A Tiny Rust Cookbook

Department of Computer and Information Science, NTNU

Revision 2.0

Michael Gimle Peder Bergebakken Sundt

This document is compiled to teach you all the necessary basics needed to get the assignments done using Rust. It is very lean, and attempts to avoid the vast majority of details unrelated to completing the assignment.

We're using Rust for a bunch of reasons, primarily:

- Better tooling in general
- Easier compilation across platforms *(although mac is not allowed for the coursework)*
- Many of the mistakes students traditionally have been making in this course are a lot harder to make in Rust

You will not have to learn a lot of Rust to be able to complete the course. The majority of the challenge will be learning to use OpenGL.

Should you be interested in learning more about the language, we *highly* recommend reading "*The Rust Programming Language*", a free book online that will easily help you understand Rust: https://doc.rust-lang.org/book/

Additionally, here are a few quick resources that could be useful.

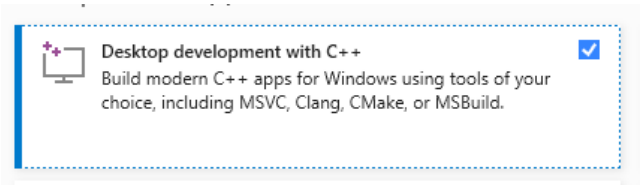
- **Rust by Example:** https://doc.rust-lang.org/stable/rust-by-example/index.html
- **Rust in easy English:** https://github.com/Dhghomon/easy_rust/blob/master/README.md#writing-rust-in-easy-english
- **Learn Rust in Y Minutes:** https://learnxinyminutes.com/docs/rust/
- **Rust cheat sheet:** https://cheats.rs/
  (Bear in mind that this sheet is designed for people that already know the language, or at least are familiar with its concepts.)

1 Getting Started

First, go to one of these links to get the Rust compiler up and running:

> https://www.rust-lang.org/tools/install
> https://doc.rust-lang.org/book/ch01-01-installation.html

On Windows, when installing the Build Tools for Visual Studio 2019, select:



An often preferred code editor for writing Rust is Visual Studio Code (https://code.visualstudio.com/). With it you should install and use the official Rust extension to get proper syntax highlighting and error squiggles (i.e. linting).

Also recommended is installing the "Code Runner" extension, and setting up the custom command to be "`cargo run`" for the project. To get to the settings in VSCode, use the hotkey CTRL+SHIFT+P, then type "settings" into the resulting text box. VSCode supports searching in the settings ui, making the rest pretty straight forward: type "custom command" to find the related setting.

Alternatives to VSCode include:

- Atom with the `ide-rust` plugin
- IntelliJ IDEA with `IntelliJ Rust`.
- Eclipse with `RustDT`.
- Vim with the official `rust.vim` plugin
- Emacs with `rust-mode`
- Kate with `rust-racer`.

Your mileage may vary.

If you have any problems getting up and running, feel free to show up to a lab session or contacting us by email.

2 Variables and Types

2.1 Declaring variables

```
let x = 6; // note how the type is inferred from the value
let y: i16 = 7; // here we specify a type
```

2.2 Printing values

```
println!("Hello, world!"); // Exclamation mark is necessary, as println is a macro
println!("Hello, {}", x); // Hello, 6
```

2.3 Mutability

Variables are *immutable* by default, meaning that if you want to modify them you first need to declare them as *mutable*:

```
let x = 6;      // immutable
let mut y = 6; // mutable
y = 3;
x = 3; // <- This will cause a compiler error
```

2.4 Integers

Integers (whole numbers) may include a *sign*, which determines whether they are negative or not. As such we have two different integer types: *signed* and *unsigned* integers.

Table 1: A list of signed and unsigned integer type names in Rust, with corresponding C++ type names. *(The C++ names are useful when referring to the OpenGL documentation.)*

Signed	Unsigned	C++ name	# of bits
`i8`	`u8`	(`unsigned`) `char`	8
`i16`	`u16`	(`unsigned`) `short`	16
`i32`	`u32`	(`unsigned`) `int`	32
`i64`	`u64`	(`unsigned`) `long long`	64
`i128`	`u128`	*none*	128
`isize`	`usize`	`size_t`	depends

`isize` and `usize` varies with the machine architeture, but is at least large enough to reference the whole memory address space. It is as such used to index into lists and arrays.

You *declare* the type of a variable by:

```
let a: i8 = 10; // signed, 8 bit
let b: u8 = 10; // unsigned, 8 bit
```

Alternatively you can let the Rust compiler *infer* the type of the variable from its value. Integer literals default to `i32`, but this can be changed with a suffix:

```
let a = 10; // signed, 32 bit
let b = 10i8; // signed, 8 bit
let c = 10u8; // unsigned, 8 bit
```

Type declarations are thus usually not needed, as the defaults usually are sufficient.

2.5 Floating Point

Non-whole numbers are usually represented with floating point representations: floats! In Rust, `f64` is the default, and is known in C++ as a `double`. `f32` is a 32-bits wide variant, and is known as a `float` in C++.

```
let x = 75.0; // this is inferred as a f64, the default
let y: f32 = 75.0; // this is a f32 declaration. The literal gets converted.
let z = 75.0f32; // This literal is a f32.
```

You will probably have to specify that you want 32-bit floating point values when using OpenGL.

2.6 Bool

In Rust we represent boolean values as `bool`. Its members are `true` and `false`.

```
let a: bool = 5 > 3; // true
let b: bool = 5 < 3; // false
let c: bool = a && b; // false
assert!(a || b); // true (a run-time check, will panic if false)
```

2.7 Tuples

Tuples are commonly used to store multiple different types of values in a fixed-size block. Useful for stuff like coordinates!

```
(i32, i32, f32) => (6, 7, 1.2)
let point_2d: (f32, f32) = (1.0, 3.4);
```

2.8 Arrays and vectors

To store a sequence of values of the same type, use either *arrays* or *vectors*. The size of *arrays* must be known at compile-time, and can therefore be stored on the program *stack*. *Vectors* may be dynamically sized and are as such stored in dynamic memory: the *heap*.

Arrays

Storing arrays of numbers on the stack can be done like this:

```
let numbers = [
    1, 2, 3, 4, 5, 6, 7
];
// or
let f32_numbers: [f32; 5] = [
    1.0, 2.0, 3.0, 4.0, 5.0
];
```

Vectors

You should be using vectors rather than arrays for these assignments. Storing arrays of numbers on the heap is done like this:

```
let mut numbers_vec = vec![1, 2, 3, 4, 5];
// or
let numbers_vec_2: Vec<f32> = Vec::new();
// or
let numbers_vec_3: Vec<f32> = vec![1.0, 2.0, 3.0];
```

Storing numbers on the heap allows you to add new elements:

```
numbers_vec.push(6); // A vector need to be `mut` if you want to push to it
```

2.9 Iterating over arrays and vectors

The easiest way to iterate over a vector is with a `for` loop:

```
for number in numbers_vec {
    println!("number is {}", number);
}
```

2.10 Borrowing and Ownership

Although we provide a brief overview of "ownership" here, we recommend you read the following to get a deeper understanding:

> https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

Ownership is one of the things that makes Rust unique, and is the feature that allows the language to not have to be *garbage collected* during runtime while still retaining memory safety guarantees.

If you want to "move" a variable somewhere without losing ownership, you can create a *borrow*. The key takeaway: a variable is removed from memory when it goes out of *scope*, but not when a borrow goes out of scope. "Moving" is only a problem for "complicated" objects and data – the kind of values you can't store on the stack. The specific reason why is that only the "simple" types implement the *trait* `Copy` by default, which means they get copied when they are attempted to be moved. You can read more about Traits in the Rust book.

```
{ // this is the beginning of a *scope*
    let b = 10; // declaration of b
} // b goes out of scope here
{
    let c = String::from("foo");
    let g = String::from("bar");
    let n = 13;
    {
        let borrowed_c = &c;
        let copied_n = n;
        let moved_g = g;
    } // borrowed_c goes out of scope, c is untouched
      // copied_n goes out of scope, n is still fine
      // moved_g goes out of scope, g no longer valid
    println!("c: {}", c); // no problem
    println!("n: {}", n); // Also fine
    println!("g: {}", g); // ERROR!
} // c,g,n goes out of scope
```

3 Functions

Calling a function will very often require you to borrow OpenGL was designed for C and C++. We use simple bindings to the C interface in Rust. the arguments with `&` instead of passing them in directly:

```
let rotate_amount: f32 = 1.0;
let y_axis = glm::vec3(0.0, 1.0, 0.0);
let rot = glm::rotation(rotate_amount, &y_axis);
```

Simple types such as numbers can be used directly, but vectors has to be borrowed. If you get errors while calling functions, inserting a `&` usually fixes it.

Some functions ask you for a *mutable reference* to some value. As such it is not enough to simply make the variable itself mutable in this situation, you actually need to explicitly say that a reference to the variable permits mutation. The way to do that is:

```
let mut variable = 0;
some_function(&mut variable);
```

3.1 Generic functions

Calling a generic function with a *type parameter* requires you to use the "turbofish" operator to specify the type:

```
let byte_size_of_f32 = mem::size_of::<f32>();
//                                   ^ type argument
```

3.2 Defining functions

A function signature in Rust looks like this:

```
fn function_name(argument_1: f32, argument_2: i8) -> u8 {
    // Function body                                 ^ return type
}
```

A function with no return value looks like this:

```
fn void_func() {
    // Function body
}
```

A function will return the final expression in its body by default, which lets your functions look a bit less cluttered

```
fn double_value(value: f32) -> f32 { value * 2 }
```

Semicolons turn expressions into void statements:

```
fn no_return_value(value: f32) { value * 2; }
```

The `return` statement still works as expected:

```
fn double_value(value: f32) -> f32 { return value * 2; }
```

4 Interfacing with OpenGL

OpenGL was designed for C and C++. We use simple bindings to the C interface in Rust.

4.1 Unsafe

Some things, such as dereferencing[1] a raw pointer, or calling a function from outside of Rust is always illegal in "safe" Rust. There is simply no way for the compiler to guarantee that nothing will go wrong (read; segfault) when doing it, so you have to tell the compiler that you know what you're doing in order to be allowed such power.

C and C++ on the other hand always operate in such an unsafe manner, and OpenGL interface was designed assuming so.

We signal to the compiler that we're about to do something dangerous by using the `unsafe` keyword:

```
unsafe {
    let p: *const i32 = std::ptr::null();
    let extremely_dangerous = *p; // Hard crash!
}
```

When you need to return a value from an unsafe block, there are a couple of different ways to go about doing so. The first is to declare the variable outside of the block without assigning it a value, and then assigning it inside of the unsafe block.

```
let value;
unsafe {
    value = unsafe_operation();
}
```

The second option is to return a value from the unsafe block, as Rust supports using blocks on the right hand side of an assignment:

```
let value = unsafe {
    unsafe_operation() // note the lack of a semicolon: an implicit return
}; // This semicolon terminates the `let` statement
```

We can also mark whole functions:

```
unsafe fn myfunction() {
    // something dangerous
}
```

Usage of the `unsafe` keyword should be minimized as much as possible in general, making it easier to locate mistakes.

[1] "*Dereferencing*" a pointer means accessing the place in memory a pointer points at.

4.2 The OpenGL C bindings

Calling OpenGL functions is our way of communicating with the GPU. Every single OpenGL function call is `unsafe`, as they are simply raw bindings to C functions. This means that *all* calls to OpenGL functions must be within an `unsafe` block or a function (`fn`) marked `unsafe`.

While OpenGL functions names in C++ will have `gl` as a *prefix*, in Rust they are instead *namespaced* under `gl`. In other words: `gl::SomeFunction`. This also applies to the "`GL_`" part of OpenGL enumerators. So if you find a function you want to use online, you'll have to *translate* it a little to be able to use it in Rust:

```
glEnableVertexAttribArray(0);
```

, in C or C++ will in Rust become:

```
unsafe { gl::EnableVertexAttribArray(0) };
```

4.3 Pointers

Pointers are commonly used in C and C++, and simply point to some address in memory. They are quite powerful, but commonly reguarded as unsafe. Pointers may include the *type* of the data it is pointing at. In C and C++, `*int` is read as "int pointer", and `*void` is read as "void pointer"[2]. In Rust they equate to `*mut i32` and `*mut c_void` respectably.

Sometimes when the Hitchhiker's guide to OpenGL suggests you use `0` as the input value, Rust will complain that it expects a pointer. While C++ is perfectly happy to implicitly cast (coerce) `0` into a pointer, Rust requires you to be a bit more explicit about it. You can make Rust happy by either using `ptr::null()` or `0 as *const c_void` in these situations.

One way OpenGL returns values is by modifying variables provided as pointers. This is because most calls to OpenGL already return an error code, using pointers is simply a work-around for the limitation of functions returning a single value.

```
let mut result: u32 = 0; // initialized to 0, doesn't matter really
gl::PerformSomeAction(&mut result as *mut u32); // we provide a mutable pointer to `result`
println!("Result is: {}", result); // the function modified `result` as a side-effect by
                                   // directly writing to where in memory it is located
```

[2] "Void" means missing. "Void functions" return nothing. "Void pointers" may point at *anything*.