Module 3) Introduction to OOPS Programming

Introduction to C++

Question 1: What are the key differences between Procedural Programming and Object Oriented Programming (OOP)?

Answer: Below are detailed explanations and a comparative table that highlights conceptual, structural, and practical differences between Procedural Programming (often abbreviated POP) and Object-Oriented Programming (OOP).

Conceptual overview:

- Procedural Programming (POP): Focuses on procedures or routines (functions). Programs are structured as sequences of instructions and function calls. Emphasis is on actions (what the program does).
- Object-Oriented Programming (OOP): Focuses on objects which bundle data and behavior together. Emphasis is on modeling real-world entities with state (data members) and behavior (methods).

Key differences table:

Aspect	Procedural Programming (POP)	Object-Oriented Programming (OOP)
Primary unit of decomposition	Functions/procedures that operate on data	Classes/objects that encapsulate data and behavior
Approach	Top-down design: break system into functions and routines	Bottom-up design: define objects then compose them to build systems
Data handling	Data often stored separately and passed to functions; global variables common	Data (attributes) are encapsulated inside objects and accessed via methods
Modularity	Functions grouped by functionality; can be modular but often less cohesive	High cohesion: classes encapsulate related state and behavior
Reusability	Limited; reuse via functions and libraries	High; reuse via inheritance, composition, and polymorphism
Encapsulation & Information Hiding	Weak by default; global data can be modified anywhere	Strong via access specifiers (private, protected, public)

Abstraction	Achieved with functions and modules; less natural mapping to real world	Natural: classes represent abstract data types and interfaces
Polymorphism	Usually absent or ad-hoc via function pointers	Built-in: function overloading, virtual functions, and templates
Typical use-cases	Small programs, utilities, numeric algorithms, system programming (e.g., C)	Large-scale software, GUIs, simulations, enterprise applications (e.g., C++, Java)
Complexity management	Can become difficult as program grows due to global state and function dependencies	Better for large codebases due to encapsulation and separation of concerns

Practical consequences:

- Maintenance: OOP tends to be easier to maintain for large systems because changes are localized inside classes; POP can cause ripple effects when global state or widely used functions change.
- Testing: Objects can be tested in isolation (unit testing) more naturally; functions also can be tested but may depend on shared state.
- Performance: POP can sometimes be simpler and slightly faster due to direct function calls and less indirection; OOP may introduce virtual dispatch overhead where runtime polymorphism is used, but modern compilers optimize aggressively.

Question 2: List and explain the main advantages of OOP over POP

Answer: Object-oriented programming provides many features that make software engineering for medium-to-large projects more robust. Below are the main advantages with explanations:

- **Encapsulation (Data Hiding):**
  - Classes package data and functions that operate on that data. Access specifiers (private, protected, public) restrict direct access to internal representation, reducing bugs caused by unauthorized modification.
  - Example benefit: a class can change its internal data structures without affecting code that interacts with the class through its public interface.
- **Abstraction:**
  - OOP allows designers to expose only essential features of an object while hiding internal complexity. This simplifies reasoning about systems and reduces cognitive load.
- **Inheritance:**
  - Enables creating new classes from existing ones by reusing and extending behavior. This reduces code duplication and supports polymorphic behavior.
  - Example: a 'Vehicle' base class; derived classes 'Car' and 'Bike' inherit common properties.
- **Polymorphism:**

    - Supports writing code that operates on abstract types (interfaces) allowing different concrete implementations to be used interchangeably. Runtime polymorphism via virtual functions enables flexible and extensible designs.
- **Modularity & Maintainability:**
    - OOP leads to highly cohesive modules. Classes provide natural boundaries for modification and debugging, making large codebases manageable.
- **Reusability & Extensibility:**
    - Libraries of classes can be reused across projects. Design patterns and inheritance/composition make it straightforward to extend functionality.
- **Better Mapping to Real-World Problems:**
    - Objects model entities and their interactions, which often matches problem domains closely (e.g., GUI widgets, bank accounts).
- **Safety & Robustness:**
    - Encapsulation + strong typing reduces certain classes of errors (e.g., accidental mutation) and helps enforce invariants.
- **Support for Modern Design Techniques:**
    - OOP fits well with design patterns, SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion), and agile/flexible development.

Caveats:

- OOP is not a silver bullet — poor class design leads to complicated code.
- Overuse of inheritance instead of composition can make code brittle.
- Some domains (e.g., low-level system programming) still prefer procedural approaches.

Question 3: Explain the steps involved in setting up a C++ development environment.

Answer: Setting up a C++ environment means installing and configuring tools required to write, compile, run, and debug C++ programs. Below are step-by-step instructions for common platforms and additional tips for a robust workflow.

1) Choose and install a compiler:

- Windows:
  →Install MinGW-w64 (GCC for Windows) or Microsoft Visual C++ (MSVC) via Visual Studio.
  → MinGW: download installer, choose architecture (x86_64 recommended), and add `bin` folder to PATH.
  → Visual Studio: download Community edition (includes MSVC compiler and debugger).
- Linux:
  → Use package manager: `sudo apt install build-essential` (Debian/Ubuntu) or `sudo dnf install gcc-c++` (Fedora).
  → Clang is an alternative (`sudo apt install clang`).

- macOS:
  → Install Xcode command line tools: `xcode-select --install` which provides clang and make.
  → Or install GCC/Clang via Homebrew (`brew install gcc`).

2) Choose and install an IDE or text editor:

- Full IDEs: Visual Studio (Windows), CLion (cross-platform, commercial), Code::Blocks, Eclipse CDT.
- Lightweight editors: Visual Studio Code (with C++ extensions), Sublime Text, Vim/Neovim (with plugins).
- Install language extensions for features like IntelliSense, linting, formatting, and debugging integration.

3) Configure build tools and debugger:

- On small projects, compile directly with the compiler: `g++ main.cpp -o main`.
- For multi-file projects, use build systems: Make (Makefile), CMake (recommended for portable projects), Meson, or Ninja.
- Configure debugger integration (GDB for GCC/Clang on Linux/macOS; WinDbg or Visual Studio debugger on Windows).

4) Create a sample project and test toolchain:

Compile and run a simple program to ensure everything works:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, C++ world!" << endl;
    return 0;
}
```

Build commands examples:

Linux/macOS using g++: `g++ -std=c++17 main.cpp -O2 -Wall -Wextra -o main`

Windows using MSVC (Developer Command Prompt): `cl /EHsc main.cpp`

5) Recommended additional tools and configuration:

- Version control: Git (set up GitHub/GitLab repository).
- Code formatter: clang-format or Artistic Style for consistent style.
- Static analyzers: clang-tidy, cppcheck to catch bugs early.
- Package/dependency manager: vcpkg or Conan for third-party libraries.
- Continuous Integration: GitHub Actions, GitLab CI or similar to run builds and tests.

6) Set language standard and flags:

- Specify a C++ standard (e.g., `-std=c++17` or `-std=c++20`).
- Enable important warnings: `-Wall -Wextra -Wpedantic` to improve code quality.
- Use optimization (`-O2` or `-O3`) for release builds and `-g` for debug builds.

Question 4: What are the main input/output operations in C++? Provide examples.

Answer: C++ I/O can be broadly classified into console I/O, string I/O (in-memory), and file I/O. The standard library provides stream abstractions (iostream) that are type-safe, extensible, and composable.

1) Console I/O: `std::cin` and `std::cout`

- `std::cout` (output stream): Used to send data to the standard output (console). Supports stream insertion operator `<<` and formatting via manipulators (e.g., `std::endl`, `std::setw`).
- `std::cin` (input stream): Used to read formatted input from standard input using extraction operator `>>`. Note that `>>` stops at whitespace; use `std::getline` for entire lines.

```
// Example: console I/O
#include <iostream>
#include <string>
using namespace std;
int main() {
    string name;
    cout << "Enter your name: ";
    getline(cin, name); // reads full line including spaces
    cout << "Hello, " << name << "!" << endl;
    return 0;
}
```

2) String streams (`std::stringstream`, `std::istringstream`, `std::ostringstream`):

Useful for parsing or building strings using stream operators. They treat string buffers like streams.

```
// Example: stringstream
#include <sstream>
#include <string>
#include <iostream>
using namespace std;

int main() {
    string data = "10 20 30";
    istringstream iss(data);
```

```
    int a, b, c;
    iss >> a >> b >> c; // parse ints from string
    cout << (a + b + c) << endl; // 60
}
```

3) File I/O: `std::ifstream`, `std::ofstream`, `std::fstream`

- `std::ifstream`: input file stream for reading files.
- `std::ofstream`: output file stream for writing files.
- `std::fstream`: bidirectional file stream.
- Always check stream state (e.g., `if (infile) { ... }`) and close streams when done (they close automatically in destructor).

```
// Example: file I/O
#include <fstream>
#include <iostream>
using namespace std;

int main() {
    ofstream ofs("output.txt");
    if (!ofs) { cerr << "Cannot open file for writing" << endl;
return 1; }
    ofs << "Line 1\n" << "Line 2\n";
    ofs.close();
    return 0;
}
```

4) I/O manipulators and formatting:

Include `<iomanip>` for manipulators like `std::setw`, `std::setprecision`, `std::fixed`, `std::scientific`, `std::left`/`std::right`. These help produce formatted textual output.

5) Low-level I/O:

For performance-critical apps, POSIX read/write or platform-specific APIs may be used. Also, buffered I/O (std::istream::rdbuf) and `std::getline` for line-based processing are common. For binary I/O, use `ofstream`/`ifstream` with `ios::binary`.

Variables, Datatypes and Operators

Question 1: What are the different datatypes available in C++? Explain with examples.

Answer: C++ provides a rich set of data types. They can be broadly categorized into fundamental (built-in) types, derived types, and user-defined types. Below is a structured breakdown with explanation and examples.

Fundamental (Primitive) types:

- Integer types: `short`, `int`, `long`, `long long` — may be signed or unsigned. Use `sizeof(type)` to find size on a given platform. Example: `int age = 21;`
- Character types: `char`, `signed char`, `unsigned char`, `wchar_t`, `char16_t`, `char32_t` — used for characters/text. Example: `char ch = 'A';`
- Floating-point types: `float`, `double`, `long double` — for real numbers. Example: `double pi = 3.141592653589793;`
- Boolean: `bool` — stores `true` or `false`. Example: `bool ok = true;`
- Void: `void` — used for functions that return nothing and for pointers to unspecified types (`void*`).

Derived types:

- Pointers: `int* p` — store memory addresses. Example: `int x=5; int* p=&x;`
- References: `int& r = x` — an alias to another object; cannot be null and must be initialized.
- Arrays: `int arr[10]` — contiguous collection of elements of same type.
- Functions / function pointers: function types and pointers to functions.
- Pointers-to-members and other compound types.

User-defined types:

- `struct` — aggregate of fields (public by default).
- `class` — similar to struct but members are private by default; supports methods, constructors, destructors, access control.
- `union` — overlapping storage for different types.
- `enum` and enum class — enumerated types; `enum class` provides scoped and strongly typed enumerations.
- `typedef` and using — create type aliases.

Modern additions (C++11 onward):

- `auto` — type deduction by compiler from initializer. Example: `auto x = 3.14; // double`

- `decltype` — deduce type from an expression.
- Smart pointers (`std::unique_ptr`, `std::shared_ptr`) manage dynamic memory safely.
- `std::array` and `std::vector` — safer container alternatives to raw arrays.

Examples:

```
// Example
int i = 42;
double d = 3.14;
char c = 'Z';
bool flag = false;
int *p = &i;
std::vector<int> v = {1,2,3};
std::array<int,3> a = {4,5,6};
auto inferred = 123; // int
```

Question 2: Explain the difference between implicit and explicit type conversion in C++.

Answer: Type conversion (casting) is the process of converting a value from one type to another. C++ supports both implicit conversions (performed by the compiler automatically) and explicit conversions (performed by the programmer).

Implicit conversion (type promotion):

- Happens automatically when the compiler can safely (or by language rules) convert one type to another.
- Common in arithmetic expressions (`int` promoted to `double`), function calls with compatible parameter types, and promotions of smaller integer types to `int`.
- The compiler follows the usual arithmetic conversions (to align operands) and integral promotions.
- Pitfall: implicit narrowing (e.g., double → int) usually does not happen automatically without a warning in certain contexts (list-initialization forbids narrowing).

Explicit conversion (casting):

- Done deliberately by the programmer to force a conversion using a cast.
- C++ provides several cast operators: `static_cast<T>(expr)`, `reinterpret_cast<T>(expr)`, `const_cast<T>(expr)`, and `dynamic_cast<T>(expr)` (for polymorphic types).
- C-style casts `(T)expr` and function-style casts `T(expr)` are also available but are less safe and harder to reason about.

- Use `static_cast` for well-defined conversions (numeric conversions, pointer up/down-casts when safe), `dynamic_cast` for safe downcasts in class hierarchies (requires RTTI), `reinterpret_cast` for low-level reinterpretation of bits (dangerous), and `const_cast` to add/remove const-qualifiers.

Examples:

```
// Implicit conversion
int a = 10;
double b = a; // ok: int -> double

// Explicit conversion
double x = 3.9;
int y = static_cast<int>(x); // y becomes 3 (fraction discarded)

// C-style cast (not recommended)
int z = (int)x;
```

Safety and best practices:

- Prefer `static_cast` and other C++ casts over C-style casts for clarity and safety.
- Avoid `reinterpret_cast` unless you know the exact bit-level representation and alignment requirements.
- Be mindful of integer overflow, precision loss, and undefined behavior when converting between incompatible types.

Question 3: What are the different types of operators in C++? Provide examples of each.

Answer: Operators in C++ are special symbols used to perform operations on variables and values. They are essential components of the C++ language, enabling developers to implement logic, manipulate data, and perform computations.

C++ supports a wide variety of operators, which are categorized as follows:

1. Arithmetic Operators

- These operators perform basic mathematical operations.

Operators:

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus)

Example:

```
int a = 10, b = 3;
int sum = a + b;         // 13
int diff = a - b;        // 7
int product = a * b;     // 30
int quotient = a / b;    // 3
int remainder = a % b;   // 1
```

2. Relational (Comparison) Operators

➢ Used to compare two values.

Operators:

- == (Equal to)
- != (Not equal to)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)
- <= (Less than or equal to)

Example:

```
int a = 5, b = 10;
bool isEqual = (a == b);          // false
bool isNotEqual = (a != b);       // true
bool isGreater = (a > b);         // false
bool isLesser = (a < b);          // true
bool isGreaterEqual = (a >= b);   // false
bool isLesserEqual = (a <= b);    // true
```

3. Logical Operators

➢ Used to combine or invert logical statements.

Operators:

- && (Logical AND)
- || (Logical OR)
- ! (Logical NOT)

Example:

```
bool a = true, b = false;
bool result1 = a && b;  // false
bool result2 = a || b;  // true
bool result3 = !a;      // false
```

4. Assignment Operators

➢ Used to assign or update values in variables.

Operators:

- = (Simple assignment)
- += (Add and assign)
- -= (Subtract and assign)
- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulus and assign)

Example:

```
int a = 10;
a += 5;   // a = 15
a -= 3;   // a = 12
a *= 2;   // a = 24
a /= 4;   // a = 6
a %= 5;   // a = 1
```

5. Increment and Decrement Operators

➢ Used to increase or decrease a value by one.

Operators:

- ++ (Increment)
- -- (Decrement)

Types:

- Prefix: ++a, --a
- Postfix: a++, a--

Example:

```
int a = 5;
int b = ++a;  // a = 6, b = 6 (prefix)
int c = a--;  // a = 5, c = 6 (postfix)
```

6. Bitwise Operators

➢ Used to perform bit-level operations.

Operators:

- & (Bitwise AND)
- | (Bitwise OR)
- ^ (Bitwise XOR)
- ~ (Bitwise NOT)
- << (Left shift)
- >> (Right shift)

Example:

```
int a = 5;  // Binary: 0101
int b = 3;  // Binary: 0011


int andResult = a & b;   // 1  (0001)
int orResult = a | b;    // 7  (0111)
int xorResult = a ^ b;   // 6  (0110)
int notResult = ~a;      // -6 (Two's complement)
int leftShift = a << 1;  // 10 (1010)
int rightShift = a >> 1; // 2  (0010)
```

7. Conditional (Ternary) Operator

- A shorthand for if-else conditions.

Syntax: condition ? expression1 : expression2;

Example:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;  // max = 20
```

8. Sizeof Operator

- Returns the size of a variable or data type in bytes.

Example:

```
int a = 5;
size_t size = sizeof(a);            // Typically 4
size_t sizeDouble = sizeof(double); // Typically 8
```

9. Typecast Operator

- Used to convert one data type into another.

Syntax: (type) expression;

Example:

```
int a = 10, b = 3;
double result = (double)a / b; // 3.33333
```

10. Scope Resolution Operator (::)

- Used to access global variables or to define class members outside the class.

Example:

```
int x = 10; // Global variable

class Example {
public:
    static int x;
};

int Example::x = 20;
```

```
int main() {
    std::cout << ::x << std::endl;        // Access global x
    std::cout << Example::x << std::endl; // Access class x
}
```

11. Member Access Operators

- Used to access members of structures, classes, or unions.

Operators:

- . (Dot operator)
- -> (Arrow operator)

Example:

```
struct Point {
    int x, y;
};

Point p1;
p1.x = 10;
p1.y = 20;

Point* p2 = &p1;
int x1 = p2->x; // Access using arrow operator
```

12. Pointer Operators

- Used in pointer handling.

Operators:

- * (Dereference operator)
- & (Address-of operator)

Example:

```
int a = 10;
int* ptr = &a;   // Get address of a
int b = *ptr;    // Dereference to get value of a
```

13. Comma Operator

- Allows multiple expressions to be evaluated in a single statement.

Example:

```
int a, b, c;
a = (b = 3, c = b + 2); // b = 3, c = 5, a = 5
```

14. New and Delete Operators

- Used for dynamic memory allocation and deallocation.

Example:

```
int* ptr = new int;  // Allocate memory
*ptr = 10;
delete ptr;          // Deallocate memory
```

15. Cast Operators (C++ Style)

- C++ provides type-safe casting options.

Types:

- static_cast<type>(expression)
- dynamic_cast<type>(expression)
- const_cast<type>(expression)
- reinterpret_cast<type>(expression)

Example:

```
int a = 10;
double b = static_cast<double>(a); // Convert int to double
```

Question 4: Explain the purpose and use of constants and literals in C++.

Answer: Constants and literals represent fixed values in the source code. They are used to provide readable, safe, and sometimes optimized code.

Constants:

- `const` keyword: declares variables whose values cannot be changed after initialization. Example: `const double PI = 3.14159;`
- `constexpr` (C++11 onward): indicates compile-time constant expressions and allows the compiler to evaluate values at compile time where possible. Example: `constexpr int MAXN = 1000;`
- Benefits: safety (prevent accidental modification), potential compiler optimizations, self-documentation.

Literals:

- Numeric literals: integers (`42`), floating-point (`3.14`), with suffixes (e.g., `42u`, `1.5f`, `2LL`).
- Character literals: `'A'`, wide-char `L'A'`, Unicode char literals such as `u'\u00E9'` (char16_t) or `U'\U0001F600'` (char32_t).
- String literals: `"hello"`, raw string literals `R"(raw
text)"`.
- Boolean literals: `true` and `false`.
- Null pointer literal: `nullptr` (since C++11) — safer than `NULL` or `0`.

Usage tips:

- Prefer `constexpr` for values known at compile time (enables more optimization).
- Use named constants rather than magic numbers to improve readability and maintainability.
- Be careful with literal sizes and signedness when mixing with variables (e.g., unsigned vs signed arithmetic).

Control Flow Statements

Question 1: What are conditional statements in C++? Explain the if-else and switch statements.

Answer: Conditional statements control the flow of execution depending on boolean conditions. The two primary conditional constructs in C++ are `if-else` and `switch`.

- **if-else statement:**

  - `if` evaluates a boolean expression. If true, it executes the associated block. Optionally followed by `else` to handle the false branch.
  - Syntax variations include single `if`, `if-else`, and `if-else if-else` chains.
  - Example and notes:

```
// if-else example
int x = 10;
if (x > 0) {
    cout << "Positive" << endl;
} else if (x == 0) {
    cout << "Zero" << endl;
} else {
    cout << "Negative" << endl;
}
```

- **Switch statement:**

  - `switch` is a multi-way branch statement that selects a case based on an integral or enum value. Each `case` label must be a compile-time constant.
  - Important details: `switch` uses fall-through semantics by default (execution continues into subsequent cases unless a `break` is used).
  - `switch` is typically used for multiple discrete branches and can be more efficient/readable for such scenarios.
  - Limitations: `switch` does not work directly with floating-point values or strings (unless using hash-based approaches or switch on enums/integers).

```
// switch example
int ch = 2;
switch (ch) {
    case 1:
        cout << "Option 1" << endl;
        break;
    case 2:
        cout << "Option 2" << endl;
```

```
            break;
        default:
            cout << "Invalid option" << endl;
    }
```

Question 2: What is the difference between for, while and do-while loops in C++?

Answer: All three are repetition constructs but they differ in how and when they evaluate the loop condition and typical use-cases:

Aspect	for Loop	while Loop	do-while Loop
Definition	Used when the number of iterations is known in advance.	Used when the number of iterations is not known in advance.	Similar to while, but guarantees at least one execution of the loop body.
Syntax	for (initialization; condition; update) { /* code */ }	while (condition) { /* code */ }	do { /* code */ } while (condition);
When Condition is Checked	Before entering the loop (pre-test loop).	Before entering the loop (pre-test loop).	After executing the loop body (post-test loop).
Minimum Executions	Zero – loop may not run if condition is false initially.	Zero – loop may not run if condition is false initially.	One – loop body is executed at least once, even if condition is false.
Initialization	Done in the loop header.	Done before the loop starts.	Done before the loop starts.
Update/Increment	Done in the loop header.	Done inside the loop body.	Done inside the loop body.
Readability	More concise and readable when the loop control is simple and known.	More flexible, but can be harder to read if not properly structured.	Less commonly used; ideal when the first iteration must always occur.
Use Case	Iterating a fixed number of times (e.g., arrays, counters).	Repeating based on a condition evaluated before the first iteration.	Menu-driven programs, input validation – when loop must execute at least once.
Example Code	for (int i = 1; i <= 5; i++) {	int i = 1; while (i <= 5)	int i = 1; do { std::cout <<

	std::cout << i << " "; }	{ std::cout << i << " "; i++; }	i << " "; i++; } while (i <= 5);
Output of Example	1 2 3 4 5	1 2 3 4 5	1 2 3 4 5
Control Structure Location	All in one line (header).	Spread out – condition in header, updates in body.	Spread out – condition at end, updates in body.
Best For	Counting loops with predictable iteration limits.	Conditional loops where exit condition is dynamic.	Executing code at least once, then checking condition.

Extra: C++11 range-based for loop:

`for (auto &elem : container)` is convenient for iterating containers without an index and avoids errors related to iterator bounds.

Question 3: How are break and continue statements used in loops? Provide examples.

Answer: `break` and `continue` control iteration flow within loops:

- `break`: Immediately exits the innermost loop or `switch` and continues execution after it. Useful for early termination when a condition is met.
- `continue`: Skips the remainder of the current iteration and moves control to the loop's next iteration (checking the loop condition).

Example

```
// break and continue example
for (int i = 1; i <= 10; ++i) {
    if (i % 2 == 0) continue; // skip even numbers
    if (i > 7) break; // stop when i > 7
    cout << i << " "; // prints: 1 3 5 7
}
```

Notes:

- In nested loops, `break`/`continue` affect only the innermost loop. To break out of outer loops, use flags or `goto` (not recommended) or restructure code into functions and `return`.
- `continue` in `for` loop still executes the increment expression (`++i`) after skipping the body. In `while` loop, control goes directly to condition check.

Question 4: Explain nested control structures with an example.

Answer: Nested control structures are control statements placed within other control statements (e.g., a loop inside a loop, or an if inside a loop). They are used to express multi-dimensional iteration or conditional logic that depends on multiple variables.

Example: Nested loops to print a multiplication table and complexity analysis:

```
// Nested loops example: multiplication table
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 5; ++i) {
        for (int j = 1; j <= 5; ++j) {
            cout << (i * j) << "\t";
        }
        cout << endl;
    }
    return 0;
}
```

Complexity:

- If outer loop runs N times and inner loop runs M times, time complexity is O(N*M). For square nested loops with both running N times, complexity is O(N^2).
- Be mindful of nested conditionals and loops as they can lead to combinatorial explosion for large inputs.

Functions and Scope

Question 1: What is a function in C++? Explain the concept of function declaration, definition and calling.

Answer: A function is a named block of code that performs a specific task; it can accept parameters and optionally return a value. Functions enable modularity, code reuse, and better organization of logic.

Declaration (prototype):

- A declaration introduces the function's name, return type, and parameter types to the compiler without providing the body. Example: `int add(int a, int b);`
- Prototypes are typically placed in header files (.h/.hpp) so that multiple translation units can use the function.

Definition:

- The definition provides the function body (implementation). Example: `int add(int a, int b) { return a + b; }`
- A function must be defined exactly once across the program (unless inline or templated with appropriate rules).

Calling a function:

- Invoke a function by its name and supply required arguments: `int s = add(3,4);`
- Parameter passing: pass-by-value (default) copies the argument; pass-by-reference (`int&`) passes an alias; pass-by-const-reference (`const Type&`) avoids copying while preventing modification.

Example

```
// header-like declaration
int add(int a, int b);

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

int main() {
    int result = add(2, 3);
    cout << result << endl; // 5
}
```

Question 2: What is the scope of variables in C++? Differentiate between local and global scope.

Answer: Scope refers to the region of program where an identifier (variable, function, type) is visible and can be referred to. Lifetime (storage duration) is related but distinct: it denotes how long an object exists in memory.

Local scope:

- Variables declared inside a function or block (`{ ... }`) have block scope and are visible only within that block. Example: `int x = 0;` inside `main()`.
- Automatic storage duration (unless declared `static`) — their lifetime is tied to block execution.
- Advantages: avoids name clashes, safer and easier to reason about.

Global scope:

- Variables declared outside all functions are global and visible across the translation unit (and possibly other units using `extern`).
- Global variables have static storage duration — they exist for the lifetime of the program.
- Disadvantages: can lead to tight coupling, harder to reason about and test; prefer minimizing global state.

Other scopes and storage:

- `static` local variables: retain value between calls, but have function/block scope.
- Namespace scope (`namespace` keyword) groups names and avoids global collisions.
- Class scope: member variables are scoped within their class; access controlled by specifiers.

Question 3: Explain recursion in C++ with an example.

Answer: Recursion occurs when a function calls itself to solve a subproblem. A correct recursive function must have:

- A base case (one or more) that terminates recursion.
- A recursive case that reduces the problem towards the base case.

Example: factorial computation (illustrates recursion and stack behavior):

```
// Recursive factorial
int factorial(int n) {
    if (n <= 1) return 1; // base case
    return n * factorial(n - 1); // recursive case
}

int main() {
    cout << factorial(5) << endl; // 120
```

```
}
```

Complexity and considerations:

- Time complexity depends on recurrence (factorial O(n)).
- Recursive calls consume stack frames; for deep recursion, risk of stack overflow exists. Tail recursion (if supported) can mitigate this by reusing frames, but standard C++ does not guarantee tail-call optimization.
- Some recursive algorithms (e.g., naive Fibonacci) have exponential complexity; transform to iterative or use memoization/dynamic programming for efficiency.

Question 4: What are the function prototypes in C++? Why are they used?

Answer: A function prototype (declaration) informs the compiler about a function's name, return type, and parameter types before its first use. This allows modular compilation, type checking, and separate compilation of source files.

Reasons to use prototypes:

- Enable the compiler to check calls for correct number and types of arguments.
- Allow functions to be called before their definitions appear (useful in header/source separation).
- Support separate compilation: header files declare prototypes while `.cpp` files provide definitions.
- Help avoid implicit int return types or implicit conversions that could mask bugs.

Example:

```
// header.h
int sum(int a, int b);

// main.cpp
#include "header.h"
int main(){ cout << sum(2,3); }

// header.cpp
int sum(int a,int b){ return a+b; }
```

Note: In modern C++ projects, prototypes live in header files and are included where needed; use include guards or `#pragma once` to prevent multiple inclusions.

Arrays and Strings

Question 1: What are arrays in C++? Explain the differences between single-dimensional and multi-dimensional arrays.

Answer: An array is a contiguous block of memory that holds elements of the same type. Arrays provide O(1) access to elements by index and are widely used for fixed-size collections.

Single-dimensional arrays (1D):

- Linear sequence of elements indexed from 0 to n-1.
- Declaration example: `int arr[5] = {1, 2, 3, 4, 5};`
- Memory layout is contiguous, supports pointer arithmetic (`arr + i` points to element `i`).

Multi-dimensional arrays (2D, 3D, ...):

- Conceptually arrays of arrays. A 2D array looks like a matrix with row-major storage in C++.
- Declaration example: `int mat[3][4];` — 3 rows, 4 columns.
- Access uses two indices: `mat[i][j]`. Memory layout is contiguous for row-major order: rows stored one after another.

Dynamic and STL alternatives:

- For flexible sizing, prefer `std::vector<T>` (1D) or `std::vector<std::vector<T>>` (2D) for dynamic matrices.
- `std::array<T,N>` is a safer wrapper around fixed-size arrays (supports range checking in debug builds and standard container interfaces).

Question 2: Explain string handling in C++ with examples.

Answer: C++ supports two main paradigms for text: C-style null-terminated strings (character arrays) and the safer `std::string` class. Use `std::string` for most high-level tasks.

C-style strings (char arrays):

- Represented as a sequence of characters terminated by a null character `\0`.
- Common functions from `<cstring>`: `strlen`, `strcpy`, `strncpy`, `strcat`, `strcmp`, `strchr`, `strstr`.
- Pitfalls: buffer overruns, forgetting null-termination, and manual memory management for dynamic allocations.

std::string (C++ standard library):

- Dynamic, resizable string with convenient methods (`length`, `substr`, `find`, `replace`, `append`, `insert`).
- Works with streams directly using `<<` and `>>` and supports `std::getline` for lines.
- Internals manage memory; capacity vs size: `reserve()` controls capacity to avoid frequent reallocations.

```
// std::string example
#include <string>
#include <iostream>
using namespace std;
int main(){
    string s = "Hello";
    s += ", world";
    cout << s.substr(0,5) << endl; // Hello
    cout << s.find("world") << endl; // index of w
}
```

Question 3: How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

Answer: Arrays can be initialized at the point of declaration using brace-enclosed initializers, or element-wise later. For dynamic arrays, initialization depends on the allocation method.

1D array initialization:

- `int arr[5] = {1, 2, 3, 4, 5};` — full init.
- `int arr[5] = {1, 2};` — remaining elements zero-initialized (3 zeros).
- `int arr[] = {10, 20, 30};` — size deduced as 3.
- `std::vector<int> v = {1,2,3};` — dynamic and safer.

2D array initialization:

- `int mat[2][3] = {{1,2,3}, {4,5,6}};` — explicit rows.
- `int mat[2][3] = {1,2,3,4,5,6};` — flattened initializer fills row-major.
- `vector<vector<int>> mat(2, vector<int>(3, 0));` — dynamic 2D vector initialized with zeros.

Example:

```
// 1D and 2D initialization examples
int a[3] = {10, 20, 30};
int b[5] = {1, 2}; // {1,2,0,0,0}
int mat[2][2] = {{1,2},{3,4}};
std::vector<std::vector<int>> dyn(3, std::vector<int>(4, -1));
```

Question 4: Explain string operations and functions in C++.

Answer: `std::string` provides a rich API for manipulating text. Below are frequently-used operations and function examples:

- **Concatenation:** `s1 + s2` or `s1.append(s2)` or `s1 += s2`. Example: `string s = "a" + string("b");`
- **Length:** `s.size()` or `s.length()` returns number of characters.
- **Substring:** `s.substr(pos, len)` returns a substring.
- **Find:** `s.find("needle")` returns position or `string::npos` if not found.
- **Replace:** `s.replace(pos, len, newStr)` replaces part of the string.
- **Insert/erase:** `s.insert(pos, other)` and `s.erase(pos, len)` modify string contents.
- **Conversion:** `stoi`, `stol`, `stod` convert strings to numbers; `to_string` converts numbers to strings.
- **C-string access:** `s.c_str()` returns a `const char*` for interoperability with C APIs.

Example

```
// std::string common operations
std::string s = "Hello";
s += " World"; // concatenation
size_t pos = s.find("World");
std::string sub = s.substr(0, 5); // "Hello"
int val = std::stoi("123"); // 123
```

Introduction to Object-Oriented Programming (OOP)

Question 1: Explain the key concepts of Object-Oriented Programming

Answer: OOP has four primary pillars: encapsulation, abstraction, inheritance, and polymorphism. Each plays a specific role in designing modular, reusable, and maintainable software.

Class:

- A class is a user-defined type that encapsulates data members and member functions.
- Key class features: constructors, destructors, access specifiers, member functions, static members, const member functions, and operator overloads

Object:

- An object is a concrete instance of a class.
- It is a must for accessing the members and properties of a class

Encapsulation:

- Combines data (attributes) and functions (methods) into a single unit (class).
- Controls access via access specifiers (private, protected, public).
- Helps enforce invariants and prevents external code from depending on internal representation.

Abstraction:

- Exposes only relevant behavior through interfaces while hiding implementation details.
- Helps manage complexity by representing essential features and ignoring unnecessary details.

Inheritance:

- Enables a class (derived) to inherit properties and behavior from another class (base), promoting reuse.
- Types of inheritance: single, multiple, multilevel, hierarchical, hybrid. Use `virtual` inheritance to resolve the diamond problem.

Polymorphism:

- Compile-time polymorphism: function overloading and templates.
- Runtime polymorphism: virtual functions and dynamic dispatch enabling different behaviors for derived classes through base pointers/references.
- Virtual destructors are essential to ensure derived destructors are called when deleting via base pointers.

Other useful OOP concepts:

- Composition (has-a) vs Inheritance (is-a): composition often preferred for flexible designs.
- Design patterns (Factory, Singleton, Strategy, Observer, etc.) leverage OOP principles to solve recurring design problems.
- SOLID principles guide OOP designs to be maintainable and scalable.

Question 2: What are classes and objects in C++? Provide an example.

Answer: A class is a user-defined type that encapsulates data members and member functions. An object is a concrete instance of a class.

Key class features: constructors, destructors, access specifiers, member functions, static members, const member functions, and operator overloads.

Example with constructors, destructor, getters/setters, and copy semantics:

```
// Class example
#include <iostream>
#include <string>
using namespace std;

class Student {
private:
    string name;
    int age;
public:
    // Constructor
    Student(const string &n, int a) : name(n), age(a) {}
    // Getter
    string getName() const { return name; }
    int getAge() const { return age; }
    // Setter
    void setAge(int a) { age = a; }
    // Method
    void display() const { cout << name << " (" << age << ")" <<
endl; }
};

int main() {
    Student s("Alice", 20);
    s.display();
}
```

Question 3: What is inheritance in C++? Explain with an example.

Answer: Inheritance allows a new class (derived) to acquire properties and behavior of an existing class (base), enabling reuse and polymorphism. Access control (public/protected/private) affects how base members are inherited and accessed.

Simple example showing single inheritance and overriding a virtual function:

```
// Inheritance example
#include <iostream>
using namespace std;

class Animal {
public:
    virtual void speak() const { cout << "Animal sound" << endl;
}
    virtual ~Animal() = default; // virtual destructor
};

class Dog : public Animal {
public:
    void speak() const override { cout << "Woof" << endl; }
};

int main() {
    Animal *a = new Dog();
    a->speak(); // prints "Woof" due to polymorphism
    delete a;
}
```

Notes:

- Member visibility: `public` inheritance preserves public members as public, `protected` as protected. `private` inheritance makes inherited members private.
- Use `virtual` for base class methods intended for overriding; use `override` in derived classes to express intent and enable compiler checks.
- Multiple inheritance is supported in C++ but should be used with care due to complexity and potential ambiguities.

Question 4: What is encapsulation in C++? How is it achieved in classes?

Answer: Encapsulation is the bundling of data and methods that operate on that data within a class, together with access control to restrict direct access to some members. This enforces invariants and reduces coupling.

Mechanisms to achieve encapsulation:

- Access specifiers: `private`, `protected`, `public` control visibility of members.
- Getters and setters (accessor/mutator functions) provide controlled access, validation, and side-effects when needed.
- Friend functions/classes can be used sparingly to grant special access while keeping most members hidden.
- `const` member functions promise not to modify object state and support read-only access.

```
// Encapsulation example
class BankAccount {
private:
    double balance;
public:
    BankAccount(double b = 0.0) : balance(b) {}
    void deposit(double amt) { if (amt > 0) balance += amt; }
    bool withdraw(double amt) {
          if (amt > 0 && amt <= balance)
          {
               balance -= amt;
               return true;
          }
          return false;
     }
    double getBalance() const {
           return balance;
     }
};
```

Best practices:

- Keep data members private unless there is a compelling reason to expose them.
- Provide clear and minimal public interfaces; prefer functions that preserve object invariants.
- Favor composition over exposing internal implementation details to encourage encapsulation.