Basic’s

Rust Memory Diagram

Basic’s

✅ Memory Diagram (4 Bytes, Decimal Values)

lua

```
Variable: i = 2 (i32)

+-----------+----------------+-----------------------------+
| Address   | Byte (Decimal) | Comment                     |
+-----------+----------------+-----------------------------+
|  0x1000   |  2             | Least significant byte (LSB) |
+-----------+----------------+-----------------------------+
|  0x1001   |  0             |                             |
+-----------+----------------+-----------------------------+
|  0x1002   |  0             |                             |
+-----------+----------------+-----------------------------+
|  0x1003   |  0             | Most significant byte (MSB)  |
+-----------+----------------+-----------------------------+
```

Basic’s

✅ Memory Diagram for `i8`

sql

```
Variable: x = 5 (i8)

+-----------+----------------+
| Address   | Byte (Decimal) |
+-----------+----------------+
| 0x3000    | 5              |   (Only 1 byte)
+-----------+----------------+
```

Rust Movement of Address

```rust
let x: i32 = 42;
let ptr: *const i32 = &x;
```

Here, `ptr` is a **raw pointer** pointing to variable `x`.

```
Variable: x = 42 (i32)

x (Data):
+-----------+-----------------+
| Address   | Byte (Decimal) |
+-----------+-----------------+
| 0x2000    | 42             |  (Stored in 4 bytes as i32)
+-----------+-----------------+
| 0x2001    | 0              |
+-----------+-----------------+
| 0x2002    | 0              |
+-----------+-----------------+
| 0x2003    | 0              |
+-----------+-----------------+
```

```
Pointer Variable: ptr

ptr (Pointer):
+-----------+-----------------+
| Address   | Pointer Value  |
+-----------+-----------------+
| 0x3000    | 0x2000         |  (Address of x)
+-----------+-----------------+
```

Rust Movement of Address

📦 Concept: Two Pointers, Single Data

- The **data** stays in **one place in memory**.
- Pointers simply **store the address** of that data.
- If a pointer is reassigned or moved, only the **pointer address** changes.

Rust Movement of Address

✅ Memory Diagram – Two Pointers, One Data

```
pgsql

Data Variable: x = 55 (Stored in Stack)

0x5000    +-----------+
          |   55      |   (Value of x)
          +-----------+

Pointer 1 (ptr1):
0x6000    +-----------------+
          | Pointer to 0x5000 |   (ptr1 holds address of x)
          +-----------------+

Pointer 2 (ptr2):
0x7000    +-----------------+
          | Pointer to 0x5000 |   (ptr2 holds same address)
          +-----------------+
```

```rust
fn main() {
    let x: i32 = 55;

    let ptr1: *const i32 = &x;
    let ptr2: *const i32 = ptr1;  // Copying/moving pointer (not data)

    println!("{:p}", ptr1);       // Same address
    println!("{:p}", ptr2);       // Same address
}
```

Rust Movement of Address

✅ Memory Diagram

lua

```
DATA SEGMENT (read-only):
+-------------+
| "coderrange" |
+-------------+

STACK:
+-------------+
| names       |  --->  Points to "coderrange"
+-------------+
```

```
let names: &str = "coderrange";
```

Rust Movement of Address

📦 Diagram – Movement of `Box` Pointer

lua

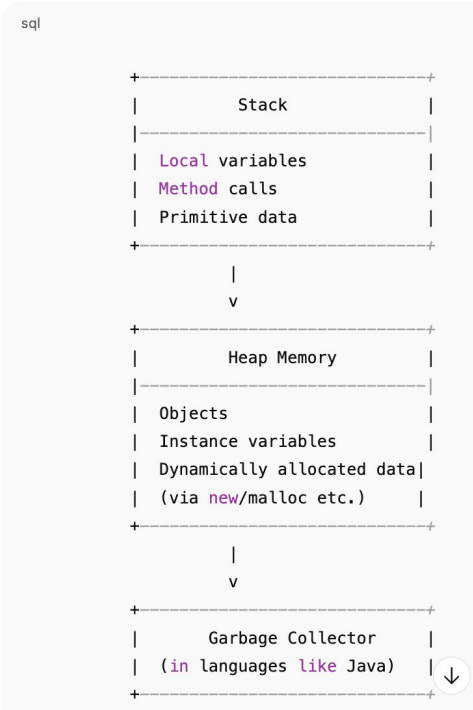
```
HEAP:
+--------+
|   10   |   (Owned by Box)
+--------+

STACK (before move):
+------+        +------+
| b1   | -----> | 10   |  (Box pointer)
+------+        +------+

STACK (after move):
b1 = INVALID

+------+        +------+
| b2   | -----> | 10   |  (Box pointer now in b2)
+------+        +------+
```

```rust
let b1 = Box::new(10);    // Box owns data on HEAP

let b2 = b1;              // Ownership of pointer (Box) MOVES to b2

// println!("{}", b1);    // ❌ Error: b1 is invalid after move

println!("{}", b2);       // ✅ b2 is now the owner
```

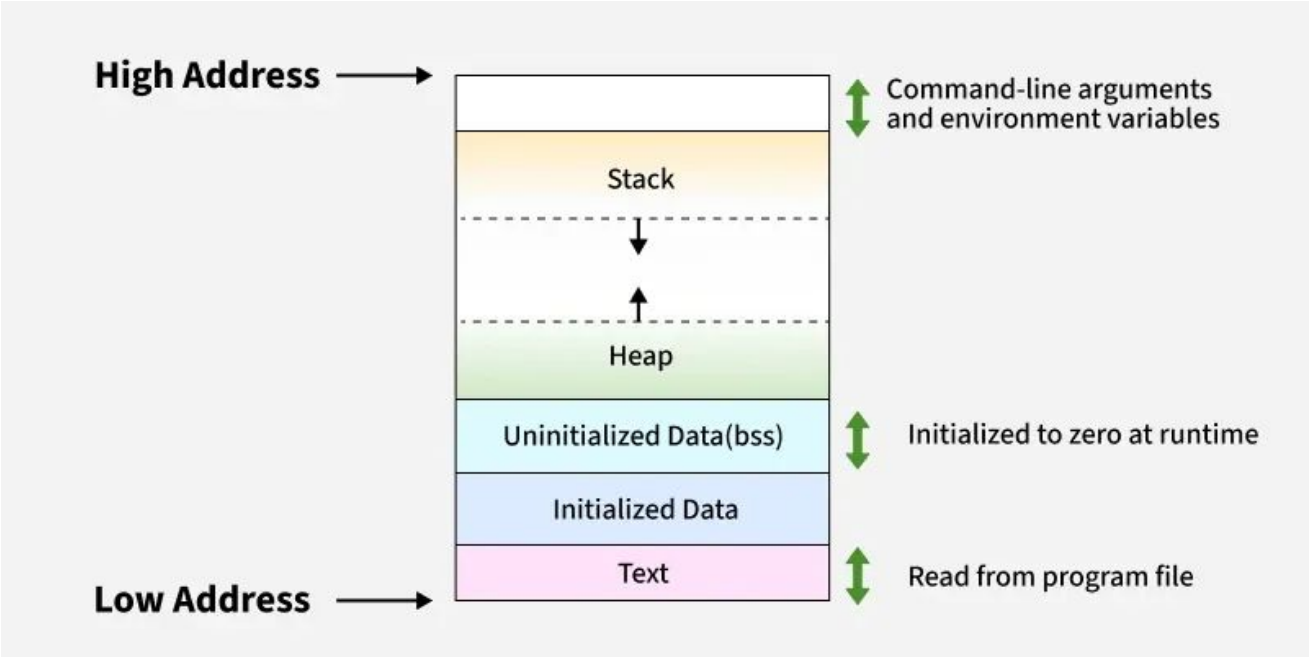
Memory Diagram

🧠 Heap Memory – Conceptual Diagram

sql



Memory Diagram



Memory Diagram

✅ Visual Memory Diagram

vbnet

```
let name = String::from("Rust");

STACK:
+--------------------------+
| Pointer to Heap (name)   |
| Length = 4               |
| Capacity >= 4            |
+--------------------------+

HEAP:
+----+----+----+----+
| R  | u  | s  | t  |
+----+----+----+----+
```

When you do `name.push_str(" Language");`, r

Memory Diagram

1️⃣ Pointers vs. References

pgsql

```
+-----------+              +-----------------+
|  speed    | 100          |  ref: &speed    |
+-----------+              +-----------------+
                            Points to 'speed'
```

pgsql

```
+-----------+              +----------------------+
|  speed    | 100          |  raw_ptr: *const T   |
+-----------+              +----------------------+
                            Unsafe pointer to 'speed'
```

Memory Diagram

3️⃣ Ownership and Moves

rust

```
let a = String::from("Hello");

+-------+      +------------------+
|   a   | ---> | "Hello" in HEAP  |
+-------+      +------------------+

let b = a;    // Ownership moved to b

+-------+      +------------------+
|   b   | ---> | "Hello" in HEAP  |
+-------+      +------------------+

a is now invalid (moved).
```

Memory Diagram

4️⃣ Borrowing (`&T` and `&mut T`)

Immutable Borrow:

kotlin

```
let data = 42;
let ref1 = &data;   // Immutable borrow

+-------+      +--------+
| data  | 42   | ref1   |
+-------+      +--------+
                Points to 'data'
```

Memory Diagram

Mutable Borrow (Only One Allowed):

wasm

```
let mut data = 42;
let ref_mut = &mut data;   // Mutable borrow

+-------+      +-------------+
| data  | 42   | ref_mut     |
+-------+      +-------------+
                Points to 'data'
```

Memory Diagram

5️⃣ Lifetimes

rust

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str
```

Diagram:

rust

```
+--------------------+
|  'a lifetime scope|
+--------------------+
|   x: &'a str      |   Both references valid
|   y: &'a str      |   for same lifetime 'a
|   return: &'a str |
+--------------------+
```

Memory Diagram

✅ Best RAII + Heap Memory Diagram

sql

```
1️⃣ Resource Allocation (Vec Created)
+-----------------------------+
| v (Vec) metadata on STACK   |
| - pointer to heap           |
| - length, capacity          |
+-----------------------------+
            |
            v
+-----------------------------+
| Heap Memory Allocated       |
| [1, 2, 3]                   |
+-----------------------------+
```

```
2️⃣ Ownership Scope
+-----------------------------+
| v owns the heap memory      |
| (RAII: resource tied to v)  |
+-----------------------------+
```

```
3️⃣ Scope Ends (Drop Triggered)
+-----------------------------+
| v goes out of scope         |
| Drop::drop() called         |
+-----------------------------+
            |
            v
+-----------------------------+
| Heap Memory Automatically   |
| Deallocated (Freed)         |
+-----------------------------+
```

Memory Diagram

1️⃣ `mut` vs. `&mut` – (Most Common Confusion)

Situation	Code Snippet	Meaning
`mut` before variable	`let mut x = 10;`	`x` itself is mutable.
`mut` in reference	`let y = &mut x;`	Reference to mutable `x`. You can mutate **what y points to**, not y itself.

Memory Diagram

2️⃣ Raw Pointers: `*const` vs. `*mut`

Keyword	Example	Meaning
`*const T`	`let ptr: *const i32`	Immutable raw pointer.
`*mut T`	`let ptr: *mut i32`	Mutable raw pointer.
Dereferencing	`unsafe { *ptr }`	Always inside `unsafe` block.

Memory Diagram

3️⃣ `ref` vs. `&`

Keyword	Use Case	Example
`ref`	Pattern matching	`let ref y = x;`
`&`	Reference in binding	`let y = &x;`

`ref` is rarely used in modern Rust; using `&` is clearer for most cases.

Memory Diagram

4️⃣ `static` vs. `const`

Keyword	Example	Stored In	Lifespan	Mutable?
`const`	`const MAX: u32 = 100;`	DATA/TEXT	Compile-time	Never
`static`	`static COUNT: u32 = 10;`	DATA	Entire program	Yes (unsafe with `static mut`)

Memory Diagram

5️⃣ Shadowing with `let`

Confusion Point

Variable can be redefined using `let` again in the same scope.

rust

```rust
let x = 5;
let x = x + 1;   // Shadowing, not mutation
```

Memory Diagram

6️⃣ Ownership / Borrowing Errors

Confusion Point	Mistake Example
Trying to use moved value	After `let b = a;`, `a` becomes invalid if non-Copy.
Borrow checker complaints	Holding both `&mut` and `&` simultaneously causes compiler errors.

Memory Diagram

7️⃣ Lifetime Annotations (`'a`)

- Even experienced devs struggle with:
  - Missing lifetime annotations.
  - Incorrect `'a` placement.
  - Functions returning references to local variables.

Memory Diagram

8️⃣ Interior Mutability Confusion

- Use of:
  - `RefCell<T>` – run-time borrow checking.
  - `Cell<T>` – copyable interior mutability.
  - `Mutex<T>` – for multi-threaded interior mutability.

Memory Diagram

✅ RAII + Box – Best Box Diagram

pgsql

```
1️⃣ Resource Allocation (Box Created)

+-----------------------------+
|  b (Box pointer)            |   // Stored on STACK
|  Points to heap memory      |
+-----------------------------+
              |
              v
+-----------------------------+
| Heap Memory Allocated       |
| [ 500 ]                     |   // Data stored on HEAP
+-----------------------------+
```

```
2️⃣ Ownership Scope (RAII Active)

+-----------------------------+
| b owns the heap memory      |
| Resource tied to b's life   |
+-----------------------------+


3️⃣ Scope Ends (RAII Cleanup)

+-----------------------------+
| b goes out of scope         |
| Drop::drop() called         |
+-----------------------------+
              |
              v
+-----------------------------+
| Heap Memory Automatically   |
| Freed (Deallocated)         |
+-----------------------------+
```

Memory Diagram

✅ Example Code Using `Box`

```rust
fn main() {
    let b = Box::new(500);   // Allocates 500 on HEAP
    println!("{}", b);       // b owns the heap memory

}   // Scope ends -> Drop called -> HEAP memory freed
```

&b ⟶ 7000
6000

6000⟶
500

Memory Diagram

&b → 7000
6000

let b1 =b

Let &b1 = &b

6000→
500

Memory Diagram

✅ Example Code Using `Box`

rust

```rust
fn main() {
    let b = Box::new(500);   // Allocates 500 on HEAP
    println!("{}", b);       // b owns the heap memory

}   // Scope ends -> Drop called -> HEAP memory freed
```

&boxing → 10004
6007

6007 → heap
500

&boxer → 7000
10004

Memory Diagram

✅ 1️⃣ Tuple Memory Diagram

```rust
let t = (10, 3.14, true);
```

```
+-----------+-----------+-----------+
|   10 (i32) | 3.14 (f64) | true (bool)|
+-----------+-----------+-----------+
```

Memory Diagram

✅ 2️⃣ Array Memory Diagram

rust

```rust
let arr = [1, 2, 3, 4];
```

```
+-----+-----+-----+-----+
|  1  |  2  |  3  |  4  |   (All i32)
+-----+-----+-----+-----+
```

Memory Diagram

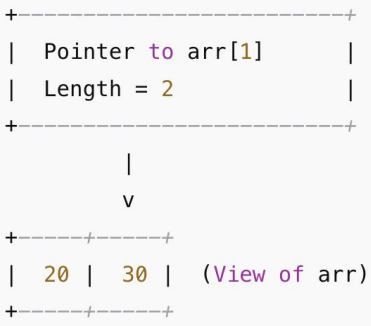
✅ 3️⃣ Slice Memory Diagram

rust

```rust
let arr = [10, 20, 30, 40];
let slice = &arr[1..3];  // [20, 30]
```

- A **pointer** to start of data.
- A **length** (number of elements in slice).

pgsql



Memory Diagram

✅ Slice – STACK (metadata) + Referenced Data (STACK

- Example:

```rust
let arr = [10, 20, 30, 40];
let slice = &arr[1..3];  // slice = [20, 30]
```

Memory Diagram

🔍 When Slice Points to Heap:

- Example:

```rust
let v = vec![10, 20, 30, 40];  // Data on HEAP
let slice = &v[1..3];          // Slice view over HEAP data
```

Memory Diagram

```
Tuple:
+-----------+-----------+-----------+
| Element 1 | Element 2 | Element 3 |   (All on STACK)
+-----------+-----------+-----------+

Array:
+-----+-----+-----+-----+
|  1  |  2  |  3  |  4  |   (All on STACK)
+-----+-----+-----+-----+

Slice (of an array):
+------------------------+
| Pointer + Length (STACK)
+------------------------+
           |
           v
+-----+-----+
|  20 |  30 |   (Data in STACK or HEAP depending on source)
+-----+-----+
```

Memory Diagram

1 Tuple + `mut` — Challenges

rust

```rust
let mut t = (1, 2.0, true);  // Entire tuple is mutable

t.0 = 5;         // ✅ Can modify individual elements
t.1 = 3.14;      // ✅
t.2 = false;     // ✅
```

2 Array + `mut` — Challenges

rust

```rust
let mut arr = [1, 2, 3, 4];

arr[0] = 99;     // ✅ Modify elements directly
```

3 Slice + `mut` — Challenges

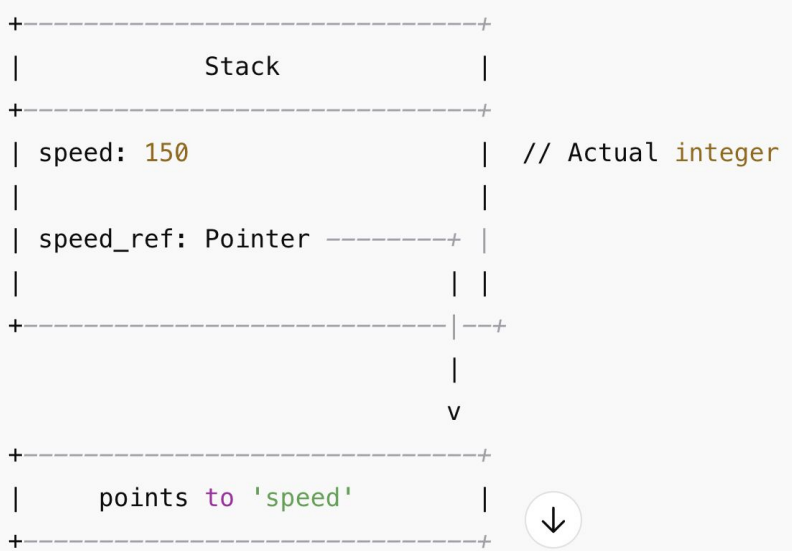
rust

```rust
let mut arr = [10, 20, 30, 40];
let slice = &mut arr[1..3];   // Mutable slice of arr

slice[0] = 99;     // ✅ Modifies arr[1]
```

Memory Diagram

✅ Best Memory Diagram

pgsql



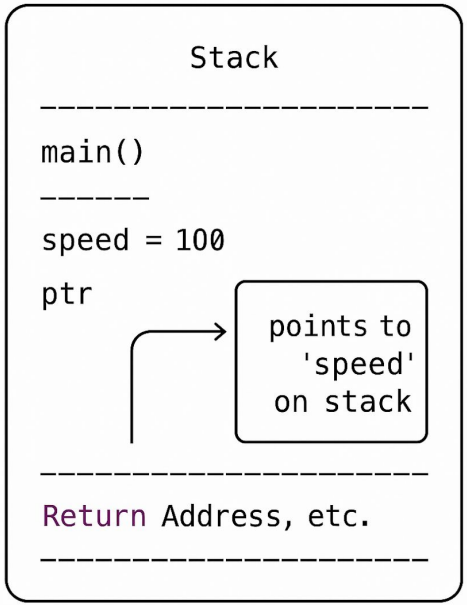
Memory Diagram 0

✅ Rust Code (Raw Pointer Example)

rust

```rust
fn main() {
    let speed = 100;                    // stack allocation
    let ptr: *const i32 = &speed;       // raw pointer to speed

    unsafe {
        println!("Value via ptr: {}", *ptr);
    }
}
```

🦀 Rust Raw Pointer – Memory Diagram



Coding Mistakes

```
Box             ::  new
String          ::  string
                    push_str
```

Memory Diagram 1

✅ Rust Code (Stack Object)

rust

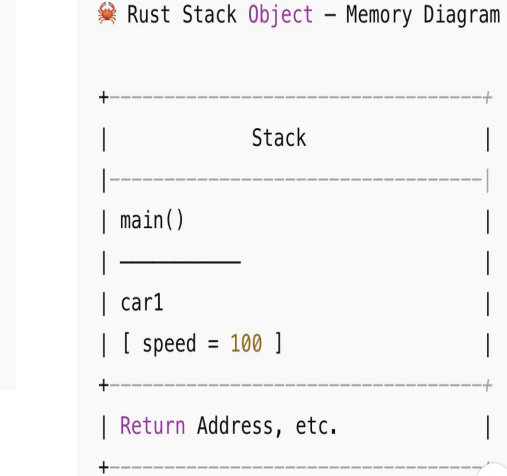
```rust
struct Car {
    speed: u32,
}

fn main() {
    let car1 = Car { speed: 100 }; // stack allocation
    println!("Speed: {}", car1.speed);
}
```

🧠 Memory Diagram – Rust Stack Object

pgsql

Copy

🦀 Rust Stack Object – Memory Diagram

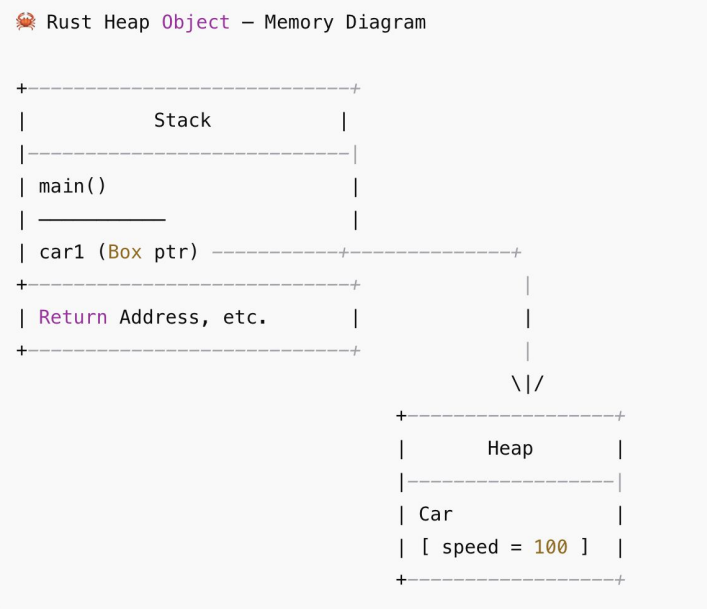


Memory Diagram 1

✅ Rust Code (Heap Allocation)

rust

```rust
struct Car {
    speed: u32,
}

fn main() {
    let car1 = Box::new(Car { speed: 100 }); // heap allocation
    println!("Speed: {}", car1.speed);
}
```



Memory Diagram 2

```rust
fn foo(x: i32) {
    let double = x * 2;
    println!("Double: {}", double);
}

fn main() {
    let num = 10;
    foo(num); // function call
}
```

```
🦀 Rust Function Call — Memory Diagram

+-------------------------------------------------+
|                    Stack                        |
|-------------------------------------------------|
| foo() Frame                                     |
| ————————————                                    |
| x = 10                                          |
| double = 20                                     |
+-------------------------------------------------+
| main() Frame                                    |
| ————————————                                    |
| num = 10                                        |
| return address to resume after foo()            |
+-------------------------------------------------+
| Lower Stack (OS, env, etc.)                     |
+-------------------------------------------------+
```

Memory Diagram 2

```
HEAP:
+--------------------+
| "coderrange"       |   <- vec[0]
+--------------------+

STACK:
vec (pointer, len, cap)
+-----------+
| vec ptr   | ---> points to HEAP
| len       |
| capacity  |
+-----------+

ref_to_first (borrows from HEAP)
+----------------------+
| &vec[0]              |
| Points to HEAP data  |
+----------------------+

Lifetime ties:
ref_to_first cannot outlive vec
```

Memory Diagram 2

```
HEAP:
+--------------+
| "rustacean"  |  <- Stored in HEAP (owned by String s)
+--------------+

STACK:
s (String): owns pointer to heap

r (&String): immutable borrow, points to heap data
Lifetime 'a: r cannot outlive s
```

Memory Diagram 2

✅ Quick Code Summary

Element	Memory Location	Role
`s` (String)	Stack (pointer + len + cap) + Heap	Owns heap data (`"rustacean"`)
`r` (`&String`)	Stack	Borrowed reference, points to heap
Lifetime	Logical rule	Ensures r doesn't outlive s

Memory Diagram 2

```rust
fn main() {
    let s = String::from("rustacean");

    let r = borrow_string(&s);

    println!("{}", r);  // ✅ Safe: r cannot outlive s
}
```

Memory Diagram 2

✅ **Return Ownership:**

```rust
fn get_ref() -> String {
    let s = String::from("coderrange");
    s    // Returning ownership
}
```

Usage:

```rust
fn main() {
    let result = get_ref();
    println!("{}", result);   // Safe: result now owns the String
}
```

Memory Diagram 2

```rust
fn get_ref<'a>(input: &'a String) -> &'a String {
    input    // Borrowing from input safely
}
```

Usage:

`&'a`

rust

```rust
fn main() {
    let s = String::from("coderrange");
    let r = get_ref(&s);

    println!("{}", r);  // ✅ Safe: r cannot outlive s
}
```

Memory Diagram 2

&’a String

&’a Str

implicit lifetimes,

```
fn news(args: &String) -> &String {
    args
}
```

```
fn news<'a>(args: &'a String) -> &'a String { ... }
```

Memory Diagram 3

✅ `{}` vs `{:?}` in Rust – Internally Handled by Traits

Syntax	Trait Used	Purpose
`{}`	`std::fmt::Display`	Human-readable output
`{:?}`	`std::fmt::Debug`	Developer/debug output

Memory Diagram 3

```rust
#[derive(Debug, Clone, PartialEq)]
struct User {
    name: String,
    age: u32,
}
```

📦1️⃣ `Debug` – Auto-Generated Example:

Rust creates an implementation that enables formatted printing using `{:?}`.

```rust
impl std::fmt::Debug for User {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        f.debug_struct("User")
            .field("name", &self.name)
            .field("age", &self.age)
            .finish()
    }
}
```

Memory Diagram 3

```rust
println!("{:?}", my_data);
```

➡️ The compiler looks for an implementation of the `Debug` trait:

```rust
impl std::fmt::Debug for MyType {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        f.debug_struct("MyType")
         .field("some_field", &self.some_field)
         .finish()
    }
}
```

Memory Diagram 3

Method	Behavior
`.unwrap()`	Panic if None/Err
`.expect()`	Panic with custom message
`.unwrap_or()`	Use fallback value if None/Err
`.unwrap_or_else()`	Use closure to handle error

Memory Diagram 3

📦 `.unwrap()` – Source Code in `Option<T>`

From Rust standard library:

rust

```rust
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

Memory Diagram 3

🧠 C++ Function Arguments – Memory Diagram

Assume:

```cpp
void example(int x, int& y, int* z);
```

Then we call:

```cpp
int a = 10, b = 20, c = 30;
example(a, b, &c);
```

📦 Stack Memory Layout During Function Call

```pgsql
+-----------------------------------+
|      Stack Frame: example()       |
|-----------------------------------|
|  Return Address                   |
|-----------------------------------|
|  x = 10                           | ← Pass-by-Value (copy of a)
|-----------------------------------|
|  y → refers to b                  | ← Pass-by-Reference (alias)
|-----------------------------------|
|  z → points to &c                 | ← Pass-by-Pointer (explicit addr)
+-----------------------------------+
```

Memory Diagram 4

Pointer

📦 Diagram:

pgsql

```
+-------------------------+
|      Stack Memory       |
|-------------------------|
|  x = 10                 |  ← Variable (int)
|-------------------------|
|  ptr = 0xABF4           |  ← Pointer storing address of x
+-------------------------+

            |
            |  ptr points to →
            ▼

+-------------------------+
|   Address: 0xABF4       |
|-------------------------|
|  Value: 10              |  ← Actual value stored in memory
+-------------------------+
```

🧠 C++ Reference Variable – Memory Diagram

🎯 Scenario:

You have an `int x = 10;` and `int& ref = x;`

📦 Memory Diagram:

pgsql

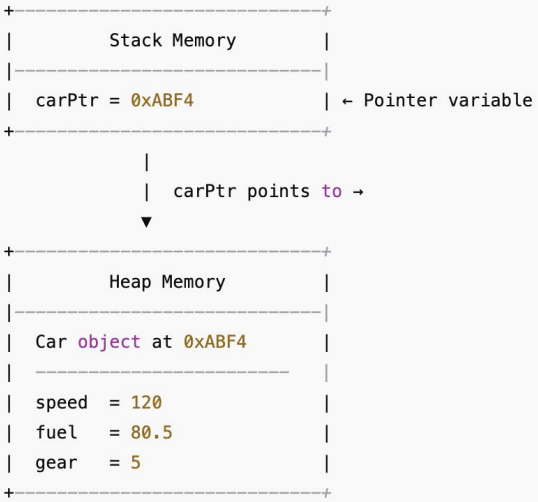
```
+-------------------------+
|      Stack Memory       |
|-------------------------|
|  x = 10                 |  ← Original variable
|-------------------------|
|  ref ↔ x                |  ← Reference (alias to x)
+-------------------------+
```

Memory Diagram 5

🧠 Object with variables– Memory Diagram

📦 Memory Layout

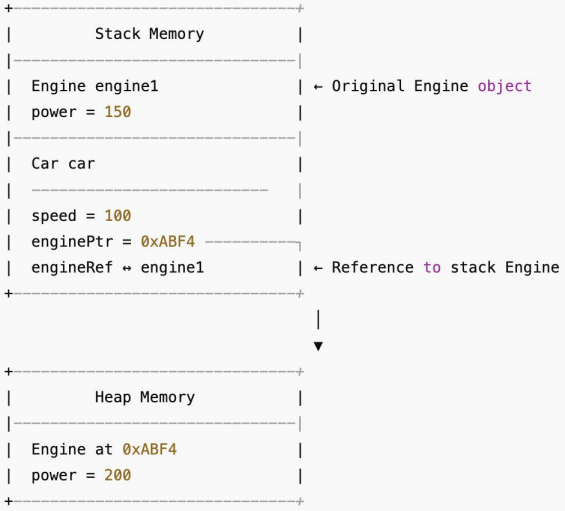
pgsql



🧠 C++ Class with Int, Pointer, and Reference – Memory Diagram

📦 Full Memory Diagram

pgsql



Memory Diagram 6

📦 Shallow Copy – Memory Diagram

Let's assume:

```cpp
class Car {
public:
    int speed;
    int* fuel;
};
```

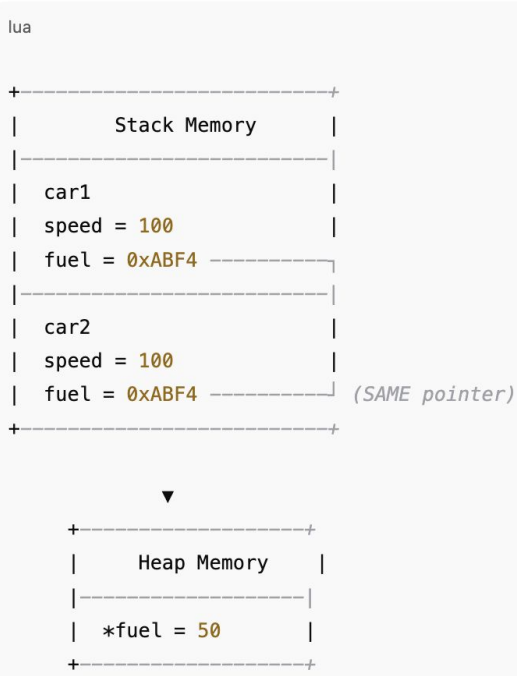
We do:

```cpp
Car car1;
car1.speed = 100;
car1.fuel = new int(50);

Car car2 = car1;  // shallow copy
```

Saved memory full

🧠 Memory Layout:

lua



Memory Diagram 7

🧠 C++ Deep Copy – Memory Diagram

🎯 Scenario:

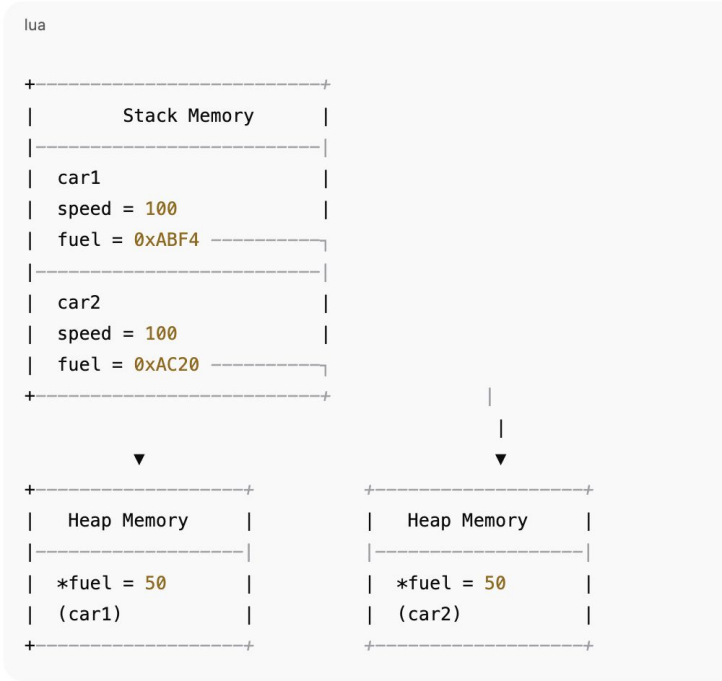
You have a class like:

```cpp
class Car {
public:
    int speed;
    int* fuel;
};
```

You want to **deep copy** `car1` to `car2`, so that:

- A new memory location is created for `fuel`.
- Both objects are fully independent.

📦 Deep Copy – Memory Layout



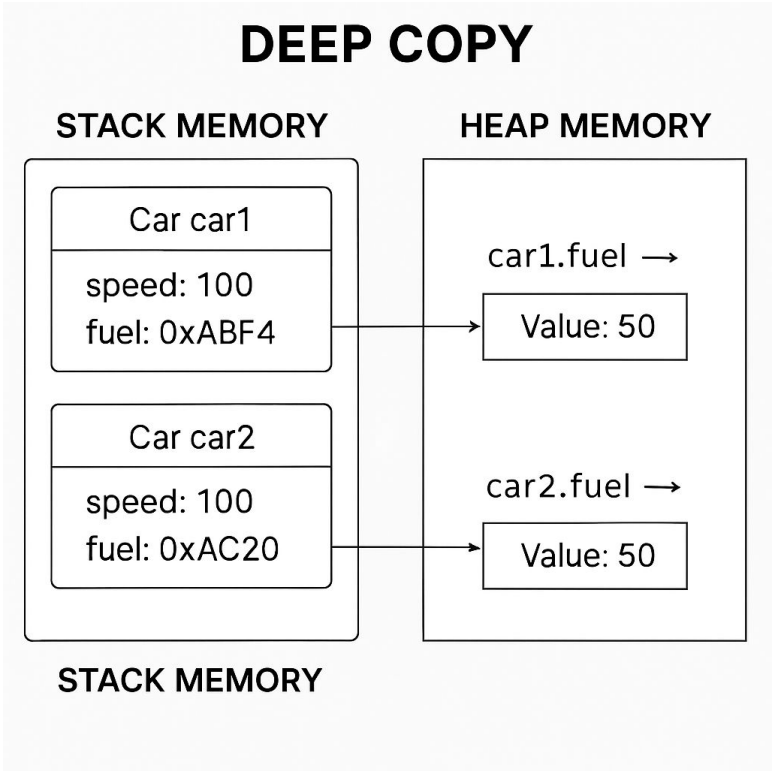
Memory Diagram 8

📦 A. Default `=` Operator (Shallow Copy) – Memory Diagram

lua Copy

```
+-----------------------------+
|        Stack Memory         |
|-----------------------------|
|  car1                       |
|  speed = 100                |
|  fuel = 0xABF4 ------------┐|
|-----------------------------|
|  car2                       |
|  speed = 100                |
|  fuel = 0xABF4 ------------┘| ← same address (shared)
+-----------------------------+

          ▼

      +--------------------+
      |   Heap Memory      |
      |--------------------|
      |  *fuel = 50        |
      +--------------------+
```

❗ **Risk: Double delete, shared mutation, crash**



Memory Diagram 9

🎯 Object Memory Layout on Stack

pgsql

```
                STACK MEMORY (No vptr, No dynamic dispatch)

+-----------------------------------------------+
| Dog d;                                        |
|-----------------------------------------------|
| Base class part → Animal:                     |
|   - speak() function (statically bound)       |
|-----------------------------------------------|
| Derived class part → Dog:                     |
|   - speak() function (separate, hides base)   |
+-----------------------------------------------+

+-----------------------------------------------+
| Animal* ptr = &d;                             |
|-----------------------------------------------|
| Points to base part of Dog object             |
| ptr->speak() → Calls Animal::speak() ONLY     |
+-----------------------------------------------+

🚫 No vptr
🚫 No vtable
🚫 No runtime dispatch
```

📦 Memory Layout (NO virtual, STATIC binding)

pgsql

```
                    STACK MEMORY
+-----------------------------------------------+
| Dog d;                                        |
|-----------------------------------------------|
| Base class: Animal                            |
|   - age          = 5                          |
|                                               |
| Derived class: Dog                            |
|   - tailLength   = 20                         |
+-----------------------------------------------+

+-----------------------------------------------+
| Animal* ptr = &d;                             |
|-----------------------------------------------|
| Points to base part of Dog                    |
| ptr->speak() → Calls Animal::speak()          |
+-----------------------------------------------+

🚫 No vptr
🚫 No vtable
✅ Static memory layout
```

Memory Diagram 10

📉 Memory Layout Diagram — No `virtual`

✅ STACK MEMORY

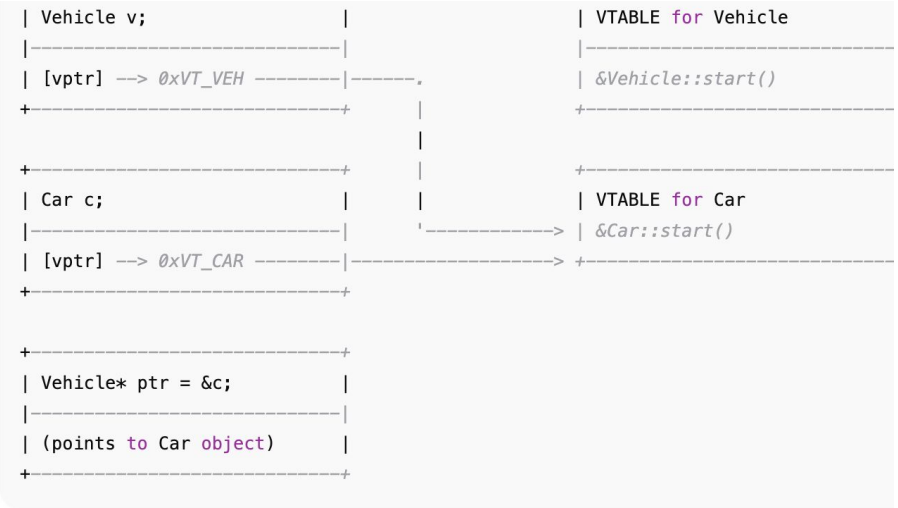
pgsql

Copy Edit

```
                STACK MEMORY                          NO VTABLE EXISTS
+-------------------------------+                +-------------------------------+
| Vehicle v;                    |                | 🚫 No vtable generated        |
|-------------------------------|                | 🚫 No vptr in object          |
| start() -> Vehicle::start() |                | 🚫 No dynamic dispatch        |
+-------------------------------+                +-------------------------------+

+-------------------------------+
| Car c;                        |
|-------------------------------|
| start() -> Car::start()     |
+-------------------------------+

+-------------------------------+
| Vehicle* ptr = &c;            |
|-------------------------------|
| Points to Car object          |
| Call ptr->start()             |
| 👉 But it calls Vehicle::start() |
+-------------------------------+
```

Memory Diagram 11

🧠 Correct Diagram — Stack Object & Vtable with Overriding

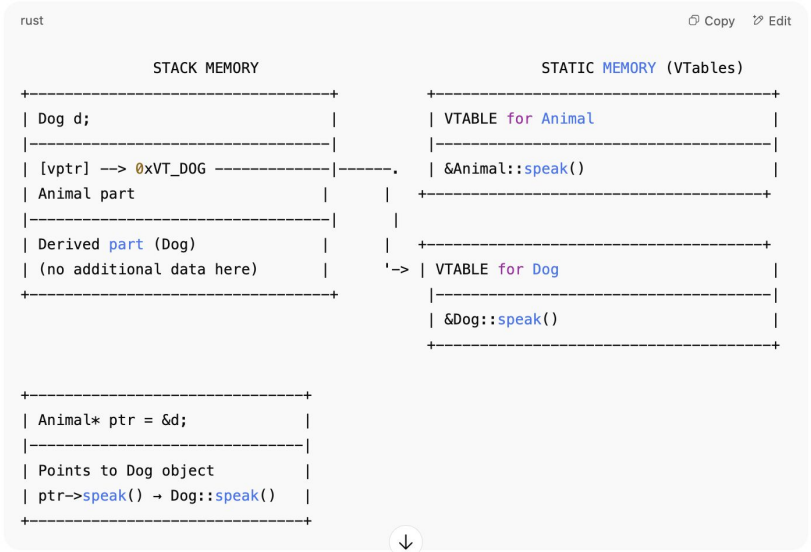


Memory Diagram 11

🧠 Correct Diagram — Stack Object & Vtable with Overriding

✅ INHERITANCE OBJECT DIAGRAM (STACK + VTABLE)

rust Copy Edit



Memory Diagram 12

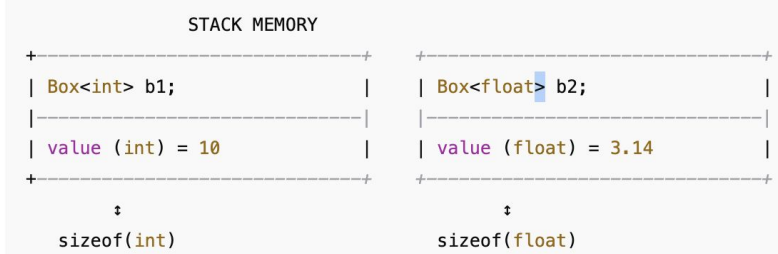
📌 Code

cpp

```cpp
Box<int> b1;
Box<float> b2;
```

🎯 Memory Layout

pgsql



✅ No vptr
✅ No shared code
✅ No dynamic dispatch

🧪 Template Function Example

cpp

```cpp
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

Memory Diagram 12

🧪 Template Function Example

cpp

```cpp
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

💾 STATIC + STACK MEMORY LAYOUT

pgsql

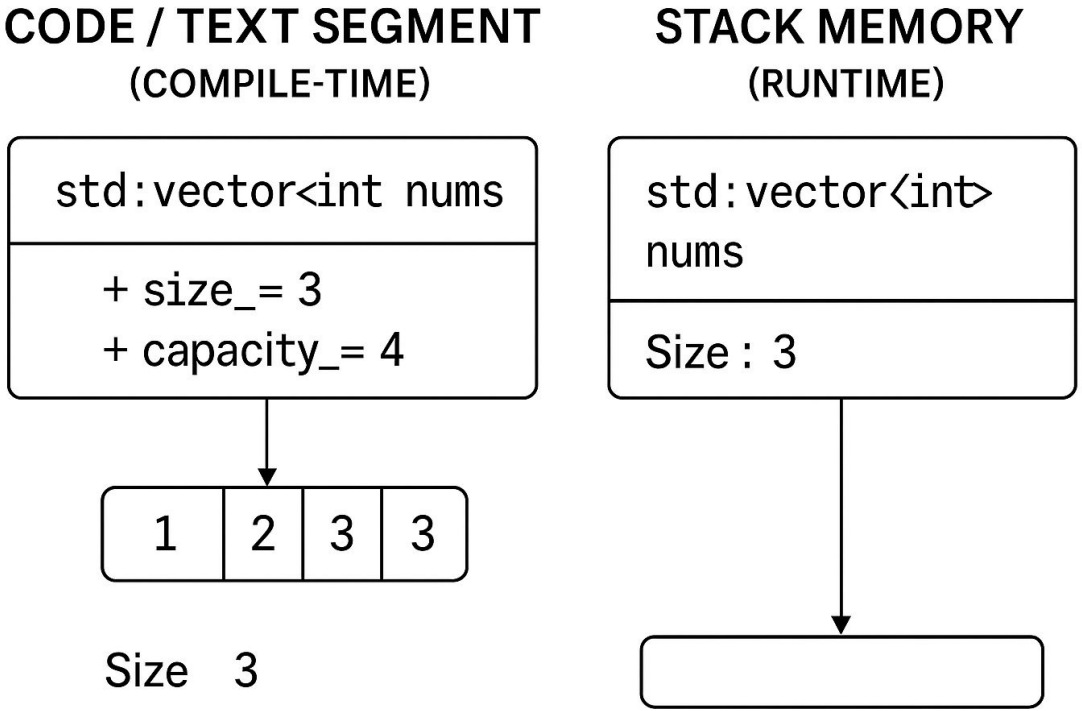
```
                CODE / TEXT SEGMENT (COMPILE-TIME)
+-------------------------------------------------------+
| Function: add_int(int, int)                           |
|  Address: 0x1000                                      |
|  Code: a + b (int addition logic)                     |
+-------------------------------------------------------+
| Function: add_float(float, float)                     |
|  Address: 0x1050                                      |
|  Code: a + b (float addition logic)                   |
+-------------------------------------------------------+

                    STACK MEMORY (RUNTIME)
+-------------------------------+       +-------------------------------+
| int x = add(10, 20);          |       | float y = add(1.5, 2.5);      |
|-------------------------------|       |-------------------------------|
| Call → add_int                |       | Call → add_float              |
| Parameters:                   |       | Parameters:                   |
|   a: 10                       |       |   a: 1.5f                     |
|   b: 20                       |       |   b: 2.5f                     |
| Return → 30                   |       | Return → 4.0f                 |
+-------------------------------+       +-------------------------------+
```

Reference

https://chatgpt.com/share/686bce8a-e55c-8002-b96d-a0b3eaf1fe08

https://chatgpt.com/share/686bcea6-a310-8002-933a-f2489dd24615

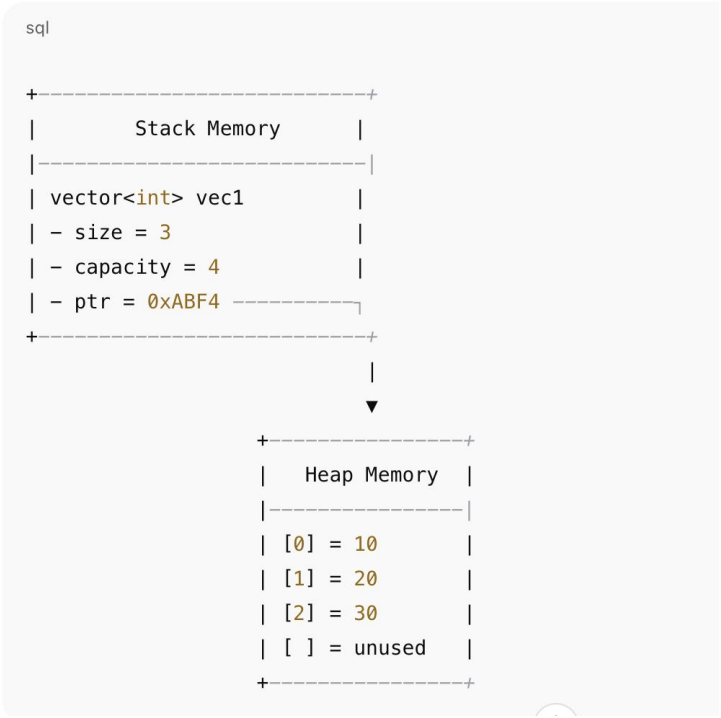
Memory Diagram 13



Memory Diagram 13

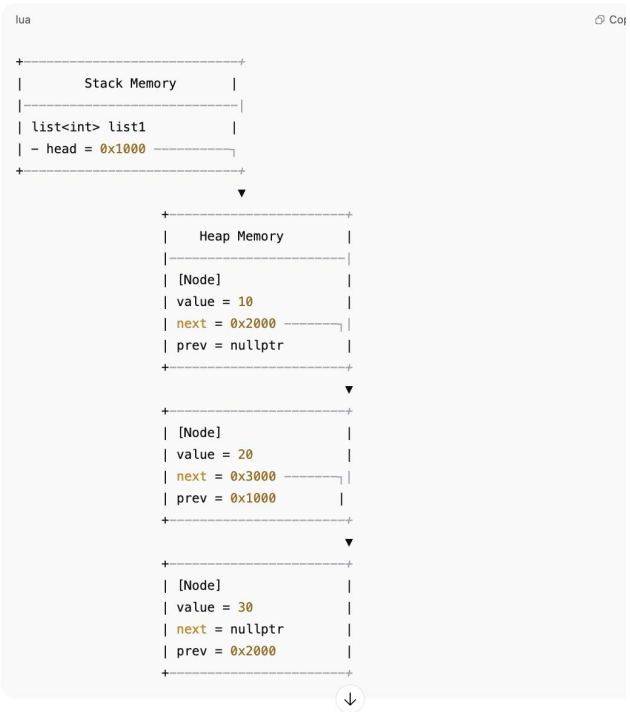
📦 A. `std::vector` – Memory Diagram

sql

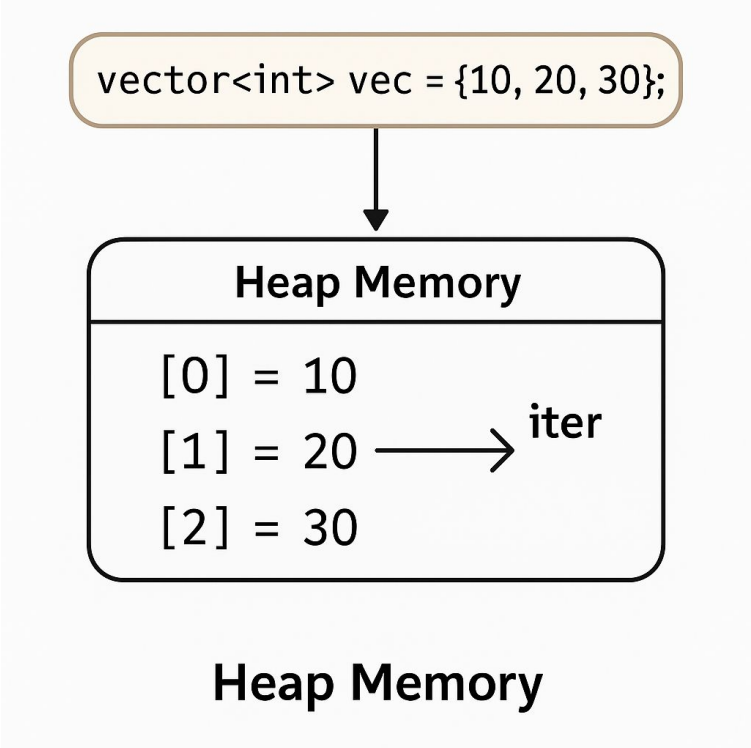


📦 B. `std::list` – Memory Diagram (Doubly Linked List)

lua



Memory Diagram 13



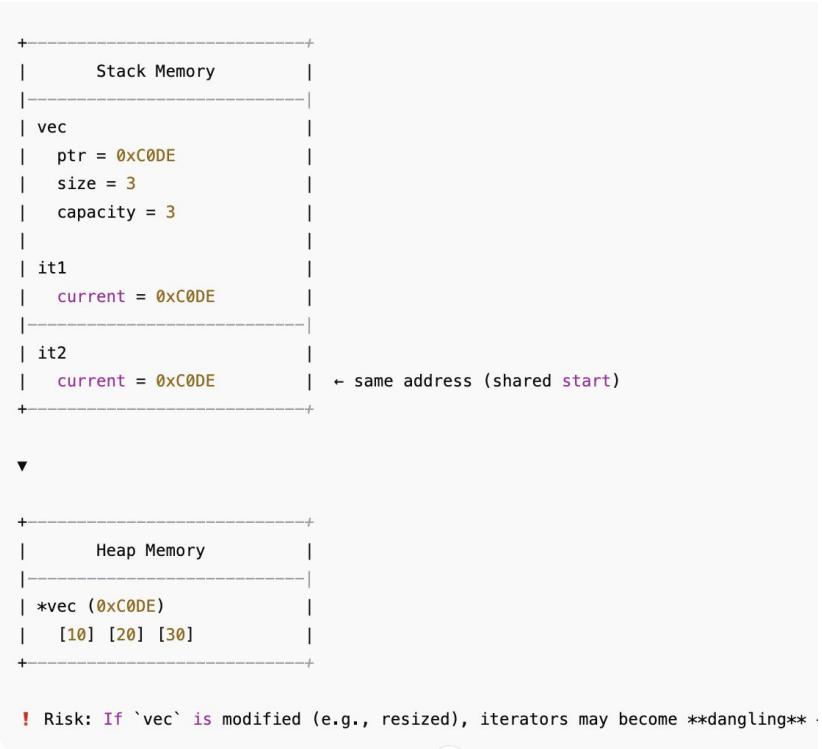
📦 C. `std::map` – Memory Diagram (Red-Black Tree)

sql

Copy Edit

```
+-------------------------------+
|        Stack Memory           |
|-------------------------------|
| map<int, string> m1           |
| - root = 0x1500 ------------- |
+-------------------------------+
                ▼
+-------------------------------+
|        Heap Memory            |
|-------------------------------|
| [Node]                        |
| key = 2                       |
| value = "B"                   |
| left = 0x1400 -------------   |
| right = 0x1600 ------------   |
| parent = nullptr              |
+-------------------------------+
                ▼
+-------------------------------+   +-------------------------------+
| [Node]                        |   | [Node]                        |
| key = 1                       |   | key = 3                       |
| value = "A"                   |   | value = "C"                   |
| left = nullptr                |   | right = nullptr               |
| right = nullptr               |   | left = nullptr                |
| parent = 0x1500               |   | parent = 0x1500               |
+-------------------------------+   +-------------------------------+
```

Memory Diagram 14 - Iterator



Memory Diagram 15

🧠 1. `std::unique_ptr<T>` — Owns Object Exclusively

cpp

```cpp
std::unique_ptr<Car> car1 = std::make_unique<Car>(100);
```

pgsql

```
+-----------------------------+
|      Stack Memory           |
|-----------------------------|
| car1                        |
|   ptr = 0xABF4              |
+-----------------------------+

▼

+-----------------------------+
|      Heap Memory            |
|-----------------------------|
| *Car (0xABF4)               |
|   speed = 100               |
+-----------------------------+
```

✅ Ownership: Single owner (car1)
❌ No copies allowed

🔁 2. `std::shared_ptr<T>` — Shared Ownership (Ref Counted)

cpp

```cpp
std::shared_ptr<Car> car1 = std::make_shared<Car>(100);
std::shared_ptr<Car> car2 = car1;
```

sql

```
+-----------------------------+
|      Stack Memory           |
|-----------------------------|
| car1                        |
|   ptr = 0xBEEF              |
|-----------------------------|
| car2                        |
|   ptr = 0xBEEF              | ← shared
+-----------------------------+

▼

+-----------------------------+
|      Heap Memory            |
|-----------------------------|
| *Car (0xBEEF)               |
|   speed = 100               |
|                             |
| Ref Count = 2               |
+-----------------------------+
```

✅ Shared ownership
🧱 Automatically deleted when Ref Count = ↓

Memory Diagram 15

👀 3. `std::weak_ptr<T>` — Non-owning Observer

cpp

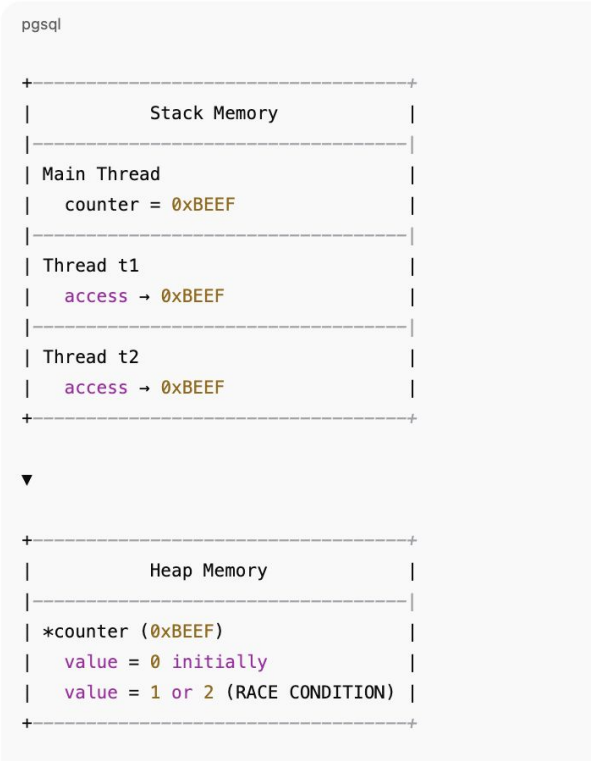
```cpp
std::shared_ptr<Car> car1 = std::make_shared<Car>(100);
std::weak_ptr<Car> weak_car = car1;
```

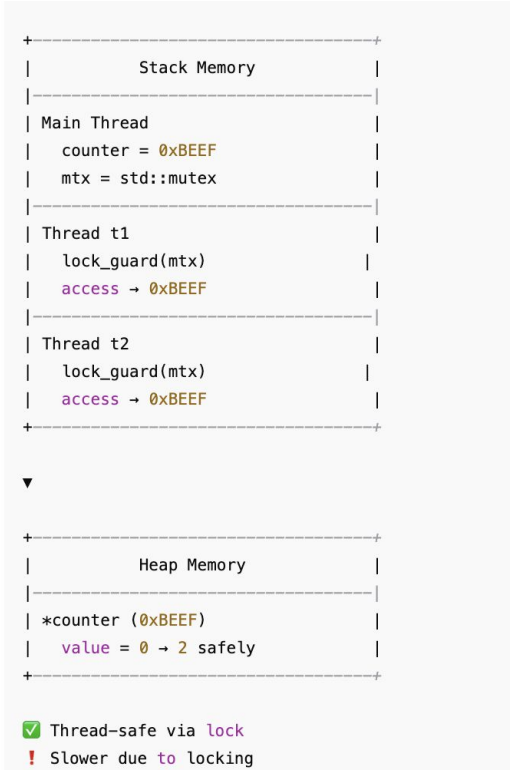
pgsql

```
+-----------------------------+
|      Stack Memory           |
|-----------------------------|
| car1                        |
|   ptr = 0xDEAD              |
|-----------------------------|
| weak_car                    |
|   ptr = 0xDEAD (weak)       | ← no ref count
+-----------------------------+

▼

+-----------------------------+
|      Heap Memory            |
|-----------------------------|
| *Car (0xDEAD)               |
|   speed = 100               |
|                             |
| Ref Count = 1               |
+-----------------------------+
```

Memory Diagram 16

pgsql





Memory Diagram 17

✅ Correct Singleton Memory Diagram (Single Thread)

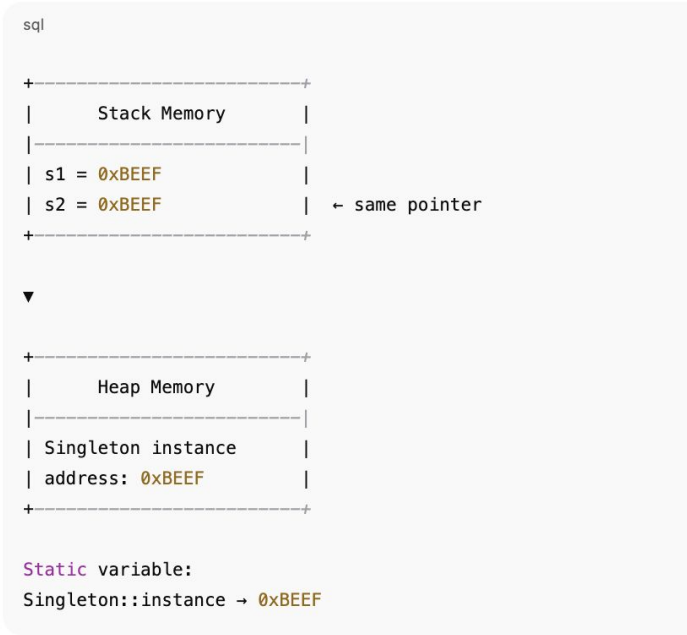
Code:

```cpp
class Singleton {
public:
    static Singleton* getInstance() {
        if (instance == nullptr)
            instance = new Singleton();
        return instance;
    }

private:
    static Singleton* instance;
    Singleton() {}  // private constructor
};

// Define static member
Singleton* Singleton::instance = nullptr;
```

When you call:

```cpp
Singleton* s1 = Singleton::getInstance();
Singleton* s2 = Singleton::getInstance();
```

✅ Simple Stack + Heap Diagram:



Memory Diagram 18

✅ Virtual Inheritance — Memory Diagram

sql

```
+-----------------------------+
|          Stack              |
|-----------------------------|
| obj (of class D)            |
|-----------------------------|
| B → [vptr to A]             |
| C → [vptr to A]             |
|       ↓ shared              |
| A::x = 10                   |
+-----------------------------+

Memory layout:
D
├── B ─┐
|      └──┐
└── C ────┴──> A::x (only one copy)

✅ One shared A instance → unambiguous
```

Memory Diagram 18

✅ When VTables Are Used

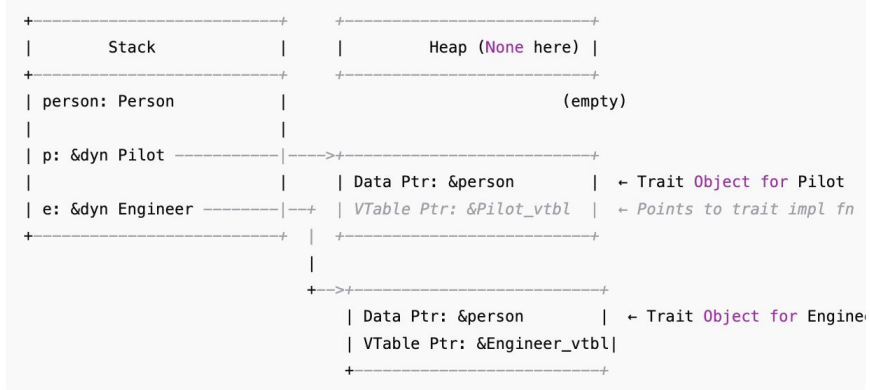
Usage	VTable Exists?	Dispatch Type
`&dyn Trait`	✅ Yes	Dynamic
`Box<dyn Trait>`	✅ Yes	Dynamic
`Arc<dyn Trait>`	✅ Yes	Dynamic
`&impl Trait`	❌ No	Static
`struct T; impl Trait for T`	❌ No	Static

Memory Diagram 18

```rust
trait Pilot {
    fn fly(&self);
}

trait Engineer {
    fn work(&self);
}

struct Person;

impl Pilot for Person {
    fn fly(&self) {
        println!("Pilot flying");
    }
}

impl Engineer for Person {
    fn work(&self) {
        println!("Engineer working");
    }
}

fn main() {
    let person = Person;
    let p: &dyn Pilot = &person;      // Trait object
    let e: &dyn Engineer = &person;   // Trait object
}
```

🧠 **Memory Diagram (Simplified)**

pgsql — Copy — Edit



Best Real Time Formula

📊 Refined Formula for Rust Development Thinking:

#	Concept	Is it Core to Rust?	Notes
1	**Memory Allocation (Stack / Heap)**	✅ Core	Ownership, borrowing, move semantics are directly tied to memory model.
2	**Memory Initialization**	✅ Core	Rust enforces mandatory initialization before use.
3	**Condition Handling (`if`, `match`)**	✅ Core	Pattern matching is a powerful feature in Rust.
4	**Communication (Function Calls, Return)**	✅ Core	Functions, methods, trait methods are essential, with explicit data movement (ownership).
5	**Loops (`for`, `while`)**	✅ Core	Rust also promotes iterator-based loops (preferred idiom).
6	**Exception Handling**	⚠️ Partial	Rust uses `Result` and `Option` for error handling instead of exceptions. `panic!` is for unrecoverable errors.
7	**Business Logic**	✅ Domain-specific	Core to solving actual problems. Rust doesn't enforce domain patterns but is capable.
8	**Transformation (Data Processing)**	✅ Strong	Iterators, functional programming patterns (map, filter, fold) make Rust great for transformation logic.
9	**Resources (Files, DB, Threads, OS)**	✅ Strong	`std::fs`, `tokio`, `threads`, `async` handle resource interaction safely.
10	**Domain Knowledge**	✅ Important	Language-agnostic but essential for applying Rust to real-world problems.
11	**Functionality (Deliverables)**	✅ Result	This is your outcome. All above points lead to implementing meaningful features.

Best Real Time Formula

✅ Suggested Rust Mindmap / Formula:

plaintext

1. Ownership / Borrowing / Lifetimes
2. Memory Management (Stack / Heap / Move / Copy)
3. Pattern Matching / Enums
4. Error Handling (Result, Option)
5. Traits / Trait Objects (Abstraction)
6. Concurrency / Async (Safe Parallelism)
7. Functional Data Transformations (Iterators)
8. Resource Safety (Files, Threads, DB)
9. Business Logic & Domain Modeling
10. Deliver Features Safely & Performantly