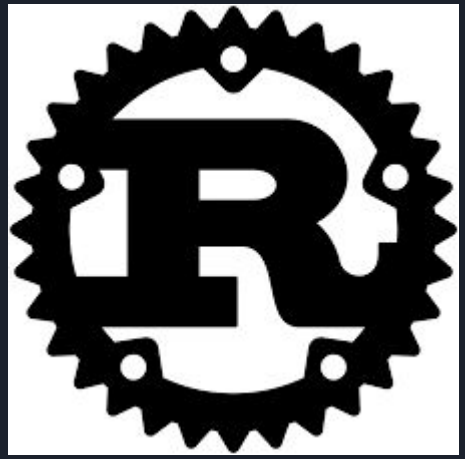
Rust



Nate and Kyan

What is Rust?

Similar C and C++ but with modern features

First created by Graydon Hoare and later developed by **Mozilla Research**

- First Stable Release was in 2015

Systems Programing Language

- **Performance**, **Efficiency**, and **Control**
- Creates low-level programing
- Interacts with computer hardware

Key Features of Rust

Variables

- Rust is **statically typed:** variable types are determined at **compile time** not at runtime (which would be dynamically typed)
- All variables are **immutable** by default

Compilation in Rust:

- Rust is compiled in its previous version
- Was originally compiled in OCaml

Fast:

- Rust is **faster** than Python and around the same speed or faster than C++

Key Features of Rust Cont.

Memory Safety Without Garbage Collection:

- **ownership system** to manage memory
  a. will not crash due to dangling pointers, null references, or memory leaks.
- Rust doesn't rely on garbage collection

Concurrency Without Fear:

- Rust's strict compile-time checks eliminate **data races**

Zero-Cost Abstractions:

- Rust gives you high-level constructs like iterators and smart pointers without sacrificing performance.

Variables in Rust

- Most variables in Rust are immutable by default
  - Ex: integers, floats, characters, strings, arrays, references, functions, etc.
- Variables must be referenced and changed within their scope
- Variable types can be explicit or inferred with the let keyword

Which of these is correct?

```
fn main(){
    let var: i32 = -10;
    println!("{var}");
    var = 10;
    println!("{var}")
}
```

```
fn main(){
    let mut var: i32 = -10;
    println!("{var}");
    var = 10;
    println!("{var}")
}
```

Error messages in Rust

```
   Compiling playground v0.0.1 (/playground)
error[E0384]: cannot assign twice to immutable variable `var`
 --> src/main.rs:5:5
  |
3 |     let var: i32 = -10;
  |         --- first assignment to `var`
4 |     println!("{var}");
5 |     var = 10;
  |     ^^^^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
3 |     let mut var: i32 = -10;
  |         +++
```

Errors vs. Panic! in Rust

- **Rust has two error types, recoverable and unrecoverable**
- Recoverable errors can be corrected with some help, and do not need to stop the program.
- Unrecoverable errors need to stop the program so something else can be done

Example Code

Memory Allocation in Rust

- Most variables allocate memory in the stack when created in rust
- There are a few exceptions which dynamically allocate memory on the heap
- Other languages offer other wise to handle this heap memory, Rust uses a borrow checker

Borrow Checker

Approach	Pros	Cons	Example languages
Manual memory allocation and deallocation	Developer is in control	Difficult to get right	C, C++, Zig
Garbage collection	No need to worry about memory safety, faster (memory safe) development	Runtime costs, unpredictable performance	Java, Python, JavaScript, Erlang
Borrow checker	Best of both worlds	Need to abide by strict compiler rules	Rust

Source: https://fitech101.aalto.fi/programming-languages/rust/1-introduction/

Python vs. Rust

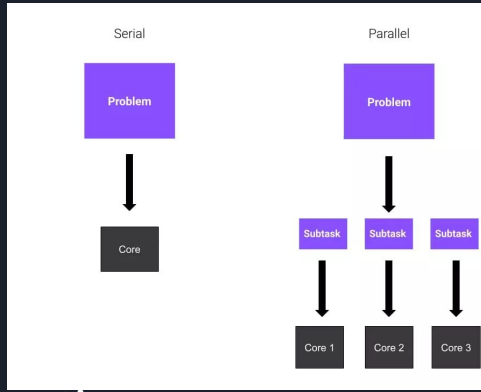
- Our goal:
  - Set x = 42
  - Set y = x
  - Display the value of x, then the value of y, then the value of x

- Python Code Example

- Rust Code Example

What is Parallel Computing?

Parallel computing means **dividing a task into smaller subtasks that can be processed simultaneously**, often on multiple CPU cores or threads



Core Components:

1. **Lightweight**
2. **Shared Resources**
3. **Concurrency**

Common Usage:
Video Games - NPC AI, physics engine and rendering happen at the same time
Streaming videos while shopping in another tab

Advantages: Faster execution, Efficient use of hardware

Rust’s Concurrency Solution

1. Memory Safety
    1.1. Enforce memory safety through unique ownership model
    1.2. AT COMPILE - check how all memory is accessed and looks for any unsafe memory operations
    1.3. Avoids **buffer overflows, dangling pointers, or accessing freed memory**
2. Ownership Model
    2.1. Data is either owned by single threads or shared by all threads
        2.1.1. Owned data CANNOT be shared unless exceptions like Mutex calls are made
        2.1.2. Shared data used synchronization primitives like Mutex and RWLocks
3. Data Race Prevention
    3.1. Ensures prevention through memory safety model
        3.1.1. Will immediately terminate if found and not compile

Concurrency Cont.

1. Concurrency Primitives
   1.1. Threads - Lightweight and easy to use
   1.2. Channels - Message-passage between threads (inspired by Go)
   1.3. Mutex and RWLock - Locking shared data
   1.4. Atomic Types - Low-level primates for operations on shared data
2. Powerful Ecosystem
   2.1. Includes libraries within the standard release of Rust
   2.2. Rayon - Powerful library for creating threads
   2.3. Tokio - Used for asynchronous runtime for handling tasks like networking and file I/O
   2.4. Crossbeam - Used for working with thread-safe data types
3. Zero-Cost Abstractions
4. Compile-Time Guarantees

Rust Thread Example

```rust
fn main() {
    // Make a vector to hold the children which are spawned.
    let mut children: Vec<JoinHandle<()>> = vec![];

    for i: u32 in 0..NTHREADS {
        // Spin up another thread
        children.push(thread::spawn(move || {
            println!("this is thread number {}", i);
        }));
    }

    for child: JoinHandle<()> in children {
        // Wait for the thread to finish. Returns a result.
        let _ = child.join();
    }
}
```

```
this is thread number 0
this is thread number 3
this is thread number 6
this is thread number 8
this is thread number 9
this is thread number 5
this is thread number 7
this is thread number 2
this is thread number 4
this is thread number 1
```

Thread Example: Rust vs Python

RUST

```
Total sum: 20000000100000000
Time taken: 1.50s
```

```rust
use std::thread;
use std::time::Instant;

fn main() {
    let ranges = vec![
        (1, 50_000_000),
        (50_000_001, 100_000_000),
        (100_000_001, 150_000_000),
        (150_000_001, 200_000_000),
    ];

    let start_time = Instant::now(); // Start the timer

    let mut handles = vec![];

    // Start threads for each range
    for (start, end) in ranges {
        let handle = thread::spawn(move || {
            (start..=end).fold(0u64, |acc, x| acc + x)
        });
        handles.push(handle);
    }

    let mut total_sum = 0;

    // Collect results from all threads
```

Thread Example: Rust vs Python

PYTHON

```
Total sum: 20000000100000000
Time taken: 15.08 seconds
```

```python
import threading
import time

def compute_sum(start, end, results, index):
    results[index] = sum(range(start, end + 1))

if __name__ == "__main__":
    ranges = [
        (1, 50_000_000),
        (50_000_001, 100_000_000),
        (100_000_001, 150_000_000),
        (150_000_001, 200_000_000),
    ]

    results = [0] * len(ranges)
    threads = []

    start_time = time.time()  # Start the timer

    # Start threads for each range
    for i, (start, end) in enumerate(ranges):
        thread = threading.Thread(target=compute_sum, args=(start, end, results, i))
        threads.append(thread)
        thread.start()

    # Wait for all threads to complete
    for thread in threads:
```

Use Cases

- Priority towards memory safety and resource management
  - Game Development, Backend Development,Blockchain
- Cybersecurity
  - Can minimize human-caused bugs
  - Null pointers
  - Buffer Overflow
  - Dangling Pointers
  - Accessing Freed Memory
- Only not useful in situations where a slower language is acceptable or a faster one is required