C++

- C++ Concepts

C++ Concepts

Resource Acquisition Is Initialization

> RAII is a design pattern where resource lifetime is bound to object lifetime. When the object is created, it acquires the resource. When the object is destroyed, it releases the resource automatically via the destructor.

Resources Include :

- heap memory
- file handles
- sockets
- mutex locks
- db connections
- GPU buffers
- threads

File I/O RAII Example

```
#include <iostream>
#include <fstream>
#include <string>

// ifstream abides by RAII design pattern
// Resource (file) is acquired during scope
// Destructor naturally runs when escaping scope
int main() {
  std::ifstream file{"file.txt"};
  if (file.isOpen()) {
    std::cout << "file is open" << std::endl;
  } else {
    std::cout << "file is close" << std::endl;
  }
}
```

Mutex RAII Example

```
std::mutex m;

// No RAII object.
// if some condition is true, m.unlock never runs -> deadlock
void foo() {
  m.lock();

  if (some condition) {
    return;
  }

  m.unlock();
}

// With RAII object
// destructor runs when leaving scope
void bar() {
  std::lock_guard<std::mutex> lock(m);

  if (some condition) return;
}
```

lvalue vs rvalue

lvalue - An expression that refers to a persistent object with an identifiable memory location.

```
// persists
// has address in memory
int x = 10;
```

value - A temporary expression that does not persist beyond the full expression.

```
// temporary
10;
std::string("hello");
foo();
```

Rule of 5

- Destructor
- Copy constructor
- Copy assignment
- Move constructor
- Move assignment

> We use the rule of 5 : when a class owns a resource, the default copy behavior is unsafe, so we **must explicitly define copy and move semantics**

```
class Buffer {
  // std::vector already implements RAII
  // RAII container
  std::vector<int> data;
};
```

> Raw pointers are mainly used for non-owning references

Copy or Move

```
// double delete error
std::unique_ptr<int> p1 = std::make_unique<int>(5);
std::unique_ptr<int> p2 = p1;

// if this is allowed, it could cause double frees/memory corruption/unexpected behavior
```

1. copy constructor tries to run because p1 is an lvalue
2. `std::unique_ptr` deletes its copy constructor
3. compilation fails

You have to move ownership

```
// unique_ptr is move only
std::unique_ptr<int> p2 = std::move(p1);
```

Move Semantics

```
// function f takes ownership
// simplest - forces move
void f(std::unique_ptr<int> p);
  // but not
  std::unique_ptr<int> p = std::make_unique<inst>(5);
  f(p);
  // it tries to copy p but there is no copy constructor

// function g is only BORROWING access
// cannot modify pointer
void g(const std::unique_ptr<int>& p);

// function h borrows mutable
void h(std::unique_ptr<int>& p);

// function i expects something moveable
// Caller must pass temporary or std::move
void i(std::unique_ptr<int>&& p);
```