Rust for C++ Programmers

Radu Popescu

CERN SFT Group, “Software We Love” Meetup, Feb 24th 2017

“Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.”

-the person who wrote the Rust website

Rust is only about safety

- Ownership and lifetimes are a means to eliminate certain types of memory errors

- All is done at compile time (minimal runtime library is needed, always statically linked)

- Safer concurrency

- Uses LLVM as backend

Rust is not only about safety

Abstractions “stolen” from high-level languages:

- Algebraic data types
- Pattern matching
- Error handling
- Type inference
- Polymorphism (using traits ~ type classes)

Zero cost abstractions*

Brief history of Rust

- 2006 - Started as a personal project of Graydon Hoare (@graydon_pub) at Mozilla
- 2009 - Mozilla **sponsorship** begins
- 2011 - Rust compiler is **self-hosted** (previously OCaml)
- 2012 - First pre-alpha
- 2015 - **Version 1.0** - first stable release, **API stability guaranteed**
- Since 1.0, a new stable minor release every ~6 weeks.
- Backed by Mozilla, **Open Source**

```
fn twice(val: i32) -> i32 {
    2 * val
}

fn main() {
    let mut some_var: i32 = 3;
    println!("Two times {} makes {}”,
                some_var, twice(some_var));

    let arrrr = [1, 2, 3];
    println!("Arrrr {:?}!", arrrr);
    println!("Segmentation fault!!!!");
}
```

Primitive types

- Booleans: `true`, `false`
- Char (Unicode): `‘a’`, `‘b’`, `‘c’`
- Numeric type: signed and unsigned integers of various widths, single and double floats
- Arrays: `[1, 2, 3, 4, 5]`, `[2.0, 3.0, 4.0]` etc.
- Slices (views into arrays and collections): `&[T]`
- `str`: string slices (pointer + length)
- Tuples: `(1, “two”, true)`
- Functions (pointers):

Functions

```rust
fn some_function(x: f64, y: f64) -> f64 {
    let z = x + y;
    z * z
}
```

Algebraic data types

Sum types: enums (tagged unions, variants)

```
enum Event {
    Quit,
    NewPosition(i32, i32, i32),
    Clone { x: i32, y: i32 },
    Write(String),
}
```

Algebraic data types

Product types: structs, tuple-structs, empty structs

```
struct Point {
    x: i32,
    y: i32,
}

struct Point(i32, i32, i32);

struct Electron {}
```

Pattern matching

Match expressions:

```
match some_enum_value {
    Event::Quit => {
        do_something();
    },
    Event::NewPosition(x, y, z) => {
        do_something_else(x, y, z);
    },
    Event::Clone { x: i32, y: i32 } => {
        maybe_this(x, y);
    },
    Event::Write(msg) => {
        println!(msg);
    },
}
```

Pattern matching

Match expressions:

```
match some_struct_value {
    Point { x, y } => { x + y },
}

match some_struct_value {
    Point { x, .. } => { x + y },
}
```

Methods

```
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn new(x1: f64, y1: f64) -> Point {
        Point { x: x1, y: y1 }
    }
    fn dist(&self) -> f64 {
        f64::sqrt(self.x * self.x + self.y * self.y)
    }
}

fn main() {
    let p = Point::new(1.0, 2.0);
    println!("{}", p.dist());
}
```

Ownership

Variables have ownership of data they are bound to:

```
let a = [1,2,3];
```

Rust has move semantics by default:

```
let a = [1,2,3];
let b = a;
println!(“{:?}", a); // compilation error.
```

Borrowing

- It's possible to borrow values by taking references
- Either **multiple read-only** references (`&T`) to the same data can exist at the same time, or a single mutable reference (`&mut T`)

```
fn some_fun(v: &i32) -> i32 {
    2 * v
}

fn main() {
    let mut a = 2;
    let bref = &mut a;

    some_fun(&a); // compilation error
}
```

Lifetimes

The following usage is ambiguous and causes compilation errors:

```
fn fn_with_references(r1: &i32, r2: &i32) -> &i32 {
    r1
}
```

```
struct Container {
    val: &i32,
}

fn main() {
    let a = 2;
    let b = 3;
    fn_with_references(&a, &b);

    let c = Container { val: &a };
}
```

Lifetimes

We need to introduce explicit lifetime parameters to track validity of references:

```
fn fn_with_references<'a, 'b>(r1: &'a i32, r2: &'b i32) -> &'a i32 {
    r1
}
```

```
struct Container<'a> {
    val: &'a i32,
}

fn main() {
    let a = 2;
    let b = 3;
    fn_with_references(&a, &b);

    let c = Container { val: &a };
}
```

One weird trick…

A struct allocated on the stack, passing references to its data members, references which are guaranteed at compile time to not outlive the instance of the struct itself:

```
struct X {
  y: Y
}

impl X {
  fn y(&self) -> &Y { &self.y }
}
```

http://robert.ocallahan.org/2017/02/what-rust-can-do-that-other-languages.html

Polymorphism: Generics

Generic functions:

```rust
fn some_function<T>(x: T) {
    // Do something with `x`.
}
```

Generic data structures:

```rust
enum Option<T> {
    Some(T),
    None,
}
```

```rust
struct Point<T> {
    x: T,
    y: T,
}
```

Polymorphism: Traits

- Traits (a.k.a. type classes) describe what functionality a set of types must provide

- Similar to interfaces in OO languages, or Concepts (Lite?) in C++

- The implementation of a trait for a given type is done outside of the definition of the type

- Much functionality is implemented with traits: Copy, Clone, Drop, From, Display, Debug etc.

Polymorphism: Traits

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn print_area<T>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Memory management

- Stack allocation is preferred (and made safe and efficient with move semantics, borrowing, lifetimes etc.)
- Full range of smart pointers for heap allocation, as in C++:
  - `Box<T>` - similar to `std::unique_ptr<T>`
  - `Rc<T>`, `Arc<T>` - similar to `std::shared_ptr<T>`

Error handling

Prefer explicit error handling:

```rust
enum Option<T> {
    Some(T),
    None,
}
```

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

Can be used in pattern matching!

Error handling

Can use combinator functions for Option and Result:

```
fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    File::open(file_path)
        .map_err(|err| err.to_string())
        .and_then(|mut file| {
            let mut contents = String::new();
            file.read_to_string(&mut contents)
                .map_err(|err| err.to_string())
                .map(|_| contents)
        })
        .and_then(|contents| {
            contents.trim().parse::<i32>()
                    .map_err(|err| err.to_string())
        })
        .map(|n| 2 * n)
}
```

Error handling

Can use early return macro `try!` and the early return operator `?`:

```
fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = try!(File::open(file_path).map_err(|e| e.to_string()));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(|e| e.to_string()));
    let n = try!(contents.trim().parse::<i32>().map_err(|e| e.to_string()));
    Ok(2 * n)
}
```

```
fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = File::open(file_path).map_err(|e| e.to_string())?;
    let mut contents = String::new();
    file.read_to_string(&mut contents).map_err(|e| e.to_string())?;
    let n = contents.trim().parse::<i32>().map_err(|e| e.to_string())?;
    Ok(2 * n)
}
```

Concurrency

- Native threads
- All the synchronisation primitives: atomic types, barriers, mutexes, rw-locks, condition variables etc.
- Channels
- Two traits describe the semantics in a multi-threaded context:
  - Send - ownership of a value can be transferred between threads
  - Sync - can safely be used from multiple threads through shared references
- Crossbeam - additional high-performance algorithms and data structures in the package (https://docs.rs/crossbeam/0.2.10/crossbeam/) - Scoped threads!
- Rayon - a parallelism library (https://github.com/nikomatsakis/rayon)

More concurrency

- Asynchronous programming with (zero-cost futures) Futures: https://aturon.github.io/blog/2016/08/11/futures/
    - Can be composed with similar set of combinators as `Option<T>` and `Result<T,E>`!
- Tokyo - high-performance network programming based on Futures: https://tokio.rs/
    - Separate your application-specific parts (`tokio-service`, `tokio-protocol`) from the underlying state-machine: `tokio-core`.
    - Middleware: SSL, timers, logging, etc.
    - Support for: TCP, UDP, sockets, signals, processes, databases, inotify etc.

Other language features

Important topics that weren't covered here:

- iterators
- closures
- interior mutability
- macros
- trait objects (dynamic dispatch)
- associated types
- …

The Rust Book covers everything!

Development tools

- rustup - install and update Rust, add components, toolchains, targets for cross-compilation etc.
- Cargo - build tool with dependency management; one stop shop for project configuration
- Can use GDB, LLDB, Perf, macOS Instruments etc.
- crates.io - online database of packages (“crates”), used by Cargo
- docs.rs - online database with documentation of Rust + crates.io
- Editor and IDE plugins (Vim, Emacs, Visual Studio Code, Sublime Text, Atom, IntelliJ etc.)
- Rust Language Server, Racer - autocompletion and smart editing
- rustfmt - formats all source code files, eliminates bikeshedding
- Clippy* - very smart linter

Testing, benchmarks, documentation

Cargo has built-in support for:

- Unit tests
- Integration tests
- Doc-tests (examples in documentation are compiled and tested)!
- Micro-benchmarks*

Demo

Unsafe Rust

- Sometimes it may be necessary to loosen the restriction of the compiler: interact with native C or C++ code, compiler is unable to verify correctness

- Can use the “unsafe” block in this case

- Unsafe Rust can ONLY do three extra things:

    - Access and mutate static mutable variables

    - Dereference a raw pointer

    - Call unsafe functions

- Programmer must be sure to manually enforce the invariants of the compiler!

The bad parts

- Language is young. API stability since 1.0, but things can still change

- Smaller community, for now

- There is still much to be improved: ergonomics, resources, adding/stabilising missing features

- Fewer libraries. Many crates are wrappers around C or C++ libs (not necessarily a bad thing)

- Fewer posts on StackOverflow (may be a great thing!)

Resources: Tools

- Rustup (https://rustup.rs/) - install and update Rust, Cargo, rustdoc, rust-gdb, rust-lldb

- crates.io (https://crates.io) - database of Rust packages

- Racer (https://github.com/phildawes/racer) - Rust autocompletion for your editor/IDE

- rustfmt (https://github.com/rust-lang-nursery/rustfmt) - automatic formatting for Rust source code files

- Clippy (https://github.com/Manishearth/rust-clippy) - smart linter for Rust code

Resources: Documentation

- docs.rs (https://docs.rs) - documentation for Rust std library and all the packages on crates.io

- The Rust Book (https://doc.rust-lang.org/stable/book/) - In depth presentation of the language, referenced by this tutorial!

- Learning with examples (http://rustbyexample.com/)

- Online Rust playground (https://play.rust-lang.org/)

- The Rustonomicon (https://doc.rust-lang.org/nomicon/) - WIP Guide to Unsafe Rust

Resources: Community

- https://www.rust-lang.org/en-US/community.html
- https://users.rust-lang.org/
- https://www.reddit.com/r/rust/
- Industrial users of Rust: https://www.rust-lang.org/en-US/friends.html

Thank you!

I hope you enjoy Rust!