Lecture 1: Hello, Rust!

Alexander Stanovoy

February 1, 2022

alex.stanovoy@gmail.com

What is this course about?

The Rust programming language, its basic and advanced features.

- Basics: syntax, collections, traits...
- Some nightly features, such as trait specialization.
- Parallel and concurrent computing.
- Metaprogramming.
- Tooling around language.
- System safety.

Prerequisites

- Solid knowledge of C++, including C++20 standard.
- Understanding of parallel computing and concurrency.
- Comprehension of computer architecture and operating systems.
- Passion for backend engineering.

Are you ready? :)

The creation of Rust: Short history background

A brief history of programming languages

Machine codes

Pros:

- Direct execution of code written by engineer.
- Don’t require any runtime.

Cons:

- Easy to make a mistake when ”typing”.
- Very hard to read.
- Difficult to debug.
- Hard to reuse existing code.
- Requires a lot of time to write even a simple program.
- Code changes from machine to machine.

A brief history of programming languages

Assembly language

Pros:

- Direct execution of the code written by an engineer.
- Don’t require any runtime.
- Not possible to type a nonexistent command or write an incorrect offset.
- Mnemonics are easier to read than machine codes.

Cons:

- Difficult to debug.
- Still hard to write complex logic.
- Hard to reuse existing code.
- Requires a lot of time to write even a simple program.
- Usually still requires code changes from machine to machine.

A brief history of programming languages

C language

Pros:

- Don't require any runtime.
- Provides a thin layer of abstraction, allowing to write complex but still readable code: structures, functions, pointers...
- Abstraction over hardware.

Cons:

- Segfaults, buffer overflows, null pointers, data races, **undefined behavior**...
- Not very productive lacks syntax sugar.
- No unified build system and dependency management.

Remember dreadful undefined behaviour?

CVE-2008-0166

Bug in `glibc` that resulted in vulnerability in OpenSSL.

- `srandom()` - set seed for non-cryptographic pseudorandom number generator.
- If read from `/dev/random` failed, the following code is executed:

```
struct timeval tv;
unsigned long junk;
gettimeofday(&tv, NULL);
srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```

Remember dreadful undefined behaviour?

CVE-2008-0166

Bug in `glibc` that resulted in vulnerability in OpenSSL.

- `srandom()` - set seed for non-cryptographic pseudorandom number generator.
- If read from `/dev/random` failed, the following code is executed:

```
struct timeval tv;
unsigned long junk;
gettimeofday(&tv, NULL);
srandom(~~(getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec~~ ^ junk);
```

One day, the compiler decided to remove everything except junk :D

Remember dreadful memory unsafety?

Memory unsafety

- Code has full access to memory of the whole process.
- Side effects can be unpredictable.

```
char line[1024];
ssize_t len = getline(&line, sizeof(line), stdin);
```

Remember dreadful memory unsafety?

A story about `memcpy`

```
void *memcpy(void *dest, const void *src, size_t n);
```

Remember dreadful memory unsafety?

A story about `memcpy`

```
void *memcpy(void *dest, const void *src, size_t n);
```

The `memcpy()` function copies `n` bytes from memory area `src` to memory area `dest`. The memory areas must not overlap.

Remember dreadful memory unsafety?

A story about `memcpy`

```
void *memcpy(void *dest, const void *src, size_t n);
```

The `memcpy()` function copies `n` bytes from memory area `src` to memory area `dest`. The memory areas must not overlap.

- Summer 2010: software developers from Intel added a patch to `glibc`. Now `memcpy()` sometimes can copy from right to left.
- September 29, 2010: Fedora Linux user found a bug - Flash Player cracks instead of playing sound.

Remember dreadful memory unsafety?

A story about `memcpy`: Who is at fault?

The only stupidity is crap software violating well known rules that have existed forever.

Andreas Schwab, `glibc` *developer*

You can call it "crap software" all you like, but the thing is, if `memcpy` doesn't warn about overlaps, there's no test coverage, and in that case, even well-designed software will have bugs.

Linus Torvalds

A brief history of programming languages

C++ language

Pros:

- Don’t require any runtime.
- Provides a thick layer of abstraction, allowing to write complex readable code: classes, templates, lambda functions, iterators...
- Abstraction over hardware and memory due to OOP.

Cons:

- Segfaults, buffer overflows, null pointers, data races, **undefined behaviour**...
- Cluttered standard with a lot of gaps due to previous decisions.
- No unified build system and dependency management.

Remember dreadful undefined behaviour?

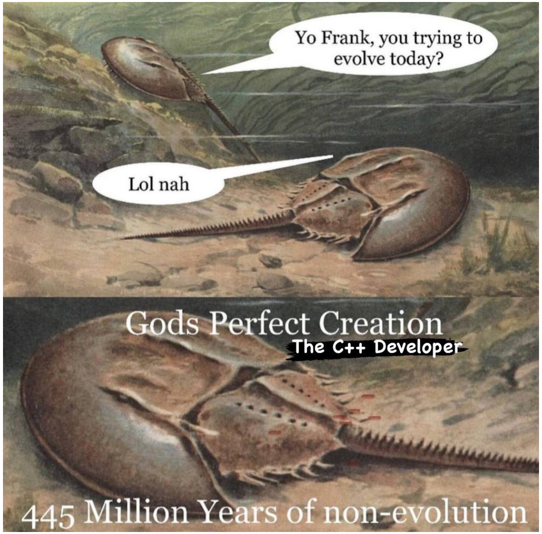
An example of regular C++'s UB

```
std::vector<std::string> createStrings() {
  // Some code
}

for (char c : createStrings().at(0)) {
  // undefined behaviour
}
```

I love C++!

Fun fact: it will not be fixed until C++26 because the proposal in C++23 was rejected.


Yo Frank, you trying to evolve today?
Lol nah
Gods Perfect Creation
The C++ Developer
445 Million Years of non-evolution

A brief history of programming languages

In the meantime: languages such as garbage collection.

Java, C# and similar languages

Pros:

- Require runtime but execution is quite fast due to JIT.
- In general, provide thick layer of abstraction.
- Run code everywhere where runtime exists.
- Safer than C, C++ or similar languages, but implementation-defined and unspecified behavior exist.

A brief history of programming languages

In the meantime: languages with garbage collection.

Java, C# and similar languages

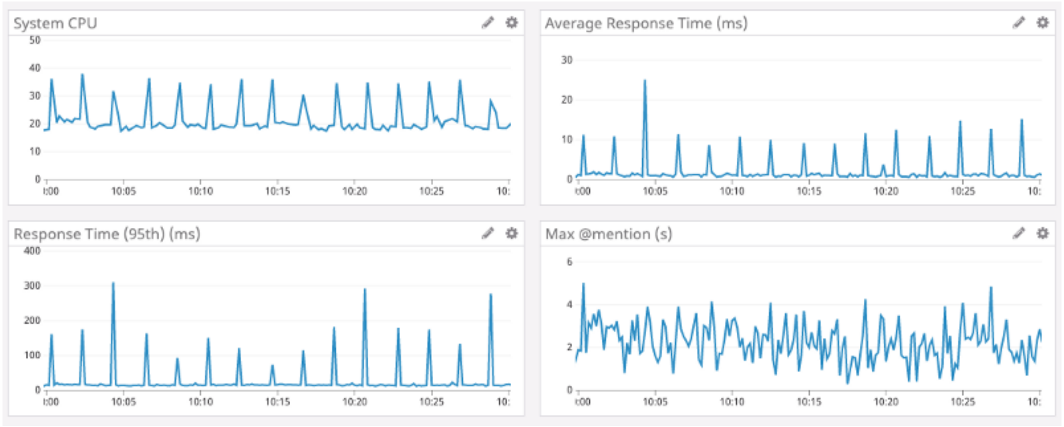
Cons:[1]

- Expensive: no matter what type of garbage collection is used, there will always be nontrivial memory overhead.
- Disruptive: drop what you're doing - it's time for GC!
- Non-deterministic: when will the next GC pause be? Who knows! It depends on how much memory is being used.

[1] CS110L Lecture 1 Slides.

Garbage Collection cons

Why Discord is swithing from Go to Rust?



Garbage Collection cons

Eliminating Large JVM GC Pauses Caused by Background IO Traffic

In our production environments, we have seen unexplainable significant STW pauses (> 5 seconds) in our mission-critical Java applications.

LinkedIn Engineering

A brief history of programming languages

In the meantime: languages with garbage collection.

Java, C# and similar languages

Cons:

- Precludes manual optimization: in some situations, you may want to structure your data in memory in a specific way to achieve high cache performance. GC can't know how you will use memory, so it optimizes for the average use case.
- Have poor interoperability.
- **Are not suitable for system programming.**[2]

[2] Actually suitable, but GC limits use cases.

A brief history of programming languages

What is system programming?

System programming - the process of creation of programs that communicate with hardware and other programs, not users.

Examples:

- Operating systems, firmware.
- Databases: PostgreSQL, MySQL
- Virtual machines: JVM, CLR.
- Browsers and their engines: Blink, WebKit, Servo.

A brief history of programming languages

What is system programming?

System programming - the process of creation of programs that communicate with hardware and other programs, not users.

Usual properties:

- Fast execution time with stable latency.
- High sensitivity to bugs.
- High security.

At the same time, we try to pursue two goals: performance and safety.

A brief history of programming languages

Why is C still used for system programming?

- Assembly is ”strictly better” than Machine codes.
- C is nearly ”strictly better” than Assembly.
- C++ **is not** ”strictly better” than C.
- Languages with runtime **is not** ”strictly better” than C.

Will there be a language ”strictly better” than C?

A brief history of programming languages

Rust language

Pros:

- Don’t require any runtime.
- Provides a thick layer of abstraction, allowing to write complex readable code: structures, generics, traits, closures, iterators…
- **No memory unsafety and undefined behavior.**[3]
- Modern standard without any[4] incorrect decisions.
- Unified build system and dependency management.

According to Microsoft and Chromium, 70% of bugs involve memory unsafety.

[3] Unless you or your dependencies use `unsafe` incorrectly.
[4] There exist not good decisions.

A brief history of programming languages

Rust language

Cons:

- Difficult to use and especially learn.
- Compiled, thus no such simple coding as in, for instance, Python.
- If you’re using a lot of optimizations or writing software that frequently works with pointers, you’ll have to use **a lot** of `unsafe`. It may be inconvenient.

What is safety after all?

Let’s take a look at C code of simple vector and find some bugs on it to intuitively understand the Rust’s features.

What is safety after all?

Dangling pointers

```
Vec* vec_new() {
    Vec vec;
    vec.data = NULL;
    vec.length = 0;
    vec.capacity = 0;
    return &vec;
}
```

What is safety after all?

Dangling pointers

```
Vec* vec_new() {
    Vec vec;
    vec.data = NULL;
    vec.length = 0;
    vec.capacity = 0;
    return &vec;
}
```

Wouldn’t it be nice if the compiler realized that vec ”lives” within those two curly braces, and therefore its address shouldn’t be returned from the function?

What is safety after all?

Double frees

```
void main() {
    Vec *vec = vec_new();

    /* ... */

    free(vec->data);
    vec_free(vec);
}
```

What is safety after all?

Double frees

```
void main() {
    Vec *vec = vec_new();

    /* ... */

    free(vec->data);
    vec_free(vec);
}
```

Wouldn't it be nice if the compiler enforced that free is called on a variable, that variable can no longer be used?

What is safety after all?

Iterator Invalidation

```c
void main() {
    /* ... */

    int *n = &vec->data[0];
    vec_push(vec, 110);
    printf("%d\n", *n);

    /* ... */
}
```

What is safety after all?

Iterator Invalidation

```
void main() {
    /* ... */

    int *n = &vec->data[0];
    vec_push(vec, 110);
    const char* format = "%d\n";
    printf(format, *n); // I'm sorry for 'format'

    /* ... */
}
```

Wouldn’t it be nice if the compiler stopped us from modifying the data `n` was pointing to (as it does in `vec_push`)?

What is safety after all?

Memory leaks

```
void vec_push(Vec *vec, int n) {
    if (vec->length == vec->capacity) {
        /* ... */

        vec->data = new_data;
        vec->capacity = new_capacity;
    }

    /* ... */
}
```

What is safety after all?

Memory leaks

```
void vec_push(Vec *vec, int n) {
    if (vec->length == vec->capacity) {
        /* ... */

        vec->data = new_data;
        vec->capacity = new_capacity;
    }

    /* ... */
}
```

Wouldn't it be nice if the compiler noticed when a piece of heap data no longer had anything pointing to it? (and so then could easily be freed?)

Is Rust safe?

Rust is theoretically proven to be safe.

Understanding and Evolving the Rust Programming Language, Ralf Jung, August 2020.

Awards:

- 2021 Otto Hahn Medal.
- Honorable Mention for the 2020 ACM Doctoral Dissertation Award.
- 2021 ETAPS Doctoral Dissertation Award.

Is Rust safe?

RustBelt - formal model of Rust that includes core conceptions of language (borrowing, lifetimes, lifetime inclusion).

- Proof of safety of Safe[5] Rust.
- Definition of sufficiency conditions for every type `T` to consider it safe abstraction.
- Proof of soundness using Coq + Iris: `Cell`, `RefCell`, `thread::spawn`, `rayon::join`, `Mutex`, `RwLock`, `Arc`.

[5]There exists Unsafe Rust, but we will return to it later.

A brief history of Rust[6]

1. **(2006-2010)** Started at Mozilla by Graydon Hoare as a personal project.
2. **(2010-2012)** Rust is now a Mozilla project.
   - The team slowly grows allowing Rust to grow faster.
   - The aim is to make language that can catch critical mistakes before code even compiles.
3. **(2012-2014)** Rust improves type system.
   - To make the language safe, the team thought they need a garbage collector, but they figured out they don't need it: everything can be done in the level of type system!
   - Birth of Cargo - the Rust package manager. Influenced by `ruby` and `npm`.
4. **(2014-present)** Rust grows!

[6]The History of Rust talk.

Is it popular of just a niche language?

Google

- Pushing Rust to Linux Kernel.
- Developing new OS Fuchsia in Rust.[7]
- Enabled support of Rust in Android.

Meta [8]

- Mononoke - version control system.
- Diem - blockchain.
- Metaverse - virtual reality.

[7] Count of lines of code in different languages.
[8] A brief history of Rust at Facebook

Is it popular of just a niche language?

Amazon[9]

- Hired core developers of Tokio (the most popular framework for async).
- Firecracker - open source virtualization technology.
- Bottlerocket OS - open-source Linux-based operating system meant for hosting containers.
- Nitro - compute environments; underlying platform for Amazon EC2.

Microsoft

- Rewrited Windows component in Rust.
- Official Rust WinAPI wrapper.

[9] How our AWS Rust team will contribute to Rust's future successes.

Why companies sometimes do not use Rust?

- Already wrote a lot of code in another language.
- Company's internal tools do not support Rust, and maintenance could be costly.
- In a big company, you should have your committee to help support language in the company.
- Hard to find developers in such a difficult and fresh language.
- Rust developers are usually talented people with high salary expectations.

Why companies sometimes do not use Rust?

- Already wrote a lot of code in another language.
- Company's internal tools do not support Rust, and maintenance could be costly.
- In a big company, you should have your committee to help support language in the company.
- Hard to find developers in such a difficult and fresh language.
- Rust developers are usually talented people with high salary expectations.

At the same time, startups and small companies do not hesitate to use the language.

**Silly blockchain joke*.*

Finally some Rust

First of all: Hello, World!

How to write Hello World in Rust?

```rust
fn main() {
    println!("Hello, World!");
}
```

First of all: Hello, World!

How to write Hello World in Rust?

```rust
fn main() {
    println!("Hello, World!");
}
```

```
$ rustc main.rs  # no optimizations
$ ./main
Hello, World!
```

First of all: Hello, World!

How to write Hello World in Rust: assembly edition.

```
#![no_main]

#[link_section=".text"]
#[no_mangle]
pub static main: [u32; 9] = [
    3237986353,
    3355442993,
    120950088,
    822083584,
    252621522,
    1699267333,
    745499756,
    1919899424,
    169960556,
];
```

First of all: defining variables

Integer variable types:

Bits count	8	16	32	64	128	32/64
Signed	`i8`	`i16`	`i32`	`i64`	`i128`	`isize`
Unsigned	`u8`	`u16`	`u32`	`u64`	`u128`	`usize`

`usize` - size of the pointer.

First of all: defining variables

To define a variable, use `let` keyword:

```rust
let idx: usize = 92;
```

Literals:

```rust
let y = 92_000_000i64;
let hex_octal_bin = 0xffff_ffff + 0o777 + 0b1;
```

In Rust there's **type inference**. For integer type, the default type is `i32`.

Variables are **immutable** by default. To make a variable mutable, use `mut` keyword:

```rust
let mut idx: usize = 0x1022022;
```

First of all: defining variables

Floats and bools: `f32`, `f64` and `bool`.

In Rust, `bool` can have only two values: `true` and `false`:

```rust
let mut x = true;
x = false;
// error: expected `bool`, found integer!
// x = 1;
```

At the same time, it's 1 byte in memory (will be important later).

Arithmetic

- Basic arithmetic: `+`, `-`, `*`, `/`, `%`
- `/`, `%` round to 0.

```
let (x, y) = (15, -15);
let (a1, b1) = (x / -4, x % -4);
let (a2, b2) = (y /  4, y %  4);
// outputs "-3 3 and -3 -3"
println!("{a1} {b1} and {a2} {b2}");
```

- Bitwise and logical operations `!`, `<<`, `>>`, `|`, `&`
- No `++`
- Functions `(-92i32).abs()`, `0b001100u8.count_ones()`, `2i32.pow(10)` ans so on.
- Full list of operators here.

Type casting

In Rust, there’s no type implicit casting:

```
let x: u16 = 1;

// error: mismatched types
// let y: u32 = x;

// Just filling leading bits with zeros
let z: u16 = y as u16;
let y: u32 = x.into();

// Cutting leading zeros
let to_usize = 92u64 as usize;
let from_usize = 92usize as u64;
```

`as` - explicit casting operator.

Type casting

Note: Casting is not transitive, that is, even if:

```
e as U1 as U2
```

Is a valid expression, the expression:

```
e as U2
```

It is not necessarily so.

Overflow

Overflow is a programmer's mistake. In debug build vs in release build:

```
fn main() {
    let x = i32::max_value();
    let y = x + 1;
    println!("{}", y);
}
```

```
$ cargo run
thread 'main' panicked at 'attempt to add with overflow',
main.rs:3:13

$ cargo run --release
-2147483648
```

Explicit arithmetic

```rust
let x = i32::MAX;

let y = x.wrapping_add(1);
assert_eq!(y, i32::MIN);

let y = x.saturating_add(1);
assert_eq!(y, i32::MAX);

let (y, overflowed) = x.overflowing_add(1);
assert!(overflowed);
assert_eq!(y, i32::MAX)

match x.checked_add(1) {
    Some(y) => unreachable!(),
    None => println!("overflowed"),
}
```

Floating point

```rust
let y = 0.0f32; // Litaral f32
let x = 0.0;    // Default value

// Point is necessary
// error: expected f32, found integer variable
// let z: f32 = 0;
let z = 0.0f32;

let not_a_number = std::f32::NAN;
let inf = std::f32::INFINITY;

// Wow, so many functions!
8.5f32.ceil().sin().round().sqrt()
```

Boolean

```
let to_be = true;
let not_to_be = !to_be;
let the_question = to_be || not_to_be;
```

`&&` and `||` are lazy.

Tuple

```
let pair: (f32, i32) = (0.0, 92);
let (x, y) = pair;
// The same as this
// Note the shadowing!
let x = pair.0;
let y = pair.1;

let void_result = println!("hello");
assert_eq!(void_result, ());

let trailing_comma = (
    "Archibald",
    "Buttle",
);
```

More on shadowing

What is the output of this code?

```rust
let x = 10;
for i in 0..5 {
    if x == 10 {
        println!("{i}");
        let x = 12;
    }
}
```

More on shadowing

What is the output of this code?

```
let x = 10;
for i in 0..5 {
    if x == 10 {
        let x = 12;
        println!("{i} {x}");
    }
}
// This code outputs 0 10\n1 10\n2 10\n3 10\n4 10\n
```

More on shadowing

Drop is compiler optimization. Godbolt.

```rust
pub fn shadowing(num: i32) -> i32 {
    let vec = vec![0, 1, 2, 3];
    let vec = vec![4, 5, 6, 7];
    vec[0]
}
```

Tuple

```rust
// Zero element tuple, or Unit
let x = ();
let y = {};
assert!(x == y); // OK

// One element tuple
let x = (42,);
```

Tuple

In memory, tuple is stored continuously.

`7`	`07 00 00 00`
`(7, 263)`	`07 00 00 00 07 01 00 00`

Tuple

Tuple is a zero-cost abstraction!

```rust
let t = (92,);
// 0x7ffc6b2f6aa4
println!("{:?}", &t as *const (i32,));
// 0x7ffc6b2f6aa4
println!("{:?}", &t.0 as *const i32);
```

Meanwhile in Python:

```python
t = (92,)
print(id(t))      # 139736506707248
print(id(t[0]))   # 139736504680928
```

Tuple

Tuple is a zero-cost abstraction!

```
let t = (92,);
// 0x7ffc6b2f6aa4
println!("{:?}", &t as *const (i32,));
// 0x7ffc6b2f6aa4
println!("{:?}", &t.0 as *const i32);
```

Meanwhile in Python:

```
t = (92,)
print(id(t))     # 139736506707248
print(id(t[0]))  # 139736504680928
```

Array

```
let xs: [u8; 3] = [1, 2, 3];
assert_eq!(xs[0], 1);    // index -- usize
assert_eq!(xs.len(), 3); // len() -- usize

let mut buf = [0u8; 1024];
```

The size of an array is a constant known at compile-time and the part of the type.

References

- Is really a pointer in compiled program.
- Cannot be `NULL`.
- Guaranties that the object is alive.
- There are & and &mut references.

```
let mut x: i32 = 92;
let r: &mut i32 = &mut 92; // Reference created explicitly
*r += 1;                   // Explicit dereference
```

References

In C++ we have to use `std::reference_wrapper` to store a reference in a vector:

```
int x = 10;
std::vector<std::reference_wrapper<int>> v;
v.push_back(x);
```

In Rust, references are a first class objects so we can push them to vector directly:

```
let x = 10;
let mut v = Vec::new();
v.push(&x);
```

Pointers

- Useless without unsafe, because you cannot dereference it.
- Can be NULL.
- Does not guarrantee that the object is alive.
- **Very rarely needed.** Examples: FFI, some data structures, optimizations...

```
let x: *const i32 = std::ptr::null();
let mut y: *const i32 = std::ptr::null();
let z: *mut i32 = std::ptr::null_mut();
let mut t: *mut i32 = std::ptr::null_mut();
```

In Rust, we read type names from left to right, not from right to left like in C++:

```
uint32_t const * const x = nullptr;
uint32_t const * y = nullptr;
uint32_t* const z = nullptr;
uint32_t* t = nullptr;
```

Box

- Pointer to some data on the heap.
- Pretty like C++'s `std::unique_ptr`, but without `NULL`

```rust
let x: Box<i32> = Box::new(92);
```

Functions

Functions are defined via `fn` keyword. Note the expressions and statements!

```
fn func1() {}
fn func2() -> () {}
fn func3() -> i32 {
    0
}
fn func4(x: u32) -> u32 {
    return x;
}
fn func5(x: u32, mut y: u64) -> u64 {
    y = x as u64 + 10;
    return y
}
fn func6(x: u32, mut y: u64) -> u32 {
    x + 10
}
```

More on expressions and statements

This works in any other scope, for instance in `if`'s:

```
let y = 42;
let x = if y < 42 {
    345
} else {
    y + 534
}
```

Structures

Structures are defined via `struct` keyword:

```
struct Example {
    oper_count: usize,
    data: Vec<i32>, // Note the trailing comma
}
```

Rust **do not** give any guarantees about memory representation by default. Even these structures can be different in memory!

```
struct A {
    x: Example,
}

struct B {
    y: Example,
}
```

Structures

Let's add new methods to `Example`:

```
impl Example {
    // Associated
    pub fn new() -> Self {
        Self {
            oper_count: 0,
            data: Vec::new(),
        }
    }

    pub fn push(&mut self, x: i32) {
        self.oper_count += 1;
        self.data.push(x)
    }

    /* Next slide */
}
```

Structures

Let's add new methods to `Example`:

```
impl Example {
    /* Previous slide */

    pub fn oper_count(&self) -> usize {
        self.oper_count
    }

    pub fn eat_self(self) {
        println!("later on lecture :)")
    }
}
```

Note: you can have multiple `impl` blocks.

Structures

Initialize a structure and use it:

```
let mut x = Example {
    oper_count: 0,
    data: Vec::new(),
};
let y = Example::new();
x.push(10);
assert_eq!(x.oper_count(), 1);
```

Simple example of generics

What about being *generic* over arguments?

```
struct Example<T> {
    oper_count: usize,
    data: Vec<T>,
}
```

Simple example of generics

What about being *generic* over arguments?

```
impl<T> Example<T> {
    pub fn new() -> Self {
        Self {
            oper_count: 0,
            data: Vec::new(),
        }
    }

    pub fn push(&mut self, x: T) {
        self.oper_count += 1;
        self.data.push(x)
    }

    /* The rest is the same */
}
```

Simple example of generics

Initialize a structure and use it:

```
let mut x = Example<i32> {
    oper_count: 0,
    data: Vec::new(),
};
let y = Example::<i32>::new(); // ::<> called 'turbofish'
let z: Example<i32> = Example {
    oper_count: 0,
    data: Vec::new(),
};
let z: Example::<i32> = Example { // Wow, this works too?!
    oper_count: 0,
    data: Vec::new(),
};
x.push(10);
assert_eq!(x.oper_count(), 1);
```

Conditions and loops: `if`, `while`, `for`, `loop`

```rust
let mut x = 2;
if x == 2 { // No braces in Rust
    x += 2;
}
while x > 0 { // No braces too
    x -= 1;
    println!("{x}");
}
```

Conditions and loops: `if`, `while`, `for`, `loop`

```
loop { // Just loop until 'return', 'break' or never return.
    println!("I'm infinite!");
    x += 1;
    if x == 10 {
        println!("I lied...");
        // Note no ;
        break
        // But this will go too
        // break;
    }
}
```

Conditions and loops: `if`, `while`, `for`, `loop`

In Rust, we can break with a value from `while` and `loop`!

```
let mut counter = 0;
let result = loop {
    counter += 1;
    if counter == 10 {
        break counter * 2;
    }
};
assert_eq!(result, 20);
```

Default `break` is just `break ()`.

Conditions and loops: `if`, `while`, `for`, `loop`

Or break on outer `while`, `for` or `loop`:

```rust
'outer: loop {
    println!("Entered the outer loop");
    'inner: for _ in 0..10 {
        println!("Entered the inner loop");

        // This would break only the inner loop
        // break;

        // This breaks the outer loop
        break 'outer;
    }
    println!("This point will never be reached");
}
println!("Exited the outer loop");
```

Conditions and loops: `if`, `while`, `for`, `loop`

Time for `for` loops!

```rust
for i in 0..10 {
    println!("{i}");
}
for i in 0..=10 {
    println!("{i}");
}
for i in [1, 2, 3, 4] {
    println!("{i}");
}
```

Conditions and loops: `if`, `while`, `for`, `loop`

Time for `for` loops!

```rust
let vec = vec![1, 2, 3, 4];
for i in &vec { // By reference
    println!("{i}");
}
for i in vec { // Consumes vec; will be discussed later
    println!("{i}");
}
```

Enumerations

Enumerations are one of the best features in Rust :)

```
enum MyEnum {
    First,
    Second,
    Third, // Once again: trailing comma
}
enum OneMoreEnum<T> {
    Ein(i32),
    Zwei(u64, Example<T>),
}

let x = MyEnum::First;
let y: MyEnum = MyEnum::First;
let z = OneMoreEnum::Zwei(42, Example::<usize>::new());
```

Enumerations: Option and Result

In Rust, there's two important enums in `std`, used for error handling:

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

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

We will discuss them a bit later in the course.

Match

`match` is one of things that will help you to work with `enum`.

```rust
let x = MyEnum::First;
match x {
    MyEnum::First => println!("First"),
    MyEnum::Second => {
        for i in 0..5 { println!("{i}"); }
        println!("Second");
    },
    _ => println!("Matched something!"),
}
```

The _ symbol

- _ matches everything in `match` (called wildcard).
- Used for inference sometimes:

```rust
// Rust does not know here to what type
// you want to collect
let mut vec: Vec<_> = (0..10).collect();
vec.push(42u64);
```

- And to make a variable unused:

```rust
let _x = 10;
// No usage of _x, no warnings!
```

Match

`match` can match multiple objects at a time:

```rust
let x = OneMoreEnum::<i32>::Ein(2);
let y = MyEnum::First;
match (x, y) {
    (OneMoreEnum::Ein(x), MyEnum::First) => {
        println!("Hello!");
    },
    // Destructuring
    (OneMoreEnum::Zwei(a, _), _) => println!("{a}"),
    _ => println!("ooooof!"),
}
```

Match

There's feature to match different values with same code:

```
let number = 13;
match number {
    1 => println!("One!"),
    2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
    13..=19 => println!("A teen"),
    _ => println!("Ain't special"),
}
```

Match

And we can apply some additional conditions called guards:

```
let pair = (2, -2);
println!("Tell me about {:?}", pair);
match pair {
    (x, y) if x == y => println!("These are twins"),
    // The ^ `if condition` part is a guard
    (x, y) if x + y == 0 => println!("Antimatter, kaboom!"),
    (x, _) if x % 2 == 1 => println!("The first one is odd"),
    _ => println!("No correlation..."),
}
```

Match

Match is an expression too:

```
let boolean = 13;
let x = match boolean {
    13 if foo() => 0,
    // You have to cover all of the possible cases
    13 => 1,
    _ => 2,
};
```

Match

Ignoring the rest of the tuple:

```
let triple = (0, -2, 3);
println!("Tell me about {:?}", triple);
match triple {
    (0, y, z) => {
        println!("First is `0`, `y` is {y}, and `z` is {z}")
    },
    // `..` can be used to ignore the rest of the tuple
    (1, ..) => {
        println!("First is `1` and the rest doesn't matter"),
    },
    _ => {
        println!("It doesn't matter what they are"),
    },
}
```

Match

Let’s define a struct:

```
struct Foo {
    x: (u32, u32),
    y: u32,
}

let foo = Foo { x: (1, 2), y: 3 };
```

Match

Destructuring the struct:

```
match foo {
    Foo { x: (1, b), y } => {
        println!("First of x is 1, b = {},  y = {} ", b, y);
    },
    Foo { y: 2, x: i } => {
        println!("y is 2, i = {:?}", i);
    },
    Foo { y, .. } => { // ignoring some variables:
        println!("y = {}, we don't care about x", y)
    },
    // error: pattern does not mention field `x`
    // Foo { y } => println!("y = {}", y),
}
```

Match

Binding values to names:

```
match age() {
    0 => println!("I haven't celebrated my birthday yet"),
    n @ 1..=12 => println!("I'm a child of age {n}"),
    n @ 13..=19 => println!("I'm a teen of age {n}"),
    n => println!("I'm an old person of age {n}"),
}
```

Match

Binding values to names + arrays:

```
let s = [1, 2, 3, 4];
let mut t = &s[..]; // or s.as_slice()
loop {
    match t {
        [head, tail @ ..] => {
            println!("{head}");
            t = &tail;
        }
        _ => break,
    }
} // outputs 1\n2\n3\n4\n
```

Enumerations

You can create custom functions for `enum`:

```
enum Test {
    A(i32),
    B,
}

impl Test {
    pub fn unwrap(self) -> i32 {
        match self {
            Self::A(x) => x,
            Self::B => panic!("error!"),
        }
    }
}
```

`if let`

Sometimes we need only one enumeration variant to do something. Can we write it in a better way?

```rust
let optional = Some(7);
match optional {
    Some(i) => {
        println!("It's Some({i})");
    },
    _ => {},
    // ^ Required because `match` is exhaustive
};
```

`if let`

Sometimes we need only one enumeration variant to do something. Can we write it in a better way?

```rust
let optional = Some(7);
if let Some(i) = optional {
    println!("It's Some({i})");
}
```

`while let`

Same with `while`:

```
let mut optional = Some(0);
while let Some(i) = optional {
    if i > 9 {
        println!("Greater than 9, quit!");
        optional = None;
    } else {
        println!("`i` is `{i}`. Try again.");
        optional = Some(i + 1);
    }
}
```

Enumerations

Let’s dive into details and compare enumerations with C++ `enum` and `union`.

- To identify the variant, we store some *bits* in fields of enum. These bits are called *discriminant*.
- The count of bits is exactly as many as needed to keep the number of variants.
- These bits are stored in unused bits of enumeration in another field. (compiler optimizations!)
- That means we have something like `union` and `enum` in C++ at the same time! (Note: in Rust there’s `union`, we’ll dive into it later)

Enumerations

```rust
enum Test {
    First(bool),
    Second,
    Third,
    Fourth,
}
assert_eq!(
    std::mem::size_of::<Test>(), 1
);
assert_eq!(
    std::mem::size_of::<Option<Box<i32>>>(), 8
);
```

Enumerations

```
enum Test1 {
    First(bool, bool),
    Field0,
    // More fields...
    Field253,
}
enum Test2 {
    First(bool, bool),
    Field0,
    // More fields...
    Field254,
}
assert_eq!(std::mem::size_of::<Test1>(), 2);
assert_eq!(std::mem::size_of::<Test2>(), 3);
```

Vector

Dynamic array, just like in C++.

```
pub struct Vec<T> {
    ptr: *const T,
    length: usize,
    capacity: usize,
    // PhantomData<T>?
}
```

Vector

```
use std::mem::size_of;
assert_eq!(
    size_of::<Vec<i32>>(),
    size_of::<usize>() * 3,
);

let mut xs = vec![1, 2, 3];
// To declare vector with same element and
// specific count of elements, write
// vec![42; 113];
xs.push(4);
assert_eq!(xs.len(), 4);
assert_eq!(xs[2], 3);
```

Slices

We can create a slice to a vector or array. A slice is a contiguous sequence of elements in a collection.

```rust
let a = [1, 2, 3, 4, 5];
let slice1 = &a[1..4];
let slice2 = &slice1[..2];
assert_eq!(slice1, &[2, 3, 4]);
assert_eq!(slice2, &[2]);
```

Panic!

In Rust, when we encounter an unrecoverable error, we `panic!`

```rust
let x = 42;
if x == 42 {
    panic!("The answer!")
}
```

There are some useful macros that `panic!`

- `unimplemented!`
- `unreachable!`
- `todo!`
- `assert!`
- `assert_eq!`

println!

The best tool for debugging, we all know.

```rust
let x = 42;
println!("{x}");
println!("The value of x is {}, and it's cool!", x);
println!("{:04}", x); // 0042
println!("{value}", value=x); // 42
let vec = vec![1, 2, 3];
println!("{vec:?}"); // [1, 2, 3]
println!("{:?}", vec);
let y = (100, 200);
println!("{:#?}", (100, 200));
// (
//     100,
//     200,
// )
```

Inhabited type !

Rust always requires to return something correct.

```
// error: mismatched types
// expected `i32`, found `()`
fn func() -> i32 {}
```

How does this code work?

```
fn func() -> i32 {
    unimplemented!("not ready yet")
}
```

`Inhabited type !`

Rust always requires to return something correct.

```
// error: mismatched types
// expected `i32`, found `()`
fn func() -> i32 {}
```

How does this code work?

```
fn func() -> i32 {
    unimplemented!("not ready yet")
}
```

Return type that is never constructed: `!`.

Inhabited type `!`

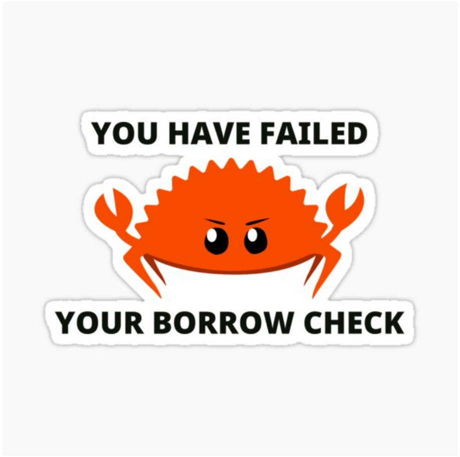
Return type that is never constructed: `!`

Same as:

```
enum Test {} // empty, could not be constructed
```

`loop` without any `break` returns `!`

Borrow Checker



What’s the problem, Rust?

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

What’s the problem, Rust?

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

```
error[E0502]: cannot borrow `v` as mutable because it is also
borrowed as immutable
 --> src/main.rs:8:5
  |
7 |     let x = &v[0];
  |              - immutable borrow occurs here
8 |     v.push(4);
  |     ^^^^^^^^^ mutable borrow occurs here
9 |     println!("{}", x);
  |                    - immutable borrow later used here
```

What’s the problem, Rust?

```rust
fn sum(v: Vec<i32>) -> i32 {
    let mut result = 0;
    for i in v {
        result += i;
    }
    result
}

fn main() {
    let mut v = vec![1, 2, 3];
    println!("first sum: {}", sum(v));
    v.push(4);
    println!("second sum: {}", sum(v))
}
```

What’s the problem, Rust?

```
error[E0382]: borrow of moved value: `v`
  --> src/main.rs:12:5
   |
10 |     let mut v = vec![1, 2, 3];
   |         ----- move occurs because `v` has type `Vec<i32>`,
   |  which does not implement the `Copy` trait
11 |     println!("first sum: {}", sum(v));
   |                                   - value moved here
12 |     v.push(4);
   |     ^^^^^^^^^ value borrowed here after move
```

Ownership rules

- Each value in Rust has a variable that’s called it’s *owner*.
- There can be only one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Ownership rules

```rust
fn main() {
    let s = vec![1, 4, 8, 8];
    let u = s;
    println!(“{:?}”, u);
    // This won't compile!
    // println!("{:?}", s);
}
```

Ownership rules

```rust
fn om_nom_nom(s: Vec<i32>) {
    println!("I have consumed {s:?}");
}

fn main() {
    let s = vec![1, 4, 8, 8];
    om_nom_nom(s);
    println!("{s:?}");
}
```

- Each ”owner” has the responsibility to clean up after itself.
- When you move `s` into `om_nom_nom`, it becomes the owner of `s`, and it will free `s` when it’s no longer needed in that scope. *Technically the `s` parameter in `om_nom_nom` become the owner.*
- That means you can no longer use it in `main`!
- In C++, we will create a copy!

Ownership rules

Given what we just saw, how can the following be the valid syntax?

```rust
fn om_nom_nom(n: u32) {
    println!("{} is a very nice number", n);
}

fn main() {
    let n: u32 = 110;
    let m = n;
    om_nom_nom(n);
    om_nom_nom(m);
    println!("{}", m + n);
}
```

Ownership rules

- Say you have a group of lawyers that are reviewing and signing a contract over Google Docs (just pretend it's true :))
- What are some ground rules we'd need to set to avoid chaos?
- If someone modifies the contract before everyone else reviews/signs it, that's fine.
- But if someone modifies the contract while others are reviewing it, people might miss changes and think they're signing a contract that says something else.
- We should allow a single person to modify, or everyone to read, but not both.

Ownership rules

- Say you have a group of lawyers that are reviewing and signing a contract over Google Docs (just pretend it's true :))
- What are some ground rules we'd need to set to avoid chaos?
- If someone modifies the contract before everyone else reviews/signs it, that's fine.
- But if someone modifies the contract while others are reviewing it, people might miss changes and think they're signing a contract that says something else.
- We should allow a single person to modify, or everyone to read, but not both.

Borrowing intuition

- I should be able to have as many ”const” pointers to a piece of data that I like.
- However, if I have a ”non-const” pointer to a piece of data at the same time, this could invalidate what the other const pointers are viewing. (e.g., they can become dangling pointers...)
- If I have at most one ”non-const” pointer at any given time, this should be OK.

Borrowing

- We can have multiple shared (immutable) references at once (with no mutable references) to a value.
- We can have only one mutable reference at once. (no shared references to it)
- This paradigm pops up a lot in systems programming, especially when you have ”readers” and ”writers”. In fact, you’ve already studied it in the course of Theory and Practice of Concurrency.

Borrowing

- The lifetime of a value starts when it's created and ends the *last time it's used*.
- Rust doesn't let you have a reference to a value that lasts longer than the value's lifetime.
- Rust computes lifetimes at compile time using static analysis. (this is often an over-approximation!)
- Rust calls the special "drop" function on a value once its lifetime ends. (this is essentially a destructor)

Drop implementation?

```
let vec = vec![];
drop(vec);
```

Drop implementation?

```
let vec = vec![];
drop(vec);
```

```
fn drop(self) {}
```

Drop flags

```
let x = Box::new(92);
let y;
let z;
if condition {
    y = x;
} else {
    z = x;
}
// Who cleans the memory?
```

Drop flags

```
let x = Box::new(92);
let y;
let z;
if condition {
    y = x;
} else {
    z = x;
}
// Who cleans the memory?
```

Invisible flag on the stack: is the variable initialized?

Borrowing

```
let mut x;
// this access would be illegal, nowhere to
// draw the flow from:
// assert_eq!(x, 42);
x = 42;
// this is okay, can draw a flow from the
// value assigned above:
let y = &x;
// this establishes a second, mutable flow from x:
x = 43;
// this continues the flow from y, which in turn
// draws from x:
assert_eq!(*y, 42);
```

Borrowing

```
let x1 = 42;
let y1 = Box::new(84);
{ // starts a new scope
    let z = (x1, y1);
    // z goes out of scope, and is dropped;
    // it in turn drops the values from x1 and y1
}
// x1's value is Copy, so it was not moved into z
let x2 = x1;
// y1's value is not Copy, so it was moved into z
// let y2 = y1;
```

Conclusion

- We studied why we need Rust and where it’s used.
- Learned the basics of language syntax.
- And passed the borrow check :)