A Case for Rust

The Safe, Fast, and Productive Programming Language of the Future

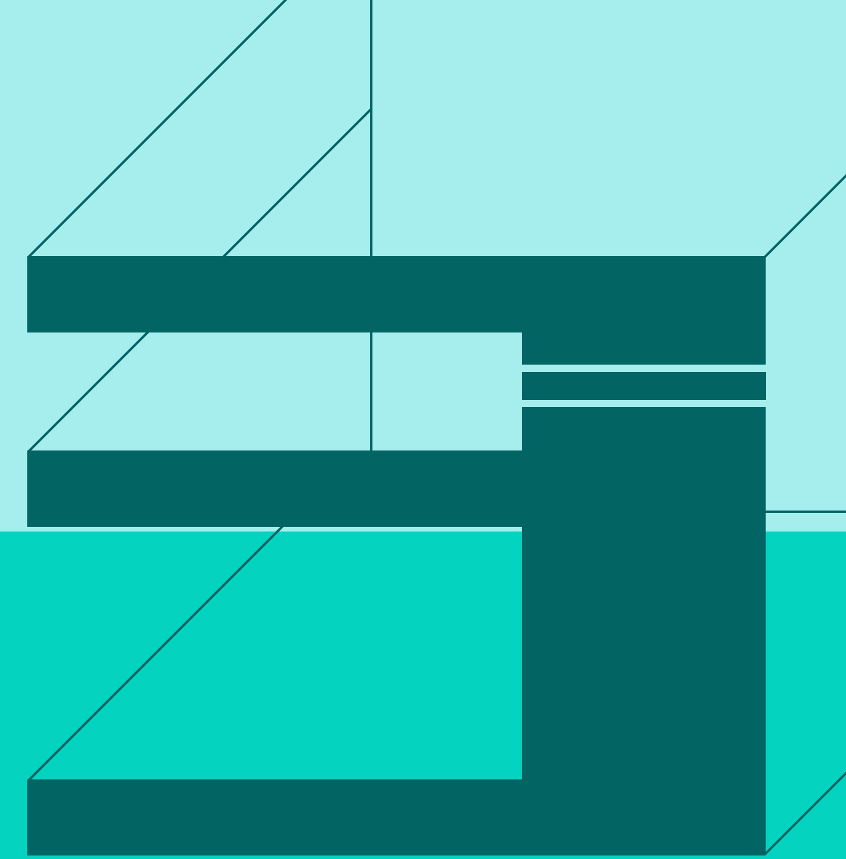
Little bit about me...

Matthew Esposito

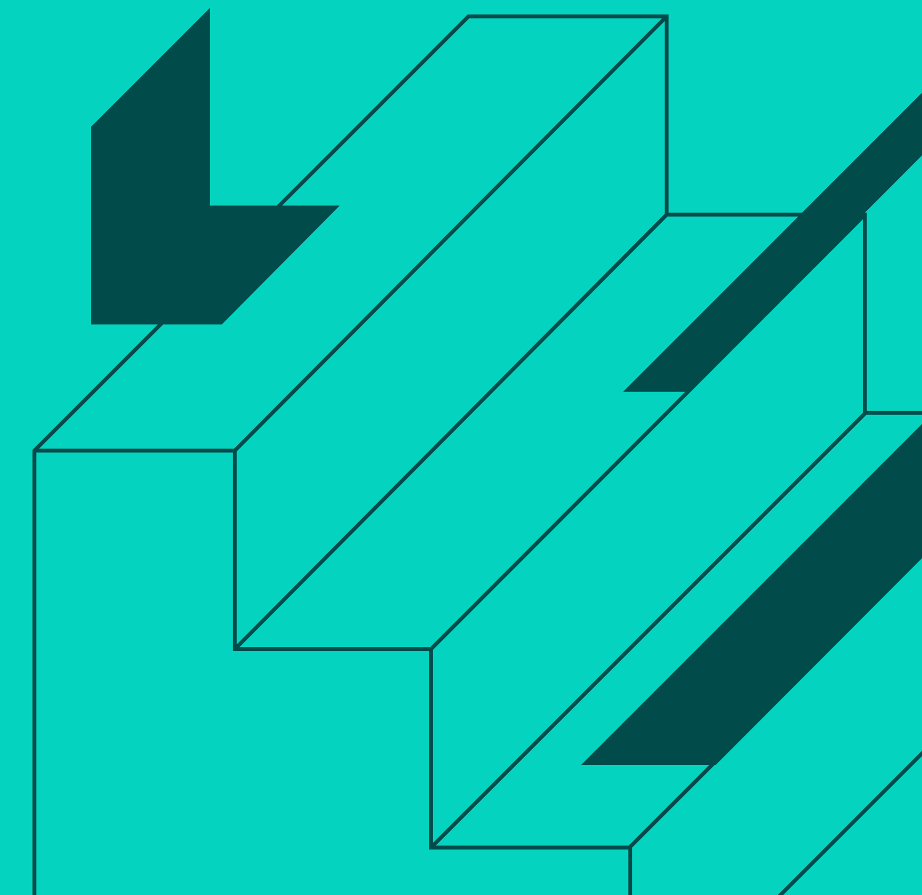
William & Mary '26

- Software Development Research Assistant @ geoLab
- Honors Thesis Research Assistant
- Rust Lang Contributor
    - Documentation, Compiler
- Open Source maintainer

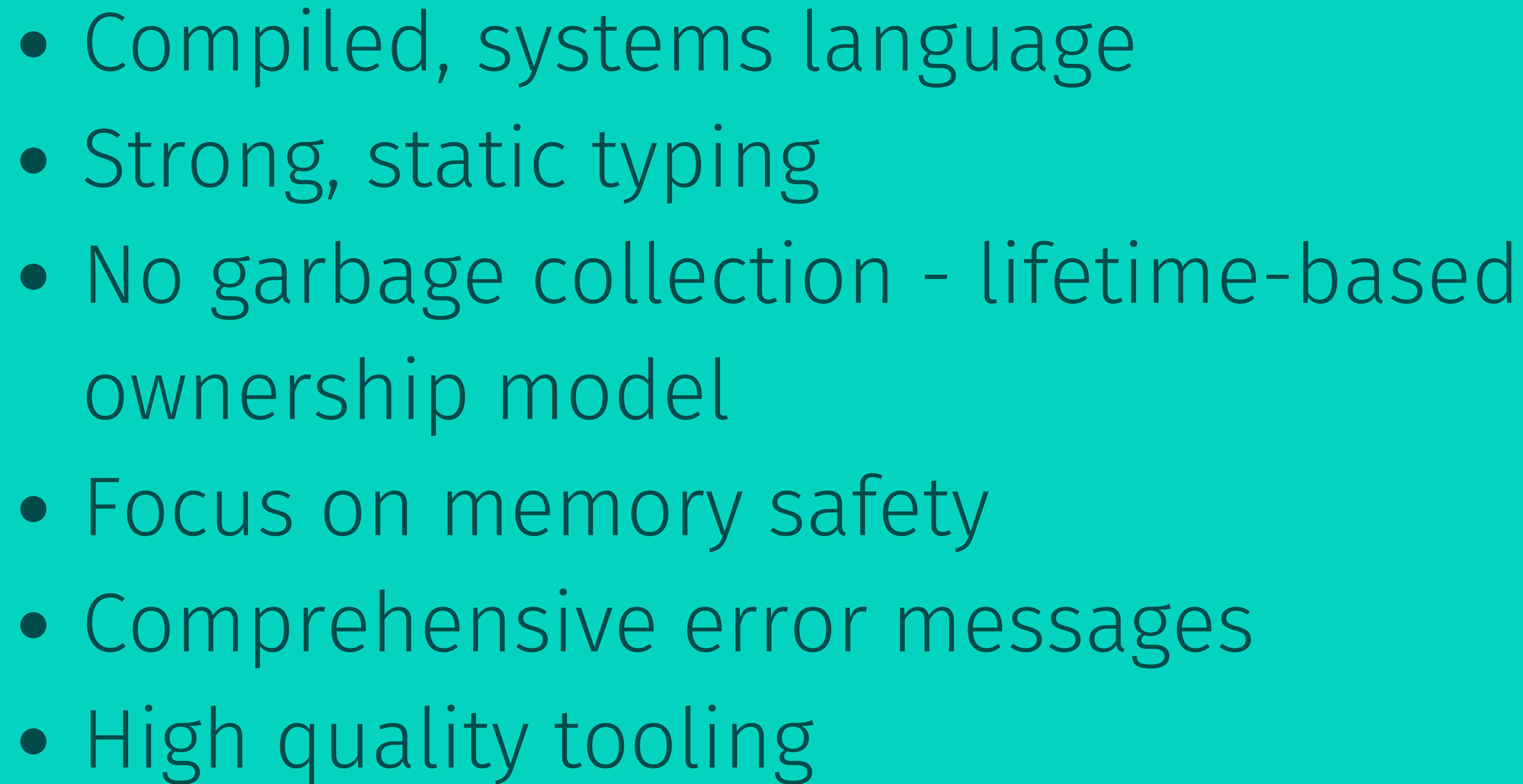
Agenda



- Technical concepts - comparisons, benefits
- Future of Rust
- Short introduction to language
- Workshop - Rustlings course



Technical Concepts


Compiled, systems language
Strong, static typing
No garbage collection - lifetime-based ownership model
Focus on memory safety
Comprehensive error messages
High quality tooling

Compiled language

- Higher speed compared to interpreted languages like Python, JavaScript, etc
- Optimizations performed at **compile-time**
- Far more power-efficient than other languages[1]
  - Python uses more than 36x more electricity
  - Go: 13x
  - JavaScript/TypeScript: 6x
  - Java: 2.3x
- This translates to efficiency gains, so the same workloads can be performed **faster** or on much **cheaper hardware**.
- Rust works on many, many devices, including embedded (Arduino, for example).

Static Types

- Strict, static types means you know exactly what data you're working with at all times, **without handling nulls** or **dynamic typing**.
- Allows better understanding of larger code-bases
- Concrete String types which only support UTF-8 encoding
  - Never handle errors related to text encoding
- Types are enforced at compile-time, never at run-time, so type errors are handled earlier in the development process
  - If you've ever tried to call a function on an object only to find out you have the wrong type, you've experienced the pain of dynamic typing.
- No Null type, only Option<T>.
  - A function that returns an integer (i64) cannot return null.
  - Only a function that returns an Option<i64> can return None.
- Dynamic typing does have its place in some forms of development, though.

Option/Result types

- These are special enum types in Rust.
- If your function can fail with a specific error in mind, you return Result<type, error_type>.
- If your function can fail without any specific error, you return Option<type>.

```
fn checked_division(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        None
    } else {
        Some(dividend / divisor)
    }
}
```

Lifetime paradigm

- Most popular high-level languages like Python, Go, JavaScript, etc use garbage collection
  - What is garbage collection?
- For performance-critical applications, garbage collection poses a serious problem
- Garbage collection passes can introduce significant lag spikes[2] and lead to unpredictable memory usage.
- Other systems programming languages handle this by manually managing memory: Allocations and deallocations are explicit
  - Examples: C, C++
- Rust does this differently: as soon as a resource is no longer used, it is automatically freed/destroyed. This is possible because of Rust's lifetime system.
  - Essentially an automatic version of the "manual memory" idea, but far **faster** and more **memory efficient.**
    - **Why?**

Memory Safety

- As a result of these rules, Rust consequently rules out entire classes of bugs, namely, memory safety.
    - 70% of all bugs involve memory safety[3].
    - Use after free, double free, memory leaks, buffer overflows, null pointers, null dereferencing, data races, out-of-bounds, etc
    - These all trigger **undefined behavior** in languages like C or C++. This frequently leads to vulnerabilities, often critical.
    - C/C++ communities have created many tools to attempt to detect these errors. These all check for undefined behavior *after* compilation.
- When writing Rust, these errors are completely impossible because they are **compile-time** errors[4].
- The cost for this safety, however, is adhering to the ownership rules.

Comprehensive Error Messages

- Rust's error messages are very beginner-friendly.
- They are designed to tell you what went wrong clearly, but more importantly, it will tell you exactly how to fix it whenever possible.

```
error[E0425]: cannot find value `num` in this scope
 --> src/main.rs:2:5
  |
2 |     num = 20;
  |     ^^^
  |
help: you might have meant to introduce a new binding
  |
2 |     let num = 20;
  |     +++
```

```
error[E0423]: expected function, found macro `println`
 --> src/main.rs:2:5
  |
2 |     println("Hello, world!");
  |     ^^^^^^^ not a function
  |
help: use `!` to invoke the macro
  |
2 |     println!("Hello, world!");
  |            +

For more information about this error, try `rustc --explain E0423`.
error: could not compile `x` (bin "x") due to previous error
```

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:3:2
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     x = 10;
  |     ^^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
warning: `x` (bin "x") generated 1 warning
error: could not compile `x` (bin "x") due to previous error; 1 warning emitted
```

```
error[E0615]: attempted to take value of method `len` on type `[{integer}; 2]`
 --> src/main.rs:3:19
  |
3 |     println!("{}", x.len);
  |                      ^^^ method, not a field
  |
help: use parentheses to call the method
  |
3 |     println!("{}", x.len());
  |                         ++
```

High quality tooling

- Rust's tooling makes dealing with the language incredibly easy.
- Cargo, the build system/package manager, has built-in dependency management.
- One command, `cargo run`, will automatically install all necessary dependencies for a project.
  - Starting to work on an existing software project is as simple as running a single command. No manual dependency installation
- Never deal with CMake/NPM/PyPi/Anaconda dependency errors again!
- Features of Cargo:
  - Dependency management
  - Linting/formatting
  - Testing/benchmarking
  - Cross-compilation
  - Documentation generator

Adoption

Amazon

Amazon utilizes Rust with many of AWS's services. AWS developed an open-source virtualization technology off of it, called Firecracker.

Google

Google is using Rust within the development of their next-generation operating system, Fuschia.

Dropbox

Dropbox was an early adopter of the Rust programming language, beginning development of core services in it as early as 2015.

Adoption

Discord

Discord rewrote much of their server code base in Rust, after requiring high performance as well as memory safety.

Facebook

Facebook wrote their source control manager, Sapling SPM, in Rust, in order to scale to "millions of files and commits"

Mozilla

Mozilla cultivated the Rust Programming Language from its creation within Mozilla Research!

Adoption

Linux

Probably one of the best testaments to Rust's success was its choice to be added to the Linux kernel. In the past, the Linux kernel was written in C. (Nearly everything you use on the internet is built off of the Linux kernel)

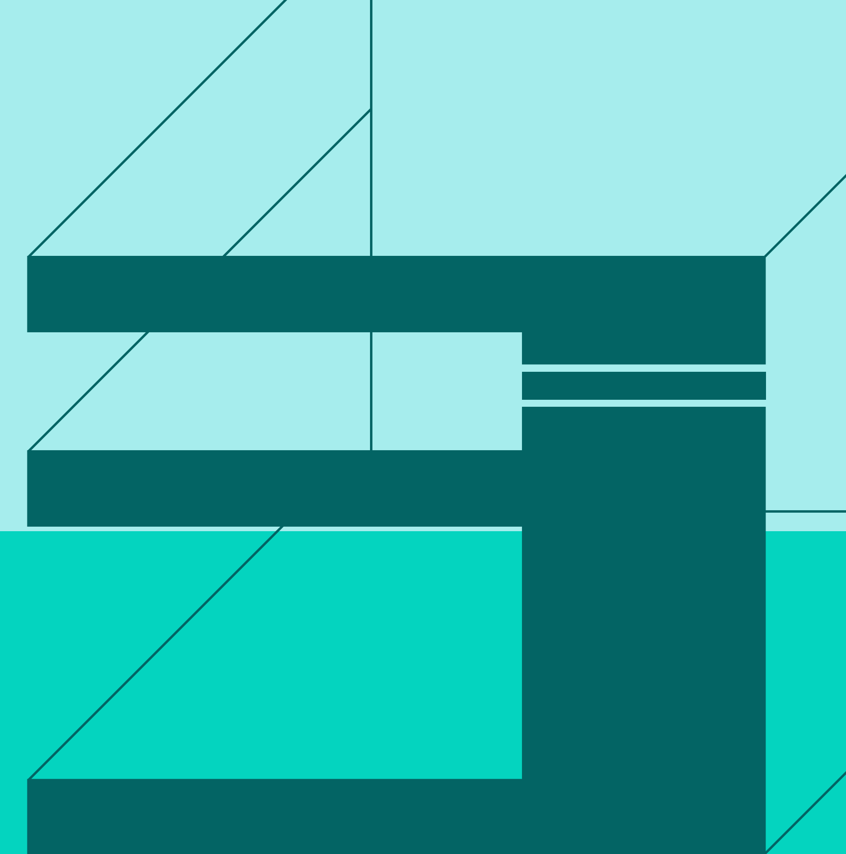
In October 2022, Rust was added to the kernel.

Rust is the only other language that has ever been trusted to run safely enough to replace some critical components of the Linux kernel.

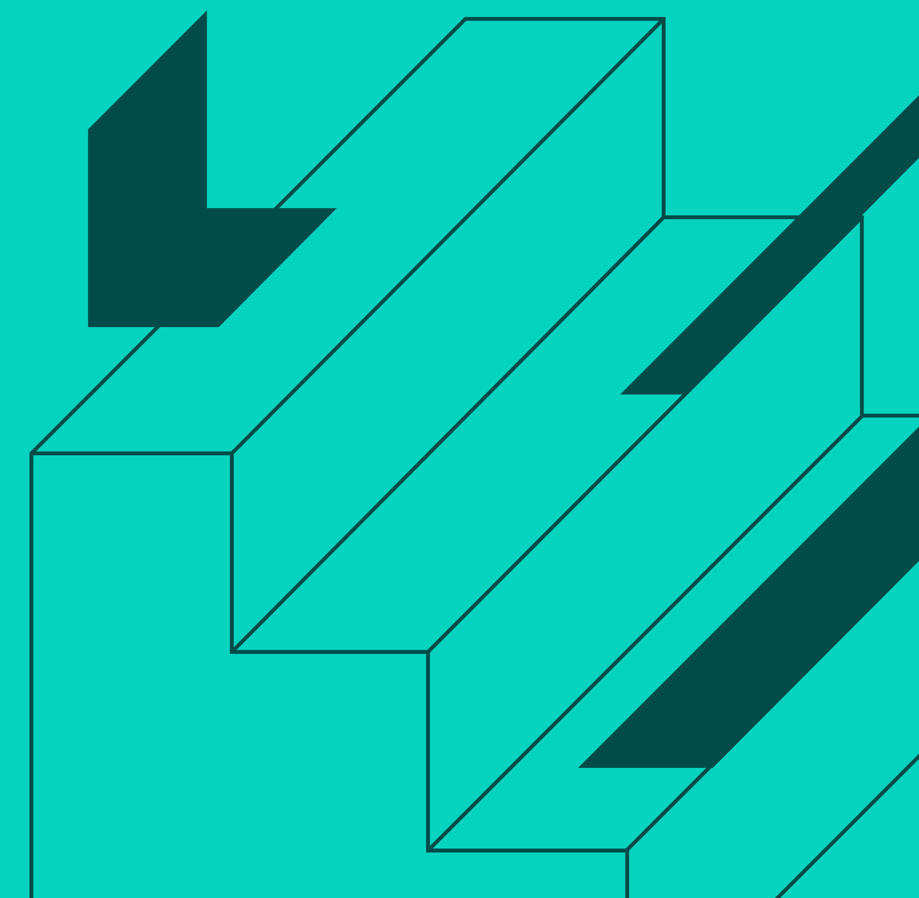
Footnotes

[1]: https://dl.acm.org/doi/abs/10.1145/3136014.3136031
[2]: https://discord.com/blog/why-discord-is-switching-from-go-to-rust
[3]: https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL
[4]: When writing *safe* Rust.

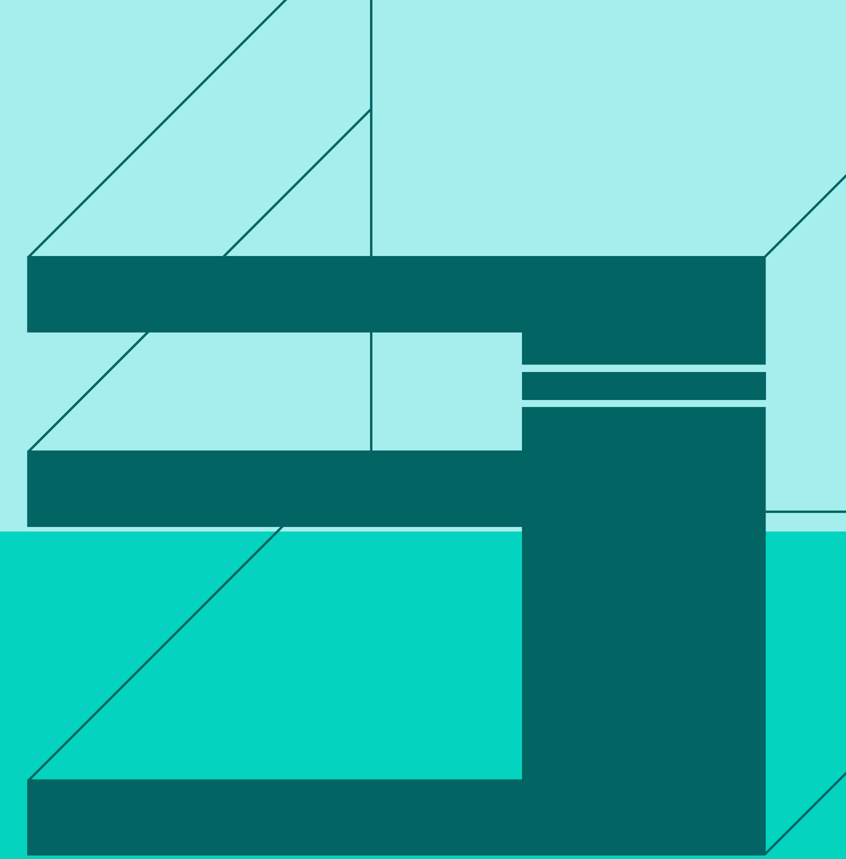
Future of Rust



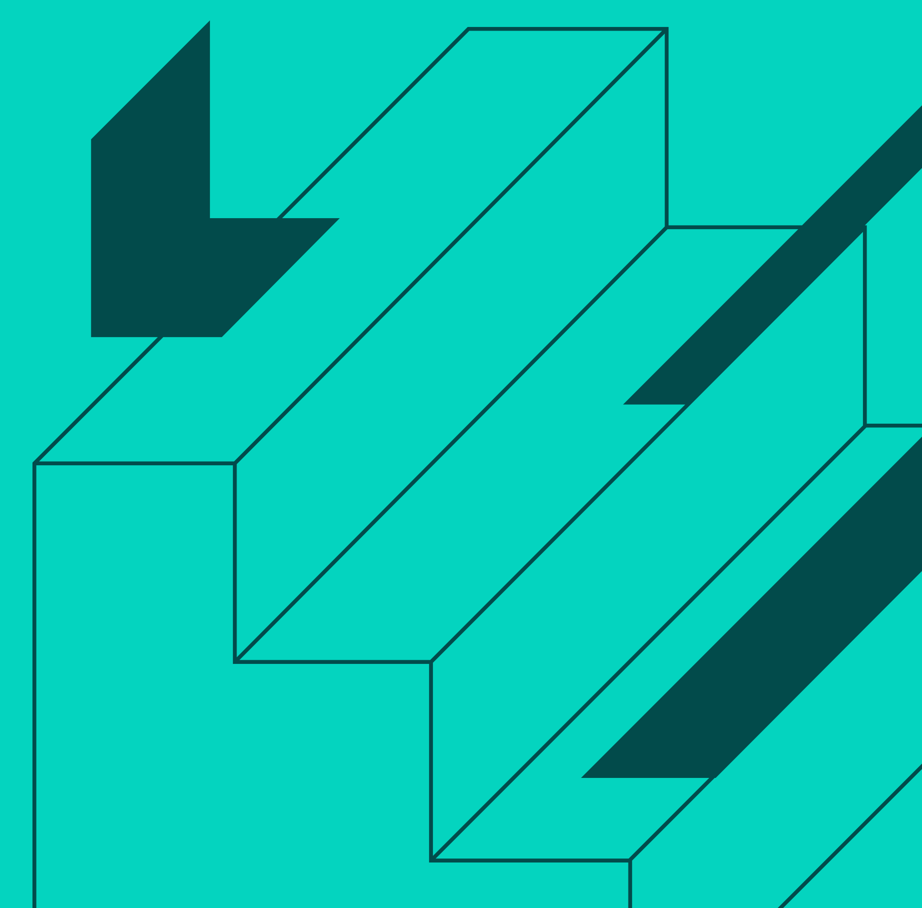
- Language specification
- **Optimizing compiler speed**
- Higher-kinded types
- Advanced const evaluation
- More platforms - gcc-rs
- **Ecosystem growth**



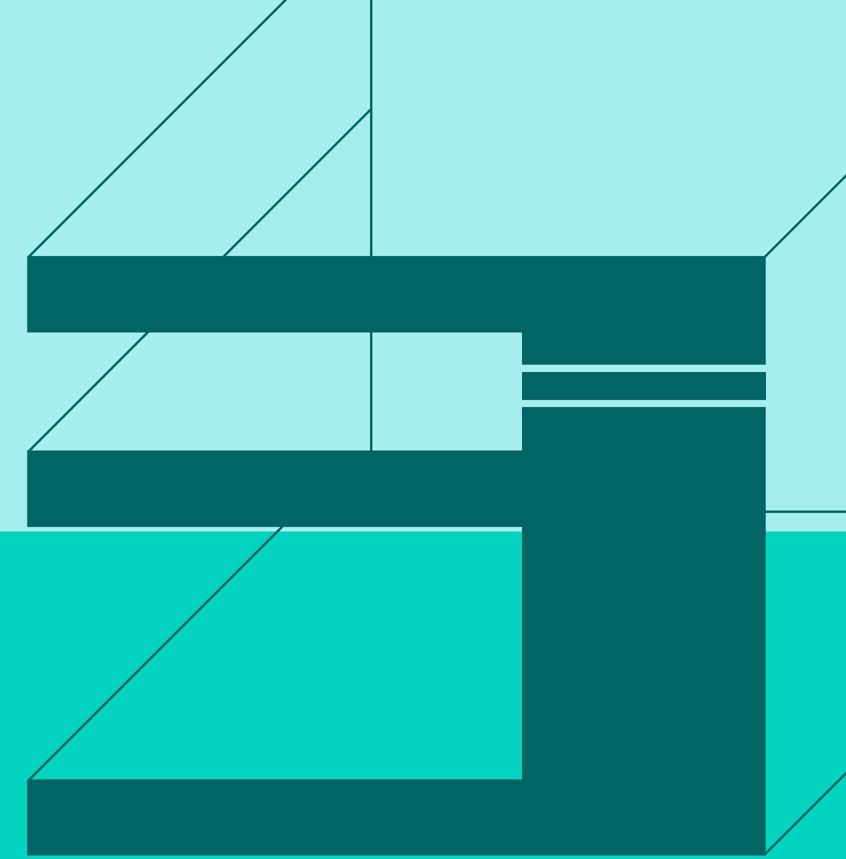
Workshop



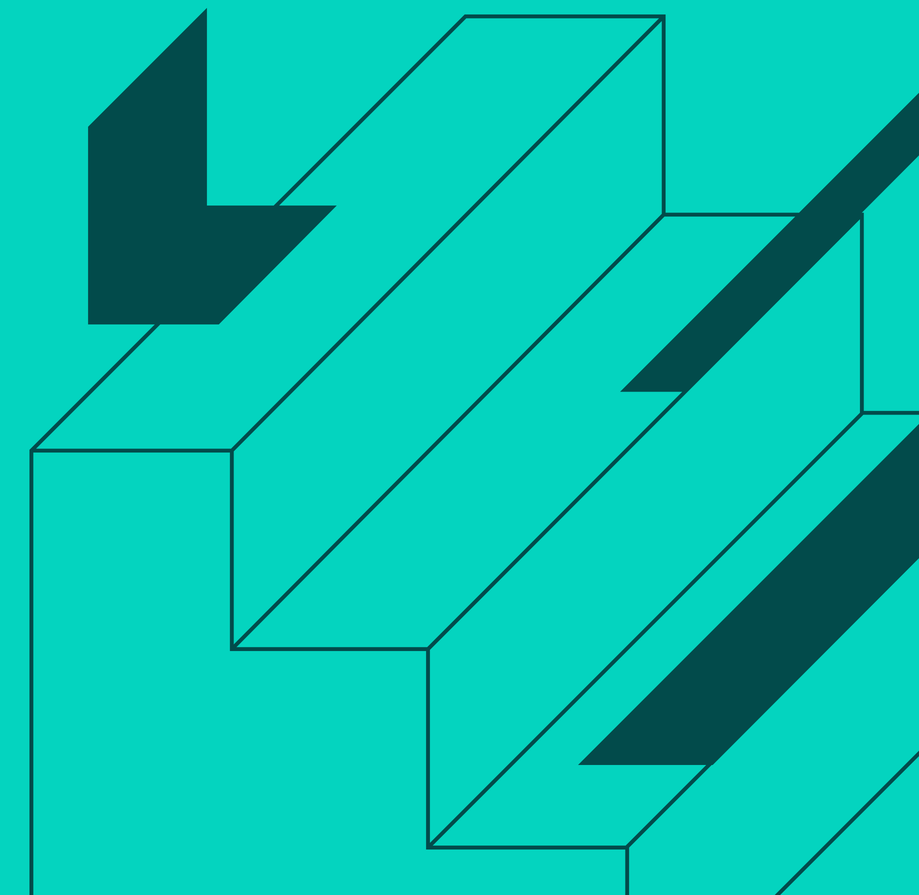
- This portion will involve your web browser.
- Please navigate to https://hurlurl.com/FZJA2
- Sign in with GitHub and let the container build - in the meantime, I'll go over some basics



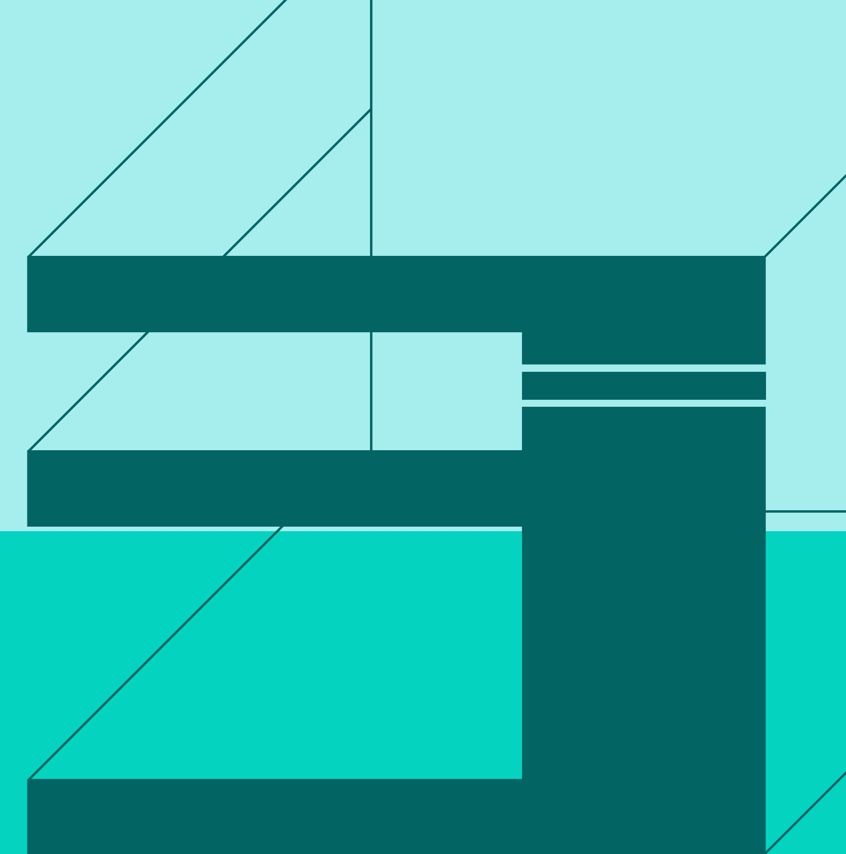
Primitive types



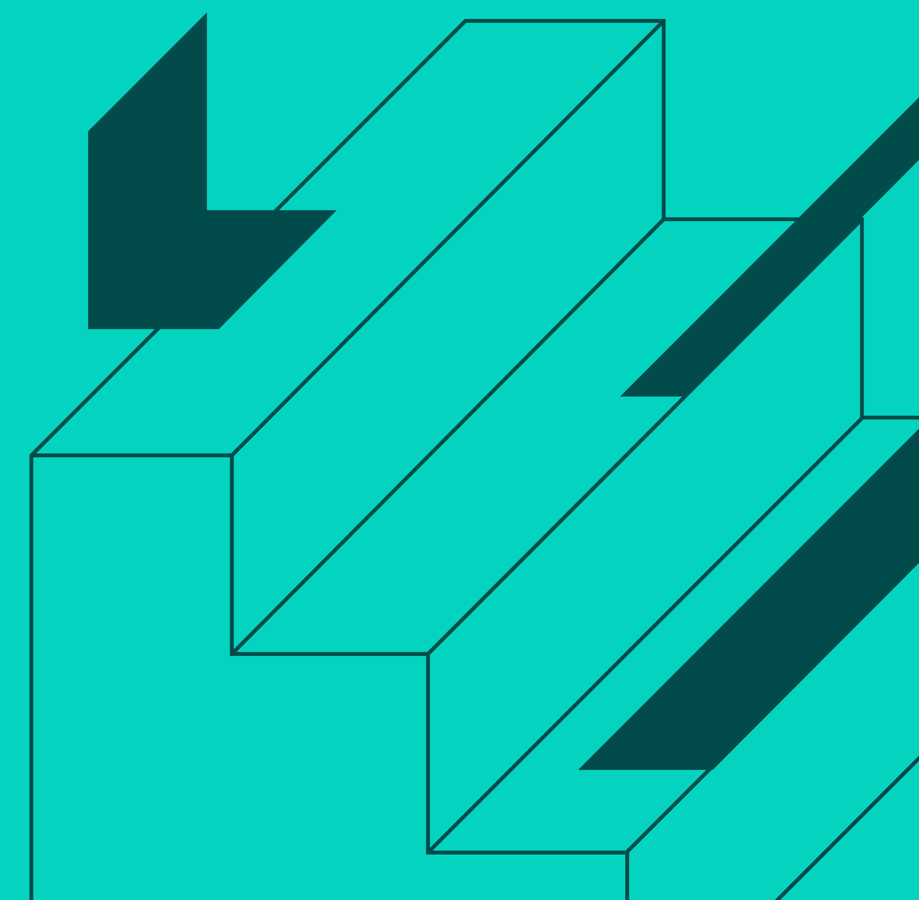
- Signed integers: i8, i16, i32, i64, i128
- Unsigned integers: u8, u16, u32, u64, u128
- Floating point: f32, f64
- char: Unicode scalar values like 'a', 'α' and '∞' (4 bytes each)
- bool: either true or false



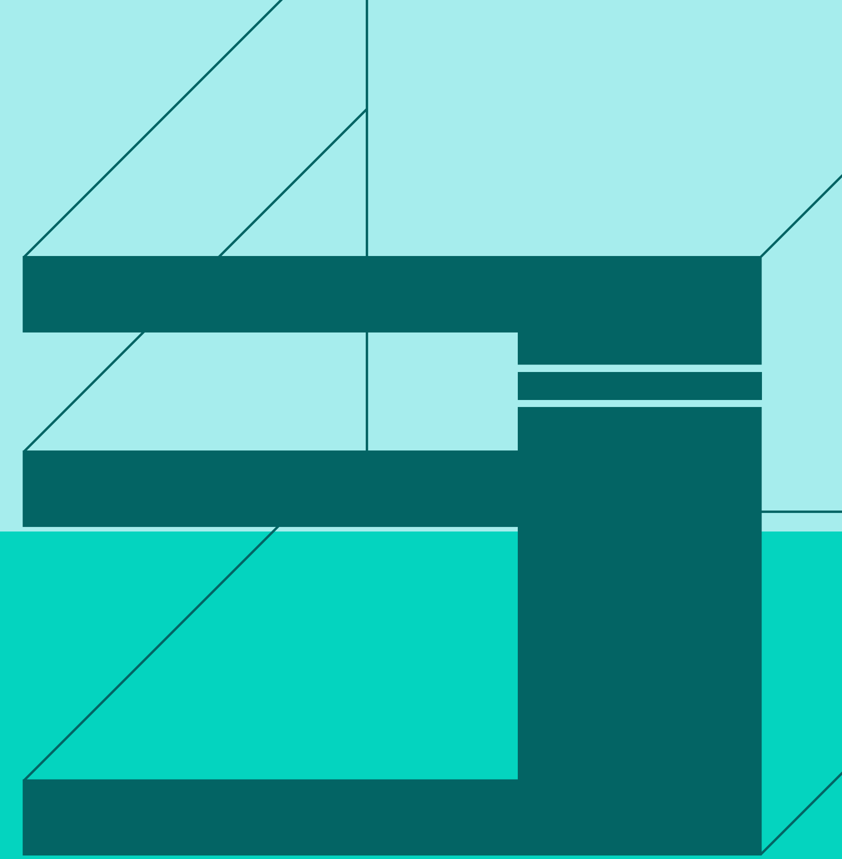
Variable declaration



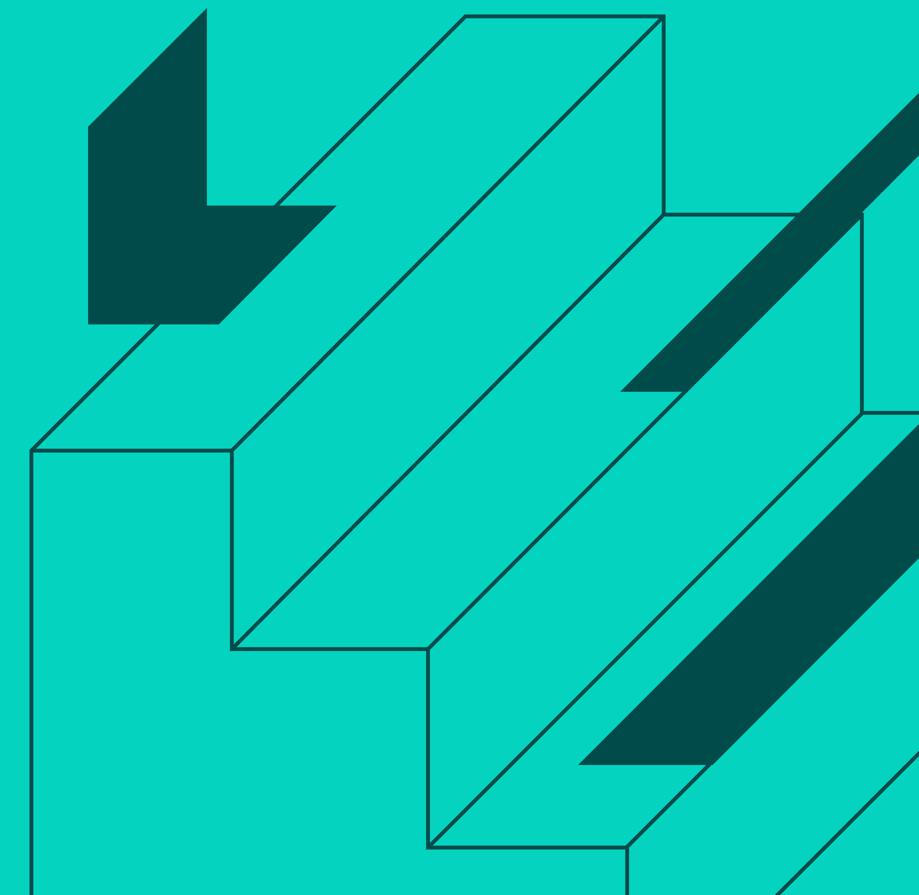
```rust
let x: i32 = 42;

// Can be written without including the type
// via type inference

let y = 42;

// Mutable variables must be declared as such.

let mut z = 42;
z += 1;
```



Tuple declaration

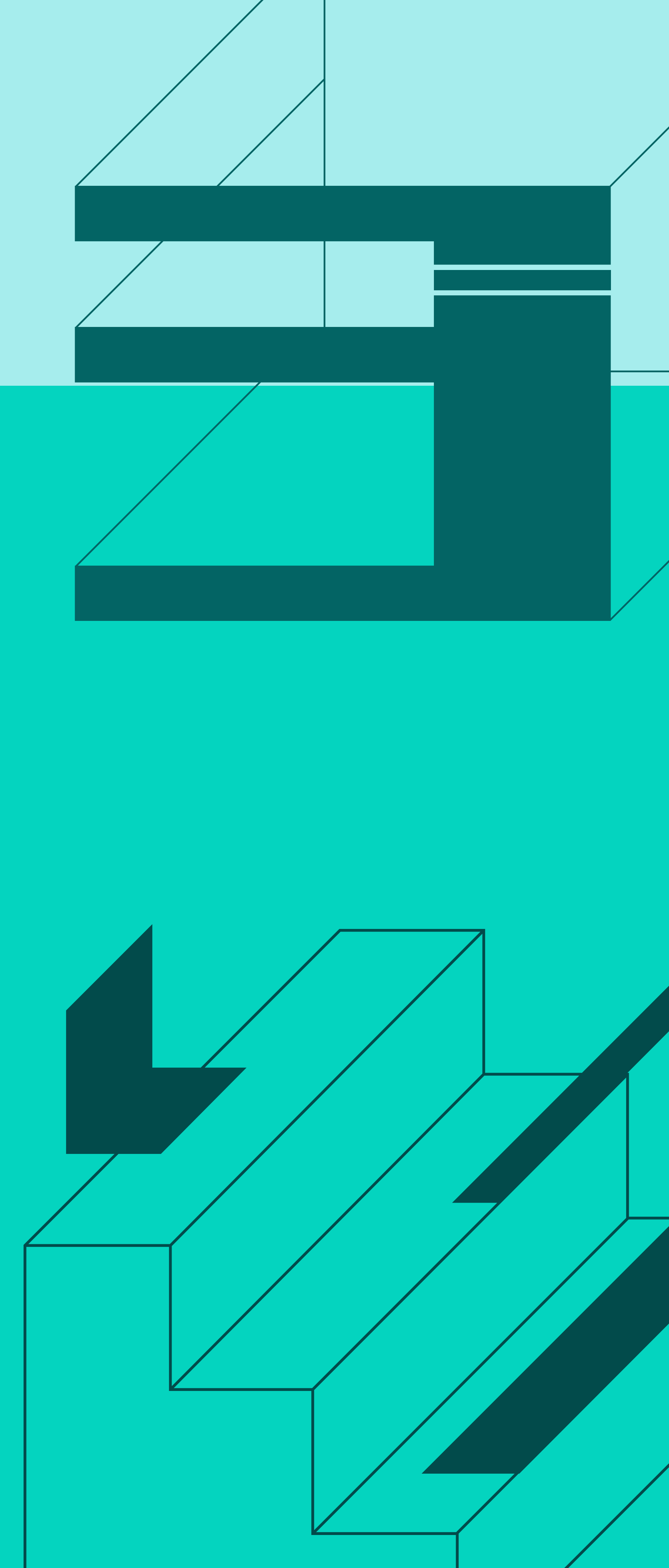


```
let x = (1, 2);

// Can be destructured via field access

let y = x.0;

println!("{y}"); // 1
```

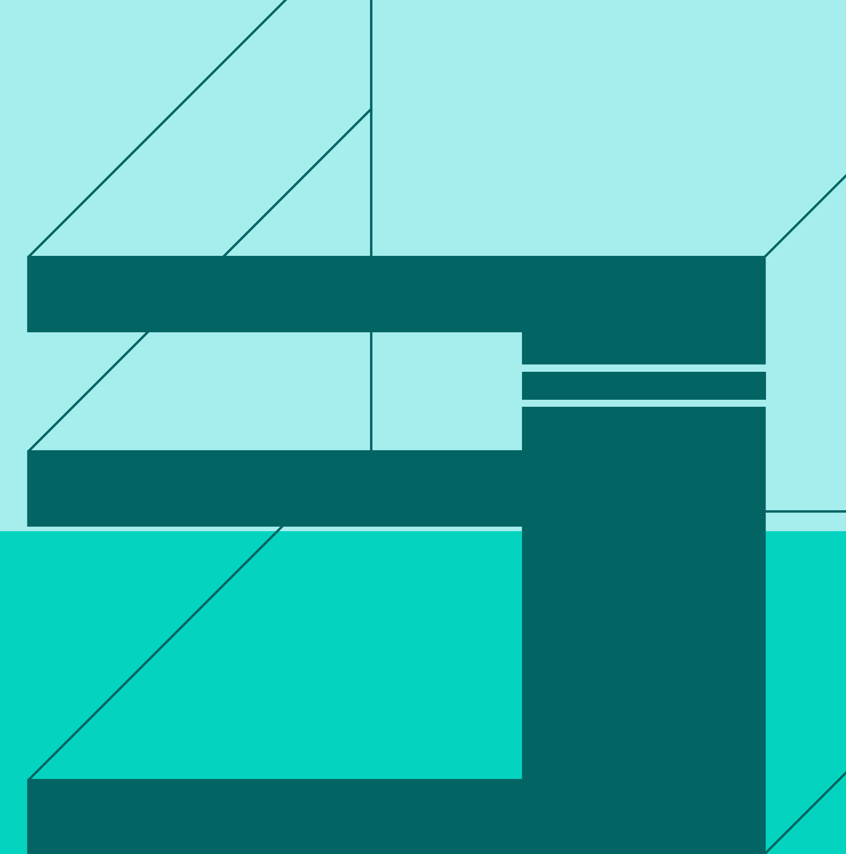


Function declaration

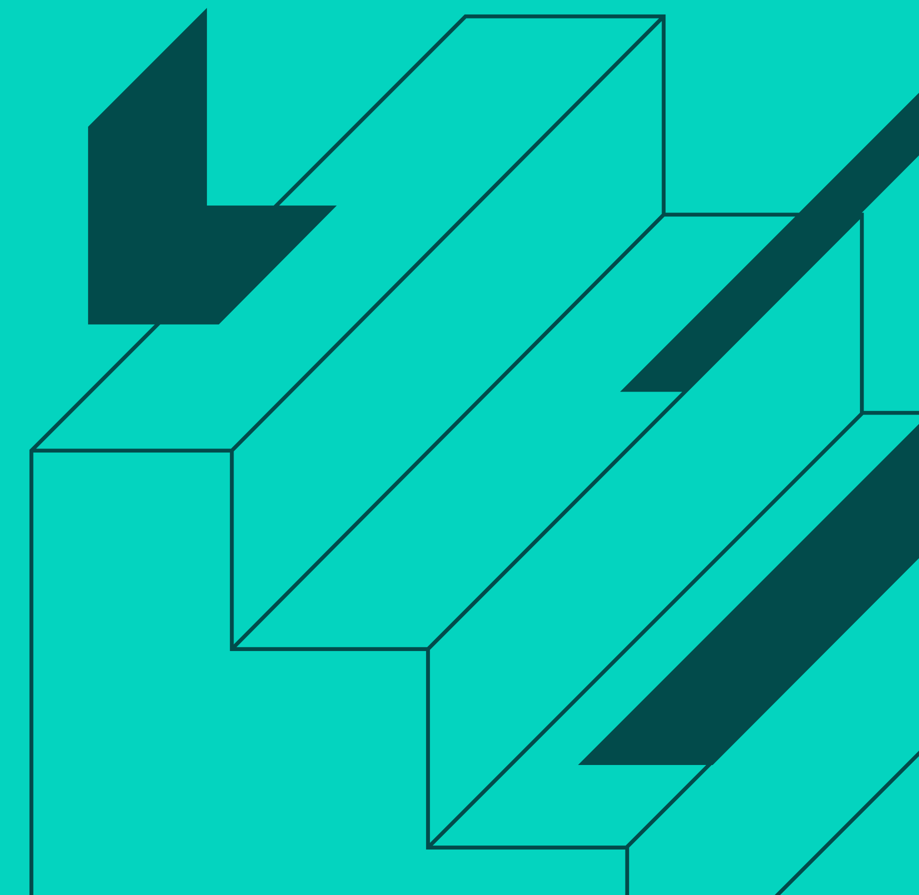
```
// Has no return type
fn hello_world() {
        println!("Hello Cypher VIII!");
}

// Returns an integer (arrow indicates return type)
fn add_one(x: i32) -> i32 {
        x + 1
}

// Example of expressions in Rust - This could be written as
        return x + 1;
// But, in Rust, the value "x + 1" evaluates as an
expression, and the last expression in a function (the
"tail") is automatically its return value.
```



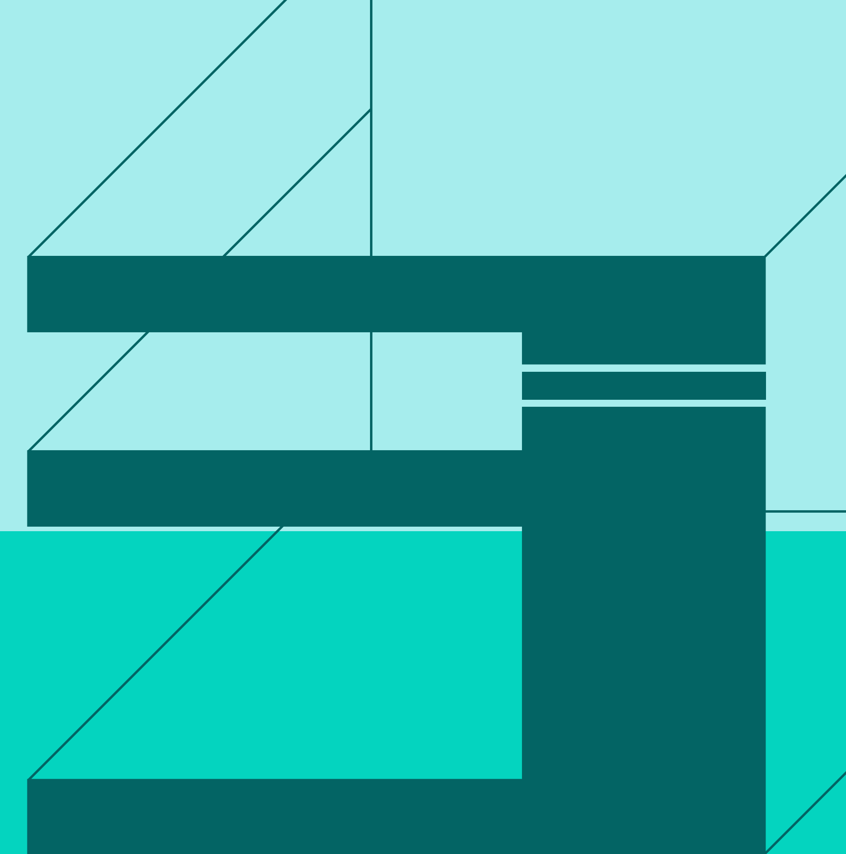
Blocks



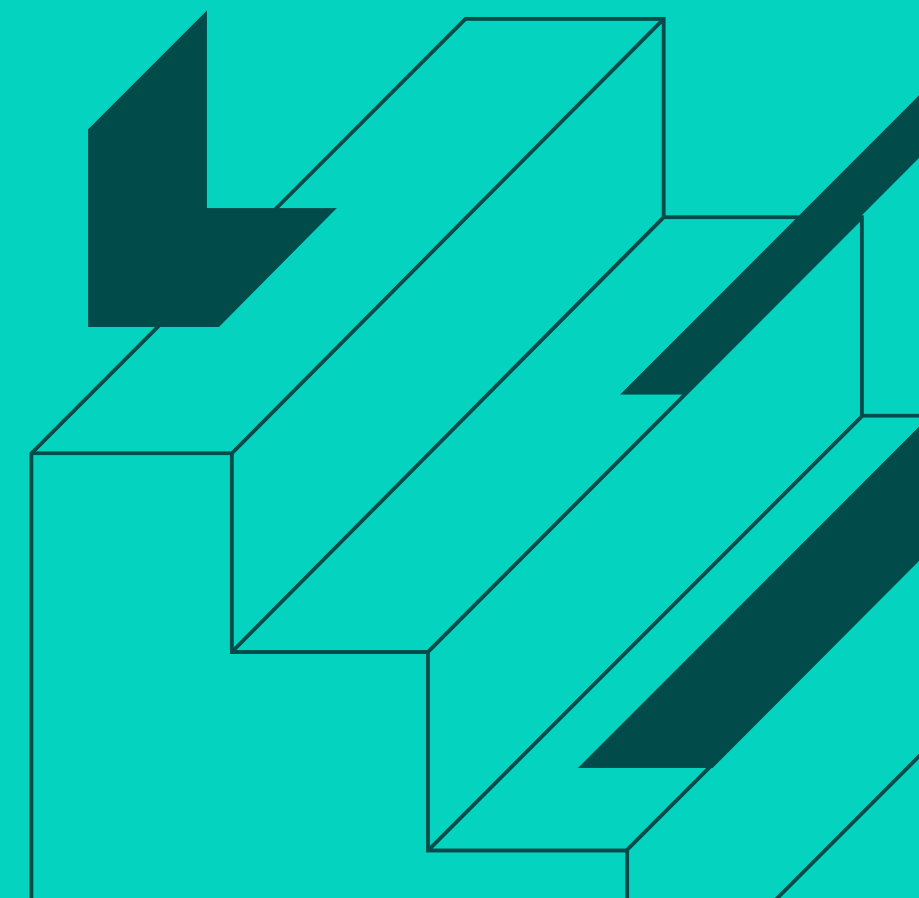
```
// this:
let x = 42;

// is equivalent to this:
let x = { 42 };

// so, we can do this:

let y = if x == 42 { "Hooray!" } else { "Aww :(" };

// y is now "Hooray!"
```



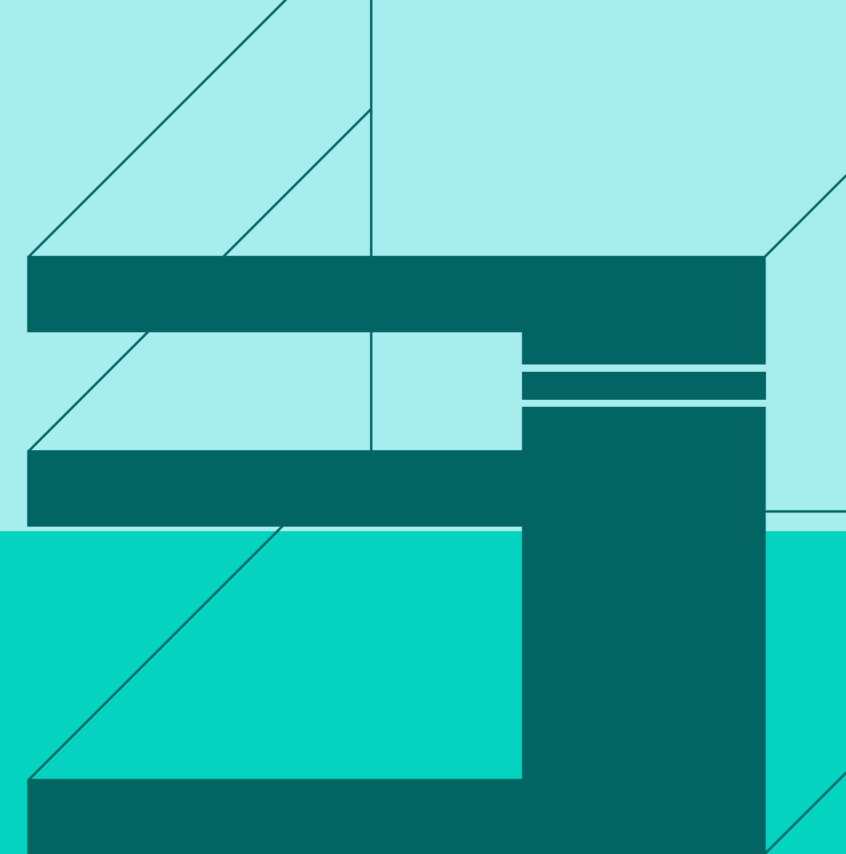
Function calls



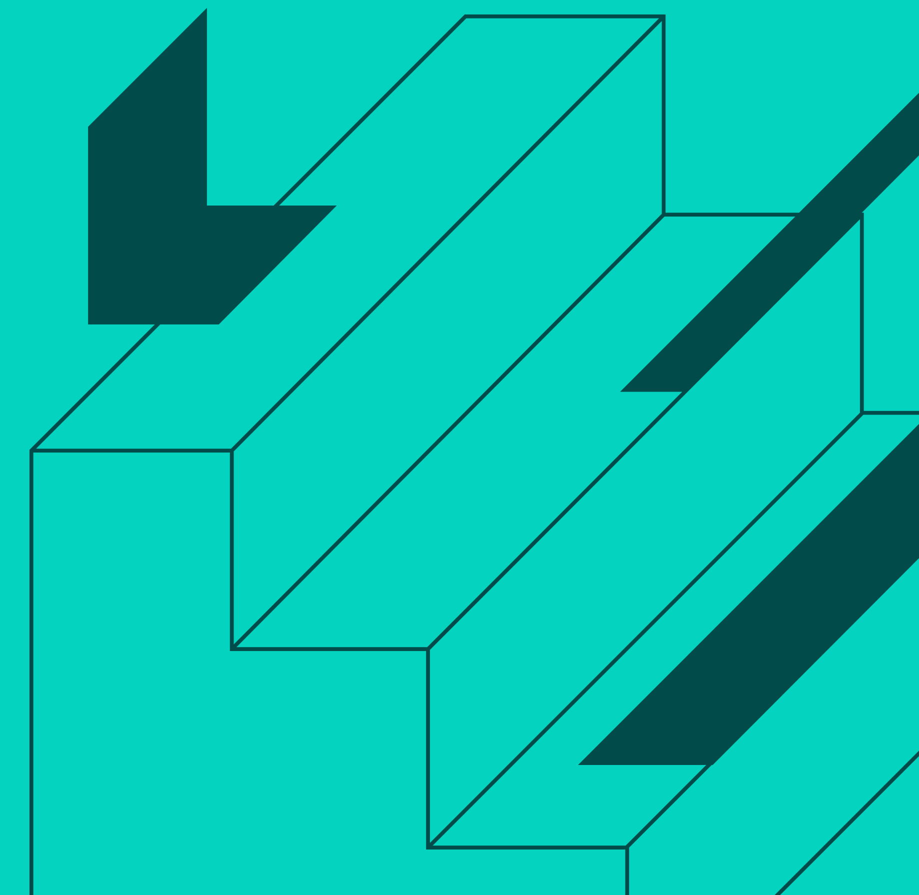
```
// If a function runs on a certain type, you can call it like so:

let string = String::from("W&M Cypher VIII");
let len = string.len();
// .len() is a function for an object of type String.
```



Structs



```
// Similar to classes, structs can be declared like so:

struct Cypher {
    participant_count: i32,
    year: i32
}

// and can be initialized like so:

let cypher8 = Cypher { participant_count: 200, year: 2023 }

// Fields can be accessed like so:

let year = cypher8.year;
```



Now, onto the workshop…