Consuming Rust bite by byte

Bite 2 – Undefined Behavior

Jim Fawcett

https://JimFawcett.github.io

Bite #1

Bite #3

Bite #2 – Undefined Behavior

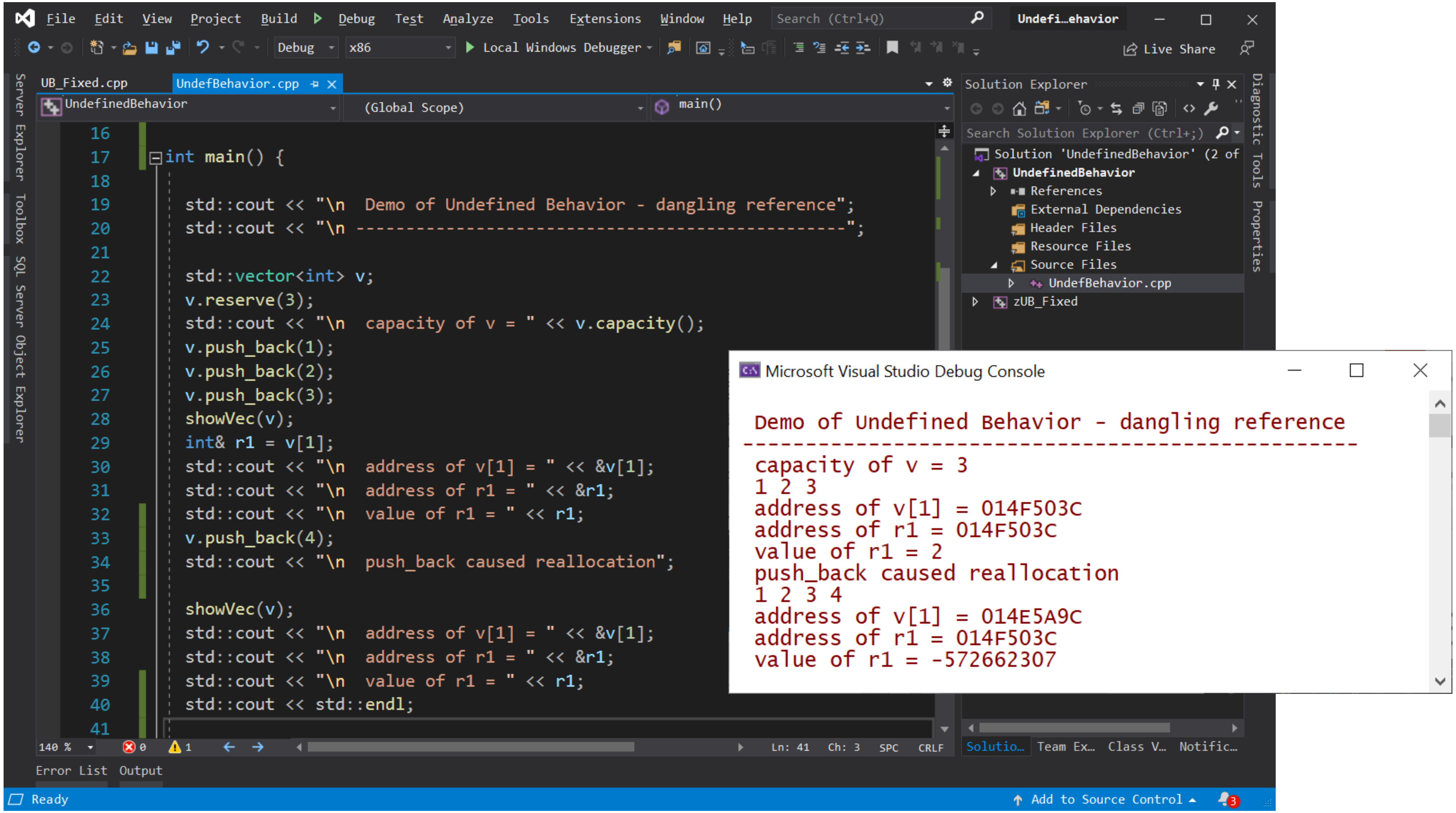
- Our goal is to observe undefined behavior in C++, and understand why that won’t happen in Rust code:

  - Invalid references
  - Indexing out of bounds
  - C++ safe by convention
  - Rust safe by construction

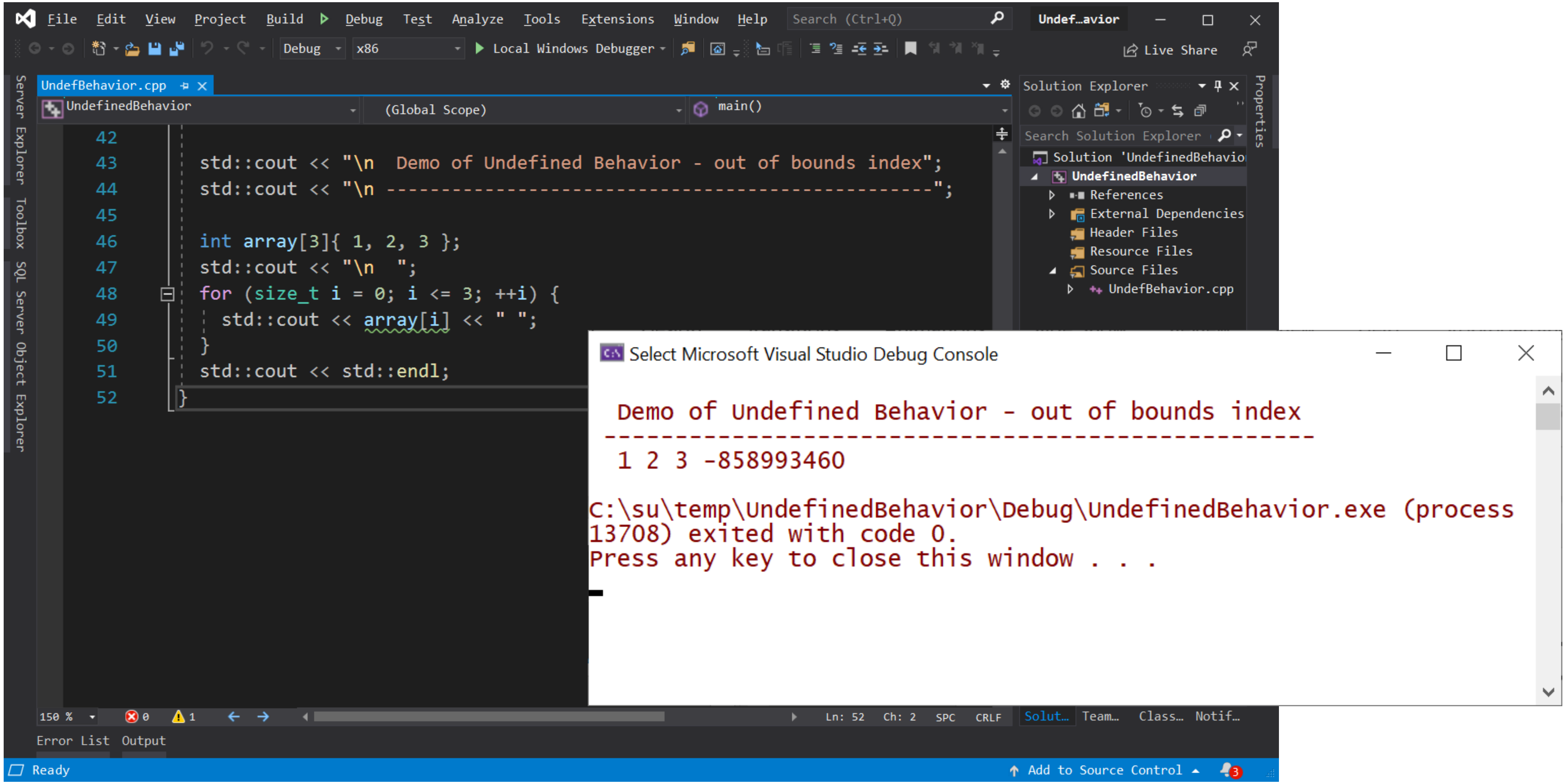
Type Safety

- A program is well defined if no execution can exhibit undefined behavior.
- A language is type safe if its type system ensures that every program is well defined.
- A non-type safe language may introduce undefined behavior with:
  - Reference invalidation
  - Integer overflow, e.g., wrap-around
  - Buffer overflow – out of bounds access
  - Use after free – access unowned memory
  - Double free – corrupt memory manager
  - Race conditions – mutation without exclusive ownership

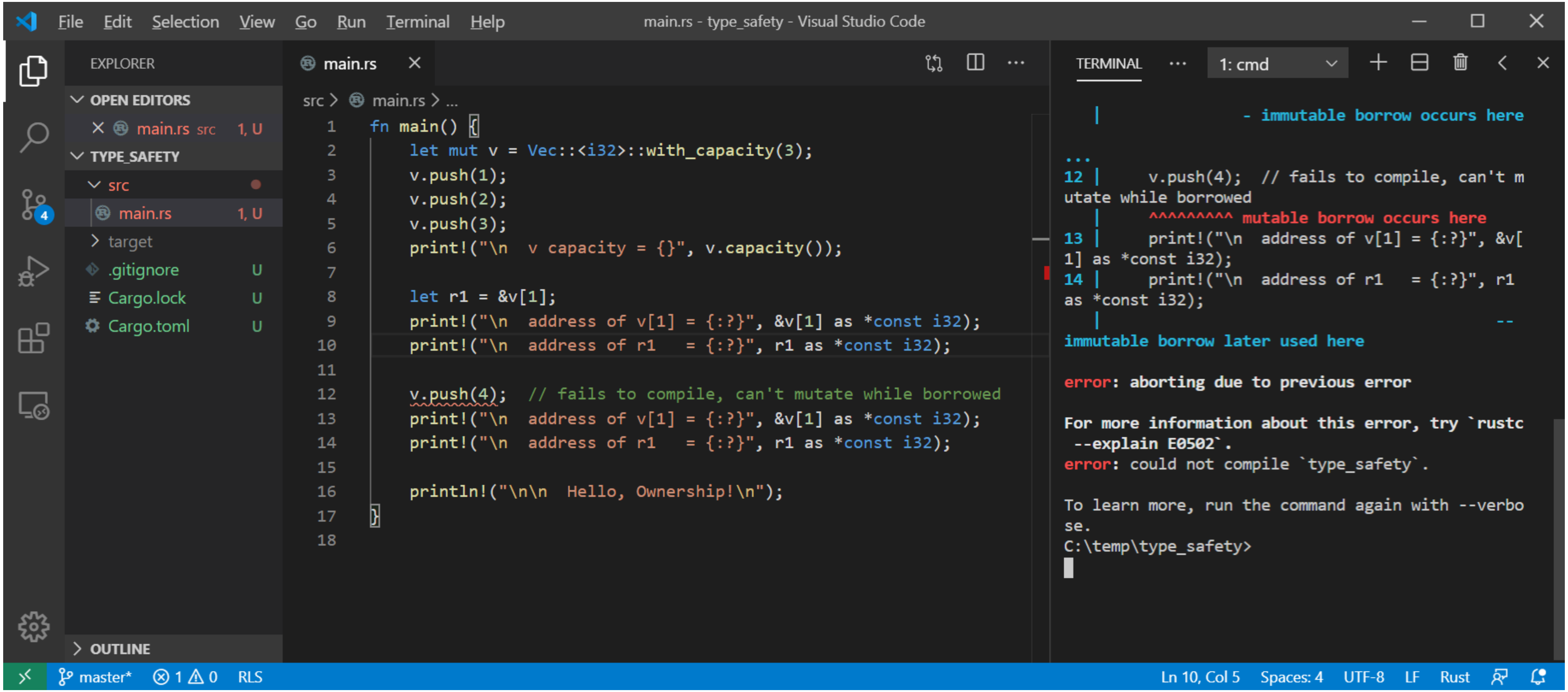
Undefined Behavior – C++ dangling reference



Undefined Behavior – C++ index out of bounds



Rust won’t allow mutation with an active reference



In defense of C++ - Dangling Reference

- If we had used an iterator:
  - `auto iter1 = ++v.begin();`
  - `v.push_back(4);`
  - `Std::cout << *iter1; // throws exception – no undefined behavior`
- It is standard practice to access containers with iterators, so well-crafted C++ will not exhibit undefined behavior.
- The difference:
  - With Rust you can’t get undefined behavior (UB) – most often programs fail to compile if they would have UB.
  - C++ code has to be well-crafted to avoid UB, errors are discovered at run-time, not compile-time.

In defense of C++ - Index out of Bounds

- If we had used a range-based for loop:
  - ```
    for(auto item : array) {
        std::cout << item << " ";
    }
    ```

  there is no chance of out-of-bounds indexing
- It is standard practice to traverse containers with range-based for loops, so well-crafted C++ will not exhibit undefined behavior.
- The difference:
  - With Rust you can't get undefined behavior (UB) – out of bounds index causes panic (exit) with no chance to access unowned memory.
  - C++ code has to be well-crafted, using standard idioms, to avoid UB.
    ```

Why Rust?

- Memory Safety
  - No dangling pointers or null references
  - No reading or writing to unowned memory
  - Rust’s type system enforces sane ownership policies.
- No Data Races
  - The same ownership policies applied to thread interactions ensures data race free operation
- Performance
  - As fast as C and C++
- Abstraction without Overhead
  - Traits and Trait objects
  - In the same ballpark as C++

Exercises

1. Create a Rust array of integers – attempt to index out of bounds.
    - What is the advantage of Rust panic over C++ allowed access?
2. Explain the difference between references in C++ and Rust?
    - Distinguish between references and pointers.
3. Consult Dr. Google to discover what you can and cannot do with pointers in safe Rust code.

References

Link	Description
ConsumingRustBite1 - Data	Bind, Copy, Move, and Clone
ConsumingRustBite3 - Ownership	Single owner, borrow
Rust Models	Expanded discussion in Rust Models presentation

That’s all until Bite #3

Bite #3 introduces Rust’s ownership model. That’s what makes Rust a safe language.