

Consuming Rust bite by byte

Bite 2 – Undefined Behavior

Jim Fawcett

<https://JimFawcett.github.io>

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

The screenshot displays the Visual Studio IDE with a C++ project named 'UndefinedBehavior'. The code in 'UndefBehavior.cpp' demonstrates a dangling reference by pushing back an element into a vector, which causes a reallocation and invalidates existing references. The output in the 'Microsoft Visual Studio Debug Console' shows the state of the program before and after the reallocation, highlighting the change in the memory address of the dangling reference.

```
16
17 int main() {
18
19     std::cout << "\n Demo of Undefined Behavior - dangling reference";
20     std::cout << "\n -----";
21
22     std::vector<int> v;
23     v.reserve(3);
24     std::cout << "\n capacity of v = " << v.capacity();
25     v.push_back(1);
26     v.push_back(2);
27     v.push_back(3);
28     showVec(v);
29     int& r1 = v[1];
30     std::cout << "\n address of v[1] = " << &v[1];
31     std::cout << "\n address of r1 = " << &r1;
32     std::cout << "\n value of r1 = " << r1;
33     v.push_back(4);
34     std::cout << "\n push_back caused reallocation";
35
36     showVec(v);
37     std::cout << "\n address of v[1] = " << &v[1];
38     std::cout << "\n address of r1 = " << &r1;
39     std::cout << "\n value of r1 = " << r1;
40     std::cout << std::endl;
41 }
```

Microsoft Visual Studio Debug Console

```
Demo of Undefined Behavior - dangling reference
-----
capacity of v = 3
1 2 3
address of v[1] = 014F503C
address of r1 = 014F503C
value of r1 = 2
push_back caused reallocation
1 2 3 4
address of v[1] = 014E5A9C
address of r1 = 014F503C
value of r1 = -572662307
```

Undefined Behavior – C++ index out of bounds

The screenshot shows the Visual Studio IDE with a C++ project named 'UndefinedBehavior'. The source file 'UndefBehavior.cpp' is open, showing a program that prints a title, a separator line, and then iterates over an array of 3 elements (1, 2, 3) using an index from 0 to 3. The index 3 is out of bounds, causing undefined behavior. The program then prints the memory address at index 3, which is -858993460. The output window shows the execution results, including the title, separator line, and the out-of-bounds value. The status bar at the bottom indicates 'Ready' and 'Add to Source Control'.

```
42  
43     std::cout << "\n Demo of Undefined Behavior - out of bounds index";  
44     std::cout << "\n -----";  
45  
46     int array[3]{ 1, 2, 3 };  
47     std::cout << "\n ";  
48     for (size_t i = 0; i <= 3; ++i) {  
49         std::cout << array[i] << " ";  
50     }  
51     std::cout << std::endl;  
52 }
```

Select Microsoft Visual Studio Debug Console

Demo of Undefined Behavior - out of bounds index

1 2 3 -858993460

C:\su\temp\UndefinedBehavior\Debug\UndefinedBehavior.exe (process 13708) exited with code 0.
Press any key to close this window . . .

Rust won't allow mutation with an active reference

```
File Edit Selection View Go Run Terminal Help
main.rs - type_safety - Visual Studio Code

EXPLORER
OPEN EDITORS
  main.rs src 1, U
TYPE SAFETY
  src
    main.rs 1, U
  target
  .gitignore U
  Cargo.lock U
  Cargo.toml U
  OUTLINE

main.rs
src > main.rs > ...
1 fn main() {
2     let mut v = Vec::<i32>::with_capacity(3);
3     v.push(1);
4     v.push(2);
5     v.push(3);
6     print!("\n v capacity = {}", v.capacity());
7
8     let r1 = &v[1];
9     print!("\n address of v[1] = {:?}", &v[1] as *const i32);
10    print!("\n address of r1 = {:?}", r1 as *const i32);
11
12    v.push(4); // fails to compile, can't mutate while borrowed
13    print!("\n address of v[1] = {:?}", &v[1] as *const i32);
14    print!("\n address of r1 = {:?}", r1 as *const i32);
15
16    println!("\n\n Hello, Ownership!\n");
17 }
18

TERMINAL
1: cmd
- immutable borrow occurs here
...
12 |     v.push(4); // fails to compile, can't m
    |               utate while borrowed
13 |         ^^^^^^^^^ mutable borrow occurs here
14 |     print!("\n address of v[1] = {:?}", &v[
    |               1] as *const i32);
15 |     print!("\n address of r1 = {:?}", r1
    |               as *const i32);
16 |     --
    |     immutable borrow later used here
error: aborting due to previous error
For more information about this error, try `rustc
--explain E0502`.
error: could not compile `type_safety`.
To learn more, run the command again with --verbo
se.
C:\temp\type_safety>
```

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.