Name :- S Venkata Surya  
 Emp Id :- T1934

* How does a range-based for loop work under the hood? What requirements must a type satisfy to be used in a range-based for loop?

Ans)

Ranged based for loop is frequently used in vectors and in arrays which is help to shrink and extens the size which we don't know the size.

Syntax:-

for(datatype varablename : name){}

the above thing must be statisfied to use the range-based for loop

-------------------------------------------------------------------------------------------------------------------

* What is the use of auto in modern C++?

Ans)

By using auto we can declare any type which is int, double and string the compiler will take care that belongs to which type is.

Syntx: auto varname = value;

------------------------------------------------------------------------------------------------

* Differentiate between std::unique\_ptr and std::shared\_ptr?

Ans)

The std::unique\_ptr, std::shared\_ptr it is a type of smart pointer which is used to mange the memory leakes and prevent the dangling pointer error (which is represents the null pointer)  
  
std::unique\_ptr:

which is used to give the ownership exclusive to one and we can move the ownership to another by using move and the previous one will be null and the owernship is accessed by new one which we move.

std::shared\_ptr:

which is used to give the ownership to all the members in the class and their count is calculated how many can have the ownership by using the **usecount()** and the last one will make this zero.

----------------------------------------------------------------------------------------------

* What is the difference between capturing variables by value [=] and by reference [&] in a lambda? What happens if you capture a local variable by reference and the lambda outlives the variable’s scope?

Ans)

Lambda is an anonymous function which there is no function name and variables by value [=] it first looks to the srrounding signs and it take the values of that sign.  
where by reference [&] we can pass that reference of that value if we change the value then it replace the before value, because it takes the address of the previous class and replacing it.If you capture a local variable by reference it take the address and we can change anything it affacts the value.

-----------------------------------------------------------------------------------------------

* When is a copy constructor called? What is a move constructor and when is it triggered?

Ans)

copy constructor:

Copy constructor is called when we assign one object to the another object this time it calls the copy constructor.

Ex: s1 = s2;

move constructor:

Move constructor is used to move the owernership to other object and the make null to before one and we can done with **std::move()** a using the && in the function parametr.

-------------------------------------------------------------------------------------------------------------------

**📱 Phone Contact Manager – Application Assignment**

**✅ Concepts to Practice:**

* Constructors (default, parameterized, copy, move)
* Destructor
* Smart Pointers (**unique\_ptr**, **shared\_ptr**)
* Lambda Functions
* Inline Functions

❌ No use of **std::vector** or sorting mechanisms  
💡 Focus is on core OOP and modern C++ features in an embedded-style phone app simulation

**💡 Objective:**

To build a simple console-based phone contact manager that allows users to create and manage a small list of contacts using smart pointers, lambdas, and inline functions.

This simulates a basic phone application like you might find in embedded systems or automotive infotainment systems.

**🧱 Class Design**

**1. Contact Class**

Represents a single contact with name and phone number.

**Attributes:**

* **std::string name**
* **std::string phoneNumber**

**Methods:**

* Default constructor
* Parameterized constructor
* Copy constructor
* Move constructor
* Destructor
* **void display() const**

**2. PhoneBook Class**

Represents a simple phone book with a fixed number of contacts (e.g., max 3).

**Attributes:**

* **std::unique\_ptr<Contact> contact1**
* **std::unique\_ptr<Contact> contact2**
* **std::unique\_ptr<Contact> contact3**

**Methods:**

* Constructor
* Destructor
* **void addContact(int position, const Contact& contact)**
* **void callContact(int position) const**
* **void showAllContacts() const**

**Sample Output:-**[LOG] Starting Phone App...

Calling: John Doe at +1234567890

Calling: Jane Smith at +0987654321

Custom Action: Message sent to John Doe

📌 **Submission Checklist**

✅ Use of constructors, destructors, smart pointers, lambda functions, and inline functions

✅ Proper comments explaining key parts

✅ Output screenshot or sample run  
  
  
**📚 Learning Goals**

**By completing this assignment, students will learn:**

* How to define and use constructors and destructors
* Why smart pointers are essential for safe memory management
* How to use lambda expressions for inline logic
* When to use inline functions for utility logic
* How to simulate real-world applications using object-oriented design

Ans)

#include<iostream>

using namespace std;

#include<memory>

#include<string>

inline void sendCustomMessage(const std::string& name) {

std::cout << "Custom Action: Message sent to " << name << std::endl;

}

class Contact {

private:

std::string name;

std::string phoneNumber;

public:

// Default constructor

Contact() : name("Unknow"), phoneNumber("Null") {

std::cout << "Default constructor called\n";

}

// Parameterized constructor

Contact(const std::string& name, const std::string& phone)

: name(name), phoneNumber(phone) {

std::cout << "Parameterized constructor called\n";

}

//Copy constructor

Contact(const Contact& other)

: name(other.name), phoneNumber(other.phoneNumber) {

std::cout << " Copy constructor called\n";

}

// Move constructor

Contact(Contact&& other) noexcept

: name(std::move(other.name)), phoneNumber(std::move(other.phoneNumber)) {

std::cout << " Move constructor called\n";

}

// Destructor

~Contact() {

std::cout << " Destructor called for " << name << "\n";

}

// Display contact details

void display() const {

std::cout << "Calling: " << name << " at " << phoneNumber << std::endl;

}

// for lambda

const std::string& getName() const { return name; }

};

class PhoneBook {

private:

std::unique\_ptr<Contact> contact1;

std::unique\_ptr<Contact> contact2;

std::unique\_ptr<Contact> contact3;

public:

// Constructor

PhoneBook() {

std::cout << " PhoneBook initialized.\n";

}

// Destructor

~PhoneBook() {

std::cout << " PhoneBook destroyed.\n";

}

void addContact(int position, const Contact& contact) {

switch (position) {

case 1: contact1 = std::make\_unique<Contact>(contact); break;

case 2: contact2 = std::make\_unique<Contact>(contact); break;

case 3: contact3 = std::make\_unique<Contact>(contact); break;

default: std::cout << " Invalid position\n"; break;

}

}

// Call a contact

void callContact(int position) const {

const Contact\* ptr = nullptr;

switch (position) {

case 1: ptr = contact1.get(); break;

case 2: ptr = contact2.get(); break;

case 3: ptr = contact3.get(); break;

default: std::cout << " Invalid position\n"; return;

}

if (ptr) ptr->display();

else std::cout << " No contact at position " << position << endl;

}

// Show all contacts

void showAllContacts() const {

std::cout << "\nContact List\n";

if (contact1) contact1->display();

if (contact2) contact2->display();

if (contact3) contact3->display();

}

};

int main() {

std::cout << "[LOG] Starting Phone App...\n";

// Create contacts

Contact john("John Doe", "+1234567890");

Contact jane("Jane Smith", "+0987654321");

// PhoneBook

PhoneBook phoneBook;

phoneBook.addContact(1, john);

phoneBook.addContact(2, jane);

// Call contacts

phoneBook.callContact(1);

phoneBook.callContact(2);

// Lambda function

auto sendMessage = [](const Contact& c) {

sendCustomMessage(c.getName());

};

sendMessage(john);

// Display all

phoneBook.showAllContacts();

return 0;

}

O/p:-

