



CC-211L Object Oriented Programming Lab
Department of Computer Science
University of the Punjab
Fall 2026

Lab 3: Destructors, Lifetime, and Dynamic Objects

After completing this lab, you will be able to:

- Define a destructor and observe when it runs for an automatic object.
- Trace construction and destruction through nested scopes.
- Use `this` to distinguish a parameter from the receiving object's data member.
- Pass an existing object by reference or by pointer without transferring ownership.
- Check a pointer against `nullptr` before using `->`.
- Create, use, and release one dynamic object with matching `new` and `delete`.
- Explain ownership as responsibility for ending a dynamic object's lifetime.

Lab duration: 1 hour 45 minutes

Part A: Automatic lifetime and destructors (20 marks)

Part B: `this`, references, and pointers (20 marks)

Part C: Dynamic objects and ownership (20 marks)

Part D: Independent programming problem (40 marks)

Grading Scheme

Question:	1	2	3	4	5	6	7	8	9	10	Total
Points:	5	15	10	10	10	5	5	25	10	5	100
Score:											

Important instructions

- Use one source file named `lifetime.cpp`.
- Do not use inheritance, operator overloading, templates, STL containers, exceptions, or file I/O.
- Do not use copy construction, assignment of class objects, or pass any class object by value.
- Do not call a destructor explicitly. Do not use `delete` on an automatic object.
- Keep the output labels exactly as specified. Do not print memory addresses.
- Submit one PDF containing answers and output evidence, plus `lifetime.cpp`.
- Name the PDF `roll_number_oop_lab3.pdf`.

1 Part A: Automatic Lifetime and Destructors

Term used in this part: An *automatic object* is an object declared inside a function or block. Its lifetime begins when its declaration is initialized and ends automatically when execution leaves its block. A *destructor* is a special member function named `~ClassName()` with no return type and no parameters. C++ runs it when that object's lifetime ends.

```
class Marker {
public:
    explicit Marker(const std::string& label) : name{label} {
        std::cout << "construct " << name << '\n';
    }

    ~Marker() {
        std::cout << "destroy " << name << '\n';
    }

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

The destructor above only prints a line. This is deliberate: it makes lifetime observable. For a class that owns a resource, a destructor can also release that resource; Part C uses that case.

1. [5 marks] A1. Implement Marker

In `lifetime.cpp`, implement the `Marker` class exactly as shown above. The data member must be private. The constructor must use a member-initializer list. The destructor must print exactly one line in the form shown above.

2. [15 marks] A2. Trace nested scopes

Add this function exactly as shown and call it once from `main`:

```
void traceAutomaticObjects() {
    std::cout << "begin trace\n";
    Marker first{"first"};
    {
        Marker second{"second"};
        std::cout << "inside block\n";
    }
    std::cout << "after block\n";
}
```

Do not print extra lines inside this function. In the PDF, answer:

- Write the output produced by calling `traceAutomaticObjects()`.
- Why is `second` destroyed before `first`?
- At which closing brace does `second`'s lifetime end?
- Does the closing brace destroy the `Marker` object directly, or does it cause C++ to run the destructor automatically?

2 Part B: this, References, and Pointers

Terms used in this part:

- `this` is the pointer available inside an ordinary member function. It points to the object receiving the call.

- A *reference* is an alias for an existing object. It must be initialized with an object and cannot represent no object.
- A *pointer* is an object whose value is an address. It can be changed and can contain `nullptr`.
- *Borrowing* means using an object without becoming responsible for destroying it. A function receiving a reference or pointer borrows unless the interface explicitly says otherwise.

For a reference, use the dot operator:

```
Account& reference{account};
reference.deposit(10);
```

For a pointer, first ensure it points to an object, then use the arrow operator:

```
Account* pointer{&account};
if (pointer != nullptr) {
    pointer->deposit(10);
}
```

The pointer and reference above both provide access to `account`; neither one becomes its owner. The code that created an automatic `account` remains responsible for its lifetime.

3. [10 marks] **B1. Implement Account and use this**

Implement this class in `lifetime.cpp`:

```
class Account {
public:
    Account(const std::string& accountName, int initialBalance);
    void setName(const std::string& accountName);
    void deposit(int amount);
    std::string getName() const;
    int getBalance() const;

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

Use these exact rules:

- The constructor stores the name and stores `initialBalance` only if it is positive; otherwise balance is 0.
- Use `this` pointer in `setName`.
- `deposit` function changes the balance only when `amount > 0`.
- Both getter functions are `const` and are not allowed to modify the object.

In main, create `Account account{"Areeba", 50}`; call `account.setName("Areeba Khan")`; call `account.deposit(25)`; and print exactly:

```
ACCOUNT name=Areeba Khan balance=75
```

4. [10 marks] **B2. Implement borrowing functions**

Implement these exact free functions:

```
void printAccount(const Account& account);
bool depositIfPositive(Account& account, int amount);
void inspectAccount(const Account* account);
```

Use these rules:

- `printAccount` prints `ACCOUNT name=<name> balance=<balance>` and does not modify the object.
- `depositIfPositive` calls `deposit` and returns `true` for a positive amount; it returns `false` and does not change the account otherwise.
- `inspectAccount` checks for `nullptr`. For a non-null pointer, it calls `printAccount`; for null, it prints exactly `ACCOUNT absent`.
- None of these functions may call `delete`. They borrow access.

In `main()`, call the functions in this order:

```
Account account{"Bilal", 10};
printAccount(account);
std::cout << "DEPOSIT result="
           << (depositIfPositive(account, 5) ? "true" : "false") << '\n';
std::cout << "DEPOSIT result="
           << (depositIfPositive(account, -2) ? "true" : "false") << '\n';
inspectAccount(&account);
inspectAccount(nullptr);
```

and show the output produced.

3 Part C: Dynamic Objects and Ownership

Terms used in this part: A *dynamic object* is created by `new` and remains alive until code executes matching `delete`. The pointer returned by `new` is not automatically an owner. The *owner* is the code or object responsible for executing `delete` exactly once.

If the only pointer to a dynamic object is overwritten or discarded before `delete`, the object becomes unreachable while its storage remains allocated. This is a *memory leak*. Assigning `nullptr` changes the pointer value; it does not destroy the object that the pointer used to address.

A resource-owning class uses its destructor to release its resource:

```
class OwnedScore {
public:
    explicit OwnedScore(int initial)
        : value{new int{initial}} {}

    ~OwnedScore() {
        delete value;
    }

private:
    int* value;
};
```

The object owns the dynamically allocated integer. When the object is destroyed, its destructor releases the integer.

5. [10 marks] C1. Implement OwnedScore

Implement `OwnedScore` exactly as shown above. Add these member functions:

```
void setValue(int value);
int getValue() const;
```

`setValue` changes the dynamically allocated integer through the pointer. The destructor must contain exactly one `delete value;` statement. Do not add a second destructor, and do not call the destructor explicitly.

In `main`, execute:

```
OwnedScore automatic{10};
automatic.setValue(15);
std::cout << "SCORE automatic=" << automatic.getValue() << '\n';
```

and show the output.

6. [5 marks] **C2. Create and release one dynamic object**

Add this exact function and call it once:

```
void useDynamicScore() {
    OwnedScore* score{new OwnedScore{20}};
    std::cout << "SCORE dynamic=" << score->getValue() << '\n';
    score->setValue(25);
    std::cout << "SCORE dynamic=" << score->getValue() << '\n';
    delete score;
    score = nullptr;
    std::cout << "POINTER cleared=" << (score == nullptr ? "true" : "false
        ") << '\n';
}
```

and show the output.

The owner in this function is the code between `new` and `delete`. The pointer is not the owner merely because it stores the address.

7. [5 marks] **C3. Diagnose a leak**

The following function is broken:

```
void broken() {
    OwnedScore* score{new OwnedScore{30}};
    score = nullptr;
}
```

Answer all three questions:

- Is the dynamically allocated `OwnedScore` destroyed? Explain.
- What happens to the pointer variable?
- Rewrite the function's two final statements so the object is released safely and the pointer is then cleared.

4 Part D: Independent Programming Problem

This part is intentionally less guided. You must design and implement the class from the problem statement. The required behavior and test values are fixed so that the program remains easy to verify.

Problem: Parking meter. A parking meter records the number of parking minutes currently available for one parking zone. The meter owns its minute count: the count must be stored in a dynamically allocated `int`, and the destructor must release it. A meter must never have a negative remaining time or more time than its maximum.

Required class: Implement a class named `ParkingMeter`. You must decide the placement of public and private, the data members, and the constructor definition.

Required public operations:

```
ParkingMeter(const std::string& zoneName, int maximumMinutes);
~ParkingMeter();
bool addMinutes(int minutes);
bool useMinutes(int minutes);
std::string getZone() const;
int getRemaining() const;
```

Required behavior:

- The constructor stores `zoneName`. If it is empty, store `UNKNOWN`.
- If `maximumMinutes` is not positive, use 60 as the maximum.
- The initial remaining time is 0.
- The maximum value must be stored as an ordinary private `int`.
- The remaining value must be stored as a private pointer to a dynamically allocated `int`.
- `addMinutes` returns `true` and increases the remaining time only when `minutes > 0` and the new value would not exceed the maximum.
- `useMinutes` returns `true` and decreases the remaining time only when `minutes > 0` and enough time remains.
- A failed operation returns `false` and must not change the remaining time.
- `getZone` and `getRemaining` are getters and must be `const`.
- The destructor must release the dynamically allocated remaining-time integer exactly once.

Ownership rule: The `ParkingMeter` object owns the dynamic integer because its constructor creates it and its destructor must delete it. A pointer variable used temporarily in `main` does not become the owner of the `ParkingMeter`; it only provides access to that object.

8. [25 marks] D1. Design and implement `ParkingMeter`

Create `ParkingMeter` in `lifetime.cpp`. Use a member-initializer list for members that can be initialized there. Use `this->` where a parameter name and a data-member name would otherwise be confused.

Your implementation receives marks for:

- correct private state and public interface: 5 marks;
- constructor validation and initial state: 5 marks;
- correct successful and failed `addMinutes` behavior: 5 marks;
- correct successful and failed `useMinutes` behavior: 5 marks;
- destructor and exactly-once resource release: 5 marks.

9. [10 marks] D2. Run the fixed boundary tests

Add the following test sequence to `main`. Use the labels exactly as shown. For Boolean results, print “true” or “false” with a conditional expression.

```
ParkingMeter meter{"P-12", 30};
std::cout << "D1 remaining=" << meter.getRemaining() << '\n';
std::cout << "D2 add10=" << (meter.addMinutes(10) ? "true" : "false")
          << " remaining=" << meter.getRemaining() << '\n';
std::cout << "D3 add25=" << (meter.addMinutes(25) ? "true" : "false")
          << " remaining=" << meter.getRemaining() << '\n';
std::cout << "D4 use4=" << (meter.useMinutes(4) ? "true" : "false")
          << " remaining=" << meter.getRemaining() << '\n';
std::cout << "D5 use10=" << (meter.useMinutes(10) ? "true" : "false")
          << " remaining=" << meter.getRemaining() << '\n';
std::cout << "D6 addNegative="
```

```

        << (meter.addMinutes(-2) ? "true" : "false")
        << " remaining=" << meter.getRemaining() << '\n';

ParkingMeter invalid{"", -1};
std::cout << "D7 fallback zone=" << invalid.getZone()
        << " remaining=" << invalid.getRemaining() << '\n';

ParkingMeter* dynamicMeter{new ParkingMeter{"D-7", 20}};
dynamicMeter->addMinutes(20);
std::cout << "D8 dynamic remaining=" << dynamicMeter->getRemaining() << '\n';
delete dynamicMeter;
dynamicMeter = nullptr;
std::cout << "D9 pointer cleared="
        << (dynamicMeter == nullptr ? "true" : "false") << '\n';

```

Show your output.

10. [5 marks] **D3. Explain the ownership boundary**

Answer briefly:

- Which object owns the dynamically allocated integer inside `meter`?
- Which function must release that integer, and why?
- In the D2 dynamic test, what does `dynamicMeter` point to, and what does `delete dynamicMeter` destroy?
- Why would assigning `dynamicMeter = nullptr` before `delete dynamicMeter` create a leak?

Submission and grading checklist

Submit one PDF and `lifetime.cpp` only. The PDF must contain your full name, roll number, the compile command, answers, and output evidence. The grader will check:

- exact destructor and lifetime output;
- private data members and required function signatures;
- correct use of `this`, references, pointers, `nullptr`, `new`, and `delete`;
- no deletion in borrowing functions;
- exact D2 output for all nine fixed tests;
- warning-free C++17 compilation.

Name the PDF exactly `roll_number_oop_lab3.pdf`. Do not submit executable files.

Suggested timing

Part A: automatic lifetime and destructor tracing	20 minutes
Part B: <code>this</code> , references, and pointers	20 minutes
Part C: dynamic objects and ownership	20 minutes
Part D: independent programming problem	35 minutes
Evidence and submission packaging	10 minutes
Total	105 minutes