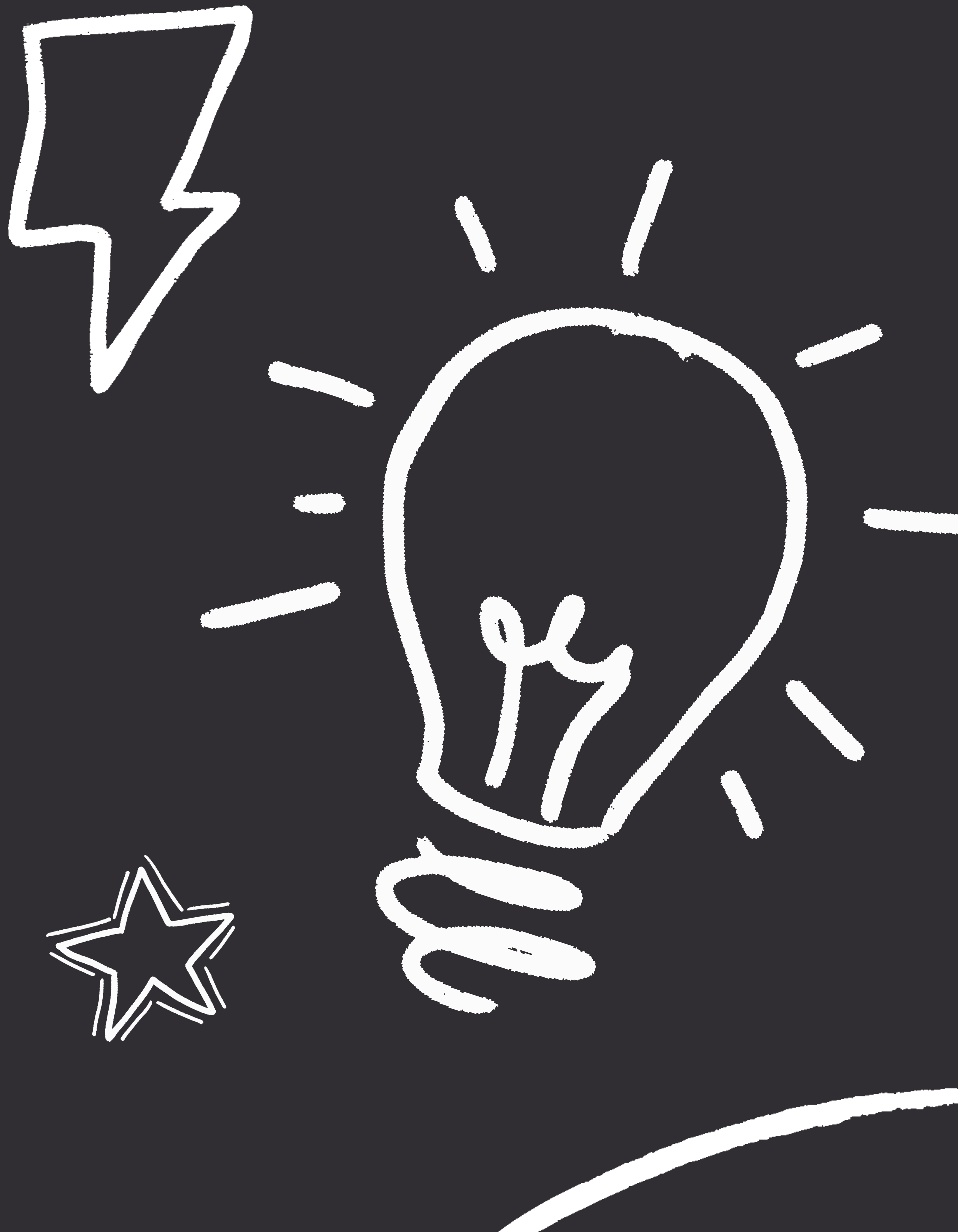
COLBY COLLEGE

Rust: The Modern Marvel of Systems Programming



Presenters:

Sameer Khan

Date:

17 May 2024

Introduction

- Did you know Rust is a systems programming language developed by Mozilla Research? Its first stable release came out in 2015.

- Have you ever wondered what makes Rust stand out? It's designed to be safe, concurrent, and practical, offering memory safety without sacrificing performance (without garbage collection).

- Why was Rust created? It was developed to tackle the common problems in system-level programming, especially those related to safety and concurrency.

- Did you know that these issues often lead to bugs and security vulnerabilities in languages like C and C++? Rust aims to provide a more secure and efficient way to write system-level code.

- Curious about how Rust achieves this? It combines modern language features with a strong focus on safety and performance, making it a great choice for systems programming.

Tonight's Presenter

Sameer Khan

Rust

Why Rust?

Agenda

- Growing Popularity
- Use Cases of Rust
- Polymorphism
- Memory Management
- Additional Feature: Concurrency
- Thoughts
- Q&A

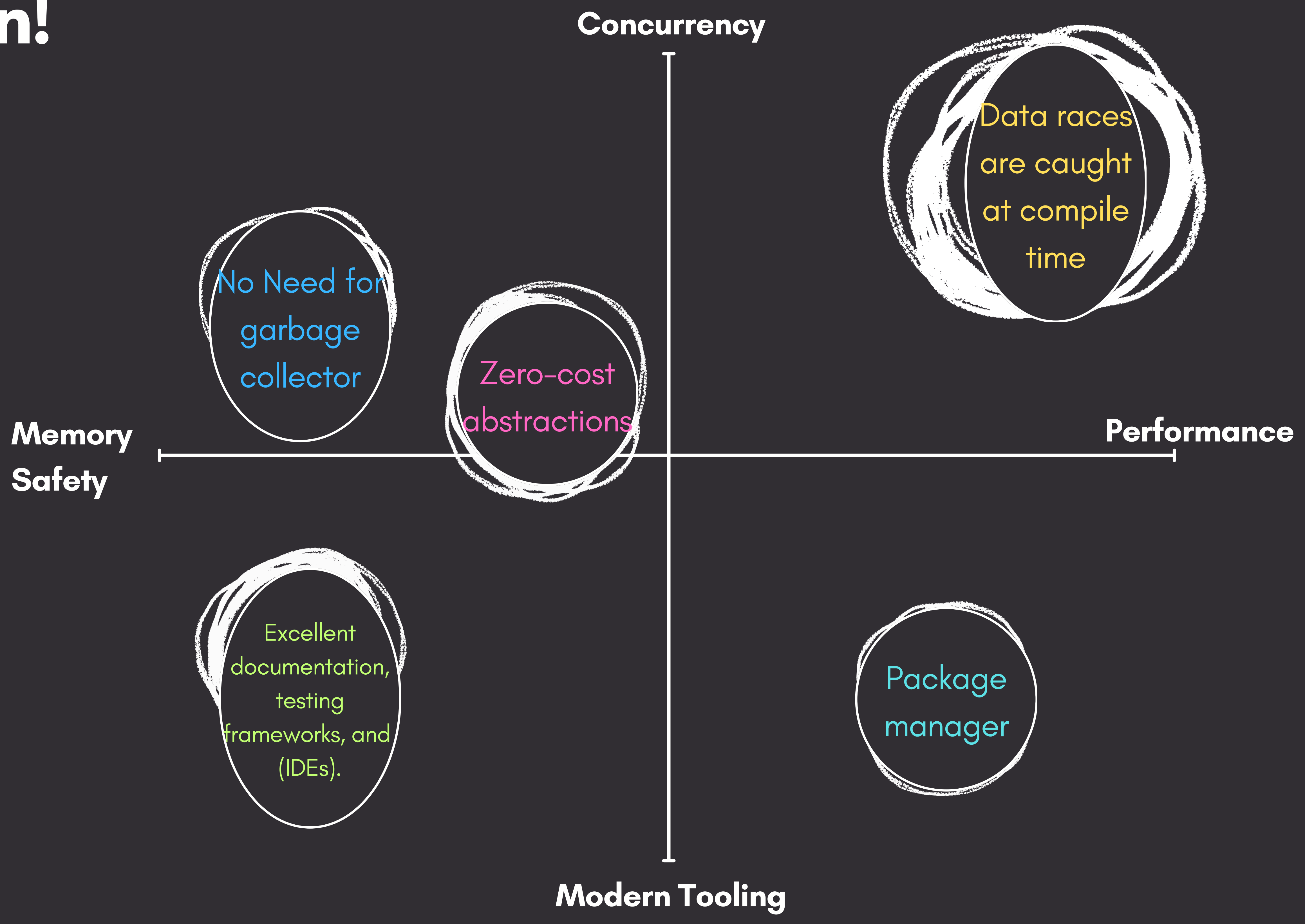
EXTRAS

Comparision
with
C- Language

Popularity!

- Popularity: Rust has been voted the "most loved programming language" in the Stack Overflow Developer Survey for several consecutive years.
- Adoption: Many companies, including Mozilla, Microsoft, Dropbox, and Amazon, use Rust in production for various applications.
- Community: The Rust community is known for being welcoming and inclusive, contributing to extensive documentation, tooling, and libraries.

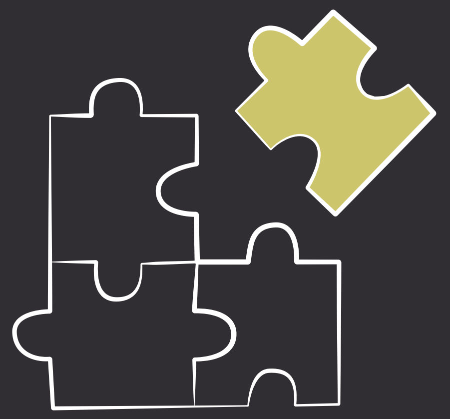
Gaining Attention!



Use Cases of Rust

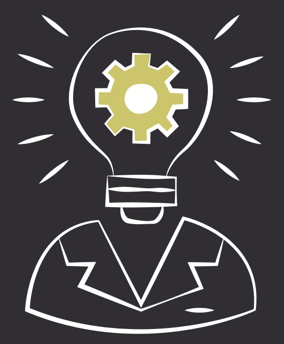
- **WebAssembly:** Ideal for efficient browser execution due to performance and safety.
- **Major Adoption:** Used by Mozilla, Microsoft, Amazon, and Dropbox for system utilities and web services.
- **Developer Experience:** Helpful error messages and clear, readable code with detailed compiler feedback.
- **Versatility:** Suitable for system programming, embedded systems, web development, and more.
- **Sustainability:** Backward compatibility ensures older code works with newer versions, simplifying long-term maintenance.

Polymorphism



Definition

A fundamental concept in object-oriented programming that allows objects of different types to be treated as objects of a common super type. It enables a single function, method, or operator to work in different ways based on the type of input or the context in which it is used.



Compile-time Polymorphism (Static Binding)

- Method Overloading: Multiple methods in the same class with the same name but different parameters.
- Operator Overloading: Defining different behaviors for an operator (e.g., +, -) based on the types of its operands.

Run-time Polymorphism (Dynamic Binding)

- Method Overriding: A subclass provides a specific implementation of a method that is already defined in its superclass.
- Interfaces/Abstract Classes: A class implements an interface or inherits from an abstract class, and provides concrete implementations of the abstract methods.

Thoughts

A common feature but it is implemented in various ways depending on the language's paradigms and type systems. **Java and C++** use method overriding and overloading, **Python and JavaScript** leverage dynamic typing and duck typing, **Rust** utilizes traits and generics, and **C** relies on function pointers. Each language provides unique mechanisms to achieve polymorphism, catering to different programming needs and styles.

Polymorphism in Rust

Polymorphism in Rust is primarily achieved through traits, which are similar to interfaces in other programming languages. Traits allow you to define shared behavior that can be implemented by different types. Rust also uses generics to support compile-time polymorphism, enabling functions and types to operate on multiple data types while maintaining type safety.

1. Traits

Definition:

- Traits are a way to define shared behavior in Rust. They can be implemented by different types, allowing those types to be used interchangeably when the trait is in scope.

Key Points:

- **Trait Definition:** Traits are defined using the trait keyword, followed by the trait name and a list of method signatures.
- **Implementation:** Different types can implement the same trait, providing concrete implementations for the trait's methods.
- **Usage:** Traits enable polymorphism by allowing you to write code that can operate on any type that implements a particular trait.

```rust
// Define a trait named `Shape` with a method `area`.
trait Shape {
    fn area(&self) -> f64;
}
// Define a struct `Circle` with a field `radius`.
struct Circle {
    radius: f64;
}
// Define a struct `Square` with a field `side`.
struct Square {
    side: f64;
}
// Implement the `Shape` trait for `Circle`.
impl Shape for Circle {
    fn area(&self) -> f64 {
        3.14 * self.radius * self.radius
    }
}
// Implement the `Shape` trait for `Square`.
impl Shape for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}
// Function that takes a reference to any object implementing the `Shape` trait.
fn print_area(shape: &dyn Shape) {
    println!("The area is {}", shape.area());
}
fn main() {
    let circle = Circle { radius: 1.0 };
    let square = Square { side: 2.0 };
    // `print_area` can accept both `Circle` and `Square` because they implement the `Shape`
        trait.
    print_area(&circle);
    print_area(&square);
}
```

Polymorphism in Rust

Polymorphism in Rust is primarily achieved through traits, which are similar to interfaces in other programming languages. Traits allow you to define shared behavior that can be implemented by different types. Rust also uses generics to support compile-time polymorphism, enabling functions and types to operate on multiple data types while maintaining type safety.

2. Generics

Definition:

- Generics allow for defining functions, structures, enums, and traits that can operate on multiple types while ensuring type safety at compile time.

Key Points:

- **Generic Functions:** Functions can be defined to accept parameters of any type, constrained by traits to ensure the required behavior is available.
- **Generic Structs and Enums:** Structs and enums can be defined to hold values of any type, enabling them to be used flexibly with different data types.
- **Type Constraints:** Generics can be constrained to types that implement specific traits, ensuring that the generic types support the necessary operations.

```rust
trait Shape {
    fn area(&self) -> f64;
}

struct Circle {
    radius: f64;
}

impl Shape for Circle {
    fn area(&self) -> f64 {
        3.14 * self.radius * self.radius
    }
}
```

Polymorphism in Rust

Polymorphism in Rust has also a connection with static dispatch.

3. Static Dispatch

Definition:

- Static dispatch occurs when the method to call is determined at compile time. This is commonly used with generics and inlined code.

Characteristics:

- **Compile-Time Resolution:** The exact method to call is resolved at compile time.
- **Performance:** No runtime overhead, leading to faster execution and better optimization.
- **Type Safety:** Ensures type correctness at compile time.

```rust
fn print_area<T: Shape>(shape: &T) {
    println!("The area is {}", shape.area());
}
```

Polymorphism in Rust

Polymorphism in Rust has also a connection with dynamic dispatch.

4. Dynamic Dispatch

Definition:

- Dynamic dispatch occurs when the method to call is determined at runtime. This is commonly used with trait objects.

Characteristics:

- **Runtime Resolution:** The method to call is determined at runtime using pointers to trait objects.
- **Flexibility:** Allows for more flexible code, where the exact type isn't known until runtime.
- **Trait Objects:** Trait objects are created using &dyn Trait or Box<dyn Trait>.

```rust
fn print_area(shape: &dyn Shape) {
    println!("The area is {}", shape.area());
}
```

Polymorphism in Rust

5. Associated Types

Definition:

- Associated types are a way of associating a type placeholder with a trait, allowing trait methods to use these types.

Characteristics:

- **Type Association:** Associates a type with a trait, simplifying the use of generics and improving readability.
- **Usage in Traits:** Associated types are specified within trait definitions and implemented by the concrete types.

```rust
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

struct Counter {
    count: i32,
}

impl Iterator for Counter {
    type Item = i32;
    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;
        Some(self.count)
    }
}
```

6. Default Implementations

Definition:

- Traits can provide default implementations for methods, allowing types to inherit this behavior or override it as needed.

Characteristics:

- **Code Reuse:** Facilitates code reuse by providing default behavior that types can use directly.
- **Override Capability:** Types can override the default implementations to provide specific behavior.

```rust
trait Greet {
    fn greet(&self) {
        println!("Hello!");
    }
}

struct Person;

impl Greet for Person {
    // Using the default implementation of greet
}

struct Robot;

impl Greet for Robot {
    fn greet(&self) {
        println!("Greetings, human.");
    }
}

fn main() {
    let person = Person;
    let robot = Robot;

    person.greet(); // Outputs: Hello!
    robot.greet();  // Outputs: Greetings, human.
}
```

Do you know **C-language?**

Polymorphism in Rust and C differs significantly in terms of implementation, language features, and ease of use.

Polymorphism in Rust vs. C

1. Language Support

Rust:

- **Traits:** Built-in support for defining shared behavior across types, similar to interfaces.
- **Generics:** Enables polymorphism in functions, structs, enums, and traits with type safety.
- **Associated Types:** Simplifies and enhances readability of generic code.
- **Dispatch:** Supports both static (compile-time) and dynamic (runtime via trait objects) dispatch.

C:

- **No Direct Support:** Relies on manual implementation for polymorphism.
- **Function Pointers:** Achieves polymorphic behavior via runtime function pointer selection.
- **Structures with Function Pointers:** Mimics object-oriented behavior using structures containing function pointers.
- **Manual VTables:** Requires manual creation of VTables for dynamic dispatch.

```c
#include <stdio.h>

// Define a structure for an animal
typedef struct {
    const char* name;
    void (*make_sound)();
} Animal;

// Define functions to make specific sounds
void dog_sound() {
    printf("Woof!\n");
}

void cat_sound() {
    printf("Meow!\n");
}

int main() {
    // Create instances of Dog and Cat
    Animal dog = {"Dog", dog_sound};
    Animal cat = {"Cat", cat_sound};

    // Create an array of animals
    Animal animals[] = {dog, cat};

    // Iterate through the array and make each animal sound
    for (int i = 0; i < sizeof(animals) / sizeof(animals[0]); ++i) {
        printf("%s says: ", animals[i].name);
        animals[i].make_sound();
    }

    return 0;
}
}
```

Polymorphism in Rust vs. C

2. Implementation Complexity

Rust:

- **Ease of Use:** Rust's built-in features (traits, generics, etc.) make implementing polymorphism straightforward and type-safe.
- **Safety:** The Rust compiler enforces strict type checks, reducing the chances of runtime errors related to polymorphism.
- **Code Reusability:** Traits and generics enhance code reusability and modularity.

C:

- **Manual Effort:** Implementing polymorphism in C requires significant manual effort, including setting up function pointers and managing VTables.
- **Error-Prone:** The lack of language support means more room for errors, such as incorrect function pointer assignments or type mismatches.
- **Less Readable:** Code can become less readable and harder to maintain due to the manual setup required for polymorphism.

```c
#include <stdio.h>

// Define a structure for an animal
typedef struct {
    const char* name;
    void (*make_sound)();
} Animal;

// Define functions to make specific sounds
void dog_sound() {
    printf("Woof!\n");
}

void cat_sound() {
    printf("Meow!\n");
}

int main() {
    // Create instances of Dog and Cat
    Animal dog = {"Dog", dog_sound};
    Animal cat = {"Cat", cat_sound};

    // Create an array of animals
    Animal animals[] = {dog, cat};

    // Iterate through the array and make each animal sound
    for (int i = 0; i < sizeof(animals) / sizeof(animals[0]); ++i) {
        printf("%s says: ", animals[i].name);
        animals[i].make_sound();
    }

    return 0;
}
}
```

Polymorphism in Rust vs. C

3. Type Safety

Rust:

- **Compile-Time Checks:** Rust's type system and borrow checker ensure type safety at compile time, preventing many common errors.
- **Trait Bounds:** Traits and generics are constrained by bounds, ensuring that types adhere to expected behavior.

C:

- **Runtime Errors:** C relies on runtime checks, which can lead to errors if function pointers are incorrectly assigned or used.
- **Manual Type Management:** Type safety depends on the programmer's discipline and careful management of types and function pointers.

```c
#include <stdio.h>

// Define a structure for an animal
typedef struct {
    const char* name;
    void (*make_sound)();
} Animal;

// Define functions to make specific sounds
void dog_sound() {
    printf("Woof!\n");
}

void cat_sound() {
    printf("Meow!\n");
}

int main() {
    // Create instances of Dog and Cat
    Animal dog = {"Dog", dog_sound};
    Animal cat = {"Cat", cat_sound};

    // Create an array of animals
    Animal animals[] = {dog, cat};

    // Iterate through the array and make each animal sound
    for (int i = 0; i < sizeof(animals) / sizeof(animals[0]); ++i) {
        printf("%s says: ", animals[i].name);
        animals[i].make_sound();
    }

    return 0;
}
}
```

Polymorphism in Rust vs. C

4. Performance

Rust:

- **Static Dispatch:** Rust's static dispatch (generics) is resolved at compile time, leading to highly optimized code with no runtime overhead.
- **Dynamic Dispatch:** Rust's dynamic dispatch (trait objects) has some runtime overhead but is still efficient and type-safe.

C:

- **Function Pointers:** Using function pointers introduces some runtime overhead due to indirect function calls.
- **Manual Optimization:** Performance optimization requires manual effort, such as inlining functions or carefully managing function pointers.

```c
#include <stdio.h>

// Define a structure for an animal
typedef struct {
    const char* name;
    void (*make_sound)();
} Animal;

// Define functions to make specific sounds
void dog_sound() {
    printf("Woof!\n");
}

void cat_sound() {
    printf("Meow!\n");
}

int main() {
    // Create instances of Dog and Cat
    Animal dog = {"Dog", dog_sound};
    Animal cat = {"Cat", cat_sound};

    // Create an array of animals
    Animal animals[] = {dog, cat};

    // Iterate through the array and make each animal sound
    for (int i = 0; i < sizeof(animals) / sizeof(animals[0]); ++i) {
        printf("%s says: ", animals[i].name);
        animals[i].make_sound();
    }

    return 0;
}
}
```

Do you know How **Memory Management** works in Rust?

Memory management in Rust and C differs significantly due to their different approaches to memory safety and ownership.

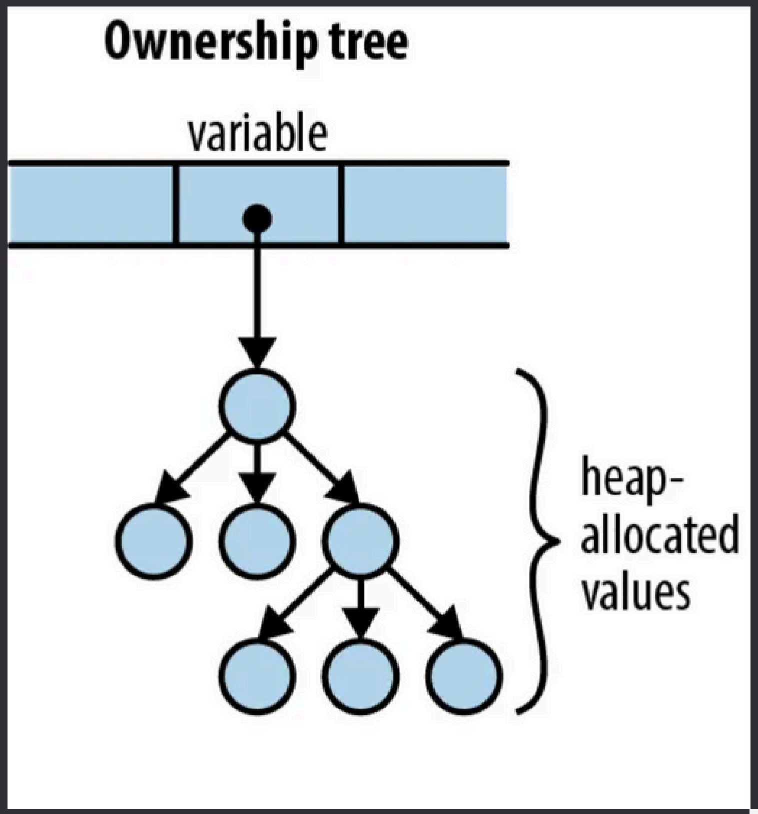
Memory Management in Rust

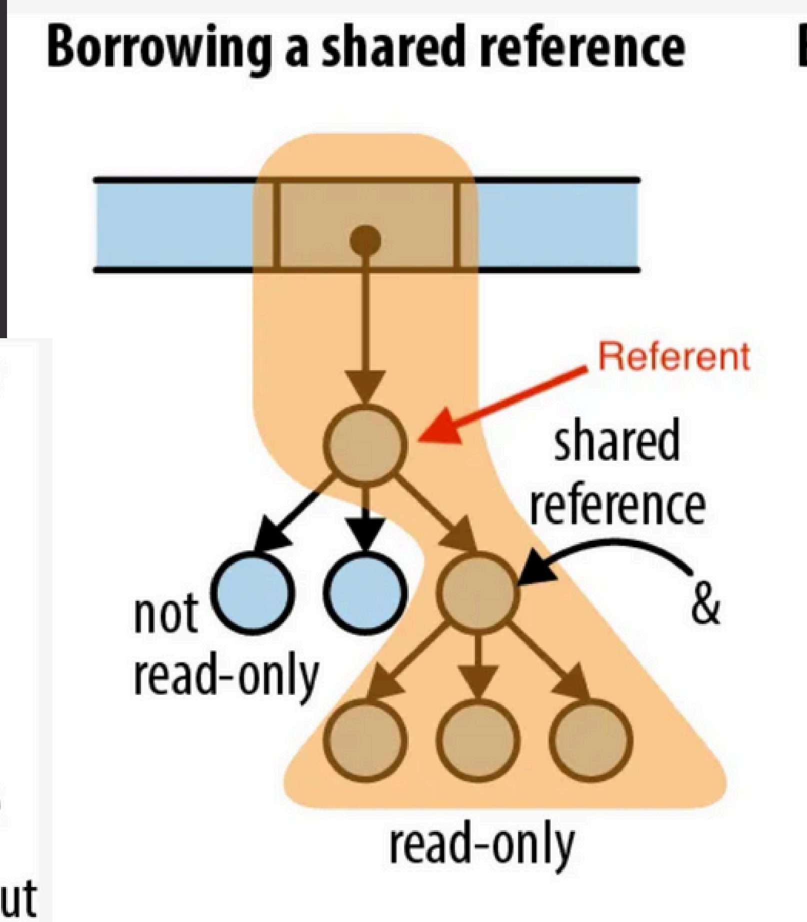
- **Ownership System:** Every value in Rust has a variable that is its owner. When the owner goes out of scope, Rust automatically deallocates the memory associated with that value. This helps prevent memory leaks.

- **Borrowing:** Rust's borrowing system allows functions to borrow references to values without taking ownership. This prevents issues like dangling pointers because the borrow checker ensures that references remain valid.

- **Lifetimes:** Rust uses lifetimes to ensure that references to memory remain valid for as long as they are used. This prevents issues such as use-after-free errors, where memory is accessed after it has been deallocated.

- **No Garbage Collection:** Unlike some other languages, Rust does not have a garbage collector. Instead, it relies on its ownership system and borrowing rules to manage memory safely and efficiently.





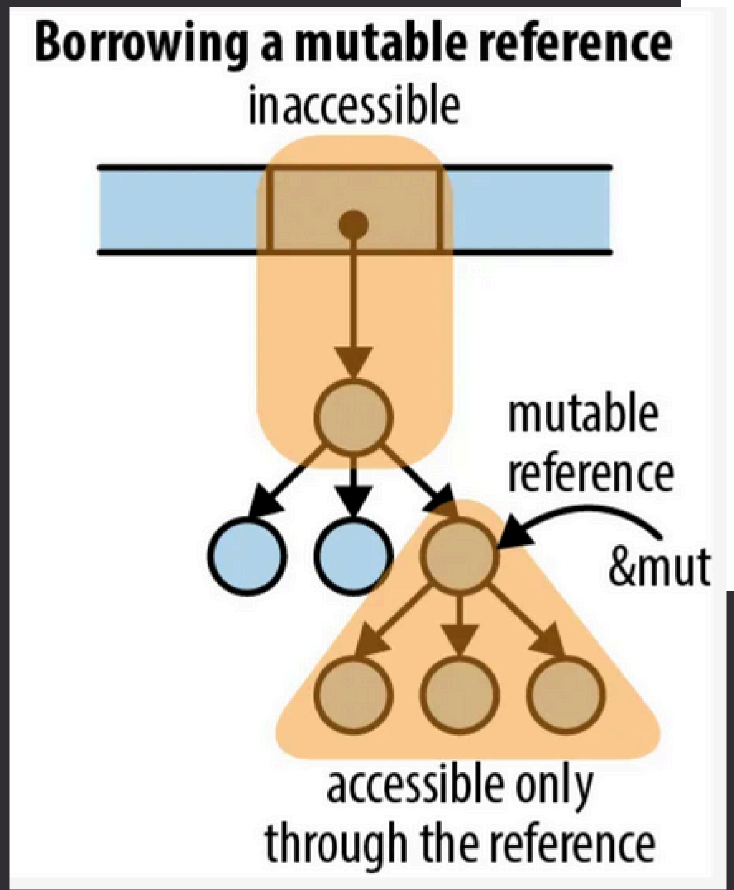


Image Source:
https://medium.com/coinmonks/understanding-ownership-in-rust-with-examples-73835ba931b1

Memory Management in C

- **Manual Memory Management:** In C, developers are responsible for explicitly allocating and deallocating memory. They use functions like malloc, calloc, and realloc to allocate memory and free to deallocate it.
- **Risk of Memory Leaks:** Memory leaks occur when memory that is no longer needed is not deallocated. This can happen if a developer forgets to call free after allocating memory, leading to wasted memory over time.
- **Risk of Dangling Pointers:** Dangling pointers occur when a pointer continues to reference memory that has been deallocated. This can lead to unpredictable behavior and crashes in a program.
- **Limited Safety Features:** C does not have built-in features to prevent common memory-related issues, such as accessing uninitialized memory or buffer overflows, which can lead to security vulnerabilities.

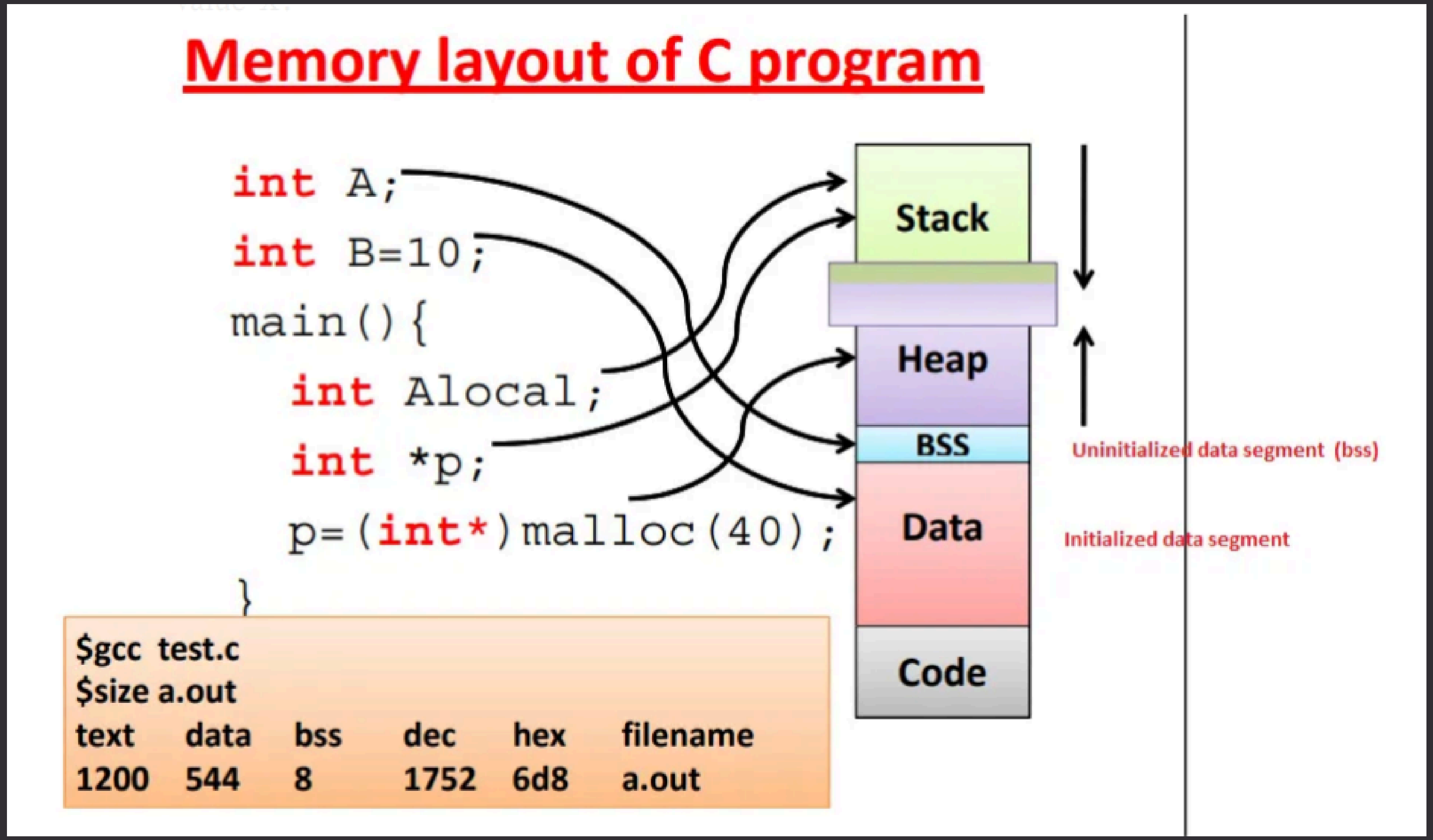


Image Source:https://medium.com/@saidsadaoy/memory-management-in-c-78797950efaa

Memory Management Comparision

- **Safety vs. Control:** Rust's approach provides a high level of safety by preventing common memory-related issues at compile time. In contrast, C gives developers more control over memory management but requires them to manually avoid pitfalls.

- **Complexity:** Rust's ownership system and borrowing rules can be more complex for developers to learn and use effectively compared to C's more straightforward manual memory management.

- **Performance:** Both languages can achieve similar levels of performance, but Rust's memory management can sometimes lead to more optimized code due to its strict rules and ability to prevent certain types of bugs.

- **Use Cases:** C is often used in systems programming and situations where low-level control over memory is necessary. Rust is also suitable for systems programming but offers additional safety features that make it appealing for applications where security and reliability are critical.

Image Source:https://medium.com/@saidsadaoy/memory-management-in-c-78797950efaa

Ugh Long Day!

Last but not the least

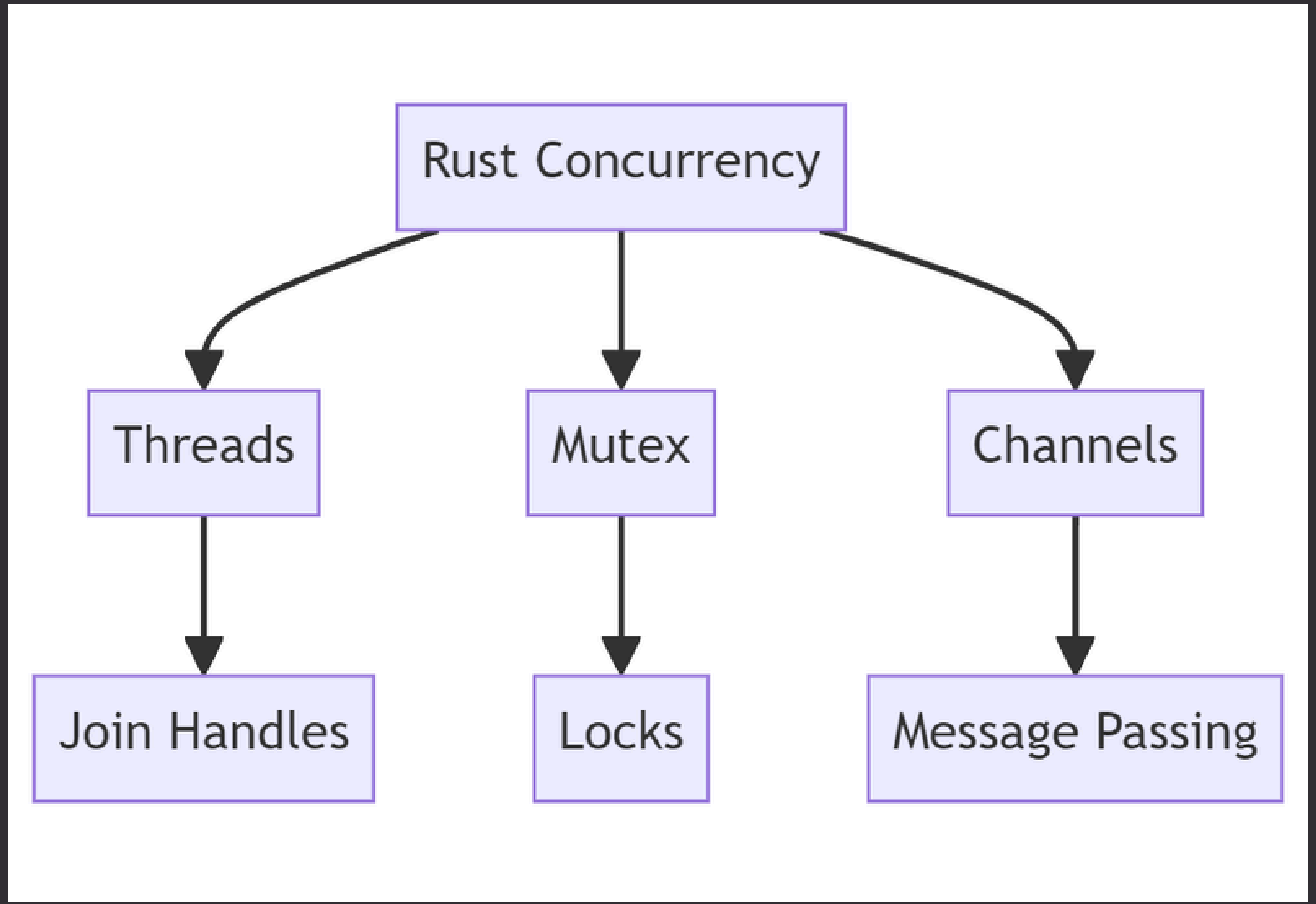
Concurrency

Concurrency is the ability of a system to handle multiple tasks or processes simultaneously.

Concurrency in Rust

Image Source: https://medium.com/coinmonks/understanding-ownership-in-rust-with-examples-73835ba931b1

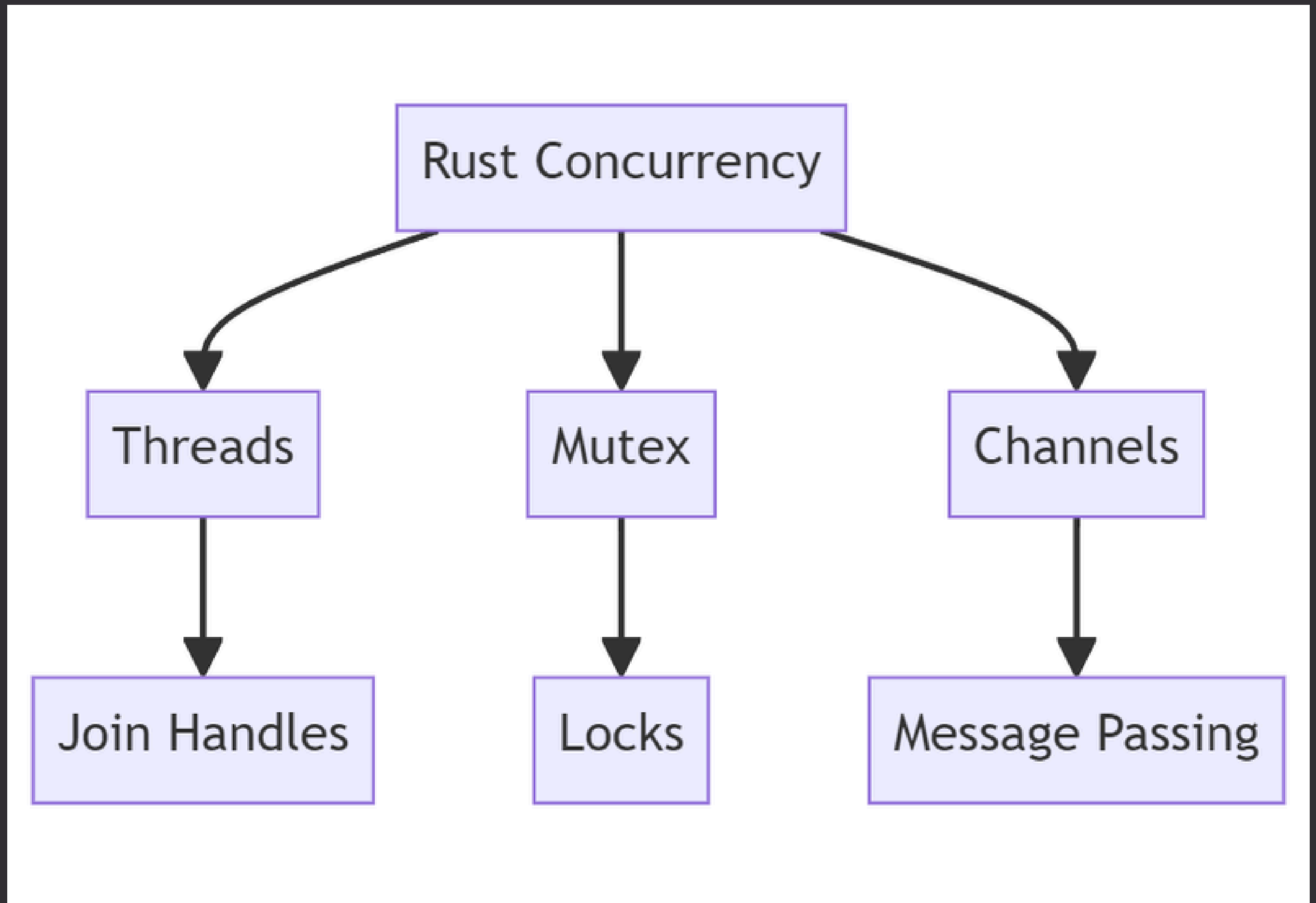
- **Ownership System:** Prevents data races by allowing only one thread to mutate data at a time.
- **Traits for Concurrency:** Provides traits like Send and Sync for safe sharing between threads.
- **Concurrency Primitives:** Offers threads, mutexes, channels, and atomics for managing concurrency.
- **Thread Safety:** Enforced by the type system, ensuring safe sharing of mutable data.
- **Asynchronous Programming:** Supports asynchronous programming for efficient concurrent tasks.
- **Fearless Concurrency:** Rust aims to make concurrent programming safe and easy, avoiding common issues like data races and deadlocks.



Concurrency in C

Image Source: https://medium.com/coinmonks/understanding-ownership-in-rust-with-examples-73835ba931b1

- **Approach:** Achieved using the POSIX thread library (pthread).
- **Thread Management:** pthread provides functions for creating, managing, and synchronizing threads.
- **Synchronization:** Utilizes primitives like mutexes and semaphores to coordinate shared resource access.
- **Asynchronous Operations:** Implemented using threads and callbacks due to the lack of built-in support.
- **Thread Safety:** Programmer's responsibility, requiring careful synchronization and memory management.



Concurrency in Rust vs. C

Image Source: https://medium.com/coinmonks/understanding-ownership-in-rust-with-examples-73835ba931b1

Safety vs. Control:

- **Rust:** Ensures safety through ownership and borrowing, reducing bugs like data races, but offers less direct memory control.
- **C:** Provides more control over memory management but requires manual effort for safety.

Complexity:

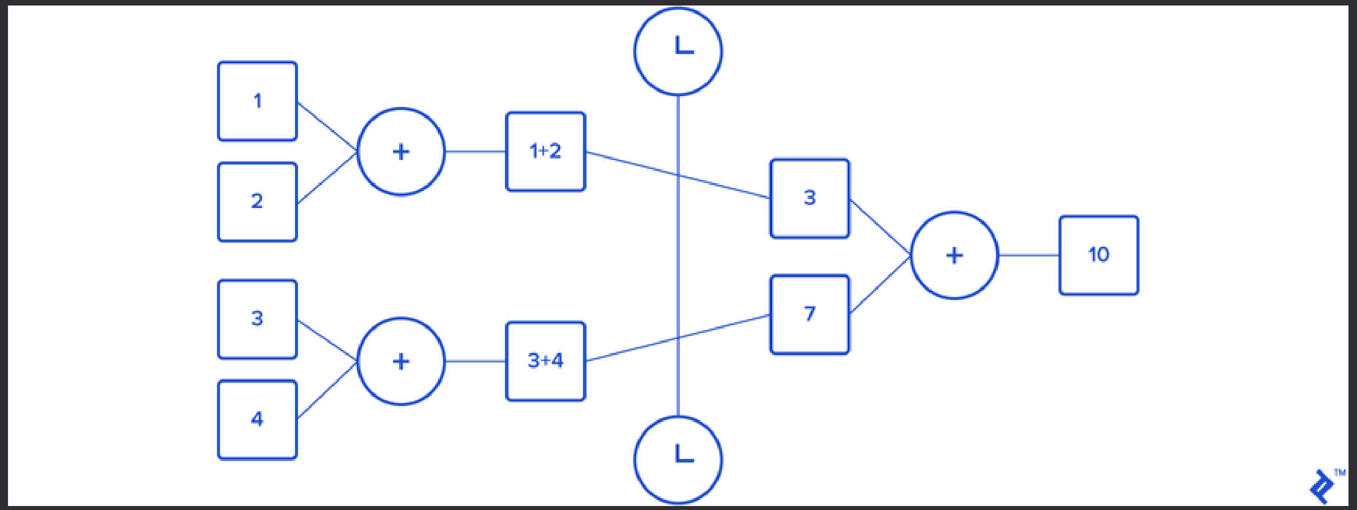
- **Rust:** Ownership and borrowing can be complex but prevent common concurrency bugs.
- **C:** Manual memory management and synchronization are straightforward but can lead to errors in complex scenarios.

Performance:

- **Rust:** Can achieve similar performance to C but with added safety features.
- **C:** Focuses on performance, but manual management can lead to issues if not careful.

Use Cases:

- **Rust:** Suited for applications needing safety and concurrency, like systems programming and web servers.
- **C:** Common in operating systems, embedded systems, and performance-critical applications.



Thank You

Q&A