



CC-211L Object Oriented Programming Lab

Department of Computer Science

University of the Punjab

Fall 2026

Lab 4: Copying, const, and Static Members

After completing this lab, you will be able to:

- Explain memberwise copying and identify when copying pointer values is unsafe.
- Implement and use a copy constructor.
- Use const objects, const member functions, and const references correctly.
- Implement static data and member functions shared by all objects.
- Deep-copy one dynamically allocated resource and verify the copies are independent.

Lab duration: 1 hour 45 minutes

Part A: Guided foundations, 50 marks, $\approx$ 45 minutes

Part B: Independent programming problem, 50 marks, $\approx$ 35 minutes

Fixed tests, written explanation, and submission evidence: $\approx$ 25 minutes

Grading Scheme

Question:	1	2	3	4	5	6	Total
Points:	10	15	25	30	15	5	100
Score:							

Important instructions

- Use C++17 and compile with `-std=c++17 -Wall -Wextra -pedantic`.
- Use one source file named `scorebox.cpp`.
- Do not add extra dynamic allocations. Part B owns exactly one dynamic `int` per `ScoreBox` object.
- Do not pass a `ScoreBox` by value. Copy only where the problem explicitly requests copy construction.
- Keep output labels exactly as specified. Do not print memory addresses.
- Submit one PDF containing answers and output evidence, plus `scorebox.cpp`.
- Name the PDF `roll_number_oop_lab4.pdf`.

1 Part A: Guided Foundations

1. [10 marks] A1. Trace memberwise copying

Consider this class and object creation:

```
class Result {
public:
    Result(std::string n, int m) : name{n}, mark{m} {}
    std::string getName() const { return name; }
    int getMark() const { return mark; }

private:
    std::string name;
    int mark;
};

Result first{"Areeba", 84};
Result second{first};
```

Answer in the PDF:

- What values do `second.getName()` and `second.getMark()` return?
- Which constructor initializes `second`?
- If `first` and `second` are separate objects, does changing one object's `mark` change the other object's `mark`? Explain.
- A class instead stores an owning `int*`. What does the default memberwise copy copy: the pointed-to integer or the address?
- State one problem that may occur if both copied objects later delete the same address.

2. [15 marks] A2. Implement a copyable value class

In `scorebox.cpp`, implement `StudentResult` with this interface:

```
class StudentResult {
public:
    StudentResult(const std::string& name, int mark);
    StudentResult(const StudentResult& other);
    void setMark(int mark);
    std::string getName() const;
    int getMark() const;

private:
    std::string name;
    int mark;
};
```

The constructors must initialize all members with member-initializer lists. The copy constructor must copy the name and mark from `other`. The getters must be `const`. Also implement:

```
void printResult(const StudentResult& result);
```

It must print exactly `RESULT name=<name> mark=<mark>`. In `main`, create `source{"Bilal", 76}`, copy-construct `copy{source}`, change the copy's mark to 91, and print both objects with `printResult`. Show the output and answer: why does changing the copy leave the source unchanged?

3. [25 marks] A3. Track live objects with a static member

Implement this class. Its static count records how many `Badge` objects are currently alive:

```

class Badge {
public:
    Badge();
    ~Badge();
    static int getLiveCount();

private:
    static int liveCount;
};

int Badge::liveCount{0};

```

Increment the count in the constructor and decrement it in the destructor. The static getter must be **const-free** and must not require an object. Add this exact function and call it once:

```

void traceBadges() {
    std::cout << "BADGE start=" << Badge::getLiveCount() << '\n';
    Badge first;
    std::cout << "BADGE after first=" << Badge::getLiveCount() << '\n';
    {
        Badge second;
        std::cout << "BADGE inside=" << Badge::getLiveCount() << '\n';
    }
    std::cout << "BADGE after block=" << Badge::getLiveCount() << '\n';
}

```

In the PDF, show the exact output. Also explain why `liveCount` is static and identify the event that decreases it from 2 to 1.

2 Part B: Independent Programming Problem

Problem: Score box. Implement a `ScoreBox` that owns one dynamically allocated integer. Copying a box must create an independent integer, not share the original address. The class also tracks how many of its objects are currently alive.

Required public interface:

```

class ScoreBox {
public:
    explicit ScoreBox(int initialScore);
    ScoreBox(const ScoreBox& other);
    ~ScoreBox();
    void setScore(int score);
    int getScore() const;
    static int getLiveCount();

private:
    int* score;
    static int liveCount;
};

```

Required behavior:

- The ordinary constructor dynamically allocates one `int` initialized to `initialScore` and increments the live-object count.
- The copy constructor dynamically allocates a different `int` initialized with the value in `other`; it also increments the live-object count.
- The destructor deletes the object's integer exactly once and decrements the live-object count.

- `setScore` changes this object's integer. `getScore` reads it and is a `const` member function.
- `getLiveCount` returns the shared count and can be called using `ScoreBox::getLiveCount()`.
- Do not implement or use copy assignment.

4. [30 marks] **B1. Design and implement ScoreBox**

Design the private state and implement all required operations in `scorebox.cpp`. Include the definition and initialization of the static data member. Use member-initializer lists where appropriate. Your implementation is graded for:

- correct state and public interface: 5 marks;
- correct ordinary construction and static count update: 5 marks;
- independent deep copy in the copy constructor: 10 marks;
- correct destructor and live-count update: 5 marks;
- correct setter, const getter, and static getter: 5 marks.

5. [15 marks] **B2. Run the fixed tests**

Add this exact sequence to `main`. Do not add extra objects or copy any `ScoreBox` by value. Print Boolean results as `true` or `false`.

```
std::cout << "LIVE start=" << ScoreBox::getLiveCount() << '\n';
{
    ScoreBox original{40};
    ScoreBox copied{original};
    std::cout << "LIVE after copy=" << ScoreBox::getLiveCount() << '\n';
    copied.setScore(99);
    std::cout << "COPY original=" << original.getScore()
               << " copied=" << copied.getScore() << '\n';
    const ScoreBox fixed{7};
    std::cout << "CONST score=" << fixed.getScore() << '\n';
    std::cout << "LIVE with const=" << ScoreBox::getLiveCount() << '\n';
}
std::cout << "LIVE after scope=" << ScoreBox::getLiveCount() << '\n';
```

Show the output produced. The grader checks the exact labels, values, and order.

6. [5 marks] **B3. Explain the copy**

Answer briefly:

- Why must the copy constructor allocate a new integer?
- What would go wrong if both objects deleted the same integer?
- Which functions can be called on `fixed`, and why?

Submission and grading checklist

Submit one PDF and `scorebox.cpp` only. The PDF must contain your full name, roll number, compile command, Part A answers, and program output evidence. Compile with:

```
g++ -std=c++17 -Wall -Wextra -pedantic scorebox.cpp -o scorebox
```

The grader will check the exact guided output, correct deep-copy independence, static live counts, `const` correctness, and warning-free compilation. Do not submit the executable.