Lecture 2: Standard library

Alexander Stanovoy

February 22, 2022

alex.stanovoy@gmail.com

In this lecture

- `Option` and `Result`
- `Vec` and `VecDeque`
- `BTreeMap` and `BTreeSet`
- `HashMap` and `HashSet`
- `BinaryHeap`
- `LinkedList`
- `String` and `&str`
- `Box` and `Rc`

`Option` and `Result`

`Option`[1] and `Result`[2]

Let's remember their definitions:

```
enum Option<T> { // 'Maybe' from functional languages
    Some(T),
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

[1] `Option` documentation
[2] `Result` documentation

Matching `Option`:

```rust
let result = Some("string");
match result {
    Some(s) => println!("String inside: {s}"),
    None => println!("Ooops, no value"),
}
```

Option API

Useful functions `.unwrap()` and `.expect()`:

```rust
fn unwrap(self) -> T;
fn expect(self, msg: &str) -> T;
```

Option API

Useful functions `.unwrap()` and `.expect()`:

```
let opt = Some(22022022);
assert!(opt.is_some());
assert!(!opt.is_none());
assert_eq!(opt.unwrap(), 22022022);
let x = opt.unwrap(); // Copy!

let newest_opt: Option<i32> = None;
// newest_opt.expect("I'll panic!");

let new_opt = Some(Vec::<i32>::new());
assert_eq!(new_opt.unwrap(), Vec::<i32>::new());
// error[E0382]: use of moved value: `new_opt`
// let x = new_opt.unwrap(); // Clone!
```

Option API

We have a magic function:

```
fn as_ref(&self) -> Option<&T>; // &self is &Option<T>
```

Let's solve a problem:

```
let new_opt = Some(Vec::<i32>::new());
assert_eq!(new_opt.unwrap(), Vec::<i32>::new());
// error[E0382]: use of moved value: `new_opt`
// let x = new_opt.unwrap(); // Clone!

let opt_ref = Some(Vec::<i32>::new());
assert_eq!(new_opt.as_ref().unwrap(), &Vec::<i32>::new());
let x = new_opt.unwrap(); // We used reference!
// There's also .as_mut() function
```

That means if type implements `Copy`, `Option` also implements `Copy`.

Option API

We can map `Option<T>` to `Option<U>`:

```
fn map<U, F>(self, f: F) -> Option<U>;
```

Example:

```
let maybe_some_string = Some(String::from("Hello, World!"));
// `Option::map` takes self *by value*,
// consuming `maybe_some_string`
let maybe_some_len = maybe_some_string.map(|s| s.len());
assert_eq!(maybe_some_len, Some(13));
```

Option API

There's **A LOT** of different `Option` functions, enabling us to write beautiful functional code:

```rust
fn map_or<U, F>(self, default: U, f: F) -> U;
fn map_or_else<U, D, F>(self, default: D, f: F) -> U;
fn unwrap_or(self, default: T) -> T;
fn unwrap_or_else<F>(self, f: F) -> T;
fn and<U>(self, optb: Option<U>) -> Option<U>;
fn and_then<U, F>(self, f: F) -> Option<U>;
fn or(self, optb: Option<T>) -> Option<T>;
fn or_else<F>(self, f: F) -> Option<T>;
fn xor(self, optb: Option<T>) -> Option<T>;
fn zip<U>(self, other: Option<U>) -> Option<(T, U)>;
```

It's recommended for you to study the documentation and try to avoid `match` where possible.

Option and ownership

There's two cool methods to control ownership of the value inside:

```
fn take(&mut self) -> Option<T>;
fn replace(&mut self, value: T) -> Option<T>;
fn insert(&mut self, value: T) -> &mut T;
```

The first one takes the value out of the `Option`, leaving a `None` in its place.

The second one replaces the value inside with the given one, returning `Option` of the old value.

The third one inserts a value into the `Option`, then returns a mutable reference to it.

Option API and ownership: `take`

```rust
struct Node<T> {
    elem: T,
    next: Option<Box<Node<T>>>,
}

pub struct List<T> {
    head: Option<Box<Node<T>>>,
}

impl<T> List<T> {
    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            node.elem
        })
    }
}
```

`Option` and optimizations

Rust guarantees to optimize the following types `T` such that `Option<T>` has the same size as `T`:

- `Box<T>`
- `&T`
- `&mut T`
- `fn`, `extern "C" fn`
- `#[repr(transparent)]` struct around one of the types in this list.
- `num::NonZero*`
- `ptr::NonNull<T>`

This is called the “null pointer optimization” or NPO.

`Result`

Functions return Result whenever errors are expected and recoverable. In the `std` crate, `Result` is most prominently used for I/O.

Results must be used! A common problem with using return values to indicate errors is that it is easy to ignore the return value, thus failing to handle the error. `Result` is annotated with the `#[must_use]` attribute, which will cause the compiler to issue a warning when a Result value is ignored.[3]

[3] The Error Model

Result API

We can `match` it as a regular `enum`:

```
let version = Ok("1.1.14");
match version {
    Ok(v) => println!("working with version: {:?}", v),
    Err(e) => println!("error: version empty"),
}
```

Result API

We have pretty the same functionality as in `Option`:

```
fn is_ok(&self) -> bool;
fn is_err(&self) -> bool;
fn unwrap(self) -> T;
fn unwrap_err(self) -> E;
fn expect_err(self, msg: &str) -> E;
fn expect(self, msg: &str) -> T;
fn as_ref(&self) -> Result<&T, &E>;
fn as_mut(&mut self) -> Result<&mut T, &mut E>;
fn map<U, F>(self, op: F) -> Result<U, E>;
fn map_err<F, O>(self, op: O) -> Result<T, F>;
// And so on
```

It's recommended for you to study the documentation and try to avoid `match` where possible.

Operator ?

Consider the following structure:

```
struct Info {
    name: String,
    age: i32,
}
```

Operator ?

```rust
fn write_info(info: &Info) -> io::Result<()> {
    let mut file = match File::create("my_best_friends.txt") {
        Err(e) => return Err(e),
        Ok(f) => f,
    };
    if let Err(e) = file
        .write_all(format!("name: {}\n", info.name)
        .as_bytes()) {
        return Err(e)
    }
    if let Err(e) = file
        .write_all(format!("age: {}\n", info.age)
        .as_bytes()) {
        return Err(e)
    }
    Ok(())
}
```

Operator ?

We can use the `?` operator to make the code smaller!

```
fn write_info(info: &Info) -> io::Result<()> {
    let mut file = File::create("my_best_friends.txt")?;
    file.write_all(format!("name: {}\n", info.name).as_bytes())?;
    file.write_all(format!("age: {}\n", info.age).as_bytes())?;
    Ok(())
}
```

Beautiful, isn't it?

We can use it for `Option` too!

{Result, Option}::transpose

It's time to link `Result` and `Option` together:

```rust
// self is Option<Result<T, E>>
fn transpose(self) -> Result<Option<T>, E>;

// self is Result<Option<T>, E>
fn transpose(self) -> Option<Result<T, E>>;
```

{Result, Option}::transpose

```
fn read_until_empty() -> io::Result<String> {
    let mut input = stdin().lines();
    let mut output = String::new();

    // input.next() gives Option<Result<String>>
    while let Some(line) = input.next() {
        let line = line?;
        if line.is_empty() {
            break;
        }
        output.push_str(&line);
    }

    Ok(output)
}
```

{Result, Option}::transpose

```rust
fn read_until_empty() -> io::Result<String> {
    let mut input = stdin().lines();
    let mut output = String::new();

    // input.next() gives Option<Result<String>>
    while let Some(line) = input.next().transpose()? {
        if line.is_empty() {
            break;
        }
        output.push_str(&line);
    }

    Ok(output)
}
```

Finally: containers

About Rust containers in general

Rust containers have some special properties:

About Rust containers in general

Rust containers have some special properties:

- We don’t want to allocate till we need this allocation. Allocations are costly, and usually, it is the last thing we want to do.

About Rust containers in general

Rust containers have some special properties:

- We don’t want to allocate till we need this allocation. Allocations are costly, and usually, it is the last thing we want to do.
- All safe methods try to return `Option` or `Result` where possible, thus containers are predictable and safe.

About Rust containers in general

Rust containers have some special properties:

- We don’t want to allocate till we need this allocation. Allocations are costly, and usually, it is the last thing we want to do.
- All safe methods try to return `Option` or `Result` where possible, thus containers are predictable and safe.
- Some methods `panic!` when some bad things happen (failed allocation, out-of-bounds in “operator `[]`”).

About Rust containers in general

Rust containers have some special properties:

- We don’t want to allocate till we need this allocation. Allocations are costly, and usually, it is the last thing we want to do.
- All safe methods try to return `Option` or `Result` where possible, thus containers are predictable and safe.
- Some methods `panic!` when some bad things happen (failed allocation, out-of-bounds in “operator `[]`”).
- We don’t have iterators like in C++, but we have references to elements.

About Rust containers in general

Rust containers have some special properties:

- We don't want to allocate till we need this allocation. Allocations are costly, and usually, it is the last thing we want to do.
- All safe methods try to return `Option` or `Result` where possible, thus containers are predictable and safe.
- Some methods `panic!` when some bad things happen (failed allocation, out-of-bounds in "operator `[]`").
- We don't have iterators like in C++, but we have references to elements.

These differences will **affect what algorithms and data structures we can use in the standard library** of Rust.

Fallible allocations

Interesting fact: since we panic on fallible allocations, we can’t use Rust in the Linux kernel currently. But the work being done.

- RFC 2116
- Email to Linus Torvalds about that

Vec[6]

Just implementation of default dynamic array. The same as in C++, not including some Rust-related differences. Asymptotics are the same as you might expect.

[4] Danila Kutenin blog (Russian). On pqsort, New sorting algorithm for LLVM libcxx
[5] GoingNative 2013 C++ Seasoning
[6] `Vec` documentation

Vec[6]

Just implementation of default dynamic array. The same as in C++, not including some Rust-related differences. Asymptotics are the same as you might expect.

Some basic algorithms. **They work on slices too**:

[4] Danila Kutenin blog (Russian). On pqsort, New sorting algorithm for LLVM libcxx
[5] GoingNative 2013 C++ Seasoning
[6] `Vec` documentation

`Vec`[6]

Just implementation of default dynamic array. The same as in C++, not including some Rust-related differences. Asymptotics are the same as you might expect.

Some basic algorithms. **They work on slices too**:

- `sort`, uses modified timsort, $O(N \log N)$.

[4]Danila Kutenin blog (Russian). On pqsort, New sorting algorithm for LLVM libcxx
[5]GoingNative 2013 C++ Seasoning
[6]`Vec` documentation

`Vec`[6]

Just implementation of default dynamic array. The same as in C++, not including some Rust-related differences. Asymptotics are the same as you might expect.

Some basic algorithms. **They work on slices too**:

- `sort`, uses modified timsort, $O(N \log N)$.
- `sort_unstable`, pdqsort[4], worst-case $O(N \log N)$ and best $O(N)$.

[4] Danila Kutenin blog (Russian). On pqsort, New sorting algorithm for LLVM libcxx
[5] GoingNative 2013 C++ Seasoning
[6] `Vec` documentation

`Vec`[6]

Just implementation of default dynamic array. The same as in C++, not including some Rust-related differences. Asymptotics are the same as you might expect.

Some basic algorithms. **They work on slices too**:

- `sort`, uses modified timsort, $O(N \log N)$.
- `sort_unstable`, pdqsort[4], worst-case $O(N \log N)$ and best $O(N)$.
- `binary_search`.

[4] Danila Kutenin blog (Russian). On pqsort, New sorting algorithm for LLVM libcxx
[5] GoingNative 2013 C++ Seasoning
[6] `Vec` documentation

`Vec`[6]

Just implementation of default dynamic array. The same as in C++, not including some Rust-related differences. Asymptotics are the same as you might expect.

Some basic algorithms. **They work on slices too**:

- `sort`, uses modified timsort, $O(N \log N)$.
- `sort_unstable`, pdqsort[4], worst-case $O(N \log N)$ and best $O(N)$.
- `binary_search`.
- `select_nth_unstable`, quick select with pdqsort, worst-case $O(N)$.

[4] Danila Kutenin blog (Russian). On pqsort, New sorting algorithm for LLVM libcxx
[5] GoingNative 2013 C++ Seasoning
[6] `Vec` documentation

`Vec`[6]

Just implementation of default dynamic array. The same as in C++, not including some Rust-related differences. Asymptotics are the same as you might expect.

Some basic algorithms. **They work on slices too**:

- `sort`, uses modified timsort, $O(N \log N)$.
- `sort_unstable`, pdqsort[4], worst-case $O(N \log N)$ and best $O(N)$.
- `binary_search`.
- `select_nth_unstable`, quick select with pdqsort, worst-case $O(N)$.
- There's `_by` and `_by_key` variants of functions to customize comparator.

[4] Danila Kutenin blog (Russian). On pqsort, New sorting algorithm for LLVM libcxx
[5] GoingNative 2013 C++ Seasoning
[6] `Vec` documentation

`Vec`[6]

Just implementation of default dynamic array. The same as in C++, not including some Rust-related differences. Asymptotics are the same as you might expect.

Some basic algorithms. **They work on slices too**:

- `sort`, uses modified timsort, $O(N \log N)$.
- `sort_unstable`, pdqsort[4], worst-case $O(N \log N)$ and best $O(N)$.
- `binary_search`.
- `select_nth_unstable`, quick select with pdqsort, worst-case $O(N)$.
- There's `_by` and `_by_key` variants of functions to customize comparator.
- Legendary[5] `rotate_left` and `rotate_right`.

[4] Danila Kutenin blog (Russian). On pqsort, New sorting algorithm for LLVM libcxx
[5] GoingNative 2013 C++ Seasoning
[6] `Vec` documentation

VecDeque[7]

Inside, it's just simple circular deque. Asymptotics are the same as you might expect. **Is not the same** as in C++.

[7] `VecDeque` documentation

VecDeque[7]

Inside, it’s just simple circular deque. Asymptotics are the same as you might expect. **Is not the same** as in C++.

Properties:

[7] VecDeque documentation

VecDeque[7]

Inside, it's just simple circular deque. Asymptotics are the same as you might expect. **Is not the same** as in C++.

Properties:

- The same functions as in `Vec`.

[7] `VecDeque` documentation

VecDeque[7]

Inside, it's just simple circular deque. Asymptotics are the same as you might expect. **Is not the same** as in C++.

Properties:

- The same functions as in `Vec`.
- And the same core differences from unsafe languages.

[7] `VecDeque` documentation

`VecDeque`[7]

Inside, it's just simple circular deque. Asymptotics are the same as you might expect. **Is not the same** as in C++.

Properties:

- The same functions as in `Vec`.
- And the same core differences from unsafe languages.
- Since it's simply circular deque, we have `make_contiguous` and `as_slices`. Moreover, we can easily use optimizations that rely on contiguous element storing, such as SIMD.

[7] `VecDeque` documentation

Do you remember `std::deque`?

In C++ standard, we require from `std::deque` not to invalidate iterators when modifying it. This leads to monstrous implementation.

Do you remember `std::deque`?

In C++ standard, we require from `std::deque` not to invalidate iterators when modifying it. This leads to monstrous implementation.

- Problem: We cannot use default circular deque since it invalidates iterators when reallocating.

Do you remember `std::deque`?

In C++ standard, we require from `std::deque` not to invalidate iterators when modifying it. This leads to monstrous implementation.

- Problem: We cannot use default circular deque since it invalidates iterators when reallocating.
- Solution: use circular deque, but instead of storing actual element we will store pointer to it.

Do you remember `std::deque`?

In C++ standard, we require from `std::deque` not to invalidate iterators when modifying it. This leads to monstrous implementation.

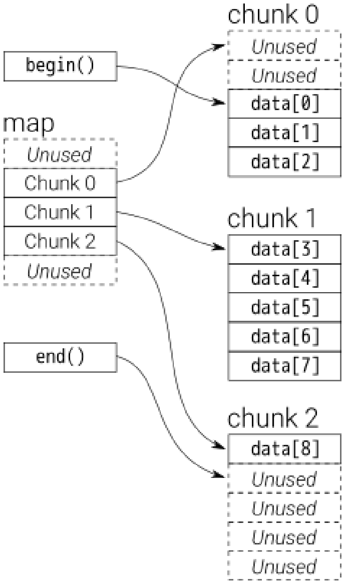
- **Problem**: We cannot use default circular deque since it invalidates iterators when reallocating.
- **Solution**: use circular deque, but instead of storing actual element we will store pointer to it.
- **Problem**: we are not cache local, the deque will be really slow.

Do you remember `std::deque`?

In C++ standard, we require from `std::deque` not to invalidate iterators when modifying it. This leads to monstrous implementation.

- **Problem**: We cannot use default circular deque since it invalidates iterators when reallocating.
- **Solution**: use circular deque, but instead of storing actual element we will store pointer to it.
- **Problem**: we are not cache local, the deque will be really slow.
- **Solution**: store chunks of elements in circular deque.

Do you remember `std::deque`?



Do you remember `std::deque`?

Although it's quite complex, it works not extremely slow in practice. It *may be* good price for having non-invalidating iterators.[8]

But don't forget: you won't be able to rely on contiguous element storing like in Rust!

[8] C++ benchmark - `std::vector` VS `std::list` VS `std::deque`

`BTreeMap`[9] and `BTreeSet`[10]

Rust also includes collections sorted by key: map and set.

[9] `BTreeMap` documentation
[10] `BTreeSet` documentation

`BTreeMap`[9] and `BTreeSet`[10]

Rust also includes collections sorted by key: map and set.

- There's a B-tree data structure inside. Thus it is cache-local and works fast on modern CPUs. Asymptotics for most operations are $O(\log_B N)$.

[9] `BTreeMap` documentation
[10] `BTreeSet` documentation

`BTreeMap`[9] and `BTreeSet`[10]

Rust also includes collections sorted by key: map and set.

- There's a B-tree data structure inside. Thus it is cache-local and works fast on modern CPUs. Asymptotics for most operations are $O(\log_B N)$.
- Lecturer's humble opinion: these names are much better than `std::map` or `std::set` since they shows what data structure it uses. B-tree, hash table, cartesian tree are also maps and sets!

[9] `BTreeMap` documentation
[10] `BTreeSet` documentation

`BTreeMap`[9] and `BTreeSet`[10]

Rust also includes collections sorted by key: map and set.

- There's a B-tree data structure inside. Thus it is cache-local and works fast on modern CPUs. Asymptotics for most operations are $O(\log_B N)$.
- Lecturer's humble opinion: these names are much better than `std::map` or `std::set` since they shows what data structure it uses. B-tree, hash table, cartesian tree are also maps and sets!
- **It is a logic error for a key to be modified in such a way that the key's ordering relative to any other key changes while it is in the map**. The behavior resulting from such a logic error **is not specified** but **will not be undefined behavior**.

[9] `BTreeMap` documentation
[10] `BTreeSet` documentation

Do you remember `std::map` and `std::set`?

In C++, we also have ordered map and set. Usually they are implemented using red-black trees.

Do you remember `std::map` and `std::set`?

In C++, we also have ordered map and set. Usually they are implemented using red-black trees.

- If B-tree data structure is really cool, why don't we use it in C++ standard library, downgrading to red-black trees?

Do you remember `std::map` and `std::set`?

In C++, we also have ordered map and set. Usually they are implemented using red-black trees.

- If B-tree data structure is really cool, why don't we use it in C++ standard library, downgrading to red-black trees?
- *Once again: iterator invalidation.*

Do you remember `std::map` and `std::set`?

In C++, we also have ordered map and set. Usually they are implemented using red-black trees.

- If B-tree data structure is really cool, why don't we use it in C++ standard library, downgrading to red-black trees?
- *Once again: iterator invalidation.*
- Since B-tree stores elements in chunks and have smaller depth, it way faster than regular BST.

Do you remember `std::map` and `std::set`?

In C++, we also have ordered map and set. Usually they are implemented using red-black trees.

- If B-tree data structure is really cool, why don’t we use it in C++ standard library, downgrading to red-black trees?
- *Once again: iterator invalidation.*
- Since B-tree stores elements in chunks and have smaller depth, it way faster than regular BST.
- This really hurts! C++ don’t have standard fast ordered map and set!

`HashMap`[12] and `HashSet`[13]

What a language without hast table? Rust have two: `HashMap` and `HashSet`. Asymptotics are predictable.

[11] CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"
[12] `HashMap` documentation
[13] `HashSet` documentation

`HashMap`[12] and `HashSet`[13]

What a language without hast table? Rust have two: `HashMap` and `HashSet`. Asymptotics are predictable.

- This hash table is quite literally the fastest universal hash table in the world currently existing. It uses quadratic probing and SIMD lookup inside.

[11] CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"
[12] `HashMap` documentation
[13] `HashSet` documentation

`HashMap`[12] and `HashSet`[13]

What a language without hast table? Rust have two: `HashMap` and `HashSet`. Asymptotics are predictable.

- This hash table is quite literally the fastest universal hash table in the world currently existing. It uses quadratic probing and SIMD lookup inside.
- More specifically, it's Rust port called Hashbrown of Google SwissTable written in C++. If you're interested in the algorithm, you can watch CppCon talk.[11]

---

[11] CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"
[12] `HashMap` documentation
[13] `HashSet` documentation

HashMap[12] and HashSet[13]

What a language without hast table? Rust have two: `HashMap` and `HashSet`. Asymptotics are predictable.

- This hash table is quite literally the fastest universal hash table in the world currently existing. It uses quadratic probing and SIMD lookup inside.
- More specifically, it's Rust port called Hashbrown of Google SwissTable written in C++. If you're interested in the algorithm, you can watch CppCon talk.[11]
- **It is a logic error for a key to be modified in such a way that the key's hash or its equality changes while it is in the map**. The behavior resulting from such a logic error **is not specified**, but **will not result in undefined behavior**.

[11] CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"

[12] `HashMap` documentation

[13] `HashSet` documentation

Do you remember `std::unordered_map` and `std::unordered_set`?

In C++, we also have unordered map and set. Usually they are implemented in some strange way.

Do you remember `std::unordered_map` and `std::unordered_set`?

In C++, we also have unordered map and set. Usually they are implemented in some strange way.

- Again: if Google Swiss table is really cool, why don't we use it in C++ standard library, donwgrading to some strange implementation?

Do you remember `std::unordered_map` and `std::unordered_set`?

In C++, we also have unordered map and set. Usually they are implemented in some strange way.

- Again: if Google Swiss table is really cool, why don't we use it in C++ standard library, donwgrading to some strange implementation?
- *And once more: iterator invalidation!* At least it prevents us to use open addressing.

Do you remember `std::unordered_map` and `std::unordered_set`?

In C++, we also have unordered map and set. Usually they are implemented in some strange way.

- Again: if Google Swiss table is really cool, why don't we use it in C++ standard library, donwgrading to some strange implementation?
- *And once more: iterator invalidation!* At least it prevents us to use open addressing.
- This leads to not suitable for production standard library hash tables.

`BinaryHeap`[14]

A priority queue implemented with a binary heap. **This will be a max-heap**. The same as in C++. Asymptotics are predictable.

[14] `BinaryHeap` documentation

`BinaryHeap`[14]

A priority queue implemented with a binary heap. **This will be a max-heap**. The same as in C++. Asymptotics are predictable.

- Lecturer's humble opinion: this name is much better than `std::priority_queue` since it shows what data structure it uses. For instance, binomial or fibonacci heaps are also priority queues!

[14] `BinaryHeap` documentation

`BinaryHeap`[14]

A priority queue implemented with a binary heap. **This will be a max-heap**. The same as in C++. Asymptotics are predictable.

- Lecturer's humble opinion: this name is much better than `std::priority_queue` since it shows what data structure it uses. For instance, binomial or fibonacci heaps are also priority queues!
- **It is a logic error for an item to be modified in such a way that the item's ordering relative to any other item changes while it is in a heap**. The behavior resulting from such a logic error **is not specified** but **will not be undefined behavior**.

[14] `BinaryHeap` documentation

`LinkedList`[16]

A doubly-linked list with owned nodes. The same as in C++. Asymptotics are predictable.

- You don't need it in almost any situation. It's slow and not memory efficient. Trust me.
- Since it's not C++, we don't have iterators pointing to elements. It's not convenient.
- Writing your list without unsafe is possible, but quite a challenge! Do it if you want to have a borrow checker as your best friend.[15]

[15] Learn Rust With Entirely Too Many Linked Lists
[16] `LinkedList` documentation

`String` and `&str`

`String`[17]

A Rust way to store a string. **Totaly differs** from C++ `std::string`.

[17] `String` documentation

`String`[17]

A Rust way to store a string. **Totaly differs** from C++ `std::string`.

- It's **UTF-8–encoded**.

[17] `String` documentation

`String`[17]

A Rust way to store a string. **Totaly differs** from C++ `std::string`.

- It's **UTF-8–encoded**.
- Growable like a `Vec`. It also made up of three components: a pointer to some bytes, a length, and a capacity. This even gives us many functions same to `Vec`.

[17] `String` documentation

`String`[17]

A Rust way to store a string. **Totaly differs** from C++ `std::string`.

- It's **UTF-8–encoded**.
- Growable like a `Vec`. It also made up of three components: a pointer to some bytes, a length, and a capacity. This even gives us many functions same to `Vec`.
- UTF-8 is a variable-width character encoding, so you cannot index it since it's UTF-8. To find N-th symbol, you should iterate over string, parsing code points.

[17] `String` documentation

String API

```rust
struct String {
    vec: Vec<u8>,
}

impl String {
    fn new() -> String;
    fn with_capacity(capacity: usize) -> String;
    fn from_utf8(vec: Vec<u8>) -> Result<String, FromUtf8Error>;
    fn from_utf16(v: &[u16]) -> Result<String, FromUtf16Error>;
    fn into_bytes(self) -> Vec<u8>;
    fn as_bytes(&self) -> &[u8];
}
```

What will this code print?

```rust
let s = String::from("привет");
println!("{}", s.len());
```

String in depth

What will this code print?

```
let s = String::from("привет");
println!("{}", s.len());
```

This outputs 12, since `.len()` gives count of *bytes* in string.

String in depth

To work directly with a string like in C++, you must convert it to `Vec<char>`

```
let s = String::from("привет");
let t = s.chars().collect::<Vec<_>>();
println!("{:?}", t); // ['п', 'р', 'и', 'в', 'е', 'т']
```

`.chars()` is function that creates iterator over chars of the string.

The `char` type

The `char` type represents a single character.

More specifically, since “character” isn’t a well-defined concept in Unicode, `char` is a “**Unicode scalar value**”, which is similar to, but not the same as, a “**Unicode code point**”.

char type

```
let mut chars = "é".chars();
// U+00e9: 'latin small letter e with acute'
assert_eq!(Some('\u{00e9}'), chars.next());
assert_eq!(None, chars.next());

let mut chars = "é".chars();
// U+0065: 'latin small letter e'
assert_eq!(Some('\u{0065}'), chars.next());
// U+0301: 'combining acute accent'
assert_eq!(Some('\u{0301}'), chars.next());
assert_eq!(None, chars.next());
```

`char` type

The size of `char` is always 4 bytes:

```
assert_eq!(std::mem::size_of::<char>(), 4);
```

`&str`

`&str` is a slice type of `String`, similar to `std::string_view`. Just like:

```
let vec = vec![1, 2, 3, 4];
let vec_slice = &vec[1..3]; // &[2, 3]
let s = String::from("hello");
let s_slice = &s[1..3]; // "el"
```

&str

Ok, let’s take a UTF-8 slice!

```
let s = String::from("привет");
let s_slice = &s[1..3];
```

&str

Ok, let's take a UTF-8 slice!

```
let s = String::from("привет");
let s_slice = &s[1..3];
// thread 'main' panicked at 'byte index 1 is
// not a char boundary; it is inside 'п'
// (bytes 0..2) of `привет`'
```

&str

Ok, let's take a UTF-8 slice!

```
let s = String::from("привет");
let s_slice = &s[1..3];
// thread 'main' panicked at 'byte index 1 is
// not a char boundary; it is inside 'п'
// (bytes 0..2) of `привет`'
```

That means `&str` also have a UTF-8 invariant checked at runtime.

&str

As a string slice, `&str` have most functions `String` have:

```
fn as_bytes(&self) -> &[u8];
fn chars(&self) -> Chars<'_>;
fn trim(&self) -> &str;
fn split<'a, P>(&'a self, pat: P) -> Split<'a, P>;
fn replace<'a, P>(&'a self, from: P, to: &str) -> String;
// And so on
```

&str

All string constants are `&str`.

```rust
let s: &str = "Hello world!";
let t1 = s.to_string();
let t2 = s.to_owned(); // The same as t1
```

Box and Rc

Box

We are already familiar with **Box** type. Let's check one advanced function:

```
fn leak<'a>(b: Box<T, A>) -> &'a mut T;
fn into_raw(b: Box<T, A>) -> *mut T;
```

Example:

```
let x = Box::new(41);
let static_ref: &'static mut usize = Box::leak(x);
*static_ref += 1;
assert_eq!(*static_ref, 42);
```

Box

But stop! Rust is the safe language, no memory unsafety, no undefined behavior, what's wrong!?

Box

But stop! Rust is the safe language, no memory unsafety, no undefined behavior, what's wrong!?

In reality, when you're creating global objects or interacting with other languages, you ***have to*** *leak objects. Moreover, it's* ***safe*** *to leak memory, just not good!*

Rc

Rc is single-threaded reference-counting pointer. “Rc” stands for “Reference Counted”.

```
let rc = Rc::new(());
let rc2 = rc.clone(); // Clones Rc, not what inside!
let rc3 = Rc::clone(&rc); // The same
```

Rc is dropped when all instances of Rc are dropped.

Primary functions:

```
fn get_mut(this: &mut Rc<T>) -> Option<&mut T>;
fn downgrade(this: &Rc<T>) -> Weak<T>;
fn weak_count(this: &Rc<T>) -> usize;
fn strong_count(this: &Rc<T>) -> usize;
```

Rc

References to the variable inside `Rc` are controlled at runtime:

```
let mut rc = Rc::new(42);
println!("{}", *rc);

*Rc::get_mut(&mut rc).unwrap() -= 41;
println!("{}", *rc);

let mut rc1 = rc.clone();
println!("{}", *rc1);
// thread 'main' panicked at 'called `Option::unwrap()`
// on a `None` value'
// *Rc::get_mut(&mut rc1).unwrap() -= 1;
```

`get_mut` guarantees that it will return mutable reference only if there's only one pointer. If there are more, you won't have a chance to modify `Rc`.

Weak

`Rc` is a **strong** pointer, while `Weak` is a **weak** pointer. Both of them have *ownership over allocation*, but only `Rc` have *ownership over the value inside*:

You can upgrade `Weak` to `Rc`:

```rust
fn upgrade(&self) -> Option<Rc<T>>;
```

Weak

```rust
let rc1 = Rc::new(String::from("string"));
let rc2 = rc1.clone();
let weak1 = Rc::downgrade(&rc1);
let weak2 = Rc::downgrade(&rc1);
drop(rc1); // The string is not deallocated
assert!(weak1.upgrade().is_some());
drop(weak1); // Nothing happens
drop(rc2); // The string is deallocated
assert_eq!(weak2.strong_count(), 0);
// If no strong pointers remain, this will return zero.
assert_eq!(weak2.weak_count(), 0);
assert!(weak2.upgrade().is_none());
drop(weak2); // The Rc is deallocated
```

`Arc`

There’s also `Arc` - a thread-safe reference-counting pointer. `Arc` stands for “Atomically Reference Counted”.

We’ll need it to share data safely across threads in the future.

Conclusion

- We studied API of Option and Result.
- Get acquainted with default collections.
- Learned about smart pointers in Rust.