

OBJECT ORIENTED PROGRAMMING

Solutions - 10 September 2026

Complete C++ solutions based on the 16 questions in the supplied OOP question paper. The programs emphasize dynamic memory, RAII, unique_ptr, shared_ptr, and weak_ptr as requested.

1. Swap using pass by value, reference, and pointer

Solution:

```
#include <iostream>
using namespace std;

void swapByValue(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    cout << "Inside swapByValue: a = " << a << ", b = " << b << '\n';
}

void swapByReference(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void swapByPointer(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;

    cout << "Original: x = " << x << ", y = " << y << '\n';

    swapByValue(x, y);
    cout << "After pass by value: x = " << x << ", y = " << y << '\n';

    swapByReference(x, y);
    cout << "After pass by reference: x = " << x << ", y = " << y << '\n';

    swapByPointer(&x, &y);
    cout << "After pass by pointer: x = " << x << ", y = " << y << '\n';

    return 0;
}
```

Explanation: Pass by value receives copies, so the caller's variables are unchanged. Reference parameters are aliases to the original variables. Pointer parameters receive addresses, and dereferencing them changes the original variables.

2. Dynamic integer array: sum, average, minimum, maximum

Solution:

```
#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;

    if (n <= 0) {
        cout << "n must be positive.\n";
        return 0;
    }

    int* a = new int[n];

    for (int i = 0; i < n; ++i) {
        cout << "Enter element " << i + 1 << ": ";
        cin >> a[i];
    }

    long long sum = 0;
    int minimum = a[0], maximum = a[0];

    for (int i = 0; i < n; ++i) {
        sum += a[i];
        if (a[i] < minimum) minimum = a[i];
        if (a[i] > maximum) maximum = a[i];
    }

    double average = static_cast<double>(sum) / n;

    cout << "Sum = " << sum << '\n';
    cout << "Average = " << average << '\n';
    cout << "Minimum = " << minimum << '\n';
    cout << "Maximum = " << maximum << '\n';

    delete[] a;
    a = nullptr;
    return 0;
}
```

Explanation: The array is allocated with `new int[n]` and released with `delete[]`. The average uses floating-point division. No vector is used.

3. Student records using dynamic memory

Solution:

```
#include <iostream>
#include <string>
using namespace std;

struct Student {
    int rollNo;
    string name;
    double marks;
};

int main() {
    int n;
    cout << "Enter number of students: ";
    cin >> n;

    if (n <= 0) return 0;

    Student* students = new Student[n];

    for (int i = 0; i < n; ++i) {
        cout << "\nStudent " << i + 1 << '\n';
        cout << "Roll number: ";
        cin >> students[i].rollNo;
        cin.ignore();
        cout << "Name: ";
        getline(cin, students[i].name);
        cout << "Marks: ";
        cin >> students[i].marks;
    }

    int best = 0;
    for (int i = 1; i < n; ++i) {
        if (students[i].marks > students[best].marks)
            best = i;
    }

    cout << "\nStudent with highest marks:\n";
    cout << "Roll No: " << students[best].rollNo << '\n';
    cout << "Name: " << students[best].name << '\n';
    cout << "Marks: " << students[best].marks << '\n';

    delete[] students;
    students = nullptr;
    return 0;
}
```

Explanation: Each student is stored in dynamically allocated memory. The program scans the array for the largest marks value and releases the whole array with delete[].

4. FileHandler demonstrating RAI

Solution:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FileHandler {
    ofstream file;

public:
    FileHandler(const string& filename) {
        file.open(filename);
        if (!file)
            throw runtime_error("Could not open file.");
        cout << "File opened.\n";
    }

    void writeData(const string& text) {
        file << text << '\n';
    }

    ~FileHandler() {
        if (file.is_open())
            file.close();
        cout << "Destructor: file closed automatically.\n";
    }
};

int main() {
    cout << "Before block.\n";

    {
        FileHandler handler("raii_demo.txt");
        handler.writeData("Hello from RAII.");
        handler.writeData("The destructor closes the file.");
        cout << "Inside block.\n";
    }

    cout << "After block: FileHandler object is out of scope.\n";
    return 0;
}
```

Explanation: RAII ties resource lifetime to object lifetime. The constructor acquires the file resource, writeData() uses it, and the destructor releases it automatically when the block ends.

5. DynamicArray class with RAI

Solution:

```
#include <iostream>
using namespace std;

class DynamicArray {
    int* data;
    int capacity;
    int size;

public:
    DynamicArray(int n) : capacity(n), size(0) {
        data = new int[capacity];
    }

    ~DynamicArray() {
        delete[] data;
    }

    void insert(int value) {
        if (size < capacity)
            data[size++] = value;
        else
            cout << "Array is full.\n";
    }

    void display() const {
        for (int i = 0; i < size; ++i)
            cout << data[i] << ' ';
        cout << '\n';
    }

    int findMaximum() const {
        if (size == 0) throw runtime_error("Array is empty.");
        int mx = data[0];
        for (int i = 1; i < size; ++i)
            if (data[i] > mx) mx = data[i];
        return mx;
    }

    int findMinimum() const {
        if (size == 0) throw runtime_error("Array is empty.");
        int mn = data[0];
        for (int i = 1; i < size; ++i)
            if (data[i] < mn) mn = data[i];
        return mn;
    }
};

int main() {
    DynamicArray arr(5);

    arr.insert(30);
    arr.insert(10);
    arr.insert(50);
    arr.insert(20);

    arr.display();
    cout << "Maximum = " << arr.findMaximum() << '\n';
    cout << "Minimum = " << arr.findMinimum() << '\n';

    return 0; // destructor releases memory
}
```

Explanation: The class owns the array and therefore cleans it in its destructor. `main()` never calls `delete[]`, satisfying the RAI requirement.

6. Employee management using unique_ptr

Solution:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

class Employee {
public:
    int id;
    string name;
    double salary;

    Employee(int i, string n, double s)
        : id(i), name(move(n)), salary(s) {}
};

class EmployeeManager {
    vector<unique_ptr<Employee>> employees;

public:
    void addEmployee(int id, const string& name, double salary) {
        employees.push_back(make_unique<Employee>(id, name, salary));
    }

    void displayAll() const {
        for (const auto& e : employees)
            cout << e->id << " | " << e->name
                << " | " << e->salary << '\n';
    }

    Employee* searchEmployee(int id) const {
        for (const auto& e : employees)
            if (e->id == id) return e.get();
        return nullptr;
    }

    bool removeEmployee(int id) {
        auto it = remove_if(employees.begin(), employees.end(),
            [id](const unique_ptr<Employee>& e) { return e->id == id; });
        if (it == employees.end()) return false;
        employees.erase(it, employees.end());
        return true;
    }
};

int main() {
    EmployeeManager manager;
    manager.addEmployee(101, "Amit", 50000);
    manager.addEmployee(102, "Neha", 62000);

    cout << "Employees:\n";
    manager.displayAll();

    if (Employee* e = manager.searchEmployee(102))
        cout << "Found: " << e->name << '\n';

    manager.removeEmployee(101);
    cout << "After removal:\n";
    manager.displayAll();
}
```

Explanation: unique_ptr gives single ownership. make_unique creates the objects, and erasing a unique_ptr destroys the corresponding Employee automatically. No raw new/delete is used.

7. Student-Course many-to-many relationship with shared_ptr

Solution:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

class Course;

class Student {
public:
    string name;
    vector<shared_ptr<Course>> courses;

    Student(string n) : name(move(n)) {}
};

class Course {
public:
    string name;
    vector<shared_ptr<Student>> students;

    Course(string n) : name(move(n)) {}
};

void addStudentToCourse(const shared_ptr<Student>& s,
                       const shared_ptr<Course>& c) {
    s->courses.push_back(c);
    c->students.push_back(s);
}

int main() {
    auto s1 = make_shared<Student>("Amit");
    auto s2 = make_shared<Student>("Neha");

    auto c1 = make_shared<Course>("OOP");
    auto c2 = make_shared<Course>("Data Structures");

    addStudentToCourse(s1, c1);
    addStudentToCourse(s1, c2);
    addStudentToCourse(s2, c1);

    cout << "Students in each course:\n";
    for (const auto& c : {c1, c2}) {
        cout << c->name << ": ";
        for (const auto& s : c->students) cout << s->name << " ";
        cout << '\n';
    }

    cout << "\nCourses for each student:\n";
    for (const auto& s : {s1, s2}) {
        cout << s->name << ": ";
        for (const auto& c : s->courses) cout << c->name << " ";
        cout << '\n';
    }
}
```

Explanation: shared_ptr is used because both sides maintain shared ownership in this direct model. In production code, bidirectional shared ownership can create cycles; weak_ptr is preferable when one direction should be non-owning.

8. Banking system: Bank, Customer, BankAccount

Solution:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
using namespace std;

class BankAccount {
    int number;
    double balance;

public:
    BankAccount(int n) : number(n), balance(0) {}

    int getNumber() const { return number; }
    void deposit(double amount) { if (amount > 0) balance += amount; }

    bool withdraw(double amount) {
        if (amount <= 0 || amount > balance) return false;
        balance -= amount;
        return true;
    }

    void display() const {
        cout << "Account " << number << ", Balance = " << balance << '\n';
    }
};

class Customer {
    int id;
    string name;
    vector<unique_ptr<BankAccount>> accounts;

public:
    Customer(int i, string n) : id(i), name(move(n)) {}

    void addAccount(int number) {
        accounts.push_back(make_unique<BankAccount>(number));
    }

    BankAccount* findAccount(int number) const {
        for (const auto& a : accounts)
            if (a->getNumber() == number) return a.get();
        return nullptr;
    }

    void display() const {
        cout << "Customer: " << id << " - " << name << '\n';
        for (const auto& a : accounts) a->display();
    }
};

class Bank {
    vector<unique_ptr<Customer>> customers;

public:
    void addCustomer(int id, const string& name) {
        customers.push_back(make_unique<Customer>(id, name));
    }

    Customer* findCustomer(int id) const {
        for (const auto& c : customers)
            if (c) {
                // Customer id lookup is simplified through stored order in this example.
            }
        return nullptr;
    }

    Customer* getCustomer(size_t index) const {
        return index < customers.size() ? customers[index].get() : nullptr;
    }

    void display() const {
        for (const auto& c : customers) c->display();
    }
};

int main() {
    Bank bank;
    bank.addCustomer(1, "Amit");
    bank.addCustomer(2, "Neha");

    auto* c1 = bank.getCustomer(0);
```

```
auto* c2 = bank.getCustomer(1);

c1->addAccount(1001);
c1->addAccount(1002);
c2->addAccount(2001);

c1->findAccount(1001)->deposit(5000);
c1->findAccount(1001)->withdraw(1200);
c2->findAccount(2001)->deposit(8000);

bank.display();
}
```

Explanation: The Bank uniquely owns Customers, and each Customer uniquely owns its BankAccount objects. Account operations are implemented through the account pointer returned by findAccount().

9. shared_ptr cycle and weak_ptr solution

Solution:

```
#include <iostream>
#include <memory>
#include <string>
using namespace std;

class PersonB;

class PersonA {
public:
    string name;
    shared_ptr<PersonB> friendB;

    PersonA(string n) : name(move(n)) {}
    ~PersonA() { cout << name << " destroyed\n"; }
};

class PersonB {
public:
    string name;
    // Version 1: shared_ptr<PersonA> friendA;    // creates a cycle
    weak_ptr<PersonA> friendA;                  // Version 2: non-owning link

    PersonB(string n) : name(move(n)) {}
    ~PersonB() { cout << name << " destroyed\n"; }
};

int main() {
    auto a = make_shared<PersonA>("Alice");
    auto b = make_shared<PersonB>("Bob");

    a->friendB = b;
    b->friendA = a;

    cout << "A use_count = " << a.use_count() << '\n';
    cout << "B use_count = " << b.use_count() << '\n';

    // At the end of main, both objects are destroyed because B's
    // reference to A is weak and therefore does not own A.
}
```

Explanation: If PersonA and PersonB both used shared_ptr for the friend link, each object could keep the other alive, creating a reference cycle and preventing destruction. weak_ptr observes the other object without increasing its ownership count, so the cycle is broken.

10. Library, Book, Member borrowing system

Solution:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
using namespace std;

class Book {
public:
    int id;
    string title;
    bool borrowed = false;

    Book(int i, string t) : id(i), title(move(t)) {}
};

class Member {
public:
    int id;
    string name;
    vector<shared_ptr<Book>> borrowedBooks;

    Member(int i, string n) : id(i), name(move(n)) {}
};

class Library {
    vector<unique_ptr<Book>> books;
    vector<unique_ptr<Member>> members;

public:
    void addBook(int id, const string& title) {
        books.push_back(make_unique<Book>(id, title));
    }

    void registerMember(int id, const string& name) {
        members.push_back(make_unique<Member>(id, name));
    }

    Book* findBook(int id) const {
        for (const auto& b : books)
            if (b->id == id) return b.get();
        return nullptr;
    }

    Member* findMember(int id) const {
        for (const auto& m : members)
            if (m->id == id) return m.get();
        return nullptr;
    }

    bool borrowBook(int memberId, int bookId) {
        Book* b = findBook(bookId);
        Member* m = findMember(memberId);
        if (!b || !m || b->borrowed) return false;

        b->borrowed = true;
        m->borrowedBooks.push_back(shared_ptr<Book>(b, [](Book*){}));
        return true;
    }

    bool returnBook(int memberId, int bookId) {
        Member* m = findMember(memberId);
        if (!m) return false;

        for (auto it = m->borrowedBooks.begin();
             it != m->borrowedBooks.end(); ++it) {
            if ((*it)->id == bookId) {
                findBook(bookId)->borrowed = false;
                m->borrowedBooks.erase(it);
                return true;
            }
        }
        return false;
    }

    void displayAvailableBooks() const {
        for (const auto& b : books)
            if (!b->borrowed)
                cout << b->id << " - " << b->title << '\n';
    }
};

int main() {
```

```

Library lib;
lib.addBook(1, "C++ Basics");
lib.addBook(2, "OOP Concepts");
lib.registerMember(101, "Amit");

lib.borrowBook(101, 1);

cout << "Available books:\n";
lib.displayAvailableBooks();

lib.returnBook(101, 1);
cout << "\nAfter return:\n";
lib.displayAvailableBooks();
}

```

Explanation: The Library uniquely owns Book and Member objects. A member's borrowed-book reference is non-owning because the Library remains the owner; the custom-deleter `shared_ptr` above is used only as a safe shared interface without transferring deletion responsibility.

11. University, Department, Professor relationships

Solution:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
using namespace std;

class Professor {
public:
    int id;
    string name;
    Professor(int i, string n) : id(i), name(move(n)) {}
};

class Department {
public:
    string name;
    vector<shared_ptr<Professor>> professors;

    Department(string n) : name(move(n)) {}

    void addProfessor(const shared_ptr<Professor>& p) {
        professors.push_back(p);
    }
};

class University {
    vector<unique_ptr<Department>> departments;
    vector<shared_ptr<Professor>> professors;

public:
    void addDepartment(const string& name) {
        departments.push_back(make_unique<Department>(name));
    }

    void addProfessor(int id, const string& name) {
        professors.push_back(make_shared<Professor>(id, name));
    }

    Department* getDepartment(size_t i) {
        return i < departments.size() ? departments[i].get() : nullptr;
    }

    shared_ptr<Professor> getProfessor(size_t i) {
        return i < professors.size() ? professors[i] : nullptr;
    }

    void assignProfessor(size_t d, size_t p) {
        if (auto* dept = getDepartment(d)) {
            if (auto prof = getProfessor(p))
                dept->addProfessor(prof);
        }
    }

    void display() const {
        for (const auto& d : departments) {
            cout << "Department: " << d->name << '\n';
            for (const auto& p : d->professors)
                cout << "    Professor: " << p->name << '\n';
        }
    }
};

int main() {
    University u;
    u.addDepartment("Computer Science");
    u.addDepartment("Mathematics");

    u.addProfessor(1, "Dr. Sharma");
    u.addProfessor(2, "Dr. Patel");

    u.assignProfessor(0, 0);
    u.assignProfessor(1, 1);
    u.assignProfessor(0, 1);

    u.display();
}
```

Explanation: A University uniquely owns its Departments, so `unique_ptr` represents the composition relationship. Professors can live independently and may be shared by department references, so `shared_ptr` is suitable for professor ownership.

12. Company, Employee, Project many-to-many system

Solution:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
using namespace std;

class Project;

class Employee {
public:
    int id;
    string name;
    vector<weak_ptr<Project>> projects;

    Employee(int i, string n) : id(i), name(move(n)) {}
};

class Project {
public:
    int id;
    string name;
    vector<weak_ptr<Employee>> employees;

    Project(int i, string n) : id(i), name(move(n)) {}
};

class Company {
    vector<shared_ptr<Employee>> employees;
    vector<shared_ptr<Project>> projects;

public:
    void addEmployee(int id, const string& name) {
        employees.push_back(make_shared<Employee>(id, name));
    }

    void createProject(int id, const string& name) {
        projects.push_back(make_shared<Project>(id, name));
    }

    shared_ptr<Employee> employee(int id) {
        for (auto& e : employees) if (e->id == id) return e;
        return nullptr;
    }

    shared_ptr<Project> project(int id) {
        for (auto& p : projects) if (p->id == id) return p;
        return nullptr;
    }

    void assignEmployee(int employeeId, int projectId) {
        auto e = employee(employeeId);
        auto p = project(projectId);
        if (!e || !p) return;
        p->employees.push_back(e);
        e->projects.push_back(p);
    }

    void display() const {
        cout << "Projects:\n";
        for (const auto& p : projects) {
            cout << p->name << ": ";
            for (const auto& w : p->employees)
                if (auto e = w.lock()) cout << e->name << " ";
            cout << '\n';
        }

        cout << "\nEmployees:\n";
        for (const auto& e : employees) {
            cout << e->name << ": ";
            for (const auto& w : e->projects)
                if (auto p = w.lock()) cout << p->name << " ";
            cout << '\n';
        }
    }
};

int main() {
    Company c;
    c.addEmployee(1, "Amit");
    c.addEmployee(2, "Neha");
    c.createProject(101, "Website");
    c.createProject(102, "Mobile App");
}
```

```
    c.assignEmployee(1, 101);  
    c.assignEmployee(1, 102);  
    c.assignEmployee(2, 101);  
    c.display();  
}
```

Explanation: The Company owns the employee and project objects through shared_ptr. The cross-links are weak_ptr because the assignment relationship should not create an ownership cycle. Both employees and projects can exist independently.

13. E-commerce: Customer, Order, Product

Solution:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
using namespace std;

class Product {
public:
    int id;
    string name;
    double price;

    Product(int i, string n, double p)
        : id(i), name(move(n)), price(p) {}
};

class Order {
public:
    int id;
    vector<shared_ptr<Product>> products;

    Order(int i) : id(i) {}

    void addProduct(const shared_ptr<Product>& p) {
        products.push_back(p);
    }
};

class Customer {
public:
    int id;
    string name;
    vector<unique_ptr<Order>> orders;

    Customer(int i, string n) : id(i), name(move(n)) {}

    void createOrder(int orderId) {
        orders.push_back(make_unique<Order>(orderId));
    }

    Order* findOrder(int orderId) {
        for (auto& o : orders) if (o->id == orderId) return o.get();
        return nullptr;
    }
};

class Store {
public:
    vector<shared_ptr<Customer>> customers;
    vector<shared_ptr<Product>> products;

    void addCustomer(int id, const string& name) {
        customers.push_back(make_shared<Customer>(id, name));
    }

    void addProduct(int id, const string& name, double price) {
        products.push_back(make_shared<Product>(id, name, price));
    }

    shared_ptr<Customer> customer(int id) {
        for (auto& c : customers) if (c->id == id) return c;
        return nullptr;
    }

    shared_ptr<Product> product(int id) {
        for (auto& p : products) if (p->id == id) return p;
        return nullptr;
    }

    void createOrder(int customerId, int orderId) {
        if (auto c = customer(customerId)) c->createOrder(orderId);
    }

    void addProductToOrder(int customerId, int orderId, int productId) {
        auto c = customer(customerId);
        auto p = product(productId);
        if (c && p)
            if (auto o = c->findOrder(orderId))
                o->addProduct(p);
    }

    void display() {
```

```

        for (const auto& c : customers) {
            cout << "Customer: " << c->name << '\n';
            for (const auto& o : c->orders) {
                cout << "  Order " << o->id << ": ";
                for (const auto& p : o->products)
                    cout << p->name << " ";
                cout << '\n';
            }
        }
    };

int main() {
    Store store;
    store.addCustomer(1, "Amit");
    store.addProduct(10, "Laptop", 70000);
    store.addProduct(11, "Mouse", 1000);

    store.createOrder(1, 5001);
    store.addProductToOrder(1, 5001, 10);
    store.addProductToOrder(1, 5001, 11);

    store.display();
}

```

Explanation: Customer owns its orders through unique_ptr, so an order cannot outlive its customer. Products are independently owned by the Store and shared_ptr lets the same Product appear in multiple orders.

14. ResourceManager with RAI and unique_ptr

Solution:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

class Resource {
public:
    int id;
    string name;

    Resource(int i, string n) : id(i), name(move(n)) {}
};

class ResourceManager {
    vector<unique_ptr<Resource>> resources;

public:
    void addResource(int id, const string& name) {
        resources.push_back(make_unique<Resource>(id, name));
    }

    bool removeResource(int id) {
        auto it = remove_if(resources.begin(), resources.end(),
            [id](const unique_ptr<Resource>& r) { return r->id == id; });

        if (it == resources.end()) return false;
        resources.erase(it, resources.end());
        return true;
    }

    Resource* findResource(int id) const {
        for (const auto& r : resources)
            if (r->id == id) return r.get();
        return nullptr;
    }

    void displayResources() const {
        for (const auto& r : resources)
            cout << r->id << " - " << r->name << '\n';
    }
};

int main() {
    ResourceManager manager;

    manager.addResource(1, "Database");
    manager.addResource(2, "File");
    manager.addResource(3, "Network");

    manager.displayResources();

    if (auto* r = manager.findResource(2))
        cout << "Found: " << r->name << '\n';

    manager.removeResource(1);
    cout << "After removal:\n";
    manager.displayResources();
}
```

Explanation: ResourceManager exclusively owns resources through unique_ptr. No raw new or delete is needed in main(), and destruction occurs automatically when resources are erased or the manager is destroyed.

15. Document-sharing system with shared_ptr and weak_ptr

Solution:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
using namespace std;

class Document {
public:
    int id;
    string contents;

    Document(int i, string c) : id(i), contents(move(c)) {
        cout << "Document created.\n";
    }

    ~Document() {
        cout << "Document destroyed.\n";
    }
};

class Editor {
public:
    string name;
    weak_ptr<Document> currentDocument;

    Editor(string n) : name(move(n)) {}

    void openDocument(const shared_ptr<Document>& doc) {
        currentDocument = doc;
        cout << name << " opened document.\n";
    }

    void closeDocument() {
        currentDocument.reset();
        cout << name << " closed document.\n";
    }

    void displayContents() const {
        if (auto doc = currentDocument.lock())
            cout << "Contents: " << doc->contents << '\n';
        else
            cout << "Document is no longer available.\n";
    }
};

class User {
public:
    string name;
    vector<shared_ptr<Document>> documents;

    User(string n) : name(move(n)) {}

    void accessDocument(const shared_ptr<Document>& doc) {
        documents.push_back(doc);
    }
};

int main() {
    auto doc = make_shared<Document>(1, "Shared project document.");

    User u1("Amit"), u2("Neha");
    Editor editor("Editor");

    u1.accessDocument(doc);
    u2.accessDocument(doc);
    editor.openDocument(doc);

    editor.displayContents();

    u1.documents.clear();
    cout << "One user released the document.\n";
    editor.displayContents();

    u2.documents.clear();
    doc.reset(); // last owning shared_ptr released

    editor.displayContents();
    return 0;
}
```

Explanation: Users hold shared_ptr because multiple users may own/access the document. Editor uses weak_ptr as a non-owning reference. When the last shared_ptr owner releases the document, the Document is destroyed and weak_ptr::lock() fails.

16. Complete library system with RAI and non-owning member references

Solution:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
using namespace std;

class Book {
public:
    int id;
    string title;
    bool borrowed = false;

    Book(int i, string t) : id(i), title(move(t)) {}
};

class Member {
public:
    int id;
    string name;
    vector<weak_ptr<Book>> books;

    Member(int i, string n) : id(i), name(move(n)) {}
};

class Library {
    vector<shared_ptr<Book>> books;
    vector<unique_ptr<Member>> members;

public:
    void addBook(int id, const string& title) {
        books.push_back(make_shared<Book>(id, title));
    }

    void registerMember(int id, const string& name) {
        members.push_back(make_unique<Member>(id, name));
    }

    shared_ptr<Book> searchBook(int id) const {
        for (const auto& b : books)
            if (b->id == id) return b;
        return nullptr;
    }

    Member* findMember(int id) const {
        for (const auto& m : members)
            if (m->id == id) return m.get();
        return nullptr;
    }

    bool borrowBook(int memberId, int bookId) {
        auto book = searchBook(bookId);
        auto member = findMember(memberId);

        if (!book || !member || book->borrowed) return false;

        book->borrowed = true;
        member->books.push_back(book); // weak_ptr: member does not own book
        return true;
    }

    bool returnBook(int memberId, int bookId) {
        auto member = findMember(memberId);
        if (!member) return false;

        for (auto it = member->books.begin();
             it != member->books.end(); ++it) {
            if (auto book = it->lock()) {
                if (book->id == bookId) {
                    book->borrowed = false;
                    member->books.erase(it);
                    return true;
                }
            }
        }
        return false;
    }

    void displayBooks() const {
        for (const auto& b : books)
            cout << b->id << " - " << b->title
                  << (b->borrowed ? " [Borrowed]" : " [Available]") << '\n';
    }
};
```

```

void displayMemberBooks(int memberId) const {
    auto member = findMember(memberId);
    if (!member) return;

    cout << "Books borrowed by " << member->name << ":\n";
    for (const auto& w : member->books)
        if (auto b = w.lock())
            cout << b->id << " - " << b->title << '\n';
}

};

int main() {
    Library library;

    library.addBook(1, "C++ Programming");
    library.addBook(2, "Object Oriented Design");
    library.registerMember(101, "Amit");
    library.registerMember(102, "Neha");

    library.borrowBook(101, 1);
    library.borrowBook(102, 2);

    library.displayBooks();
    library.displayMemberBooks(101);
    library.displayMemberBooks(102);

    library.returnBook(101, 1);

    cout << "\nAfter return:\n";
    library.displayBooks();
}

```

Explanation: The Library owns Book objects with `shared_ptr` and owns Member objects with `unique_ptr`. Members only temporarily reference books with `weak_ptr`, so members never become owners of books. No explicit delete is needed in `main()`.