Good practices in Rust

This is not exhaustive

Why having good practices

- They help you
  - You avoid mistakes
- They help other getting up to speed on your project
  - It makes your code more readable
- They create a de facto standard
  - Some cargo commands emit a warning if your code is not commited
- They cost some time at the beginning, but they help a lot in the long run
  - The bigger the project, the bigger the reward, e.g. git is very useful if you’re working alone, but a must-have for a team

But they do not solve everything

- They cannot cancel the effects of a poor design
- They are not all mandatory (especially on small projects)
- Don’t forget your goal: solving a problem, publishing a paper, etc.
    - you are not paid for writing test, documenting or even writing code

OK, let’s go

Ease of life rules

Respect the standards of your team

They take precedence over what you prefer.

If you want to change them:

- You have a good reason? You can change them, but don’t it alone.
- You don’t? You are bike-shedding.

Respect the standards of the community

They (still) take precedence over what you prefer.

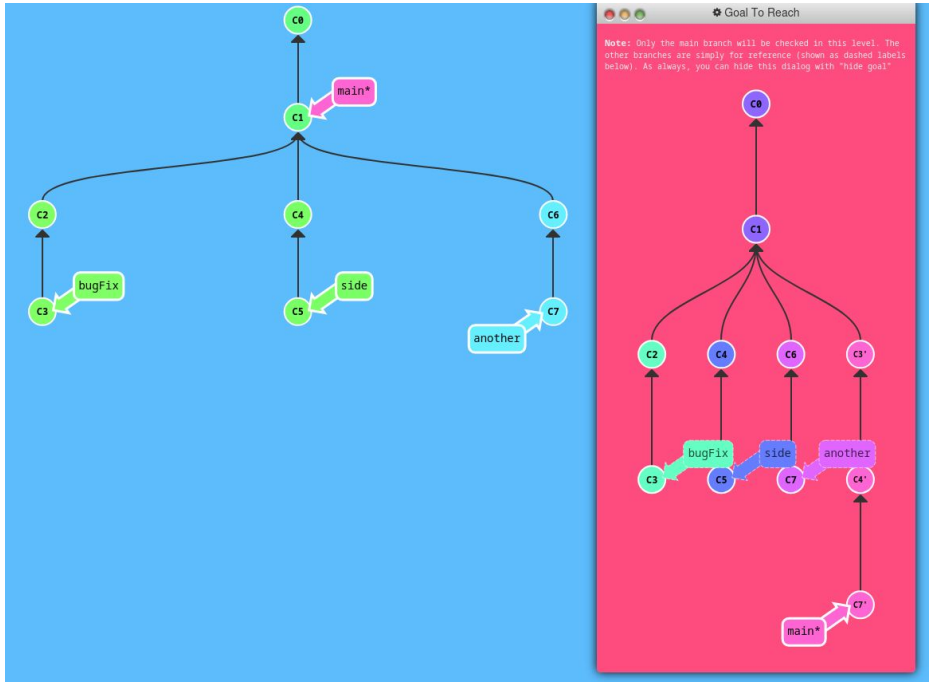
Even if you’re working alone. Mostly.

Use git

Git allows you to collaborate and to keep track of your changes

https://learngitbranching.js.org

Protect the main branch, use git hooks, etc.



Name your variables

Avoid name like a, drvr_clnt_ctrl, etc.

Exception: iteration variables.

Actually, avoid variables when possible.

Use functions

- They document your code for free
- They make you code faster
- They provide context for naming your variables
- **No out parameters**
- They have a cost, but unless proven otherwise, it is negligible.

Document your functions

Place

///

Before your functions.

Example:

```
/// Returns the arguments that this program was started with (normally passed
/// via the command line).
///
/// The first element is traditionally the path of the executable, but it can be
/// set to arbitrary text, and might not even exist. This means this property should
/// not be relied upon for security purposes.
///
/// Note that the returned iterator will not check if the arguments to the
/// process are valid Unicode. If you want to panic on invalid UTF-8,
/// use the [`args`] function instead.
```

```
/// # Examples
///
/// ```
/// use std::env;
///
/// // Prints each argument on a separate line
/// for argument in env::args_os() {
///         println!("{argument:?}");
/// }
/// ```
```

Organize your code into modules

A single file, e.g. module_name.rs.

Or a folder, module_name, containing submodules and a file mod.rs.

Document your modules

Same as functions:

//!

Example

```
//! Inspection and manipulation of the process's environment.
//!
//! This module contains functions to inspect various aspects such as
//! environment variables, process arguments, the current directory, and various
//! other important directories.
//!
//! There are several functions and structs in this module that have a
//! counterpart ending in `os`. Those ending in `os` will return an [`OsString`]
//! and those without will return a [`String`].
```

Correctness

Write unit tests

(See code)

Use assertions

- assert!(boolean) panics if boolean is false
- debug_assert!(boolean) does the same, but is removed in release mode

Avoid unsafe code

Unsafe code disables some of the compiler’s guarantees

If you’re not careful, it can lead to undefined behavior

There is a reason we didn’t see unsafe code yet

- There is usually a better alternative

Run your unsafe code with miri

https://github.com/rust-lang/miri

Miri is able to detect undefined behavior at runtime

More effort: run your unsafe code with kani

https://model-checking.github.io/kani/install-guide.html

kani tests all possible input for your code to detect undefined behavior

Setup a CI/CD pipeline

They run the tests in a standardize environment

They can protect a git branch

They can send emails when they are broken

Setup a CI/CD pipeline

```yaml
name: tests

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
```

```yaml
jobs:
  check:
    strategy:
      matrix:
        os: [ubuntu-latest]
        toolchain: [stable, nightly]
    runs-on: ${{ matrix.os }}
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Install Rust
        uses: dtolnay/rust-toolchain@stable
        with:
          toolchain: ${{ matrix.toolchain }}
      - name: Run check
```

```yaml
  tests_stable:
    strategy:
      matrix:
        os: [ubuntu-latest]
        toolchain: [stable]
    runs-on: ${{ matrix.os }}
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Install Rust
        uses: dtolnay/rust-toolchain@stable
        with:
          toolchain: ${{ matrix.toolchain }}
      - name: Run tests
        run: cargo test
```

```yaml
  tests_nightly:
    strategy:
      matrix:
        os: [ubuntu-latest]
        toolchain: [nightly]
    runs-on: ${{ matrix.os }}
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Install Rust
        uses: dtolnay/rust-toolchain@stable
        with:
          toolchain: ${{ matrix.toolchain }}
      - name: Run tests
        run: cargo test --all-features
```

Check out the code coverage

Check that critical functions are covered

It also prevent you from forgetting to write tests

```
# Generate test coverage:
# * report on terminal
# * lcov file
# * html
#
# USAGE
# -----
# ./coverage.sh
# or
# ./coverage.sh <name_of_the_module_to_test>
module_name=$1
LCOV_FILE=.lcov.info

# Clean files
rm $LCOV_FILE 2>/dev/null
cargo llvm-cov clean --workspace

# Terminal visual report
cargo llvm-cov nextest $module_name

# Generate lcov file in .lcov.info
cargo llvm-cov report --lcov --output-path $LCOV_FILE

# Generate html file in target/llvm-cov/html
genhtml $LCOV_FILE --output-directory=coverage/html
mv .lcov.info coverage/lcov.info
```

LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level			
Test:	.lcov.info	**Lines:**	5114	7481	68.4 %
Date:	2025-05-13 14:14:30	**Functions:**	496	665	74.6 %

Directory	Line Coverage		Functions	
src	54.4 %	834 / 1534	66.1 %	115 / 174
src/codec	99.7 %	686 / 688	94.7 %	36 / 38
src/codec/bmi	99.5 %	1136 / 1142	100.0 %	65 / 65
src/index	55.2 %	315 / 571	27.5 %	11 / 40
src/index/components	96.2 %	126 / 131	83.3 %	10 / 12
src/index/components/hyperkmer_parts	82.4 %	98 / 119	70.6 %	12 / 17
src/index/components/hyperkmer_parts/typical_hyperkmer_parts	97.5 %	306 / 314	77.8 %	28 / 36
src/index/components/hyperkmers_counts	36.4 %	368 / 1010	57.1 %	36 / 63
src/index/computation	0.0 %	0 / 586	0.0 %	0 / 19
src/index/computation/cache	90.3 %	112 / 124	88.2 %	15 / 17
src/serde	18.5 %	20 / 108	16.7 %	2 / 12
src/simd	96.0 %	984 / 1025	96.3 %	155 / 161
src/simd/intrinsics	100.0 %	129 / 129	100.0 %	11 / 11

```
27         49 : pub fn is_canonical(seq: &[u8]) -> bool {
28         49 :     let mut orientation_1 = same_orientation(seq);
29         49 :     let mut orientation_2 = reverse_complement(seq);
30         53 :     while let (Some(xc), Some(yc)) = (orientation_1.next(), orientation_2.next()) {
31         53 :         let xc = if unlikely(xc == b'N') { b'A' } else { xc };
32            :
33         53 :         match xc.cmp(&yc) {
34         49 :             Ordering::Less => return true,
35          0 :             Ordering::Greater => return false,
36          4 :             Ordering::Equal => {}
37            :         }
38            :     }
39            :     // in case of palindrome, prefer saying the sequence is canonical
40          0 :     true
41         49 : }
```

Performance

Release and debug mode

- Run tests in debug mode.
- Work in debug mode if possible.
- Test performance in release mode.

Do NOT optimize before benchmarking it

“We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3 %.”

Structured Programming with goto Statements

Donald E. Knuth

How to benchmark

Do not do it “manually”, e.g. with print(time) everywhere

- does not work
- time your time

Use a profiler, e.g. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler-download.html

(You may need to run “echo -1 | sudo tee /proc/sys/kernel/perf_event_paranoid” before usage) (do it at every reboot)

Use e.g. flamegraph to profile your program

Look at your CPU usage