



IP PARIS



# Crates and modules

NET7212 — Safe System Programming (in Rust)

Samuel Tardieu   Stefano Zacchiroli

2023-10-24



Distributing Rust code: crates and packages

Organizing Rust code: the module system

Testing Rust code

Property testing Rust code

Fuzzing Rust code

Takeaways

# Distributing Rust code: crates and packages

## Code reuse in Rust

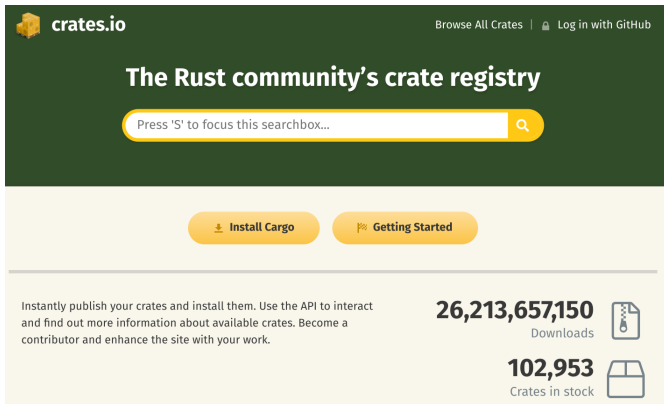
- All non trivial software projects deal with **code reuse**
  - As *downstream*: reusing software components (e.g., libraries) written by others (possibly from the same organization or team!) to avoid reinventing the wheel; and/or
  - As *upstream*: providing software components to others for the symmetric need
- Rust provide mechanisms and tools to facilitate code reuse both **in the small** (within a project) and **in the large** (across projects)
- In this lecture we will introduce you to both of them, starting from in-the-large code reuse

## (Open source) software ecosystems

- A **software ecosystem** is a set of *software components*, packaged as individually distributable software units (*packages*), that *provide* functionalities to other components and can *depend* on functionalities provided by other components. (Szyperski et al., 2002 and 2003)
- In the context of *free/open source software (FOSS)*, software components are freely distributable and inspectable, enabling large-scale software reuse.
- Several large **FOSS ecosystems** exist, e.g., npm (2M packages), PyPI (400k), Golang (800k), Maven (400k), etc.
  - See: <https://ecosyste.ms/> and <https://libraries.io> for stats and more.


## The Rust ecosystem

- <https://crates.io> is a public repository for publishing and retrieving **Rust source packages**
- It is the reference repository for the **Rust (open source) ecosystem** and community; it is integrated by default with **cargo**
- As of January 2023, Crates.io hosts ~100k Rust packages



The screenshot shows the crates.io website homepage. At the top left is the crates.io logo. To the right, there are links for "Browse All Crates" and "Log in with GitHub". The main heading is "The Rust community's crate registry". Below this is a search bar with the placeholder text "Press 'S' to focus this searchbox...". There are two buttons: "Install Cargo" and "Getting Started". At the bottom, there is a statistics section with the text "Instantly publish your crates and install them. Use the API to interact and find out more information about available crates. Become a contributor and enhance the site with your work." and two statistics: "26,213,657,150 Downloads" and "102,953 Crates in stock".

## crates.io — Package homepage example

 **pathfinding** 2.2.1

Pathfinding, flow, and graph algorithms

[#dijkstra](#) [#flow](#) [#shortest-path](#) [#graph](#) [#astar](#)

---

[Readme](#) 88 Versions Dependencies Dependents

---

### pathfinding

[crates.io](#) [v2.2.1](#) [docs](#) [passing](#) [license](#) [Apache-2.0/MIT](#)

This crate implements several pathfinding, flow, and graph algorithms in [Rust](#).


#### Algorithms


The algorithms are generic over their arguments.


#### Directed graphs

- **A\***: find the shortest path in a weighted graph using an heuristic to guide the process.
- **BFS**: explore nearest successors first, then widen the search.
- **DFS**: explore a graph by going as far as possible, then backtrack.
- **Dijkstra**: find the shortest path in a weighted graph.
- **Edmonds Karp**: find the maximum flow in a weighted graph.

#### Metadata

 2 months ago

 Apache-2.0 or MIT

 132 kB

#### Install

Add the following line to your Cargo.toml file:

```
pathfinding = "2.2.1"
```

#### Homepage

[rfc1149.net/devel/pathfinding...](#)

#### Documentation

[docs.rs/pathfinding/2.2.1](#)

#### Repository

## docs.rs — Package documentation example

Crates.io supports automated generation and publishing of package documentation (more on this later in this lecture) on <https://docs.rs>.

The screenshot shows the documentation page for the 'pathfinding' crate on docs.rs. The top navigation bar includes 'DOCS.RS', 'pathfinding-2.2.1', 'Platform', 'Feature flags', 'Releases', 'Rust', and a search bar. The main content area features a search bar with the text 'Click or press 'S' to search, '?' for more options...'. Below this, the crate name 'Crate pathfinding' is displayed with a version indicator '[-][src]'. A list of tags shows 'crates.io v2.2.1', 'docs: passing', and 'license: Apache-2.0/MIT'. The description states: 'This crate implements several pathfinding, flow, and graph algorithms in Rust.' Under the 'Algorithms' section, it notes 'The algorithms are generic over their arguments.' The 'Directed graphs' section contains a bulleted list of algorithms: A\*, BFS, DFS, Dijkstra, Edmonds Karp, and Fringe. The left sidebar contains a 'Crate pathfinding' section with 'Version 2.2.1' and a link to 'See all pathfinding's items'. Below this are 'Modules' and 'Macros' sections, with 'Modules' listing 'directed', 'grid', 'kuhn\_munkres', and 'matrix'.



## Package & crates

### Terminology

**Crate** a tree of Rust modules (more on this later, for now consider this to mean: “a software component implemented in Rust”) that can be compiled to obtain either an executable binary application (making the crate a **binary crate**) or a reusable *library* (**library crate**).

**Package** a software distribution unit containing one or more crates.

- Rust packages are *source* packages: they are distributed in source code form (e.g., as tarballs) from Crates.io to users and built on their computers on the fly
- Package formation rules:
  - A package must contain at least one crate
  - A package must contain at most one library crate
  - Examples of valid packages: a single binary crate; a single library crate; one library crate with several binary crates. Invalid package: two library crates.

## Package & crates — example

```
$ cargo new [--bin] hello # --bin is the default
Created binary (application) `hello` package
```

- Creates a *package* called `hello`, rooted in the `./hello/` directory
- `hello` contains a single binary crate, whose root source code file is located in `src/main.rs` (within the package root directory)

```
$ cargo new --lib libhello
Created library `libhello` package
```

- Package `libhello`, containing a single library crate with root source code file at `src/lib.rs`
- Note that without a binary crate you cannot `cargo run`:

```
$ cd libhello
$ cargo build
Compiling libhello v0.1.0 (/tmp/libhello)
Finished dev [unoptimized + debuginfo] target(s) in 0.37s
$ cargo run
error: a bin target must be available for `cargo run`
```

## Package layout

Noteworthy files within a Rust package on disk:

- `Cargo.toml`: metadata about the package, including dependencies
- `Cargo.lock` (automatically generated): allows to “pin” dependencies to specific versions, enabling rebuilding in the future crates in a fixed environment (warning: builds are not always bit-by-bit reproducible though!)
- `src/`: Rust source code goes in here
- `target/`: all build artifacts are stored here, for both your own code and all your dependencies. `cargo clean` will nuke this dir for you (as `rm -rf` will). DO NOT CHECK THIS DIRECTORY INTO VERSION CONTROL: it's big and not versioning friendly; you might want to exclude it from your backups too.
- Naming/path conventions are very important and handy within Rust packages, e.g.:
  - `src/lib.rs`: if present, denotes the root of a library crate in the package
  - `src/main.rs`: if present, denotes the root of a singleton binary crate in the package
  - `src/bin/*.rs`: if present, each of these denotes the root of a binary crate (in a package containing multiple binary crates)

## Cargo — Rust's Swiss Army knife

We have already used `cargo` and will learn new tricks with it in this lecture. To take a step back:

- Cargo is both a **build system** for Rust projects and a **package manager** for the Rust ecosystem.
- It is a bit of a “Swiss Army knife” that Rust programmers use daily to automate a lot of tasks.
- The Cargo CLI is organized around sub-commands, which can be extended installing new crates.
  - E.g., the `cargo fuzz` command which we will use later in this lecture is provided by the `cargo-fuzz` crate.
- By the end of this lecture and lab you will have used (at least) the following `cargo` commands:  
`build`, `check`, `clean`, `doc`, `new`, `add`, `run`, `test`, `bench`, `fuzz`
  - Try `cargo COMMAND --help` on any of them to access their documentation.

### Links

cargo: [The Cargo Book](#) 

# Organizing Rust code: the module system

## Modular programming and module systems

- **Modular programming** is the practice of implementing the functionalities that a software system should implement (as per specification) into independent, interchangeable **software modules**. It's an embodiment of the more general design principle of *separation of concerns* [↗](#) (Dijkstra, 1974).
- A **module system** is the set of abstractions and tooling that a programming language provides to support modular programming in practice.
  - All modern programming languages have module systems (e.g., Java, Python, Scala, OCaml, C++ (since C++20)); some legacy system programming languages still don't (e.g., C).
- The main functionalities that module systems offer are:
  - **Encapsulation**: definition of the interface and implementation of each module, with support for abstract data types (ADT).
  - Module organization into a (**hierarchical**) **namespace**, to avoid name clashes, fine-grained visibility control for ADTs, and more.

Rust has an expressive module system, with support for these features.

Let's dive into it to learn how to facilitate “in-the-small” code reuse within your projects.

## Rust modules

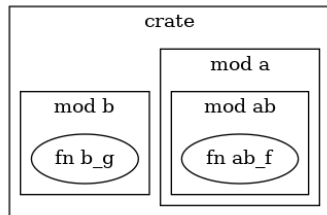
```
1 mod spores {
2     use cells::{Cell, Gene};
3
4     /// A cell made by an adult fern.
5     pub struct Spore {
6         ...
7     }
8     /// Simulate the production of a spore by meiosis.
9     pub fn produce_spore(factory: &mut Sporangium) -> Spore {
10        ...
11    }
12    /// Extract the genes in a particular spore.
13    pub(crate) fn genes(spore: &Spore) -> Vec<Gene> {
14        ...
15    }
16    /// Mix genes to prepare for meiosis (part of interphase).
17    fn recombine(parent: &mut Cell) {
18        ...
19    }
20    ...
21 }
```

- A Rust module is a collection of **items**: structures, functions, (sub-)modules, etc.
- Each item has a **visibility**: items are private by default, making them accessible only from *within* the defining module (including any of its sub-modules).
- The **pub** keyword makes an item public, allowing to access it from *outside* the module as well (more details on this later).

## Module hierarchy and paths

As modules can appear as items in other modules, Rust modules naturally form a **tree hierarchy**.

```
1 // src/main.rs
2 mod a {
3     mod ab {
4         fn ab_f() { }
5     }
6 }
7 mod b {
8     fn b_g() { }
9 }
```



We can reference items in the module hierarchy using (absolute or relative) **paths** formed by item names separated by `::`

- Absolute path from crate root: `crate::a::ab::ab_f`
- Relative paths (from within module `a`):
  - `ab::ab_f`
  - `self::ab::ab_f`
  - `super::b::b_g`



## struct visibility — abstract data types (ADTs) in Rust

```
1 mod a {
2     pub struct MyBox {
3         value: u8,
4     }
5     impl MyBox {
6         pub fn new(value: u8) -> Self {
7             // XXX enforce box invariants here
8             MyBox { value }
9         }
10        pub fn get(self: &Self) -> u8 { self.value }
11        pub fn set(self: &mut Self, v: u8) {
12            // XXX enforce box invariants here
13            self.value = v
14        }
15    }
16 }
17 fn main() {
18     let mut p = a::MyBox::new(41);
19     p.set(p.get() + 1);
20     // p.value = 42; // error: "private field"
21 }
```

- Structures have fine-grained visibility rules, you can define the visibility of:
  - the structure itself,
  - each of its field,
  - associated functions (e.g., constructor, getter, setter).

## struct visibility — abstract data types (ADTs) in Rust (cont.)

```
1 mod a {
2     pub struct MyBox {
3         value: u8,
4     }
5     impl MyBox {
6         pub fn new(value: u8) -> Self {
7             // XXX enforce box invariants here
8             MyBox { value }
9         }
10        pub fn get(self: &Self) -> u8 { self.value }
11        pub fn set(self: &mut Self, v: u8) {
12            // XXX enforce box invariants here
13            self.value = v
14        }
15    }
16 }
17 fn main() {
18     let mut p = a::MyBox::new(41);
19     p.set(p.get() + 1);
20     // p.value = 42; // error: "private field"
21 }
```

- You can implement ADTs as follows:
  - Make the struct public (so that it can be returned to code outside the module)
  - Make manipulation methods (where you will implement your invariant enforcement!) public
  - Do *not* make its fields public (so that invariant enforcement cannot be circumvented)
- You can also be more fine grained and make *some* struct fields public and some private.
  - It is safer than in C, because mere field access will not allow client code to change its content. But still, proper ADTs are better.

## Modules in separate files

```
// src/main.rs
mod a {
  mod ab {
    mod abc {
      ...
    }
    mod abd {
      ...
    }
    mod abe {
      ...
    }
  }
}
mod b {
}
```

- We can write an entire crate in a single file by arbitrary nesting modules in it.
- However, doing so in real code will quickly become unwieldy.
- Rust allows to separate modules in dedicated source code files to make this more manageable. It also allows to switch back and forth between modules-in-a-single-file and modules-in-separate-files to cope with the evolution of your software.

### Source code layout for modules

- Wherever you have `mod a { ... }` (within, say `src/main.rs`) you can replace it with `mod a;` (note the trailing semicolon). And vice-versa.
- It will make Rust look for *either* `src/a.rs` or `src/a/mod.rs` and its content (without surrounding `mod { ... }`) as the module content. (Having *both* will trigger an error.)

## Modules in separate files — example

```
// src/main.rs
pub mod a {
    pub mod ab {
        pub mod abc {
            pub fn hello() {
                print!("Hello, ");
            }
        }
    }
}

pub mod b {
    pub fn world() {
        println!("world!");
    }
}

fn main() {
    a::ab::abc::hello();
    b::world();
}
```



```
// src/main.rs
pub mod a {
    pub mod ab {
        pub mod abc; // -> src/a/ab/abc.rs
    }
}

pub mod b; // -> src/b/mod.rs
fn main() {
    a::ab::abc::hello();
    b::world();
}
```

```
// src/a/ab/abc.rs
pub fn hello() { print!("Hello, "); }
```

```
// src/b/mod.rs
pub fn world() { println!("world!"); }
```

Note how: (1) Rust enforces that the code directory structure matches module nesting; (2) client code (`main` here) is unaffected by source code layout changes.

## Shortening module paths with `use`

- The `use` keyword allows to **import definitions** from a module into the current *scope* (possibly the entire file, possibly a `{ ... }` block).
- It allows to shorten/refactor frequent use of the same paths.

```
// src/main.rs
mod a {
  mod ab { fn ab_f() {} }
  mod b { fn b_g() {} }
}
```

```
fn f() { a::ab::ab_f(); } // full path from crate (main.rs)
```

```
use a::ab;
fn f() { ab::ab_f(); } // relative path to a::ab (note: repeating "ab" as starting point)
```

```
use a::{ab, b}; // multiple imports with shared path prefix
fn g() { ab::ab_f(); b::g(); }
```

```
use a::*; // glob import (discouraged, except for precludes; more on this later)
fn g() { ab::ab_f(); b::g(); }
```

## Shortening module paths with `use` (cont.)

- You can **import a module itself** together with other entities it contains with `self`:

```
1 use std::io::{self, Result}
2 fn read_something() -> Result<String> { ... io::read(...) ... }
```

- You can resolve **import-time naming conflicts** by renaming imported entities with `as`:

```
1 mod a { pub struct S { pub v: u8 } }
2 mod b { pub struct S { pub w: u8 } }
3 use a::S;
4 // use b::S; // error[E0252]: the name `S` is defined multiple times
5 use b::S as BS;
6 fn f() { BS { w:42 }; }
```

- You can **shadow names** import via glob imports (but not explicitly imported ones):

```
1 // use std::io::{self, Error};
2 // type Error = String; // error[E0255]: the name `Error` is defined multiple times
3 use std::io::*;
4 type Error = ... ; // shadowing io::Error with our own, no error
```

## use — what to import

We can also import directly the leaf item of interest, e.g., a function in the following example:

```
// src/main.rs
mod a {
  mod ab { fn ab_f() {} }
  mod b { fn b_g() {} }
}
```

```
use a::ab::ab_f;
fn f() {
  ab_f();
}
```

In idiomatic Rust code we tend to:

- Import types directly (e.g., structures, enumerations), so that we can name it directly in constructors, pattern matching, etc. E.g., `use a::MyBox;` then `MyBox::new()`.
- Import the parent module of a function. E.g., `use a::ab` followed by `ab::ab_f()`.

## The standard prelude

- In terms of what is already imported, crates do not start with a *clean slate* of no imported items.
- A “prelude” is a list of items that Rust programs should import to work well in a given context
- The **standard prelude** is the prelude of the [Rust Standard Library](#). It is imported by default in all Rust programs. It contains types we have already used, for example:
  - the `std::marker::Copy` trait
  - Option variants: `std::option::Option::{self, Some, None}`
  - Result variants: `std::result::Result::{self, Ok, Err}`
  - the String type: `std::string::String`
- The standard prelude allows to use them without knowing their location in the stdlib module tree.
- The standard prelude can evolve over time. Its current version is `std::prelude::rust_2021`. See: <https://doc.rust-lang.org/std/prelude/>
- Other preludes exist, for example (quoting <https://doc.rust-lang.org/std/io/prelude/>):  
*The purpose of this module is to alleviate imports of many common I/O traits by adding a glob import to the top of I/O heavy modules:*

```
use std::io::prelude::*;
```



## More on visibility

Item visibility is more fine grained than public/private:

- (default): private, defining module and below
- `pub`: public, free for all
- `pub(in path)`: make the item visible for all modules in `path`; `path` must be an ancestor of the module the item belongs to
- `pub(crate)`: make the item visible in the entire crate
- `pub(super)`: make the item visible in the parent module (and below)
- `pub(self)`: make the item visible in the current module (default, useless)

Other restrictions apply and might bite you:

- You cannot make visible parts of non-visible entities (e.g., fields of a non-visible `struct`).
- To access an item, *all* path steps leading to it need to be visible (e.g., when calling `a::ab::ab_f`, all of `a`, `ab`, and `ab_f` must be accessible; it is not enough for `ab_f` to be).

## Reexporting imported items with `pub use`

- A common pattern in libraries is to offer a **public API**, which is implemented on top of **private implementation details** that might evolve over time and/or might have alternative implementations provided by private sub-modules.
- The `pub use` directive instructs the module system to re-export from the current module items imported from elsewhere.
  - It avoids the anti-pattern of implementing wrapper functions around internal functions, relying on the compiler to inline them away to avoid performance penalties.

```
1 mod implementation {
2     pub mod api {
3         pub fn f() {}
4     }
5 }
6
7 // This reexports the "api" from the current module. This is allowed because the "api" module
8 // itself is "pub" and we can reach it through the immediately visible "implementation"
9 // item. Without this reexport, modules outside the current hierarchy could not access
10 // "implementation::api" because they cannot access "implementation" which is not public.
11 pub use self::implementation::api;
```

# Testing Rust code

## Test types

- Rust has a **built-in unit testing framework**, which standardizes the way of implementing and running various types of automated tests:

**Unit tests** the usual deal, specific to a module, can access its private items

**Integration tests** for library crates; they test them via their public API only

**Documentation tests** (“doctests” for short) integration tests (with the same limitations) that are included in the library documentation

- **cargo test** will automatically execute all of the above tests that exist in a given package

## Assertions

- The **assertion language** used in Rust tests is based on macros. The most common ones are:
  - `assert!(cond, [message...])` panics (making the test fail) if `cond` doesn't evaluate to `true`
  - `assert_eq!(a, b, [message...])` panics if `a == b` (equality) does not hold
  - `assert_ne!(a, b, [message...])` panics if `a != b` (not equal) does not hold
  - `matches!(a, pattern)` returns `true` iff `a` matches `pattern`; handy in combination with `assert!`
- You can use the same macros in regular (non test) programs to verify assertions at runtime. They will make your program panic if they do not hold.
  - If you are worried about runtime overhead: these macros have counterparts prefixed by `debug_`, e.g., `debug_assert!` instead of `assert!`, that will be executed by program compiled in debug mode (which is the default for `cargo build`) but not programs compiled in release mode (`cargo build --release`).
- Any runtime panic during the execution of a test will make it fail. Hence you can also use `.unwrap()`, `.expect()` or similar methods on `Option<_>` and `Result<_, _>` types. They will make test fail if `None` or `Err(_)` are encountered.

## Unit tests

- **Unit tests** in Rust are *functions* that go in the same source code file of the “unit” being tested.
- What turns a function into a test is the `#[test]` attribute that should be added just before the `fn` declaration.

```
1 fn gcd(mut n: u64, mut m: u64) -> u64 {
2     assert!(n != 0 && m != 0);
3     while m != 0 {
4         if m < n {
5             let t = m;
6             m = n;
7             n = t;
8         }
9         m = m % n;
10    }
11    n
12 }
```

```
1 #[test]
2 fn test_gcd() {
3     assert_eq!(gcd(14, 15), 1);
4     assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
5                 3 * 7 * 11 * 13 * 19),
6                 3 * 11);
7 }
```

## Unit tests — conditional compilation

You can annotate an entity with `#[cfg(test)]` to conditionally compile it under `cargo test`.

```
1 struct Mutex { // N.B.: private struct, testable only by unit tests
2     owner: Option<ThreadId>,
3     waiting_queue: WaitingQueue,
4     #[cfg(test)] counter: usize, // not compiled/ignored outside of `cargo test`
5 }
6
7 #[test]
8 fn test_mutex() {
9     let mut mtx = Mutex::new();
10    // [...] Do things with the mutex
11    assert_eq!(None, mtx.owner, "mutex is not freed at end of test");
12    assert!(mtx.waiting_queue.is_empty(), "threads are still waiting on the mutex");
13    assert_eq!(2, mtx.counter, "mutex taken an incorrect number of times");
14 }
```

## Running tests

### Successful run:

```
1 $ cargo test
2 running 1 test
3 test test_mutex ... ok
4
5 test result: ok. 1 passed; 0 failed; [...]
```

### Failed run:

```
1 $ cargo test
2 running 1 test
3 test test_mutex ... FAILED
4
5 ---- test_mutex stdout ----
6 thread 'test_mutex' panicked at 'assertion failed: `(left == right)`
7   left: `2`,
8   right: `3`: mutex taken an incorrect number of times', src/lib.rs:44:5
9 note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
10 test result: FAILED. 0 passed; 1 failed
```



## Tests that should panic

When panicking is the right thing for your program to do (as per its specification), you need to inform the test harness that a given test should be considered successful *if and only if it panics*. That is what the `#[should_panic]` attribute is for.

```
1 #[test]
2 #[should_panic]
3 fn test_panic() { panic!("!"); }
4
5 #[test]
6 #[should_panic]
7 fn test_not_panic() { }
```

```
1 $ cargo test
2 running 2 tests
3 test test_panic ... ok
4 test test_not_panic ... FAILED
5
6 failures:
7
8 ---- test_not_panic stdout ----
9 note: test did not panic as expected
```

## Unit tests in a separate sub-module

`#[cfg(test)]` can be used to group unit tests in a separate *sub-module*:<sup>1</sup>

```
1 fn some_function() -> u32 { do_something_and_return_42() }
2
3 #[cfg(test)]
4 mod test { // Any legal module name will do
5     use super::some_function;
6
7     #[test]
8     fn test_some_function() { assert_eq!(42, some_function()) }
9 }
```

Like all modules, the test sub-module can be moved to a separate file:

```
1 #[cfg(test)] mod test; // Include only in test mode
2
3 // test.rs
4 #[test]
5 fn test_some_function() { assert_eq!(42, super::some_function()) };
```

---

<sup>1</sup>Q: why must it be a *sub-module*?

## Interlude — documenting Rust source code

- Rust has a built-in *documentation system* (`rustdoc`) that can **generate documentation automatically** from docstrings.
- Whereas normal (non-docstring) comments start with `//`, **docstrings** start with:
  - `///` to provide documentation for the *following* entity
  - `///!` to document the *enclosing* entity

```
1  ///! This is the documentation for the enclosing module. Do not miss function  
2  ///! [myfunc].  
3  
4  /// Doc for the following function with some **bold** and underline typefaces  
5  fn myfunc() { ... }  
6  
7  fn otherfunc() {  
8     ///! Nobody ever documents a function from inside but it could be done  
9  }
```

- The `cargo doc` command will generate automatically documentation for both the current crate and all its dependencies and store the result under `target/doc/`.

**Documentation tests** (or “doctests” for short) are a specific kind of *integration tests* that:

- provide examples about how to use a library (documentation);
- shows that the results of sample code in the library documentation correspond to what the library implementation do (testing, specification);
- ensures that implementation and documentation remains in sync (regression testing).

Being integration tests, doctests can only access the library **public API** (no private entities).

## Doctests — example

```
1 use std::ops::Range;
2
3 /// Return true if two ranges overlap.
4 ///
5 ///     assert_eq!(ranges::overlap(0..7, 3..10), true);
6 ///     assert_eq!(ranges::overlap(1..5, 101..105), false);
7 ///
8 /// If either range is empty, they don't count as overlapping.
9 ///
10 ///     assert_eq!(ranges::overlap(0..0, 0..10), false);
11 ///
12 pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool {
13     r1.start < r1.end && r2.start < r2.end &&
14     r1.start < r2.end && r2.start < r1.end
15 }
```

Each doctest block (two in the example above) is transformed into a standalone **test program** (a binary crate), which should terminate with exit code 0 for the test to be considered successful.

## Doctests — example (cont.), generated doc and test execution

### Function `ranges::overlap`

`[-]` `[src]`

```
pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool
```

`[-]` Return true if two ranges overlap.

```
assert_eq!(ranges::overlap(0..7, 3..10), true);  
assert_eq!(ranges::overlap(1..5, 101..105), false);
```

If either range is empty, they don't count as overlapping.

```
assert_eq!(ranges::overlap(0..0, 0..10), false);
```

```
$ cargo test # or "cargo test --doc" to only run doctests  
Compiling ranges v0.1.0 (file:///.../ranges)  
...  
Doc-tests ranges  
running 2 tests  
test overlap_0 ... ok  
test overlap_1 ... ok  
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

## Integration tests

Integration tests can also be shipped in separate source code files located in the `tests/` directory, at the top-level of a package directory (hence *alongside* `src/`, not *within* it).

Each function marked `#[test]` in `tests/*.rs` files will be **compiled and run as a separate binary crate** linked with the package library crate. Other functions in the file can be used as test helpers.

```
1 // tests/unfurl.rs - Fiddleheads unfurl in sunlight
2 use fern_sim::Terrarium;
3 use std::time::Duration;
4
5 #[test]
6 fn test_fiddlehead_unfurling() {
7     let mut world = Terrarium::load("tests/unfurl_files/fiddlehead.tm");
8     assert!(world.fern(0).is_furled());
9     let one_hour = Duration::from_secs(60 * 60);
10    world.apply_sunlight(one_hour);
11    assert!(world.fern(0).is_fully_unfurled());
12 }
```



`cargo test` run all tests, including integration tests.

You can run only a test suite subset with `cargo test [test_name_substring]`.

# Property testing Rust code



## Property testing



**Property testing**  is a *testing technique* introduced and popularized by the Haskell [QuickCheck](#) library .

In property testing, instead of making assertions on test results obtained on *fixed test inputs*, we:

1. **randomly select test inputs**,
2. assert **general properties** that are automatically verifiable on test input/output pairs,
3. verify that properties hold on the obtained test input/output pairs.

Particular care is applied to:

- Testing limit/degenerate input cases (e.g., min/max values for integer types; +/-Inf, NaN for floats, empty vectors and strings, etc.).
- When failing pairs are identified, **reduce** them to **simple/minimal cases** that show the issue.

Several Rust test frameworks support property testing. We will use the [proptest](#)  package, inspired by the Python [Hypothesis](#)  testing framework.

## proptest — example (1)

We want to verify that we can reverse byte by byte strings composed as follows: 2 to 4 lowercase latin characters followed by 1 to 4 Cyrillic characters.

```
1 use proptest::prelude::*;
2
3 proptest! {
4     #[test]
5     fn can_reverse(s in r"[a-z]{2,4}\p{Cyrillic}{1,4}") {
6         let r = s.bytes().rev().collect::<Vec<_>>(); # byte-reverse and collect into a byte vector
7         let _ = std::str::from_utf8(&r).unwrap();      # convert from bytes to string
8     }
9 }
```

- The `(s in ...)` argument syntax is interpreted by the `proptest!` procedural macro to randomize a string that matches the provided regular expression.
- For each random input the body of the function is executed as a standard test function.

`cargo test` will fail as follows (note the handy “minimal failing” example):

```
thread 'can_reverse' panicked at 'Test failed: called `Result::unwrap()` on an `Err` value:
Utf8Error { valid_up_to: 0, error_len: Some(1) };
minimal failing input: s = "aa\u{fe2e}"
```

## proptest — example (2)

Let's now test that the square of an integer is always greater or equal to it:

```
1 use proptest::prelude::*;
2
3 fn square(n: i32) -> i32 { n * n }
4
5 proptest! {
6     #[test]
7     fn can_square(a in any::<i32>()) {
8         assert!(square(a) >= a);
9     }
10 }
```

`cargo test` will point out the following issue:

```
thread 'can_square' panicked at 'Test failed: attempt to multiply with overflow;
minimal failing input: a = 46341'
```

Indeed, the maximum signed integer we can square this way is 46340; starting from 46341 the square overflows 32-bit signed integers.

## proptest — more information

`proptest` is highly configurable and allows to:

- Use and define **strategies** used to drive the randomization process; for example when you have a function with multiple parameters you can define strategies that choose subsequent parameters based on the previous (already-selected) ones.
- **Filter** values, requesting that the framework *try again* the randomization process until it obtains values that satisfy certain properties (drawback: slowing down the randomization process).
- Indicate how to randomize `enum` values.
- Store (under `proptest-regressions/`) *randomization seeds* that led in the past to the discovery of problematic cases, so that they can be retested again and again in the future for (non-)**regression testing**.

### Links

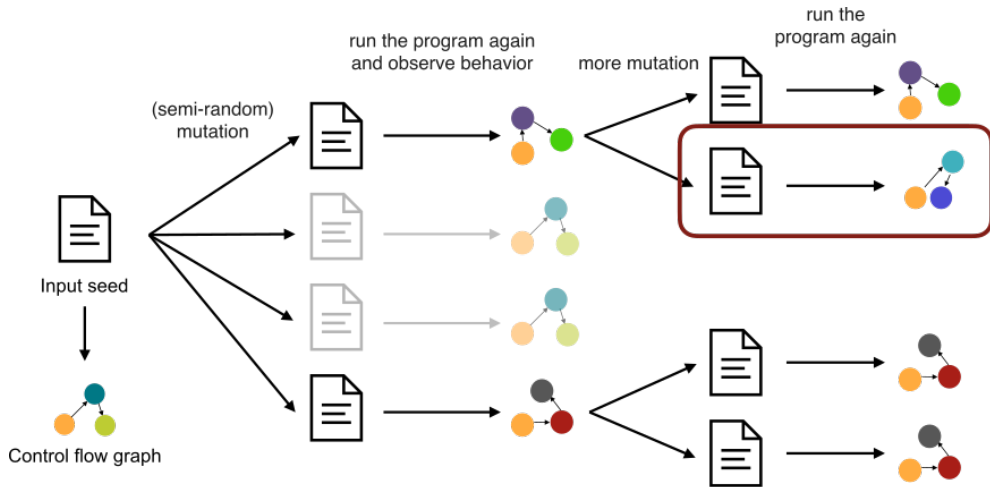
proptest: [crate](#), [documentation](#), [book](#)

## Property testing — assessment

- Property tests as implemented by `proptest` (or QuickCheck) is a good *complement* to traditional test suites composed by unit tests and integration tests.
- Property testing does not *replace* unit/integration testing and is not exhaustive (on test inputs), but can sometimes spot-check pathological and “unlucky” cases before users (or attackers) do.
- For a more thorough search of pathological inputs we should look further than property testing, specifically: *fuzzing*.

# Fuzzing Rust code

## Fuzzing (redux from a previous lecture)



(Image by Eberhardt et al., full credits on last slide.)

## cargo fuzz

`cargo fuzz` is a cargo sub-command provided by the `cargo-fuzz` crate, which offers **fuzzing support for Rust** programs, using `LibFuzzer` (which we have already used for C/C++ programs).

Let's fuzz the following function looking for issues:

```
1  /// Truncate string at n characters, adding a terminal '.'
2  pub fn truncate_at(s: &mut String, n: usize) {
3      if s.len() > n && n > 0 {
4          s.truncate(n - 1);
5          s.push('.');
6      }
7  }
8
9  #[test]
10 fn test_truncate_at() {
11     let mut s = "abcdefgh".to_owned();
12     truncate_at(&mut s, 5);
13     assert_eq!(s.as_str(), "abcd.");
14     truncate_at(&mut s, 1);
15     assert_eq!(s.as_str(), ".");
16 }
```



## cargo fuzz — setup

```
$ cd fuzzme # enter outer library crate called "fuzzme"
$ cargo test
test test_truncate_at ... ok
test result: ok. 1 passed; 0 failed
```

```
$ cargo fuzz init # create inner crate "fuzzme-fuzz", rooted at "fuzzme/fuzz/"
$ edit fuzz/fuzz_targets/fuzz_target_1.rs
```

As we did in the past with LibFuzzer, to fuzz a function we need to provide a **fuzz target**. Let's create one that passes to `truncate_at` the fuzzed payload as a mutable string:

```
1  #![no_main]
2  use fuzzme::truncate_at;
3  use libfuzzer_sys::fuzz_target;
4
5  fuzz_target!(|data: String| {
6      let mut data = data; // Make data mutable
7      truncate_at(&mut data, 5);
8  });
```

There is no assertion, but it behaves like a test, so it will fail in case of panics.

**Q:** will it find something?

## cargo fuzz — fuzzing

`cargo fuzz run` on our fuzz target quickly finds an issue and reduces the input needed to trigger the failure to a simple case:

```
1 $ cargo +nightly fuzz run fuzz_target_1
2 ==3297139== ERROR: libFuzzer: deadly signal
3
4 Failing input:
5     fuzz/artifacts/fuzz_target_1/crash-c44321f28eade1ea68ab1fcd6aa5152a34daa3f4
6
7 Output of `std::fmt::Debug`: "\n%)\u{820}\n\u{0}"
8
9 Reproduce with:
10     cargo fuzz run fuzz_target_1 fuzz/[...]/crash-c44321f28eade1ea68ab1fcd6aa5152a34daa3f4
11
12 Minimize test case with:
13     cargo fuzz tmin fuzz_target_1 fuzz/[...]/crash-c44321f28eade1ea68ab1fcd6aa5152a34daa3f4
```

Note how it also provides a complete input seed for reproducing the issue on this exact input in the future.

## cargo fuzz — diagnostic

The fuzzed string that makes `truncate_at` panics is composed by multiple characters, one of which (`\u{820}`) appears to be a Unicode code point that is represented in UTF-8 as multiple bytes.

At first sight our implementation of `truncate_at` still seems correct though:

```
1 pub fn truncate_at(s: &mut String, n: usize) {  
2     if s.len() > n && n > 0 {  
3         s.truncate(n - 1);  
4         s.push('.');  
5     }  
6 }
```

Q: what's the issue here?

We have two(!) distinct bugs:

1. From the [documentation](#) of `truncate(&mut self, new_len: usize)`: “Panics if `new_len` does not lie on a char boundary”.
2. From the [documentation](#) of `len()`: “Returns the length of this String, in bytes, not chars or graphemes”.

## cargo fuzz — correct solution

As we are dealing with UTF-8-encoded Unicode characters of variable length (in bytes), we need to iterate over them to identify where to truncate the string, like this:

```
1 pub fn truncate_at2(s: &mut String, n: usize) {
2     if n > 0 {
3         match s.char_indices().nth(n - 1) {
4             Some((index, _)) => {
5                 s.truncate(index);
6                 s.push('.');
7             }
8             None => (),
9         }
10    }
11 }
```

Bottom line: even if we didn't think about variable-length characters when implementing the first version of `truncate_at`, the fuzzer saved the day producing a minimal input that allowed us to understand and fix the problem. Keeping it around (`git add ; git commit`) as fuzzing seed will also provide a regression test for this issue in the future.

## cargo fuzz — code-driven exploration of the fuzzing space (1/2)

How smart is `cargo fuzz` in exploring the fuzzing space?

Let's fuzz the following function, which crashes rarely (w.r.t. the size of the function input space):

```
1 pub fn crash_rarely(a: u64, b: u64) {  
2     assert!(a + b != 0x76fa_abc4_7462_7364);  
3 }
```

with the following fuzz target:

```
1 #![no_main]  
2 use fuzzme::truncate_at2;  
3 use libfuzzer_sys::fuzz_target;  
4  
5 fuzz_target!(|data: [u64; 2]| {  
6     let (a, b) = (data[0], data[1]);  
7     crash_rarely(a, b);  
8 });
```

Almost instantly `cargo fuzz` finds the array `[18446744073709551383, 47244640255]`, which is a bug (overflow when adding two unsigned 64-bit integers)... but not what we were hoping to test!

## cargo fuzz — code-driven exploration of the fuzzing space (2/2)

Lather, rinse, repeat (with a non-overflowing version of `crash_rarely`):

```
1 pub fn crash_rarely(a: u64, b: u64) {  
2     assert!(a.saturating_add(b) != 0x76fa_abc4_7462_7364);  
3 }
```

If we fuzz again we will almost instantly obtain the array `[8573353700907709284, 0]`, which adds up to 8573353700907709284, i.e., 0x76faabc474627364, our magic value(!).

`cargo fuzz` does not simply randomly select values in the input space hoping to crash your program. It relies on `LibFuzzer` code instrumentation to smartly select values that make the code take execution branches not taken before, quickly converging to interesting input mutants. It is hence much more likely to select values that will lead to problematic behaviour than just spending a lot of time trying different values that are not “different enough” to be interesting for fuzzing purposes.

### Links

cargo fuzz: [homepage](#), [crate](#), [book](#)

# Takeaways

## Takeaways

- We can organize Rust code in-the-small using the **module system** to separate functionalities into reusable and well-encapsulated modules.
- We can organize Rust code in-the-large using binary and library **crates**, and publishing them as **packages** on <https://crates.io>.
- **cargo** is the go-to Rust utility for build automation, package management, and more; it is integrated by default with crates.io.
- We can automatically generate Rust **documentation** and keep it in sync with implementation code using **cargo doc**.
- We can **test** Rust code using unit tests, integration tests, and doc tests.
- We can automatically find bugs in Rust code via **fuzzing** (using **cargo fuzz**) and **property testing** (using **proptest**).



## Credits

- Some slides on the module system have been adapted from the [SE302b course](#) at Télécom Paris, by G. Duc and S. Tardieu.
- Some short examples have been adapted from the [Programming Rust](#) book: `mod spores`, `gcd`, `overlap`, `test_fiddlehead_unfurling`.
- Fuzzing image is by Ryan Eberhardt, Armin Namavari, Will Crichton, Julio Ballista, and Thea Rossman, from Stanford's course [CS 110L](#).