

# **C++ Classes**

## **Combining Everything You Know**

---

Solutions to All 10 Exercises

Each solution includes a complete working program and an explanation of the key design decisions. Try to solve each exercise yourself first!

# Table of Contents

---

## Part 1 — Absolute Beginner

- 01. Your First Class — Dog (Classes) ★■■■
- 02. Rectangle — Area and Perimeter (Classes) ★■■■
- 03. Counter Class (Classes) ★■■■
- 04. BankAccount — Deposit and Withdraw (Classes) ★■■■
- 05. Student — Grade Average (Classes) ★■■■

## Part 2 — Combining Everything

- 06. Stack Using a Vector (Classes) ★★★
- 07. Linked List of Integers (Classes) ★★★
- 08. Matrix Class with Operator Overloading (Classes) ★★★
- 09. Student Database with File I/O (Classes) ★★★
- 10. Temperature Logger Class (Classes) ★★★

---

## Part 1 — Absolute Beginner (Solutions 1–5)

---

Solution 1 of 10

### Your First Class — Dog



Topic: Classes / Private Members / Methods / Scope Resolution

---

#### Solution Code

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

class Dog {
private:
    std::string name;
    int age;

public:
    void set_values(std::string n, int a); // declared, defined outside

    void bark() { // defined inline
        std::cout << name << " says: Woof!" << std::endl;
    }
};

void Dog::set_values(std::string n, int a) {
    name = n;
    age = a;
}

int main() {
    Dog rex;
    Dog bella;

    rex.set_values("Rex", 4);
    bella.set_values("Bella", 2);

    rex.bark();
    bella.bark();
    return 0;
}
```

#### Explanation

*Making name and age private means they can only be accessed through the class's own methods. Methods defined inside the class body are implicitly inline. set\_values() is declared inside the class but defined outside using Dog:: — the scope resolution operator tells the compiler this definition belongs to Dog and gives it access to the private members. This separation of declaration from definition mirrors the real-world .h / .cpp pattern.*

## Rectangle — Area and Perimeter



Topic: Classes / Methods / Return Values

---

### Solution Code

```
#include <iostream>

class Rectangle {
public:
    double width;
    double height;

    Rectangle(double w, double h) : width(w), height(h) {}

    double area() {
        return width * height;
    }

    double perimeter() {
        return 2 * (width + height);
    }
};

int main() {
    Rectangle r(5.0, 3.0);
    std::cout << "Area: " << r.area() << std::endl;
    std::cout << "Perimeter: " << r.perimeter() << std::endl;
    return 0;
}
```

### Explanation

*Methods defined inside a class body can access all data members directly by name — no need to pass them as parameters. The class bundles data (width, height) and the functions that operate on them (area, perimeter) into a single logical unit, which is the core idea of object-oriented programming.*

## Counter Class



Topic: Classes / State / Methods

---

### Solution Code

```
#include <iostream>

class Counter {
private:
    int count;

public:
    Counter() : count(0) {}

    void increment() { count++; }
    void reset() { count = 0; }
    int getCount() { return count; }
};

int main() {
    Counter c;
    for (int i = 0; i < 5; i++) c.increment();
    std::cout << "Count: " << c.getCount() << std::endl;
    c.reset();
    std::cout << "After reset: " << c.getCount() << std::endl;
    return 0;
}
```

### Explanation

Declaring `count` as `private` means outside code can only read or change it through the public methods. This is encapsulation — the class controls its own state. `getCount()` is a getter: it exposes the value read-only without allowing direct mutation.

## BankAccount — Deposit and Withdraw



Topic: Classes / Encapsulation / Validation

### Solution Code

```
#include <iostream>

class BankAccount {
private:
    double balance;

public:
    BankAccount() : balance(0.0) {}

    void deposit(double amount) {
        if (amount > 0) balance += amount;
    }

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

    double getBalance() { return balance; }
};

int main() {
    BankAccount acc;
    acc.deposit(100);
    std::cout << "Balance: " << acc.getBalance() << std::endl;
    acc.withdraw(30);
    std::cout << "Balance: " << acc.getBalance() << std::endl;
    if (!acc.withdraw(200))
        std::cout << "Withdrawal failed." << std::endl;
    std::cout << "Balance: " << acc.getBalance() << std::endl;
    return 0;
}
```

### Explanation

*withdraw* returns *bool* so the caller can react to failure without needing exceptions. Checking *amount <= balance* before subtracting ensures the account can never go negative — the class enforces this rule every time, regardless of how many places in the program call *withdraw*. This is the key benefit of encapsulation: invariants are guaranteed by the class, not by convention.

## Student — Grade Average



Topic: Classes / Vectors / Methods

---

### Solution Code

```
#include <iostream>
#include <string>
#include <vector>

class Student {
public:
    std::string name;

private:
    std::vector<int> grades;

public:
    Student(std::string n) : name(n) {}

    void addGrade(int g) {
        grades.push_back(g);
    }

    double average() {
        if (grades.empty()) return 0.0;
        int sum = 0;
        for (int g : grades) sum += g;
        return (double)sum / grades.size();
    }

    void report() {
        std::cout << "Student: " << name << std::endl;
        std::cout << "Average: " << average() << std::endl;
    }
};

int main() {
    Student alice("Alice");
    alice.addGrade(85);
    alice.addGrade(92);
    alice.addGrade(76);
    alice.addGrade(90);
    alice.report();
    return 0;
}
```

### Explanation

*The vector is private so grades can only be added through addGrade. average() delegates to itself cleanly from report(), avoiding code duplication. The empty() check prevents division by zero. Casting sum to double before dividing ensures a decimal result.*

---

## Part 2 — Combining Everything (Solutions 6–10)

---

Solution 6 of 10

### Stack Using a Vector

★★★

Topic: Classes / Templates / Vector / Generics

---

#### Solution Code

```
#include <iostream>
#include <vector>

template <typename T>
class Stack {
private:
    std::vector<T> data;

public:
    void push(T val) { data.push_back(val); }

    T pop() {
        if (isEmpty()) {
            std::cout << "Error: stack is empty." << std::endl;
            return T{};
        }
        T val = data.back();
        data.pop_back();
        return val;
    }

    T peek() { return data.back(); }
    bool isEmpty() { return data.empty(); }
    int size() { return (int)data.size(); }
};

int main() {
    Stack<int> s;
    for (int i = 1; i <= 5; i++) s.push(i);
    std::cout << "Size: " << s.size() << std::endl;
    std::cout << "Popped: " << s.pop() << std::endl;
    std::cout << "Popped: " << s.pop() << std::endl;
    std::cout << "Top: " << s.peek() << std::endl;
    return 0;
}
```

#### Explanation

*template makes the class generic — Stack, Stack, Stack all work with the same code. The vector's back() and pop\_back() naturally implement LIFO order. T{} is value-initialisation — it returns 0 for numeric types,*

*empty string for std::string.*

## Linked List of Integers



Topic: Classes / Pointers / Dynamic Memory / Structs

---

### Solution Code

```
#include <iostream>

struct Node {
    int value;
    Node* next;
    Node(int v) : value(v), next(nullptr) {}
};

class LinkedList {
private:
    Node* head;

public:
    LinkedList() : head(nullptr) {}

    ~LinkedList() {
        Node* curr = head;
        while (curr) {
            Node* next = curr->next;
            delete curr;
            curr = next;
        }
    }

    void pushFront(int v) {
        Node* n = new Node(v);
        n->next = head;
        head = n;
    }

    void pushBack(int v) {
        Node* n = new Node(v);
        if (!head) { head = n; return; }
        Node* curr = head;
        while (curr->next) curr = curr->next;
        curr->next = n;
    }

    void print() {
        Node* curr = head;
        while (curr) {
            std::cout << curr->value;
            if (curr->next) std::cout << " -> ";
        }
    }
};
```

```
curr = curr->next;
}
std::cout << " -> null" << std::endl;
}
};

int main() {
    LinkedList list;
    list.pushFront(3);
    list.pushFront(1);
    list.pushBack(5);
    list.print();
    return 0;
}
```

### Explanation

*new allocates a Node on the heap; delete frees it. The destructor walks the list saving next before deleting so the traversal pointer is not lost. pushFront is  $O(1)$ ; pushBack is  $O(n)$  because it must traverse to the end.*

## Matrix Class with Operator Overloading



Topic: Classes / Operator Overloading / 2-D Arrays / Functions

### Solution Code

```
#include <iostream>

class Matrix2x2 {
private:
double data[2][2];

public:
Matrix2x2(double a, double b, double c, double d) {
data[0][0]=a; data[0][1]=b;
data[1][0]=c; data[1][1]=d;
}

Matrix2x2 operator+(const Matrix2x2& o) const {
return Matrix2x2(
data[0][0]+o.data[0][0], data[0][1]+o.data[0][1],
data[1][0]+o.data[1][0], data[1][1]+o.data[1][1]);
}

Matrix2x2 operator*(const Matrix2x2& o) const {
return Matrix2x2(
data[0][0]*o.data[0][0] + data[0][1]*o.data[1][0],
data[0][0]*o.data[0][1] + data[0][1]*o.data[1][1],
data[1][0]*o.data[0][0] + data[1][1]*o.data[1][0],
data[1][0]*o.data[0][1] + data[1][1]*o.data[1][1]);
}

double det() const {
return data[0][0]*data[1][1] - data[0][1]*data[1][0];
}

friend std::ostream& operator<<(std::ostream& os, const Matrix2x2& m) {
os << m.data[0][0] << " " << m.data[0][1] << "\n"
<< m.data[1][0] << " " << m.data[1][1] << "\n";
return os;
}
};

int main() {
Matrix2x2 A(1,2,3,4), B(5,6,7,8);
std::cout << "A+B:\n" << A + B;
std::cout << "A*B:\n" << A * B;
std::cout << "det(A): " << A.det() << std::endl;
return 0;
}
```

```
}
```

**Explanation**

*Overloaded operators (+, \*) are member functions that return a new Matrix2x2 by value. The const keyword at the end of each method signature promises the compiler the method won't modify the object. operator<< must be a friend function (not a member) because its left operand is std::ostream, not Matrix2x2.*

## Student Database with File I/O



Topic: **Classes / Vectors / File I/O / Pointers / Functions**

---

### Solution Code

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

class Student {
public:
    std::string name;
    double gpa;
    Student(std::string n, double g) : name(n), gpa(g) {}
};

class Database {
private:
    std::vector<Student> students;

public:
    void add(Student s) { students.push_back(s); }

    void saveTo(std::string filename) {
        std::ofstream out(filename);
        for (auto& s : students)
            out << s.name << " " << s.gpa << "\n";
    }

    void loadFrom(std::string filename) {
        students.clear();
        std::ifstream in(filename);
        std::string name; double gpa;
        while (in >> name >> gpa)
            students.push_back(Student(name, gpa));
        std::cout << "Loaded " << students.size()
            << " students." << std::endl;
    }

    Student* topStudent() {
        if (students.empty()) return nullptr;
        Student* best = &students[0];
        for (auto& s : students)
            if (s.gpa > best->gpa) best = &s;
        return best;
    }
};
```

```
int main() {
    Database db;
    db.add(Student("Alice", 3.9));
    db.add(Student("Bob", 3.6));
    db.add(Student("Carol", 3.7));
    db.saveTo("students.txt");
    db.loadFrom("students.txt");
    Student* top = db.topStudent();
    if (top)
        std::cout << "Top student: " << top->name
        << " (GPA: " << top->gpa << ")" << std::endl;
    return 0;
}
```

### Explanation

*Two separate classes keep responsibilities clean: Student holds data, Database manages the collection. Using auto& in range-based for loops avoids copying Student objects. topStudent returns a raw pointer into the vector — valid as long as the vector is not modified after the call.*

# Temperature Logger Class



Topic: Classes / Vectors / File I/O / Statistics

---

## Solution Code

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

class TempLogger {
private:
    std::vector<double> readings;
    std::string location;

public:
    TempLogger(std::string loc) : location(loc) {}

    void addReading(double t) { readings.push_back(t); }

    double average() {
        double sum = 0;
        for (double t : readings) sum += t;
        return sum / readings.size();
    }

    double maxTemp() {
        return *std::max_element(readings.begin(), readings.end());
    }

    double minTemp() {
        return *std::min_element(readings.begin(), readings.end());
    }

    void saveTo(std::string filename) {
        std::ofstream out(filename);
        out << location << "\n";
        for (double t : readings)
            out << t << "\n";
    }

    void report() {
        std::cout << "Location: " << location << std::endl;
        std::cout << "Readings: " << readings.size() << std::endl;
        std::cout << "Min: " << minTemp() << std::endl;
        std::cout << "Max: " << maxTemp() << std::endl;
        std::cout << "Average: " << average() << std::endl;
    }
};
```

```
}  
};  
  
int main() {  
    TempLogger logger("Madrid");  
    logger.addReading(36.5);  
    logger.addReading(37.1);  
    logger.addReading(35.8);  
    logger.addReading(38.2);  
    logger.addReading(36.9);  
    logger.report();  
    logger.saveTo("temps.txt");  
    return 0;  
}
```

### Explanation

*std::max\_element* and *std::min\_element* return iterators, so they must be dereferenced with *\** to get the actual value. The class encapsulates all the statistics logic — the caller in *main* just adds readings and calls *report()*, with no knowledge of how the calculations work. *saveTo()* writes the location as a header line so the file is self-describing.