

# **Memory Management in Rust, Java, and C++**

**Submitted by**

**Shiva Gowtham Kumar Vidiyala**

**Instructor Name : Jay Thom**

**Advanced Programming Language**

## Rust Program

```
fn square_in_place(x: &mut i32) {  
    *x = *x * *x;  
}  
  
fn main() {  
    let mut num = 5;  
    // Pass mutable reference → borrow, no copy  
    square_in_place(&mut num);  
  
    println!("Squared = {}", num); // 25  
} // memory automatically freed here
```

## Execution

main.rs	Output
<pre>1- fn square_in_place(x: &amp;mut i32) { 2   *x = *x * *x; 3 } 4 5- fn main() { 6   let mut num = 5; 7   // Pass mutable reference → borrow, no copy 8   square_in_place(&amp;mut num); 9 10  println!("Squared = {}", num); // 25 11 } // memory automatically freed here</pre>	<pre>Squared = 25  === Code Execution Successful ===</pre>

Rust compiler enforces that only one mutable borrow exists at a time. No dangling pointers allowed.

## Java Program

```
class Box {
    int value;
    Box(int v) { value = v; }
}

public class MemoryTest {
    static void square(Box b) {
        b.value = b.value * b.value; // object modified via reference
    }

    public static void main(String[] args) {
        Box num = new Box(5);
        square(num);
        System.out.println("Squared = " + num.value); // 25
    }
}
```

### Execution



```
1 class Box {
2     int value;
3     Box(int v) { value = v; }
4 }
5
6 public class MemoryTest {
7     static void square(Box b) {
8         b.value = b.value * b.value; // object modified via reference
9     }
10
11     public static void main(String[] args) {
12         Box num = new Box(5);
13         square(num);
14         System.out.println("Squared = " + num.value); // 25
15     }
16 }
17
```

Squared = 25

In Java, all objects are references. When you pass num into square, the method gets a reference (pointer) to the same object. GC frees memory when Box is no longer reachable.

## C++ Program

```

#include <iostream>
using namespace std;

void square_in_place(int& x) { // pass by reference
    x = x * x;
}

int main() {
    int num = 5;
    square_in_place(num);
    cout << "Squared = " << num << endl; // 25
}

```

main.cpp	Output
<pre> 1  #include &lt;iostream&gt; 2  using namespace std; 3 4  void square_in_place(int&amp; x) { // pass by reference 5      x = x * x; 6  } 7 8  int main() { 9      int num = 5; 10     square_in_place(num); 11     cout &lt;&lt; "Squared = " &lt;&lt; num &lt;&lt; endl; // 25 12 } 13 </pre>	<p>Squared = 25</p> <p>=== Code Execution Successful ===</p>

Int & is an alias to the original variable. No copying. Unlike Rust, dangling references are possible if you return refs to locals. Unlike Java, no GC—you must manage heap objects explicitly.

## Analysis of references & memory

Rust:

References (&T, &mut T) are *borrow*s with strict rules.  
Compiler prevents dangling refs or data races.  
Memory automatically freed (RAII) when the owner goes out of scope.

**Java:**

References are opaque handles to heap objects.  
No explicit free—GC reclaims unreachable objects.  
Errors come from *logical leaks* (holding references too long), not  
dangling pointers.

**C++:**

References (&) are just aliases; powerful but dangerous.  
Can dangle if you return a reference to a local variable.  
Memory must be freed manually (unless using smart pointers/RAII).

**GITHUB LINK**

[https://github.com/gowthamvidi/MSCS632\\_Assignment.git](https://github.com/gowthamvidi/MSCS632_Assignment.git)