Rust for modern C++ users

François Gindraud

23 juin 2022

$ whoami

Career in short

- PhD in computer science (HPC, low level parallel runtimes)
- ~5y of Research Engineer in bio-informatics (mostly statistical inference)

Languages : a bit of everything, but preference for compiled ones :

- C++ : heavily used in the last 10 years
- A few Python academic projects
- C, Ocaml, Bash, web stuff, ... : occasionally
- **Rust** : 4 years on mostly hobby projects (web, CLI tools)
  - Good experience of the language features and basic ecosystem
  - Average experience in library design
  - Low experience of research library availability
  - No team development experience (single dev projects)

TLDR

Compiled language (LLVM), imperative, non-garbage-collected.

- Very good expressive power for library design
  - Sum types, vocabulary types, references, ...
  - May be tedious : conversions, lifetime annotations
- Individual features can be found in other languages, but the complete package is uniquely ergonomic
- Benefits from being new
  - Overall documentation is very good
  - Low amount of legacy stuff, coherent language
  - Takes lessons from other languages (ML, C++)

History[2] & context

- Mozilla internal project initially by Graydon Hoare
- stable 1.0 in 2015.
- first big project : Servo at Mozilla
- 2018 : first *Epoch* upgrade
- progressive adoption by GAFAM, and many smaller entities
- 2020 : PhD defense or Ralf Jung[1] (Rust borrow semantics)
- 2020 : Mozilla disbands its Rust teams
- 2021 - ongoing : Rust foundation supported by industry
- 2021 : second *Epoch*

[1] `https://www.ralfj.de/research/publications.html`
[2] `https://en.wikipedia.org/wiki/Rust_(programming_language)`

Rustup

```
The Rust toolchain installer

USAGE:
    rustup [FLAGS] [+toolchain] <SUBCOMMAND>

ARGS:
    <+toolchain>    release channel (e.g. +stable) or custom toolchain to set override

SUBCOMMANDS:
    show            Show the active and installed toolchains or profiles
    update          Update Rust toolchains and rustup
    check           Check for updates to Rust toolchains and rustup
    default         Set the default toolchain
    toolchain       Modify or query the installed toolchains
    target          Modify a toolchain's supported targets
    component       Modify a toolchain's installed components
    override        Modify directory toolchain overrides
    run             Run a command with an environment configured for a given toolchain
    which           Display which binary will be run for a given command
    doc             Open the documentation for the current toolchain
    man             View the man page for a given command
    self            Modify the rustup installation
    set             Alter rustup settings
    completions     Generate tab-completion scripts for your shell
    help            Prints this message or the help of the given subcommand(s)
```

General tooling

`rust-format`

- code formatter, similar to `clang-format`
- uniformity of code bases, can be customized

Debugging

- Compiler emits normal debug info, stack frames, ...
- `rust-gdb` : loads `gdb` with pretty print scripts

Linter

- clippy

Mature IDE plugins

- *rust analyzer*[3] for VS-code, Vim, Emacs (demo)
- based on *language server* protocol

[3] `https://rust-analyzer.github.io/`

Cargo

```
Rust's package manager

USAGE:
    cargo [+toolchain] [OPTIONS] [SUBCOMMAND]

Some common cargo commands are (see all commands with --list):
    build, b    Compile the current package
    check, c    Analyze the current package and report errors, but don't build object files
    clean       Remove the target directory
    doc, d      Build this package's and its dependencies' documentation
    new         Create a new cargo package
    init        Create a new cargo package in an existing directory
    run, r      Run a binary or example of the local package
    test, t     Run the tests
    bench       Run the benchmarks
    update      Update dependencies listed in Cargo.lock
    search      Search registry for crates
    publish     Package and upload this package to the registry
    install     Install a Rust binary. Default location is $HOME/.cargo/bin
    uninstall   Uninstall a Rust binary
    # Non standard but useful
    clippy      Checks a package to catch common mistakes and improve your Rust code.
    tree        Display a tree visualization of a dependency graph
```

Project anatomy

`benches/`	Benchmarks for your crate, run via cargo bench, requires nightly by default.

`examples/`
 	
`my_example.rs`
	Examples how to use your crate, they see your crate like external user would.
	`cargo run --example my_example`.

`src/`
`main.rs`
`lib.rs`
	Actual source code for your project.
	Default entry point for applications (`cargo run`).
	Default entry point for libraries.

`src/bin/` `extra.rs`	Place for additional binaries, even in library projects. Additional binary, run with `cargo run --bin` extra.
`tests/`	Integration tests go here, invoked via cargo test. Unit tests often stay in src/ file.
`.rustfmt.toml`	In case you want to customize how cargo fmt works.
`build.rs`	Pre-build script, useful when compiling C / FFI, ...
`Cargo.toml`	Main project manifest, defines dependencies, artifacts
`Cargo.lock`	Dependency details for reproducible builds; add to git for apps, not for libs.

Cargo.toml

```
[package]
name = "slam"
version = "0.1.0"
edition = "2021"
authors = ["François Gindraud <francois.gindraud@gmail.com>"]
description = "Save multi-screen layouts and restore them when needed"
repository = "https://github.com/fgindraud/slam"
license = "MIT"
keywords = ["screen", "daemon", "x11"]

[dependencies]
anyhow = "1.0"
clap = { version = "3.1", features = ["derive"] } # cmd line parsing
log = "0.4"
regex = { git = "https://github.com/rust-lang/regex", branch = "next" }
bitflags = { path = "my-bitflags", version = "1.0" }

# xcb backend: feature "xcb"
xcb = { version = "1.1", features = ["randr"], optional = true }

[dev-dependencies]
# Drawing for layout examples
tiny-skia = "0.6.3"
palette = "0.6.0"

[[example]]
name = "layout"
```

Compilation

`$ cargo build`

- Downloads dependencies from `crates.io`
- Runs `Rustc` in parallel over all transitive dependencies
- Statically link user code and all dependencies

Compilation profiles like `-O<n>` : `$ cargo build --release`

crates.io

- Repository of versioned crates
- Risk of *supply chain attack* (like NPM for js). Mitigations :
  - alternate community owners to update crates
  - Cargo.lock pinning dependencies versions
  - crates.io *yank* of a faulty version
  - `cargo-audit` to check dep tree for vulnerabilities

Static compilation

Main reason : Rust has not defined an ABI

- Ongoing debate
- Example of C++ which is frozen by risk of ABI-break
- No ABI enables struct layout optimizations

Trade-offs

- Better optimizations (lots of templates)
- Slow compilation times, C++ level or worse
- Caching of crate code for incremental builds
- Can dynamically link (ex: libc), uses C ABI for FFI
- Advantage for research : alternative to containers ?
- Push to open source ?

Documentation ressources

Official documentation `https://doc.rust-lang.org/`

- Book : a tutorial
- Reference for language primitives and std lib
- Tool documentation : `rustdoc`, `cargo`
- Advanced documentation : unsafe, etc
- Very good quality, up to date, accessible
  - looking at you C++ standard behind IEEE walls
- Github community based evolution[4]

Third party

- Good cheat sheet `https://cheats.rs/`
- `arewe*yet.com` : status pages for specific domains `https://wiki.mozilla.org/Areweyet`

[4] `https://github.com/rust-lang/rust/issues`

`rustdoc` : a unified & useful documentation generator

Markup in comments like `Doxygen`

```
// Normal comment
/// Represents a *point* in 2d space
struct Point {
    /// Horizontal axis from left to right
    x: f64
    /// Vertical axis from bottom to up
    y: f64
}
```

But way better ergonomics due to :

- Using the same tool across the whole ecosystem
- Static compilation : all code cross-referenced and reachable
- Integrated : `cargo doc`
- Offline experience : rustup component doc for local stdlib
- *Type focused* language : lots of names to navigate around

Demo

let : constant by default

```rust
let i : i32 = 42; // const int32_t i = 42;
let j = 42usize; // auto j = std::size_t(42);
i += 1; // error
```

```
error[E0384]: cannot assign twice to immutable variable `i`
 --> a.rs:3:5
  |
1 |     let i : i32 = 42;
  |         -
  |         |
  |         first assignment to `i`
  |         help: consider making this binding mutable: `mut i`
3 |     i+=1;
  |     ^^^^ cannot assign twice to immutable variable
```

```rust
let mut i : i32 = 42; // int32_t i = 42;
i += 1; // ok

// Rebindings
let i = i; // new i (immutable) shadows previous mutable i
```

Move semantics by default

```rust
struct A { data: i32 }
let a = A { data: 42 };
let b = a; // moves ! a does not live anymore
f(a); // error
```

```
error[E0382]: use of moved value: `a`
 --> test.rs:4:10
  |
2 |     let a = A;
  |         - move occurs because `a` has type `A`, which does not implement the `Copy` trait
3 |     let b = a;
  |             - value moved here
4 |     dbg!(a);
  |          ^ value used here after move
```

```cpp
struct A { int data; };
auto a = A { 42 };
A b = a; // copies
A c = std::move(a); // moves content
f(a); // ok but a in "moved from state"
```

Copy is explicit

```
// Standard trait for performing a copy
impl Clone for A {
    fn clone(&self) -> A {
        A { data: self.data.clone() }
    }
}
let b = a.clone(); // creates a copy
f(a); // ok

// Standard trait for "always copy" : mostly for cheap types
impl Copy for A {} // Just a tag
let c = a; // implicit copy

// "derive" directive to autofill Clone/Copy with trivial impl
#[derive(Clone, Copy)]
struct A { data: i32 }
```

Block as values

```
// "Configure step then use" pattern
let database = {
    let mut db = Database::new();
    let file = File::open("data");
    fill_database_with_stuff(&mut db, file);
    db
};
```

C++ equivalent requires immediately evaluated lambdas (ugly ?)

```
auto database = [&](){
    auto db = Database();
    auto file = std::fstream("data");
    fill_database_with_stuff(db, file);
    return db;
}();
```

Could have worse scoping rules

```
for i in range(10): pass
print (i) # happily prints 9
```

If-then-else as values

```
// if-then-else as values
let values : Vec<Entry> = if database.has_data() {
    Vec::from_iter(database.iter())
} else {
    let mut v = Vec::new(); // Fallback dummy values
    for i in 0..n { v.push(rand()); }
    v
};
```

```
// Most idiomatic is "filling" a std::vector declared outside
auto values = std::vector<Entry>();
if (database.has_data()) {
    values.insert(database.begin(), database.end());
} else {
    for (std::size_t i = 0; i < n; i += 1) {
        values.push(rand());
    }
}
// values is mutable for no reason here
```

loops as values

```
let outcome : i32 = loop { // loop = infinite loop
    let draw : i32 = rand();
    if draw > i32::MAX / 2 {
        break draw; // End loop AND define "return" value
    }
    log_value(draw)
};
```

Niche but no clean C++ equivalent. Can help avoid introducing a function purely due to language rules.

```
let infinite_loop : ! = loop {}; // "never" type
let unit : () = loop { break };
let outcome : i32 = 'outer : loop {
    loop {
        // loop label for multilevel break
        break 'outer 42;
    }
};
```

Sum types (or tagged unions) with pattern matching

```
enum Variant {
    Empty,
    Fail(bool),
    Tuple(i32, String),
    StructLike { data: i32 }
}
let variant = Variant::StructLike { data: 42 };
// match statement like ML languages, returns value too
let match_statement_value = match variant {
    Variant::Tuple(i, s) => {
        let s_as_i = i32::from_str(&s);
        i + s_as_i // accepts blocks & simple expressions
    }
    Variant::StructLike { data } => data,
    Fail (really) if really => return Err("unexpected"),
    _ => 0 // wildcard, not needed if all cases handled
};
```

C++17 variants are miles behind in comparison

Sum types : expressive power

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

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

Used everywhere in APIs as vocabulary :

```rust
fn Map::find(&self, key: &K) -> Option<&V>;
fn File::open(&Path) -> Result<File, io::Error>;
fn partial_eq(lhs: &T, rhs: &T) -> Option<bool>;
```

Advantages

- Type-level documentation, reduce needs for comments
- `Result` must be matched to access value
- Reminder to handle corner cases

```rust
map<K,V>::iterator map::find(const K & key) const;
// + comment for "not found" <=> iterator == map.end()
```

Match ergonomics

```
// compound patterns useful for complex control flow
let a : Option<i32> = f();
let b : Option<String> = g();
match (a,&b) {
    (None, None) => "nothing",
    (None, Some(b)) => b, // b is implicitely &String
    (Some(a), None) => "a",
    (Some(_), Some(_)) => "both",
}
```

Result has special support for "assert" pattern :

```
fn do_things() -> Result<T, io::Error> {
    let r : Result<i32, io::Error> = read_file();
    let value : i32 = r?; // equivalent to ↓
    let value = match r {
        Ok(v) => v,
        Err(e) => return Err(e)
    };
    ...
}
```

Sum types : greater precision and correctness

```
double lhs = f();
double rhs = g();
if (lhs < rhs) {
    less_case();
} else if (lhs == rhs) {
    equal_case();
} else /*lhs > rhs*/ {
    greater_case();
}
```

Can be rewritten as

```
use std::cmp::Ordering;
let value = match PartialOrd::partial_cmp(&f(), &g()) {
    Some(Ordering::Less) => less_case(),
    Some(Ordering::Equal) => equal_case(),
    Some(Ordering::Greater) => greater_case(),
    None => panic!("Nan"), // float comparison may fail !
}
```

Sum types : small bits

Enum layout optimization

Due to absence of ABI, rust can use "holes" in type spaces to fit enum tags

```
size_of::<Option<&T>>() == size_of::<T*>();
enum {...}, NonZeroUsize, bool, struct {} // other "holed" types
```

`std::optional<T&>` is not supported due to reference semantics (and has many other problems compared to rust)

Other matchings

```
while let Some(value) = iterator.next() { ... }
if let Some(value) = map.find(key) { ... }
// Infallible
let (a, b) = (12, "blah");
let Point { x, y } = intersect(line1, line2);
```

Destructuring arrived in C++17 with weird syntax

OOP (s?)

```
class A {
    private:
        int a;
        void private_method(int b) { a += b; }
    public:
        int b;
        A(int init_value) : a(init_value) { b = a * 2; }
        int get_a() const { return a; }
};
```

Rust decouples data from methods

```
struct A {
    a: i32,
    pub b: i32,
}
impl A {
    pub fn new(a: i32) -> A {
        A {
            a,
            b: a * 2
        }
    }
    fn private_method(&mut self, b: i32) { self.a += b; }
    pub fn get_a(&self) -> i32 { self.a }
}
```

Modules & visibility

Modules combine namespaces, scoping rules and file organization[5]

```
mod a { // inline module
    struct A;
    pub struct B { private: i32 };
    pub fn new_b() -> B { B { private: 42 } }
    pub struct C { pub data: i32 };
    mod b {
        // can see private elements of all parents and self
        // parents can only see public elements
        pub fn access_b(b: &B) -> i32 { b.private }
    }
    pub use b::access_b; // re-export access_b as public
}
mod c; // file module, either 'c.rs' or 'c/mod.rs'

let a = a::A; // error A private
let b = a::B { private: 42 } // cannot see field private to build B
let b = a::new_b(); // ok
let c = a::C { data: 42 }; // ok
a::access_b(&b); // ok
```

[5] diagram for rules: https://i.redd.it/1yy98srxyvx81.png

Constructors, destructors, operators...

```
// Named constructors instead of C++ constructor overloading
impl A {
    pub fn zeros() -> A {
        A { a: 0, b: 0 }
    }
}
//     ↓ Copy is done with the Clone trait !
impl Clone for A {
    fn clone(&self) -> A { ... }
}
// Move is builtin
// Destructor is another trait
impl Drop for A {
    fn drop(&mut self) { ... }
}
// Operator overloading ? A + 42
impl std::ops::Add<i32> for A {
    type Output = A;
    fn add(self, rhs: i32) -> Self::Output { ... }
}
```

OOP with traits

Rust has something close to UFCS[6]

```
let a = A::zeros();
a.get_a(); // Equivalent to ↓
A::get_a(&a);
Clone::clone(&a); // Works with trait names too
```

Traits are *interfaces* that can be defined

```
trait PrettyPrint {
    fn pretty_print(&self);
}
impl PrettyPrint for A {
    fn pretty_print(&self) {
        println!("A(a={}, b={})", self.a, self.b)
    }
}
impl PrettyPrint for String { // extending "sealed" types !
    fn pretty_print(&self) { println!("String({})", self) }
}
```

[6] https://en.wikipedia.org/wiki/Uniform_Function_Call_Syntax

Static polymorphism

Traits are *capabilities* of types. Similar to C++20(?) *concepts*[7].

```
fn add_numbers<T>(T a, T b) -> T { a + b }
```

```
error[E0369]: cannot add `T` to `T`
 --> a.rs:1:40
  |
1 | fn add_numbers<T>(a: T, b: T) -> T { a + b }
  |                                      - ^ - T
  |                                      |
  |                                      T
  |
help: consider restricting type parameter `T`
  |
1 | fn add_numbers<T: std::ops::Add<Output = T>>(a: T, b: T) -> T { a + b }
  |                 ++++++++++++++++++++++++++
```

Rust requires the capabilities *up front*, making clear what *interface* is required for a template to work (with a name)

```
impl HashMap<K,V> {
    pub fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
        where K: Borrow<Q>, Q: Hash + Eq,
}
```

[7] https://en.cppreference.com/w/cpp/experimental/constraints

Dynamic polymorphism

```
// C++ historical interfaces force dynamic behavior
class A {
    virtual void f(); // virtual class
    A_vtable * vtable_ptr; // adds implicit vtable field to class
};
```

Rust uses on-demand dynamic polymorphism defined by traits

```
let a : A = A::zeros();
// compiler dyn object, {*a, ptr_vtable}, non owning!
let a_dyn : &dyn PrettyPrint = &a;
a_dyn.pretty_print();
// type erasure
fn pp_dyn(obj: &dyn PrettyPrint) { obj.pretty_print() }
pp_dyn(a_dyn);
// owning with unique_ptr
let a_box_dyn = Box::new(A::zeros()) as Box<dyn PrettyPrint>;
// stdlib Trait providing downcast facilities
use std::any::Any;
```

OOP Limitations

- No overloading in Rust. Can be emulated with traits but not ergonomic nor needed in practice.
- No specialization. Proposals in debate.
- Templates have similar expressing power.
  - No variadic templates in Rust (but can be emulated with effort)
  - Recent update with restricted constants
  - errors are less awful on average. It can still be madness if overused like some C++ frameworks
- Dynamic polymorphism
  - Lesser polished part of the trait system
  - Does not work with `Self`-referencing traits
  - Explicit casts to dyn objects sometimes needed

Borrow checker : motivation

```cpp
// Use after free
auto up = std::make_unique<Class>(...);
Class & ref = *up;
up = nullptr; // ref is dangling
ref.do_something(); // uh oh

// Double free
{
    auto up = std::make_unique<Class>(...);
    auto up2 = std::unique_ptr<Class>(up.get());
} // uh oh when destructors run

// Vector invalidation kills
auto vec = std::vector<T>(...);
T & ptr_to_i = vec[i];
vec.push_back(T(...)); // may invalidate storage, uh oh

// Reference types : hidden danger. No warnings with gcc -Wall c++17
auto vec = std::vector<std::string_view>(...);
vec.push_back("static blah"); // no risk
vec.push_back(std::string("dynamic") + "blah"); // ref to tmp string, uh oh

// Beware of references to temporary and reference lifetime extension
auto pair = std::minmax(std::string("blah"), std::string("uh oh")); // pair<const T&, const T&>
f(pair.min, pair.max); // UB
auto min = std::min(std::string("blah"), std::string("uh oh")); // ok, auto=string (copy)
// Due to ↓
std::pair<const T&, const T&> std::minmax(const T& a, const T& b);
const T& min(const T& a, const T& b);
```

Microsoft vulnerability statistics : $\approx 70\%$ due to memory bugs

Borrow checker : rust's answer

```rust
let mut vec : Vec<i32> = vec![42, 0, 2, 3, 4];
let ref_to_2 : &i32 = &vec[2];
vec.push(-1);
println!("vec[2]={}", ref_to_2);
```

```
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable
--> a.rs:4:5
    |
  3 |     let ref_to_2 : &i32 = &vec[2];
    |                            --- immutable borrow occurs here
  4 |     vec.push(-1);
    |     ^^^^^^^^^^^^ mutable borrow occurs here
  5 |     println!("vec[2]={}", ref_to_2);
    |                           -------- immutable borrow later used here
```

```rust
let mut vec : Vec<&str> = Vec::new();
vec.push("static !"); // &'static str
vec.push(&(String::from("dynamic") + " !"));
println!("vec[1]={}", vec[1]);
```

```
error[E0716]: temporary value dropped while borrowed
--> a.rs:4:11
    |
  4 | vec.push(&(String::from("dynamic") + " !"));
    |           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ - temporary value is freed at the end of this statement
    |           |
    |           creates a temporary which is freed while still in use
  5 | println!("vec[1]={}", vec[1]);
    |                       --- borrow later used here
    |
    = note: consider using a `let` binding to create a longer lived value
```

Borrow checker core concepts

Link a reference to borrowed value to prevent outliving it

```
let dangling_ref = {
        let a = 42;
        let r : &i32 = &a; // &'a i32
}; // a dies here so r must not outlive it
```

Borrow lifetimes are transitive and part of APIs

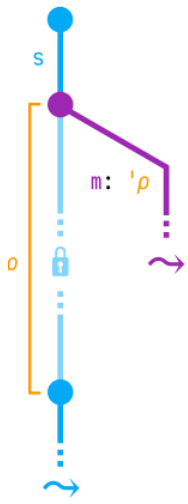
```
impl Map<K, V> {
    fn find<'s, 'k>(&'s self, key: &'k K) -> Option<&'s V>;
}
```

Reference types must expose lifetimes

```
struct StringView<'s> { r: &'s str }
let static_sv = StringView { r: &"blah" }; // 'static lifetime
let local_value = String::from("local");
let local_sv = StringView { r: &local }; // must be outlived by content
let fail_sv = StringView { r: &String::from("tmp") }; // not outlived
```

Borrow checker : & VS &mut

mutable borrow

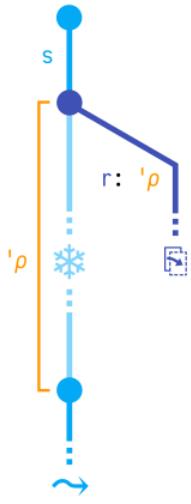



```
let mut s = String::new();

{ let m = &mut s;

    (can move m)
    (can downgrade m as &_)
    (cannot copy m)
    (cannot use s at all)

}
```

&mut

exclusive control (reference itself is movable)
mutable
cannot move referent
must not outlive its referent

borrow




```
let s = String::from("hello");

{ let r = &s;

    (can copy r)
    (can still &s)
    (cannot &mut s)
    (cannot move s)

}
```

&

nonexclusive control (reference itself is copyable)
exteriorly immutable
cannot move referent
must not outlive its referent

move (for types that do not implement Copy)


```
let s = String::from("hello");

let b = s + " world";
```

(**cannot** use s anymore)

copy (for types that do implement Copy)


```
let i = 42;

let j = i + 1;
```

(can still use i and j)

Borrow checker : capabilities

This enables *safe* reference types : lots of vocabulary

- *array slices* `&[T]` `&mut[T]`, vs owned types (`Vec<T>`)
- *string slices* `&str` = `&[u8]`+UTF-8, vs owned (`String`)
- Iterators (later)
- Great for parsers : cutting the input in pieces with slices

General tool for modeling of shared VS exclusive patterns :

```
// any number of shared &Mutex<T> can coexist, plus
fn Mutex<T>::lock<'m>(&'m self) -> Guard<'m, T>;
fn Guard<'m, T>::deref_mut<'g>(&'g mut self) -> &'g mut T;
fn Mutex<T>::get_mut<'m>(&'m mut self) -> &'m mut T; // no lock
```

Great for multi-threading safety

- Shared read-only VS exclusive mutable access
- References through thread boundaries limited though

Borrow checker : downsides

Rejects valid code that cannot be proved

```
let mut vec = vec![42, 0, 2, 3, 4];
let ref_0 = &mut vec[0]; // ref_0 tied to &mut vec
let ref_1 = &mut vec[1]; // ref_0 and ref_1 cannot coexist
// Workarounds for some patterns :
fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]);
```

API tradeoff

- Owned with copies for simplicity
- Reference for precision, efficiency, but complex

Adapt to ownership mindset

- Which object owns data, what access patterns
- Graphs are difficult (usually graph + runtime checked indexes)
- Learning curve : *fighting the borrow checker*
- Lots of new compiler error messages

Borrow checker : past improvements

Non Lexical Lifetimes (NLL)

Lifetimes initially tied strictly to scoping ⇒ artificial scoping to end lifetimes

```
fn increment(o: &mut Option<i32>) {
    match o {
        None => *o = Some(0),
        Some(i) => *i += 1
    }
}
```

Match syntax sugar

```
fn increment(o: &mut Option<i32>) {
    match o {
        &Some(ref mut i) => *i += 1, // Old
        _ => ...
    }
}
```

- Lots of work on error messages
- Implicit lifetimes on functions (not always good)

LLVM & the noalias saga

`rustc` is built on top of the LLVM compiler

- Lots of supported targets[8] : x86, ARM, ...
- Benefits from LLVM optimizations

noalias

Compiler optimization can benefit from knowing that pointers *do not alias*.
In C (not C++) this can be indicated by the rarely used *restrict* keyword :

```
void my_copy(char *restrict src, char *restrict dest, size_t n) {
    // src[0..n] MUST NOT alias dest[0..n]
}
```

Rust generates *a lot* of `noalias` guarantees everywhere :

```
fn my_copy(dest: &mut[u8], src: &[u8]) {
    // dest and src are noalias by semantics of &mut
}
```

LLVM noalias support was seldom used before, so bugs are regularly found[9]

[8] https://doc.rust-lang.org/rustc/platform-support.html
[9] https://github.com/rust-lang/rust/issues/54878

Borrow checker : unsafe for unprovable patterns

Manual pointer manipulation

```
let p : *const i32 = &42; // Convert &i32 to *const i32 is ok
*p; // error[E0133]: dereference of raw pointer is unsafe and
    // requires unsafe function or block
unsafe { *p } // ok, must ensure ptr is valid,...
// + noalias properties for mut ptr !
```

Optimized alternatives to methods

```
impl &[T] {
    fn get<I>(&self, index: I) -> Option<&T>;
    unsafe fn get_unchecked(&self, index: I) -> &T;
}
```

- Required to implement things like `Mutex`
- Prefer small abstractions : less unsafe to validate
- Find external implementations
- Rustonomicon[10] + MIRI model checker

[10] https://doc.rust-lang.org/nomicon/

stdlib

Philosophy

- Do not include too much (closer to C++ than Python)
- Role : interoperability, API to system for main cases
- Lots of vocabulary types : slice, containers, Option
- Example of feature use : typed APIs, lifetimes,...
- Provide stuff with 3rd party crates

Result

- Good quality in my opinion
- Allowed to deprecate stuff (still young)
- Could update hashmap impl using a crate
- Benefits from excellent documentation tooling
- async : no default runtime provided

stdlib : traits

```
trait Default {
    fn default() -> Self;
}
trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
trait Hash {
    fn hash<H: Hasher>(&self, state: &mut H);
}
trait From<T> {
    fn from(T) -> Self;
} // Self::from(T)
trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
} // Vec<T> deref to &[T], etc
unsafe auto trait Sync { }
unsafe auto trait Send { }
...
```

stdlib : smart pointers

```
struct A { ... }
impl A { fn f(&self); }

let b : Box<A> = Box::new(A { ... }); // unique_ptr<A>
let rc : Rc<A> = Rc::new(A { ... }); // shared_ptr<const A>
let arc : Arc<A> = Arc::new(A { ... }); // with atomic counter
let weak : Weak<A> = rc.downgrade(); // weak pointer
// All use Deref<T> for easy access
b.f(); rc.f(); ...
// Can be moved-from with nice semantics
let b_value: A = *b;
match Rc::try_unwrap(rc) {
    Ok(a) => a, // T value if rc count == 1
    Err(rc) => rc, // Gives back rc if not
}
// Rc coupled with RefCell dynamic borrow checker (RW lock)
// Enables mutability, otherwise Rc<T> only gives &T
let rc = Rc::new(RefCell::new(42));
*rc.borrow_mut() += 4;
```

stdlib : iterators

C++ : "pointers" in data structures. C++20 *ranges* : still flawed

```
for (auto it = cont.begin(); it != cont.end(); ++it) { ... }
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );
```

Rust's model : iterator is a state machine producing values

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // many defaulted "range combinators" methods
}
let vec = Vec::from_iter((0..10).map(|i| i * 2));
for reference in (&vec) { ... }
for value in vec { ... }
let mut iter = vec.into_iter();
while let Some(v) = iter.next() { ... }
for i in 0..vec.len() { ... }
```

Iterator types usually reference containers : no invalidation risk

stdlib : containers

Container lineup has all the classics[11] :

- `Vec<T>, VecDeque<T>, HashMap<K, V>, BTreeMap<K, V>, sets...`
- Personal favorite : `SortedVec<T: Ord>` made with sort

HashMap interesting features

```
// Entry API : lookup once
match map.entry(key) {
    Entry::Vacant(vacancy) => vacancy.insert(value),
    Entry::Occupied(occupied) => f(occupied.get_mut()),
    // or remove
}
// Equivalent key
let map: HashMap<String> = ...;
map.get("blah"); // compares &str values instead of tmp String
// Due to
fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
    where K: Borrow<Q>, Q: Hash + Eq,
```

[11] https://doc.rust-lang.org/std/collections/index.html

numeric and conversions

C++ numeric types have many problems (due to C legacy)

```
unsigned int a; short b; // Weird sizing guarantees
unsigned int c = -5; // Automatic doomed conversion
int c = INT_MAX + 1; // overflow is UB
char c = 'Z'; char i = 42; char *bytes; // painful ambiguity
```

Rust fixes most of those

```
let a: u32; let b: i16; // Explicit sizing
let a: u32 = -1; // Error, no implicit conversion
i32::MAX.saturating_add(1) == i32::MAX; // select behavior
i32::MAX + 1; // panic on debug, wrap around in release.
let c: char = '→'; let i: u8; let bytes : &[u8];
```

Conversions

```
let a = u32::from(42u8); // Infallible conversions
let b = u32::try_from(42i8)?; // May fail
let c = 1.0f64 as u32; // Raw, discouraged, prefer specific APIs
```

stdlib : miscellaneous

- `UTF-8` in `&str`, conversions, validation
- IO
  - Usual file io, includes socket IO too
  - Separate `Read` and `BufRead` traits (same with `Write` side)
  - `std::iostream` OOP hierarchy is legacy madness
  - Text formatting system
- OS interop / FFI
  - Exposes most of libc with safe wrappers if available
  - `CString` and `libc::c_int` for FFI
  - `OsString` for interop
- *New type* pattern used sometimes

Macros for code generation

- C++ libraries sometimes overuse templates for code generation
- Some C++ proposals for a meta-language
- Rust choice : use AST-aware *macros* for code generation

```
macro_rules! __lazy_static_create {
    ($NAME:ident, $T:ty) => {
        static $NAME: $crate::lazy::Lazy<$T> = $crate::lazy::Lazy::INIT;
    };
}
```

Typical macro uses (look for the '!')

```
panic!("message"); // abort
let a = todo!("compute a");
match map.find(key) {
    Some(value) => ...,
    None => unreachable!()
}
// Text formatting
let msg: String = format!("vec[{}]={:x}", i, vec[i]);
eprintln!("[{}] warning: {}", time, error_msg);
write!(&mut file, "entry_{}\t{}\t{}\n", a, b, c);
```

Derive macro

Builtins

```rust
// Each derive calls a macro to generate the impl
#[derive(Debug, Clone, PartialEq, Eq, hash)]
struct A(i32, String);
```

Custom complex derives, using *procedural macros* (rust code)

```rust
#[derive(clap::Parser, Debug)]
#[clap(author, version, about, long_about = None)]
struct Args {
    /// Name of the person to greet
    #[clap(short, long, value_parser)]
    name: String,

    /// Number of times to greet
    #[clap(short, long, value_parser, default_value_t = 1)]
    count: u8,
}

let args = Args::parse();
```

Test system

`cargo test` : auto detect tests from `Cargo.toml`

```rust
fn add(lhs: i32, rhs: i32) -> i32 { lhs + rhs }

// Explicit tests inside module, can inspect private state
#[test]
fn test_add() {
    assert_eq!(add(1, 2), 3);
    assert!(add(0, 1) == 1);
}

// Local module : if sub functions or imports needed
#[cfg(test)] // conditional : only keep if tests
mod tests {
    use super::add;
    fn helper() -> i32 { 3 }
    #[test] // mark as test
    fn test_add() {
        assert_eq!(add(1, 2), helper());
    }
}
```

Doctest

```rust
/// Runs until `future` finishes, and return its value.
///
/// Must not be called inside itself, or it will panic:
/// ```should_panic
/// star::block_on(async {
///     star::block_on(async {}); // panics !
/// });
/// ```
pub fn block_on<F: Future + 'static>(future: F) -> Result<F::Output, io::Error> {
    ...
}

/// `JoinHandle<T>` represents the completion (and return value) of a spawned Task.
///
/// It implements [`Future`] to support asynchronously waiting for completion:
/// ```
/// let f = async {
///     let handle = star::spawn(async { 42 });
///     handle.await
/// };
/// ```
///
/// Completion can be manually tested in a non-blocking way:
/// ```
/// star::block_on(async {
///     let handle = star::spawn(async { 42 });
///     let test = handle.try_join();
///     assert!(test.is_err()); // Should not have time to run
/// }).unwrap();
/// ```
pub struct JoinHandle<T>(TaskFrameHandle<dyn TaskPollJoin<Output = T>>);
```

Async

Compiler support to generate state machines for *Futures*

```
let future = async {
    let address = f();
    let connection = connect(address).await?; // stop 1
    let data = gather_data();
    let send_result = connection.send(data).await?; // stop 2
    connection.close();
};
// Roughly translated to
enum State {
    Init,
    Stop1 { connection },
    Stop2 { connection },
    Ended,
}
impl Future for State {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let new_state = match self {
            Init => ..., // advance to stop1
            Stop1 => ..., // check if stop1 can advance, then stop2
            ...
        }
        *self = new_state;
    }
}
```

Lots of auxiliary machinery : `Pin`, runtime (executor & reactor)

Some not-so-good things

- Explicit cases may be tedious (Add overloads for value, &self, etc)
- ndarray seems way less advanced than Eigen in C++
- numerical conversions lack some traits (f64 to int)
- Not well designed to recover from OOM errors
- Some theoretical problems : `mem::forget()`
- Compilation artifacts directories can easily reach $1G$

Will the language not suffer too much for getting older ?