unique, self};
use std::mem;
use std::ops::{Deref, DerefMut};
```

```rust
use std::marker::PhantomData;

use alloc::heap;

struct RawVec<T> {
    ptr: Unique<T>,
    cap: usize,
}

impl<T> RawVec<T> {
    fn new() -> Self {
        unsafe {
            // !0 это usize::MAX. Эта ветка должна удалиться во время компиляции.
            let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

            // heap::EMPTY служит как для "невыделения", так и для "выделения нулевого
↳  размера"
            RawVec { ptr: Unique::new(heap::EMPTY as *mut T), cap: cap }
        }
    }

    fn grow(&mut self) {
        unsafe {
            let elem_size = mem::size_of::<T>();

            // из-за того, что мы установили емкость в usize::MAX если elem_size равен
            // 0, то попадание сюда обозначает, что Vec переполнен.
            assert!(elem_size != 0, "capacity overflow");

            let align = mem::align_of::<T>();

            let (new_cap, ptr) = if self.cap == 0 {
                let ptr = heap::allocate(elem_size, align);
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let ptr = heap::reallocate(*self.ptr as *mut _,
                                           self.cap * elem_size,
                                           new_cap * elem_size,
                                           align);
                (new_cap, ptr)
            };

            // Если выделение или перераспределение памяти возвращается с ошибкой, пол
↳ учим `null`
            if ptr.is_null() { oom() }

            self.ptr = Unique::new(ptr as *mut _);
```

```
            self.cap = new_cap;
        }
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        let elem_size = mem::size_of::<T>();
        if self.cap != 0 && elem_size != 0 {
            let align = mem::align_of::<T>();

            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes, align);
            }
        }
    }
}
```

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

impl<T> Vec<T> {
    fn ptr(&self) -> *mut T { *self.buf.ptr }

    fn cap(&self) -> usize { self.buf.cap }

    pub fn new() -> Self {
        Vec { buf: RawVec::new(), len: 0 }
    }
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap() { self.buf.grow(); }

        unsafe {
            ptr::write(self.ptr().offset(self.len as isize), elem);
        }

        // Can't fail, we'll OOM first.
        self.len += 1;
    }

    pub fn pop(&mut self) -> Option<T> {
```

```
        if self.len == 0 {
            None
        } else {
            self.len -= 1;
            unsafe {
                Some(ptr::read(self.ptr().offset(self.len as isize)))
            }
        }
    }

    pub fn insert(&mut self, index: usize, elem: T) {
        assert!(index <= self.len, "index out of bounds");
        if self.cap() == self.len { self.buf.grow(); }

        unsafe {
            if index < self.len {
                ptr::copy(self.ptr().offset(index as isize),
                          self.ptr().offset(index as isize + 1),
                          self.len - index);
            }
            ptr::write(self.ptr().offset(index as isize), elem);
            self.len += 1;
        }
    }

    pub fn remove(&mut self, index: usize) -> T {
        assert!(index < self.len, "index out of bounds");
        unsafe {
            self.len -= 1;
            let result = ptr::read(self.ptr().offset(index as isize));
            ptr::copy(self.ptr().offset(index as isize + 1),
                      self.ptr().offset(index as isize),
                      self.len - index);
            result
        }
    }

    pub fn into_iter(self) -> IntoIter<T> {
        unsafe {
            let iter = RawValIter::new(&self);
            let buf = ptr::read(&self.buf);
            mem::forget(self);

            IntoIter {
                iter: iter,
                _buf: buf,
            }
        }
```

```
    }

    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            // это безопасный mem::forget. Если Drain забыт, у нас просто утечет
            // все содержимое Vec. К тому же нам нужно сделать это со *временем*
            // в любом случае, так почему не сделать это сейчас?
            self.len = 0;

            Drain {
                iter: iter,
                vec: PhantomData,
            }
        }
    }
}

impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() {}
        // allocation is handled by RawVec
    }
}

impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe {
            ::std::slice::from_raw_parts(self.ptr(), self.len)
        }
    }
}

impl<T> DerefMut for Vec<T> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe {
            ::std::slice::from_raw_parts_mut(self.ptr(), self.len)
        }
    }
}




struct RawValIter<T> {
```

```
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if mem::size_of::<T>() == 0 {
                ((slice.as_ptr() as usize) + slice.len()) as *const _
            } else if slice.len() == 0 {
                slice.as_ptr()
            } else {
                slice.as_ptr().offset(slice.len() as isize)
            }
        }
    }
}

impl<T> Iterator for RawValIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = if mem::size_of::<T>() == 0 {
                    (self.start as usize + 1) as *const _
                } else {
                    self.start.offset(1)
                };
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let elem_size = mem::size_of::<T>();
        let len = (self.end as usize - self.start as usize)
                  / if elem_size == 0 { 1 } else { elem_size };
        (len, Some(len))
    }
}

impl<T> DoubleEndedIterator for RawValIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
```

```
            None
        } else {
            unsafe {
                self.end = if mem::size_of::<T>() == 0 {
                    (self.end as usize - 1) as *const _
                } else {
                    self.end.offset(-1)
                };
                Some(ptr::read(self.end))
            }
        }
    }
}




pub struct IntoIter<T> {
    _buf: RawVec<T>, // нам не нужно волноваться об этом. Просто нужно, чтобы это жило
↳ .
    iter: RawValIter<T>,
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        for _ in &mut *self {}
    }
}




pub struct Drain<'a, T: 'a> {
    vec: PhantomData<&'a mut Vec<T>>,
    iter: RawValIter<T>,
}

impl<'a, T> Iterator for Drain<'a, T> {
```

```
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next_back() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<'a, T> Drop for Drain<'a, T> {
    fn drop(&mut self) {
        // пре-опустошение итератора
        for _ in &mut self.iter {}
    }
}

/// Прерываем процесс с ошибкой, мы получили нехватку памяти!
///
/// На практике, в большинстве ОС это мертвый код
fn oom() {
    ::std::process::exit(-9999);
}
```

11

Пример: Реализация Arc и Mutex

Знание теории - это все прекрасно, но *лучшим* способом понять что-то является ее использование. Для лучшего понимания атомарных операций и внутренней изменяемости, мы реализуем версии типов Arc и Mutex из стандартной библиотеки.

TODO: ВСЕ СДЕЛАТЬ! ОМГ!