A taste of Rust

...so that you quickly grasp its core concepts and characteristics.

Syntax: mostly similar to C++

```rust
fn bar(num_cats: u32) -> usize {
    let a: &str = "Ann has "; // <-- think of char*

    let mut b: String = String::from(a); // <-- think of std::string
    b += num_cats.to_string();
    b += " cats";
    b.push('.');

    println!("{}", &b);
    return b.len();

    // <-- b is dropped (deallocated) here - RAII mechanism employed.
}
```

Syntax: C/C++/ML/...

C/C++: Code blocks with braces, lines end with semicolons, `main` function begins every program, modules/associated functions `::` notation (e.g. `std::thread::spawn`)...

ML: Match statement, code blocks evaluated as expressions.

Inspired by functional languages (ML, Haskell)

Variables are not mutable by default

```rust
fn variables_non_mut_by_default() {
    let cant_mutate_me = 5; // <-- variable `const` by default
    // cant_mutate_me = 3; <-- compile error
    let mut can_mutate_me = 5;
    can_mutate_me += 1;
}
```

Blocks' last statement can be evaluated as an expression

```rust
fn blocks_are_expressions(flag: bool) -> i32 {
    let x = if flag {
        42 // <-- mind lack of semicolon
    } else {
        let init = -37;
        (init + 17) * (init - 3) // <-- the last statement can be returned
    };

    x // <-- same about functions - their body is a block, too.
}
```

Enum types, pattern matching

```
enum Animal {
    Cat { tail_len: usize },
    Fish,
}

fn pattern_matching(animal: Animal) {
    let tail_len = match animal {
        Cat { tail_len } => tail_len,
        Fish => 0,
    };
}
```

Functional iterators

```
let peers: Vec<Peer> = initial_peers
    .iter()
    .enumerate()
    .map(|(id, endpoint)| {
        let token = get_token(endpoint);

        Peer {
            address: endpoint.address(),
            tokens: vec![Token::new(token as i64)],
            datacenter: None,
            rack: None,
            host_id: Uuid::new_v4(),
        }
    })
    .collect();
```

What Rust lacks (on purpose)

Classes, inheritance?

```
class Vehicle {
    float estimate_cost(float route_len) = 0;
};
class Car: Vehicle { int engine_size_cm3; };
class Bike: Vehicle { int rider_stamina; };

float Car::estimate_cost(float route_len) {
    engine_size_cm3 * CAR_COST_COEFFICIENT * route_len
}

float Bike::estimate_cost(float route_len) { /* */ }
```

~~Classes, inheritance~~. Structs, traits, composition.

```
struct Car { engine_size_cm3: i32 };
struct Bike { rider_stamina: i32 };

trait Travel {
    fn estimate_cost(&self, route_len: f32) -> f32;
}

impl Travel for Car {
    fn estimate_cost(&self, route_len: f32) -> f32 {
        self.engine_size_cm3 * CAR_COST_COEFFICIENT * route_len
    }
}

impl Travel for Bike { /* */ }
```

Null value?

```
public void doSomething(SomeObject obj) {
    obj.some_method(); // BOOOM! NullPointerException!

    if obj != null {
        obj.some_method(); // Whew, we are safe now. We checked this.
    }

    /* A lot of code later */

    // Have I checked `obj` for not being null? Yeah, for sure.
    obj.some_method(); // BOOOM! NullPointerException!
}
```

~~Null value~~. Lack of value represented as an Option enum.

```
fn this_returns_something_or_nothing(password: &str) -> Result<i32, String> {
    if password == "Rust rulez!" {
        Ok(42)
    } else {
        Err("You still have to learn a lot...")
    }
}

fn main() {
    match this_can_fail_or_succeed("Rust is hard...") {
        Ok(code) => println!("Success! Code: {}", code),
        Err(msg) => println!("Oops... {}", msg),
    };
}
```

Exceptions?

```
void some_critical_operation(data: VeryFragile) {
    // We have to be extremely careful not to interrupt this.
    // Else we will end up with an invalid state!

    int const res = some_innocent_procedure(data); // BOOOM! Exception thrown.

    finalise_critical_operation(res);
}

fn main() {
    match this_can_fail_or_succeed("Rust is hard...") {
        Ok(code) => println!("Success! Code: {}", code),
        Err(msg) => println!("Oops... {}", msg),
    };
}
```

~~Exceptions~~. Errors propagated explicitly as enums.

```rust
fn this_can_fail_or_succeed(password: &str) -> Result<i32, String> {
    if password == "Rust rulez!" {
        Ok(42)
    } else {
        Err("You still have to learn a lot...")
    }
}

fn main() {
    match this_can_fail_or_succeed("Rust is hard...") {
        Ok(code) => println!("Success! Code: {}", code),
        Err(msg) => println!("Oops... {}", msg),
    };
}
```

~~Exceptions~~. Errors propagated explicitly as enums.

```
fn deserialize(
    typ: &'frame ColumnType,
    v: Option<FrameSlice<'frame>>,
) -> Result<CqlType, DeserializationError> {
    let mut val = ensure_not_null_slice::<CqlType>(typ, v)?;
    let cql = deser_cql_value(typ, &mut val)
        .map_err(deser_error_replace_rust_name::<CqlType>)?;
    Ok(cql)
}
```

Unique feature of Rust - the borrow checker

Dangling references?

```cpp
int const& bar()
{
    int n = 10;
    return n;
}

int main() {
    int const& i = bar();
    // i is a dangling reference to an invalidated stack frame...
    std::cout << i << std::endl; // May result in segmentation fault.
    return 0;
}
```

~~Dangling references~~ Lifetimes!

```rust
// This function does not compile! Reference can't borrow from nowhere.
fn bar() -> &i32 {
    let n = 10;
    &n
}
```

~~Dangling references~~ Lifetimes!

```rust
fn main() {
    let v = vec![1, 2, 3];
    let v_ref = &v;

    std::mem::drop(v);

    // This does not compile! `v` has been dropped,
    // so references to it are no longer valid.
    let n = v_ref[1];
}
```

Move semantics by default (& more lifetimes!).

```rust
fn main() {
    let mut x = vec![1, 2, 3];
    let y = x; // x's contents are moved into y.
               // No heap allocation involved, O(1).

    // This does not compile! `x`'s contents has been moved out,
    // so it is no longer valid (alive).
    // let n = x[0];

    x = vec![0, 0]; // Now x is valid (alive) again.
    let n = x[0];
}
```

Data races?

```
void thread1(shared_data: &Data) {
    while true {
        shared_data.write(next_int());
    }
}

void thread2(shared_data: const& Data) {
    while true {
        std::cout << shared_data.read() << std::endl;
    }
}

int main() {
    Data shared_data;

    // The threads race with each other!
    // A write is concurrent to another memory access (read or write)!
    std::thread t1{shared_data};
    std::thread t2{shared_data};
}
```

~~Data races~~ Aliasing XOR mutability

```
fn thread1(shared_data: &mut Data) {
    loop {
        shared_data.write(next_int());
    }
}
fn thread2(shared_data: &Data) {
    loop {
        println!("{}", shared_data.read());
    }
}
fn main() {
    let mut shared_data = Data::new();

    std::thread::scope(|s| {
        let t1 = s.spawn(|| {
            thread1(&mut shared_data);
        });
        let t2 = s.spawn(|| {
            thread2(&shared_data); // Compiler yells:
            // "cannot borrow `shared_data` as immutable because it is also borrowed as mutable"
        });
    });
}
```

Rust

A language empowering everyone to build reliable and efficient software.