Lecture: Core Concepts of Object-Oriented Programming in C++

1. Introduction to OOP

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data (attributes) and code (methods or functions). OOP aims to model real-world entities by bundling related properties and behaviors into reusable and modular units.

The four pillars of OOP are:

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

These concepts promote code reusability, modularity, maintainability, and scalability.

2. Encapsulation

Definition

Encapsulation is the bundling of data (member variables) and methods (member functions) that operate on that data into a single unit—called a class. It also involves restricting direct access to some of an object's components, which is a means of preventing unintended interference and misuse.

In C++, this is achieved using access specifiers:

- private: accessible only within the class
- protected: accessible within the class and its derived classes
- public: accessible from anywhere

2. Encapsulation

Example

```
#include <iostream>
#include <string>
class BankAccount {
double balance;  // hidden from outside access
public:
    // Constructor
    BankAccount(double initialBalance) : balance(initialBalance) {}
    // Public interface to interact with balance
    void deposit(double amount) {
        if (amount > 0)
            balance += amount;
    }
    void withdraw(double amount) {
        if (amount > 0 && amount <= balance)
            balance -= amount;
    }
// …
```

2. Encapsulation

```
    double getBalance() const {
        return balance;
    }
};

int main() {
    BankAccount account(100.0);
    account.deposit(50.0);
    account.withdraw(30.0);
    std::cout << "Current balance: $" << account.getBalance() << std::endl;
    // account.balance = -1000;  // ✕ Compilation error: 'balance' is
private
    return 0;
}
```

3. Inheritance

Definition

Inheritance allows a class (derived class or subclass) to inherit properties and behaviors (methods and attributes) from another class (base class or superclass). This promotes code reuse and establishes a natural hierarchy.

C++ supports:

- Single inheritance
- Multiple inheritance (a class can inherit from multiple base classes)

3. Inheritance

Example

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

// Base class
class Animal {
protected:
    std::string name;

public:
    Animal(const std::string& n) : name(n) {}

    void eat() const {
        std::cout << name << " is eating." << std::endl;
    }
};
```

3. Inheritance

```
// Derived class
class Dog : public Animal {
public:
    Dog(const std::string& n) : Animal(n) {}

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

int main() {
    Dog myDog("Buddy");
    myDog.eat();   // Inherited from Animal
    myDog.bark();  // Defined in Dog
    return 0;
}
```

Key Idea: Dog reuses Animal’s functionality and adds its own.

4. Polymorphism

Definition

Polymorphism (from Greek: “many forms”) allows objects of different types to be treated through a common interface. In OOP, it enables one interface to represent different underlying forms (data types).

In C++, polymorphism is implemented via:

- Function overloading (compile-time)
- Virtual functions and dynamic dispatch (run-time)
- We focus on run-time polymorphism using virtual functions.

We focus on run-time polymorphism using virtual functions.

4. Polymorphism

Example

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

class Shape {
public:
    virtual ~Shape() = default;  // virtual destructor
    virtual void draw() const = 0;  // pure virtual function → abstract class
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a Circle" << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a Rectangle" << std::endl;
    }
};
```

4. Polymorphism

```
int main() {
    std::vector<Shape*> shapes;
    shapes.push_back(new Circle());
    shapes.push_back(new Rectangle());

    for (const auto& shape : shapes) {
        shape->draw();  // Polymorphic call
    }

    // Clean up
    for (auto* shape : shapes) {
        delete shape;
    }
    return 0;
}
```

Key Idea: The same call shape->draw() behaves differently depending on the actual object type—this is dynamic polymorphism.

5. Abstraction

Definition

Abstraction means hiding complex implementation details and showing only the essential features of an object. It helps reduce programming complexity and effort.

In C++, abstraction is achieved using:

- Classes (to group relevant data and functions)
- Abstract classes (classes with at least one pure virtual function)
- Interfaces (via pure virtual functions)

5. Abstraction

Example

```
#include <iostream>
// Abstract class (interface)
class Database {
public:
    virtual ~Database() = default;
    virtual void connect() = 0;
    virtual void disconnect() = 0;
};

class MySQLDatabase : public Database {
public:
    void connect() override {
        std::cout << "Connecting to MySQL..." << std::endl;
    }

    void disconnect() override {
        std::cout << "Disconnecting from MySQL." << std::endl;
    }
};
```

5. Abstraction

```
// User code interacts only with the abstract interface
void useDatabase(Database& db) {
    db.connect();
    // ... perform operations ...
    db.disconnect();
}

int main() {
    MySQLDatabase db;
    useDatabase(db);  // No need to know internal details of MySQL
    return 0;
}
```

Key Idea: The user works with a high-level interface (Database) without caring about the specific implementation (MySQLDatabase).

6. Summary Table

Concept	Purpose	C++ Mechanism
Encapsulation	Protect data, control access	private/protected/public
Inheritance	Reuse and extend existing code	class Derived : public Base
Polymorphism	Use one interface for multiple types	virtual functions, override
Abstraction	Hide complexity, expose only essentials	Abstract classes, pure virtual funcs

7. Conclusion

Mastering these four OOP principles allows developers to write cleaner, more maintainable, and scalable C++ programs. They form the foundation for modern software design patterns (e.g., Strategy, Factory, Observer) and are essential for large-scale application development.

Remember: OOP is not just about syntax—it’s about design philosophy. Use these tools to model real-world problems effectively.

Lambda Expressions, Function Objects, and Function Pointers in C++

1. Introduction

C++ supports multiple ways to treat code as data—that is, to store, pass, and invoke functions dynamically. Three key mechanisms enable this:

- Function pointers – traditional C-style approach
- Function objects (functors) – C++ classes that overload operator()
- Lambda expressions – concise, inline anonymous functions (introduced in C++11)

These are essential for writing generic, flexible, and expressive code—especially when working with the Standard Template Library (STL), such as std::sort, std::transform, or std::for_each.

2. Function Pointers

What is a Function Pointer?

A function pointer stores the address of a function. It can be passed as an argument, returned from another function, or stored in a data structure.

Syntax:

```
return_type (*pointer_name)(parameter_types);
```

2. Function Pointers

```
#include <iostream>
void greet() {
    std::cout << "Hello from function pointer!\n";
}
int add(int a, int b) {
    return a + b;
}
int main() {
    // Declare and assign function pointers
    void (*func_ptr)() = greet;
    int (*add_ptr)(int, int) = add;

    // Call functions via pointers
    func_ptr();                     // Output: Hello from function pointer!
    std::cout << add_ptr(3, 4);     // Output: 7
    return 0;
}
```

Limitations

Cannot capture local variables from surrounding scope.

Verbose and inflexible for complex logic.

2. Function Pointers

Function pointers allow us to store the address of a function and invoke it dynamically. However, the syntax for declaring function pointers can be verbose and error-prone. To improve readability and maintainability, we can use typedef (or, in modern C++, using) to create an alias for a function pointer type.

This alias can then be used:

- To declare function pointer variables
- As a parameter type in other functions
- To simplify code when working with callbacks or strategy patterns

3. Using typedef to Simplify

We can use typedef to create a readable alias for this type:

```
typedef int (*Operation)(int, int);
```

Now, Operation is a type alias representing “pointer to a function that takes two ints and returns an int”.

Declaring Variables

```
Operation add =
          [](int a, int b) { return a + b; };  //
C++ lambda (capture-less → convertible)
Operation multiply =
          [](int a, int b) { return a * b; };
```

Note: Capture-less lambdas can be implicitly converted to function pointers.

3. Using typedef to Simplify

Or with regular functions:

```
int add(int a, int b) { return a + b; }
int multiply(int a, int b) { return a * b; }

Operation op1 = add;
Operation op2 = multiply;
```

4. Using Function Pointer Type as a Function Parameter

Once we have a typedef alias, we can use it in function declarations to accept function pointers cleanly.

Example: A Generic Calculator Function

```
#include <iostream>

// Step 1: Define the function pointer type
typedef int (*Operation)(int, int);

// Step 2: Define functions that match the signature
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }

// Step 3: Use 'Operation' as a parameter type
int calculate(int x, int y, Operation op) {
    return op(x, y);
}
```

4. Using Function Pointer Type as a Function Parameter

```
int main() {
    int a = 10, b = 4;

    std::cout << "Add: " << calculate(a, b, add) << std::endl;      // 14
    std::cout << "Subtract: " << calculate(a, b, subtract) << std::endl;
    // 6

    // Also works with capture-less lambdas
    Operation divide = [](int a, int b) { return b != 0 ? a / b : 0; };
    std::cout << "Divide: " << calculate(a, b, divide) << std::endl;   // 2

    return 0;
}
```

✅ The calculate function is generic: it works with any operation matching the Operation signature.

5. Modern C++ Alternative: using (C++11+)

Since C++11, using provides a more readable and flexible syntax:

```
using Operation = int (*)(int, int);
```

This is functionally identical to the typedef version but integrates better with templates and is generally preferred in modern code.

6. Real-World Use Case: Callbacks

Function pointer types are commonly used for callbacks—e.g., in event handling, numerical integration, or sorting.

Example: Custom Sort Comparator

```
#include <vector>
#include <algorithm>
#include <iostream>
typedef bool (*Compare)(int, int);
bool ascending(int a, int b) { return a < b; }
bool descending(int a, int b) { return a > b; }
void sortVector(std::vector<int>& v, Compare comp) {
    std::sort(v.begin(), v.end(), comp);
}
int main() {
    std::vector<int> data = {5, 2, 9, 1, 5};
    sortVector(data, descending);
    for (int x : data) std::cout << x << " ";  // 9 5 5 2 1
    return 0;
}
```

Note: In practice, lambdas or functors are often preferred for std::sort, but function pointers (with typedef) remain useful in C-compatible interfaces.

7. Function Objects (Functors)

What is a Functor?

A functor is an object of a class that overloads the function call operator operator(). It behaves like a function but can maintain state.

Why Use Functors?

Can store data between calls.

Often inlined by the compiler → high performance.

Used extensively in STL (e.g., std::less, std::plus).

7. Function Objects (Functors)

Example

```
#include <iostream>
#include <vector>
#include <algorithm>
class Multiplier {
    int factor;
public:
    Multiplier(int f) : factor(f) {}
    int operator()(int x) const {
        return x * factor;
    }
};
int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    // Use functor with std::transform
    std::transform(numbers.begin(), numbers.end(), numbers.begin(), Multiplier(10));
    for (int n : numbers)
        std::cout << n << " ";  // Output: 10 20 30 40 50
    return 0;
}
```

Note: Multiplier(10) creates a temporary object that acts like a function.

8. Lambda Expressions (C++11+)

What is a Lambda?

A lambda expression is a convenient way to define an anonymous function object inline. It's especially useful for short, one-time-use functions.

Basic Syntax

```
[ capture clause ] ( parameters ) -> return_type
{
    // body
}
```

8. Lambda Expressions (C++11+)

- Capture clause [...] specifies how variables from the surrounding scope are captured:
  - [=] – capture by value (all used variables)
  - [&] – capture by reference
  - [x, &y] – capture x by value, y by reference
  - [] – capture nothing
- Return type is often optional (compiler deduces it).

8. Lambda Expressions (C++11+)

Simple Example

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

int main() {
    std::vector<int> v = {5, 2, 8, 1, 9};
    // Sort in descending order using a lambda
    std::sort(v.begin(), v.end(), [](int a, int b) {
        return a > b;
    });
    for (int x : v)
        std::cout << x << " ";  // Output: 9 8 5 2 1
    return 0;
}
```

8. Lambda Expressions (C++11+)

Capturing Variables

```
#include <iostream>

int main() {
    int threshold = 5;

    // Capture threshold by value
    auto is_large = [threshold](int x) {
        return x > threshold;
    };

    std::cout << std::boolalpha << is_large(6);  // true
    return 0;
}
```

8. Lambda Expressions (C++11+)

By default, captured values are const. Use mutable to modify them:

```
auto counter = [count = 0]() mutable {
    return ++count;
};
std::cout << counter(); // 1
std::cout << counter(); // 2
```

Here, count is an init-capture (C++14 feature).

9. Comparison: When to Use What?

Feature	Function Pointer	Functor	Lambda
State	❌ No	✅ Yes	✅ Yes (via capture)
Capture local vars	❌ No	❌ (must pass manually)	✅ Yes
Performance	Good (function call)	Excellent (often inlined)	Excellent (compiled as functor)
Readability	Low for complex logic	Medium	High (for short logic)
STL compatibility	✅ Yes	✅ Yes	✅ Yes

Modern C++ tip: Prefer lambdas for simple, local logic. Use functors for reusable, stateful operations. Use function pointers only when interfacing with C APIs.

10. Practical Use Cases

Example: Filtering with std::copy_if

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

int main() {
    std::vector<int> data = {1, 15, 3, 22, 8};
    std::vector<int> result;
    int min_val = 10;
    std::copy_if(data.begin(), data.end(), std::back_inserter(result),
                 [min_val](int x) { return x >= min_val; });
    for (int x : result)
        std::cout << x << " ";  // Output: 15 22
}
```

10. Practical Use Cases

Example: Storing Lambdas in std::function

```
#include <iostream>
#include <functional>  // for std::function

int main() {
    std::function<int(int, int)> op;
    char choice = '*';
    if (choice == '+')
        op = [](int a, int b) { return a + b; };
    else
        op = [](int a, int b) { return a * b; };
    std::cout << op(3, 4);  // Output: 12
}
```

std::function provides type erasure—it can store any callable with matching signature.

11. Exercises

Exercise 1: Basic Lambda
Write a lambda that takes two double values and returns the smaller one. Use it with std::min_element to find the smallest value in a std::vector<double>.
Exercise 2: Capture and Modify
Create a lambda that captures an integer counter by reference and increments it every time the lambda is called. Call it 3 times and print the final value.
Exercise 3: Functor vs Lambda
Implement a functor Power that raises a number to a fixed exponent (provided in constructor). Then rewrite the same logic using a lambda with init-capture (C++14). Compare both versions.
Exercise 4: Function Pointer Callback
Write a function applyOperation that takes:

- Two int values
- A function pointer int (*op)(int, int)

It should return op(a, b). Test it with add, subtract, and a lambda (hint: you'll need to convert the lambda to a function pointer—only possible if it captures nothing).
Exercise 5: STL with Lambda
Given a std::vector<std::string>, use std::sort with a lambda to sort strings by length (shortest first). If lengths are equal, sort alphabetically.

12. Summary

Function pointers are simple but limited.

- Functors offer state and flexibility but require class definitions.
- Lambdas combine the best of both: conciseness, state (via capture), and seamless STL integration.
- Modern C++ code heavily uses lambdas for algorithms, callbacks, and event handling.

Best Practice: Use lambdas by default for local, short logic. Use std::function when you need to store or pass callables of different types with the same signature.

std::function in C++

std::function is a general-purpose polymorphic function wrapper introduced in C++11 as part of the <functional> header. It can store, copy, and invoke any callable target—including:

- Regular functions
- Function pointers
- Lambdas (even those with captures)
- Function objects (functors)
- Bind expressions (std::bind)
- Member functions (with std::bind or lambdas)

This makes std::function far more flexible than raw function pointers or functors alone.

2. Why Use std::function?

Problem with Raw Function Pointers:

- Cannot store lambdas with captures
- Cannot store functors or member functions
- Type is tied to exact signature—no polymorphism

Advantages of std::function:

✅ Supports any callable with a compatible signature

✅ Enables uniform interface for callbacks, event handlers, and strategies

✅ Integrates seamlessly with STL algorithms, asynchronous code, and APIs

3. Syntax

```
#include <functional>
std::function<Return_Type(Param_Types...)>
```

Examples:

```
std::function<int(int, int)>
// takes two ints, returns int
std::function<void()>
// takes nothing, returns nothing
std::function<double(double)>
// unary function
std::function<bool(const std::string&)>
// predicate
```

Note: std::function is a template class—the template argument is the function signature.

4. Basic Usage Examples

Example 1: Storing Different Callables

```
#include <iostream>
#include <functional>

// Regular function
int add(int a, int b) {
    return a + b;
}

// Functor
struct Multiply {
    int operator()(int a, int b) const {
        return a * b;
    }
};
```

4. Basic Usage Examples

```
int main() {
    std::function<int(int, int)> op;
    // Assign a function
    op = add;
    std::cout << op(3, 4) << std::endl;  // 7
    // Assign a functor
    op = Multiply{};
    std::cout << op(3, 4) << std::endl;  // 12
    // Assign a lambda (even with capture!)
    int factor = 10;
    op = [factor](int a, int b) { return (a + b) * factor; };
    std::cout << op(3, 4) << std::endl;  // 70
    return 0;
}
```

✅ All three callables share the same interface via std::function.

Example 2: Using std::function as a Callback Parameter

```
#include <iostream>
#include <functional>

void executeOperation(int x, int y, std::function<int(int, int)> callback) {
    std::cout << "Result: " << callback(x, y) << std::endl;
}

int main() {
    auto divide = [](int a, int b) -> int {
        return b != 0 ? a / b : 0;
    };

    executeOperation(10, 2, divide);   // Result: 5
    executeOperation(10, 3, [](int a, int b) { return a % b; }); // Result: 1

    return 0;
}
```

Performance Note

std::function may incur small overhead due to type erasure and possible heap allocation (for large lambdas or functors).

For performance-critical inner loops, prefer templates

```
template<typename F>
void call(F f) { f(); } // zero-cost abstraction
```

But for APIs, callbacks, and heterogeneous storage, std::function is the right tool.

Essential Functions from <algorithm> in C++

1. Introduction

The <algorithm> header in C++ provides a rich collection of template functions for performing operations on ranges (typically containers like std::vector, std::array, etc.). These algorithms are:

- Generic: work with any iterator type
- Efficient: often optimized (e.g., using introsort for std::sort)
- Composable: can be combined with lambdas, functors, or custom comparators

We'll cover the most frequently used algorithms in real-world C++ development.

2. Prerequisites

- Include the header: #include <algorithm>
- Most algorithms work on half-open ranges: [first, last)
- Use iterators (e.g., vec.begin(), vec.end())

3. Core Algorithms

3.1 std::sort – Sort a Range

Purpose: Sort elements in ascending order (or custom order).

Signature:

```
void sort(RandomIt first, RandomIt last);
void sort(RandomIt first, RandomIt last, Compare comp);
```

3. Core Algorithms

```
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
    std::vector<int> v = {5, 2, 8, 1, 9};
    // Default: ascending
    std::sort(v.begin(), v.end());     // v = {1, 2, 5, 8, 9}
    // Custom: descending
    std::sort(v.begin(), v.end(), std::greater<int>());     // v = {9, 8, 5, 2, 1}
    // With lambda: sort by absolute value
    std::vector<int> w = {-3, 2, -1, 4};
    std::sort(w.begin(), w.end(), [](int a, int b) {
        return std::abs(a) < std::abs(b);
    });    // w = {-1, 2, -3, 4}
    return 0;
}
```

- Time Complexity: O(N log N)
- Requires: Random-access iterators (e.g., vector, array)

3. Core Algorithms

3.2 std::find – Search for a Value

Purpose: Find the first occurrence of a value.

Signature:

```
Iterator find(Iterator first, Iterator last,
const T& value);
```

3. Core Algorithms

Example:

```
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
    std::vector<int> v = {10, 20, 30, 40};
    auto it = std::find(v.begin(), v.end(), 30);
    if (it != v.end()) {
        std::cout << "Found at index: " << std::distance(v.begin(), it) << std::endl; // 2
    }
    // Find with condition: use `std::find_if`
    auto it2 = std::find_if(v.begin(), v.end(), [](int x) {
        return x > 25;
    });
    if (it2 != v.end()) {
        std::cout << "First >25: " << *it2 << std::endl; // 30
    }
    return 0;
}
```

- std::find_if uses a predicate (callable) instead of a value.

3. Core Algorithms

3.3 std::for_each – Apply Function to Each Element

Purpose: Execute a function for every element.

Signature:

```
Function for_each(InputIt first, InputIt last,
Function f);
```

3. Core Algorithms

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

int main() {
    std::vector<int> v = {1, 2, 3, 4};
    // Print all elements
    std::for_each(v.begin(), v.end(), [](int x)
                          { std::cout << x << " "; });
    // Output: 1 2 3 4
    // Modify elements (use reference in lambda)
    std::for_each(v.begin(), v.end(), [](int& x) { x *= 2; });
    // v = {2, 4, 6, 8}
    return 0;
}
```

- Returns the function object (useful for stateful functors).

3. Core Algorithms

3.4 std::transform – Apply Function and Store Results

Purpose: Apply a function to each element and store the result (in same or another container).

Signature:

```
OutputIt transform(InputIt first, InputIt last,
OutputIt d_first, UnaryOp op);
```

3. Core Algorithms

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
int main() {
    std::vector<double> input = {1.0, 4.0, 9.0, 16.0};
    std::vector<double> output(input.size());
    // Compute square roots
    std::transform(input.begin(), input.end(), output.begin(), [](double x) {
        return std::sqrt(x);
    });
    // output = {1, 2, 3, 4}
    // In-place transform (same container)
    std::transform(output.begin(), output.end(), output.begin(), [](double x) {
        return x * x;
    });
    return 0;
}
```

3. Core Algorithms

- Use std::back_inserter for dynamic output:

```
std::vector<int> result;
std::transform(v.begin(), v.end(),
std::back_inserter(result), [](int x) { return x
* 2; });
```

3. Core Algorithms

3.5 std::copy, std::copy_if – Copy Elements

```
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
    std::vector<int> src = {1, 2, 3, 4, 5, 6};
    std::vector<int> dest;
    // Copy all even numbers
    std::copy_if(src.begin(), src.end(), std::back_inserter(dest), [](int x) {
        return x % 2 == 0;
    });     // dest = {2, 4, 6}
    // Copy first 3 elements
    std::vector<int> first3(3);
    std::copy(src.begin(), src.begin() + 3, first3.begin());     // first3 = {1, 2, 3}
    return 0;
}
```

3. Core Algorithms

3.6 std::count, std::count_if – Count Elements

Purpose: Count occurrences of a value or elements satisfying a condition.

Example:

```
std::vector<int> v = {1, 3, 3, 7, 3};
int c1 = std::count(v.begin(), v.end(), 3); // 3
int c2 = std::count_if(v.begin(), v.end(),
[](int x) { return x > 2; }); // 4
```

3. Core Algorithms

3.7 std::max_element, std::min_element

Purpose: Find iterators to the maximum/minimum element.

Example:

```
std::vector<int> v = {5, 1, 9, 3};
auto max_it = std::max_element(v.begin(), v.end());
std::cout << "Max: " << *max_it << std::endl; // 9
// With custom comparator: longest string
std::vector<std::string> words = {"cat", "elephant", "dog"};
auto longest = std::max_element(words.begin(), words.end(),
    [](const std::string& a, const std::string& b) {
        return a.size() < b.size();
    });
std::cout << "Longest: " << *longest << std::endl; // "elephant"
```

3. Core Algorithms

3.8 std::accumulate – (Note: in <numeric>, but often used with algorithms)

```
#include <numeric>
std::vector<int> v = {1, 2, 3, 4};
int sum = std::accumulate(v.begin(), v.end(),
0); // 10
int prod = std::accumulate(v.begin(), v.end(),
1, std::multiplies<int>()); // 24
```

4. Summary Table

Algorithm	Purpose	Key Notes
std::sort	Sort range	O(N log N), custom comparator
std::find	Find value	Use find_if for conditions
std::for_each	Apply function to each element	For side effects
std::transform	Map elements → new values	Can be in-place
std::copy_if	Copy elements matching predicate	Use back_inserter
std::count_if	Count elements by condition	—
std::max_element	Find largest element	Returns iterator
std::accumulate	Reduce (sum, product, etc.)	Requires <numeric>

5. Exercises

Exercise 1: Sorting by Custom Criterion

Given a std::vector<std::string> words, sort them:

- By length (shortest first)
- If lengths are equal, sort alphabetically

Exercise 2: Transform and Filter

Given a vector of integers, create a new vector containing the squares of all even numbers.

Exercise 3: Find and Replace

Write a function that replaces all occurrences of a value old_val with new_val in a vector.

(Hint: use std::replace or combine find + loop)

Exercise 4: Count Palindromes

Given a vector of strings, count how many are palindromes.

(Hint: write a lambda that checks s == std::string(s.rbegin(), s.rend()))

Exercise 5: Min/Max with Custom Logic

Given a vector of Person structs ({ std::string name; int age; }), find:

- The youngest person
- The person with the longest name