# Theory Exercise

## Introduction to C++

**Q1. What are the key differences between Procedural Programming and Object Oriented Programming (OOP)?**

**A.** The **key differences between Procedural Programming** and **Object-Oriented Programming (OOP)** can be summarized based on several core concepts:

---

- **1. Basic Concept**
  - **Procedural Programming**:
    Follows a **top-down** approach where the program is divided into functions or procedures.
  - **OOP**:
    Follows a **bottom-up** approach where the program is organized using **objects** that combine data and behavior.

---

- **2. Focus**
  - **Procedural**: Focuses on **functions** or procedures that operate on data.
  - **OOP**: Focuses on **objects** and **classes**, combining data and methods.

---

- **3. Data Handling**
  - **Procedural**: Data is **global** and can be accessed by any function.
  - **OOP**: Data is **encapsulated** within objects and protected using access modifiers (like private, public, etc.).

---

- **4. Code Reusability**
  - **Procedural**: Limited code reuse. Functions can be reused, but not easily extended.
  - **OOP**: Promotes code reuse through **inheritance**, **polymorphism**, and **composition**.

---

- **5. Security**
  - **Procedural**: Less secure, as data is freely accessible and modifiable.
  - **OOP**: More secure due to **encapsulation** and **data hiding**.

---

🔹 **6. Examples of Languages**

- **Procedural**: C, Pascal, Fortran.
- **OOP**: C++, Java, Python (partially supports both).

---

🔹 **7. Execution Flow**

- **Procedural**: Linear flow – function after function.
- **OOP**: Based on interaction between objects – messages passed to invoke behavior.

---

🔹 **8. Real-World Modeling**

- **Procedural**: Harder to map to real-world entities directly.
- **OOP**: Easier and more natural to represent real-world entities and relationships.

---

🔹 **9. Maintenance and Scalability**

- **Procedural**: Difficult to manage in large, complex programs.
- **OOP**: Easier to maintain and scale due to modular structure.

---

🔹 **10. Inheritance and Polymorphism**

- **Procedural**: Not supported.
- **OOP**: Key features; allows creating new classes from existing ones and using objects in multiple forms.

**Q3. Research and provide three real-world applications where C programming is extensively used, such as in embedded systems, operating systems, or game development.**

**A.** Here's a list of the **main advantages of Object-Oriented Programming (OOP)** over **Procedural-Oriented Programming (POP)**, with clear explanations:

---

🔹 **1. Modularity**

- **OOP**: Code is organized into self-contained **classes and objects**, making it easier to manage and understand.
- ✅ **Advantage**: You can work on different parts of a program independently and reuse components.

---

🔹 **2. Encapsulation**

- **OOP**: Keeps data and methods **together** and **hides** internal details from the outside world.
- ✅ **Advantage**: Increases **security**, prevents unintended interference, and simplifies maintenance.

---

🔹 **3. Code Reusability through Inheritance**

- **OOP**: Allows a class to **inherit** features from another class.
- ✅ **Advantage**: Reduces redundancy, saves development time, and promotes consistency.

---

🔹 **4. Improved Maintainability**

- **OOP**: Easier to update and modify code due to a modular and organized structure.
- ✅ **Advantage**: You can fix bugs or extend functionality with minimal impact on other parts of the program.

---

🔹 **5. Scalability and Flexibility**

- **OOP**: Makes it easier to scale programs and add new features by extending classes or creating new ones.
- ✅ **Advantage**: Better suited for **large and complex** applications.

---

🔹 **6. Polymorphism**

- **OOP**: Allows objects to behave differently based on their class (e.g., method overloading/overriding).
- ✅ **Advantage**: Adds **flexibility** and makes code more general and extensible.

---

🔹 **7. Real-World Modeling**

- **OOP**: Models real-world objects and relationships more naturally using classes, inheritance, and interactions.
- ✅ **Advantage**: Improves **problem-solving** and **design clarity**.

---

🔹 **8. Better Collaboration**

- **OOP**: Promotes the use of **interfaces** and **modular code**, allowing teams to work on different parts simultaneously.

- ✅ **Advantage**: Enhances **team productivity** and code integration.

---

🔹 **9. Dynamic Behavior**

- **OOP**: With polymorphism and dynamic binding, objects can adapt to different situations at runtime.
- ✅ **Advantage**: Improves program **adaptability** and **response to change**.

### Q3. Explain the steps involved in setting up a C++ development environment.

**A.** Setting up a **C++ development environment** involves installing the necessary tools (compiler, editor/IDE) and configuring the system to write, compile, and run C++ programs.

Here are the **main steps**:

---

🔧 **Step-by-Step Guide to Set Up C++ Development Environment**

---

🔹 **1. Install a C++ Compiler**

You need a **compiler** to convert your C++ code into an executable program.

✅ **Popular Compilers:**

- **GCC / G++** (for Linux/Mac/Windows via MinGW or WSL)
- **MSVC** (Microsoft Visual C++ compiler)
- **Clang** (used on macOS and Linux)

🔸 **For Windows:**

- **MinGW** or **TDM-GCC** are commonly used with GCC/G++.
- Alternatively, install **Visual Studio** (includes MSVC).

---

🔹 **2. Choose and Install a Text Editor or IDE**

An IDE provides tools like syntax highlighting, code suggestions, and debugging.

✅ **Recommended IDEs/Text Editors:**

| Tool | Description |
|---|---|
| **Visual Studio** (Windows) | Full-featured IDE with MSVC |

| Tool | Description |
| --- | --- |
| **Code::Blocks** | Lightweight, beginner-friendly |
| **Dev-C++** | Easy to set up and use |
| **VS Code** | Popular code editor with extensions |
| **CLion** | Advanced, paid IDE by JetBrains |

💡 For beginners, **Code::Blocks** or **Dev-C++** are great.
For advanced users, **Visual Studio** or **VS Code with extensions** is preferred.

---

🔹 **3. Set Up the Compiler Path (if needed)**

If you're using **VS Code** or other lightweight editors:

- You must ensure your compiler (e.g., g++) is in your **system PATH**.
- On Windows, this means adding the path to g++.exe from MinGW or TDM-GCC.

🔸 **To check:**

```
g++ --version
```

If this returns the version info, your compiler is properly installed.

---

🔹 **4. Write Your First Program**

Create a file named main.cpp and add the following code:

```
#include <iostream>
using namespace std;

int main() {
   cout << "Hello, C++!" << endl;
   return 0;
}
```

---

🔹 **5. Compile and Run the Program**

✅ **Using Terminal/Command Prompt:**

```
g++ main.cpp -o main
./main      # On Linux/Mac
main.exe    # On Windows
```

✅ **Using IDE:**

- Click **Build** or **Run** from the menu.
- The IDE handles compilation and execution automatically.

### Q4. What are the main input/output operations in C++? Provide examples.

**A.** In **C++**, the main input/output (I/O) operations are handled using the **iostream** library, which provides standard streams for input and output.

---

🔹 **Main I/O Streams in C++**

| Stream | Purpose |
|---|---|
| cin | Standard **input** |
| cout | Standard **output** |
| cerr | Standard **error output** (unbuffered) |
| clog | Standard **log/error output** (buffered) |

---

🔸 **1. Output using cout**

Used to display output to the console.

✅ **Syntax:**

```
cout << data;
```

📄 **Example:**

```
#include <iostream>
using namespace std;
```

```
int main() {
    cout << "Hello, World!" << endl;
    cout << "The result is: " << 10 + 5 << endl;
    return 0;
}
```

endl is used to move to a new line (like \n).

---

🔸 **2. Input using cin**

Used to take input from the user.

✅ **Syntax:**

```
cin >> variable;
```

📜 **Example:**

```
#include <iostream>
using namespace std;

int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "You entered: " << age << endl;
    return 0;
}
```

---

🔸 **3. Multiple Inputs**

You can take multiple inputs using one cin statement.

📜 **Example:**

```
int a, b;
cin >> a >> b;
```

---

🔸 **4. Using cerr for Errors**

Used to output **error messages** (does not buffer output).

🗒️ **Example:**

```
cerr << "Error: Invalid input!" << endl;
```

---

🔸 **5. Using clog for Logging**

Used to output **log messages** (buffered output).

```
clog << "Log: Program started" << endl;
```

# Variables, Data Types, and Operators

**Q5. What are the different data types available in C++? Explain with examples.**

**A.** In **C++**, data types define the type of data a variable can hold. They are mainly categorized into **basic**, **derived**, **enumerated**, and **user-defined** types.

---

✅ **Categories of Data Types in C++**

---

🔹 **1. Basic (Fundamental) Data Types**

| Data Type | Description | Example |
|---|---|---|
| int | Integer numbers | int age = 25; |
| float | Floating-point numbers (single precision) | float temp = 36.5; |
| double | Double precision floating-point | double pi = 3.14159; |
| char | Single character | char grade = 'A'; |
| bool | Boolean value (true or false) | bool passed = true; |
| void | Represents no value (used in functions) | void display(); |

---

### 🔹 2. Derived Data Types

These are based on fundamental types.

| Type | Description | Example |
|---|---|---|
| **Array** | Collection of elements of same type | int arr[5] = {1,2,3,4,5}; |
| **Pointer** | Stores memory address of another variable | int* ptr = &x; |
| **Function** | A block of code that performs a task | int add(int a, int b); |
| **Reference** | Alias for another variable | int &ref = x; |

---

### 🔹 3. Enumerated Data Types (enum)

Defines a list of named integer constants.

📜 **Example:**

```
enum Color { RED, GREEN, BLUE };
Color c = RED;
```

By default, RED = 0, GREEN = 1, BLUE = 2.

---

### 🔹 4. User-Defined Data Types

Created by the programmer.

| Type | Description | Example |
|---|---|---|
| struct | Groups different data types | struct Student { string name; int age; }; |
| class | Blueprint for objects in OOP | class Car { public: string brand; }; |
| union | Stores different data types in same memory | union Data { int i; float f; }; |
| typedef / using | Creates an alias for a data type | typedef int Marks; or using Marks = int; |

## Q6. Explain the difference between implicit and explicit type conversion in C++.

**A.** In **C++**, *type conversion* means changing a value from one data type to another.
It can happen in two main ways: **implicit** (automatic) or **explicit** (manual).

---

### 1. Implicit Type Conversion (Type Casting / Type Promotion)

Also called **type coercion**, this is performed **automatically by the compiler** when:

- The types are compatible.
- There is no risk of losing important data (in most cases).
- It's needed to match operands in an expression.

**Rules:**

- Smaller data types get promoted to larger ones (e.g., int → float).
- Follows the **type promotion hierarchy** (e.g., char → int → float → double).

**Example:**

```
#include <iostream>

using namespace std;


int main() {
   int a = 5;
   double b = 2.5;


   double result = a + b; // int 'a' is automatically converted to double
   cout << result;  // Output: 7.5
   return 0;
}
```

Here, a is implicitly converted to double so it can be added to b.

---

**2. Explicit Type Conversion (Type Casting)**

Also called **type casting**, this is done **manually by the programmer** to force a conversion between types.

**Two common ways:**

1. **C-style cast:**

   ```
   double x = 10.75;
   int y = (int)x; // Explicitly cast to int
   ```

2. **C++ casting operators:**
   - static_cast<type>(expression)
   - dynamic_cast<type>(expression) (for polymorphic types)
   - const_cast<type>(expression) (for removing/adding const)

- reinterpret_cast<type>(expression) (for low-level conversions)

**Example with static_cast:**

```
#include <iostream>
using namespace std;

int main() {
   double pi = 3.14159;
   int intPi = static_cast<int>(pi); // Explicitly cast to int
   cout << intPi; // Output: 3
   return 0;
}
```

Here, the fractional part is intentionally discarded.

## Q7. What are the different types of operators in C++? Provide examples of each.

**A.** In **C++**, operators are special symbols that perform operations on variables or values. They are grouped into categories based on their functionality.

---

**1. Arithmetic Operators**

Used to perform basic mathematical operations.

| Operator | Meaning | Example |
|---|---|---|
| + | Addition | 5 + 3 → 8 |
| - | Subtraction | 5 - 3 → 2 |
| * | Multiplication | 5 * 3 → 15 |
| / | Division | 5 / 2 → 2 (integer division) |
| % | Modulus (remainder) | 5 % 2 → 1 |

**Example:**

```
int a = 10, b = 3;
```

```
cout << a + b; // 13
```

---

**2. Relational Operators**

Used to compare two values; result is true or false.

| Operator | Meaning | Example |
|---|---|---|
| == | Equal to | a == b |
| != | Not equal to | a != b |
| > | Greater than | a > b |
| < | Less than | a < b |
| >= | Greater than or equal to | a >= b |
| <= | Less than or equal to | a <= b |

---

**3. Logical Operators**

Used to combine or invert conditions.

| Operator | Meaning | Example |
|---|---|---|
| && | Logical AND | (a > 0 && b > 0) |
| \|\| | Logical OR | `(a > 0 |
| ! | Logical NOT | !(a > 0) |

---

**4. Assignment Operators**

Used to assign values to variables.

| Operator | Example | Equivalent to |
|---|---|---|
| = | a = 5 | Assigns 5 to a |
| += | a += 3 | a = a + 3 |
| -= | a -= 3 | a = a - 3 |
| *= | a *= 3 | a = a * 3 |
| /= | a /= 3 | a = a / 3 |
| %= | a %= 3 | a = a % 3 |

---

**5. Increment and Decrement Operators**

Used to increase or decrease a variable by 1.

| Operator | Meaning |
|---|---|
| ++ | Increment |
| -- | Decrement |

Two types:

- **Pre-increment**: ++a (increments first, then uses the value)
- **Post-increment**: a++ (uses the value, then increments)

---

**6. Bitwise Operators**

Operate on data at the binary level.

| Operator | Meaning | Example (a=5, b=3) |
|---|---|---|
| & | Bitwise AND | 5 & 3 → 1 |
| \| | Bitwise OR | `5 |
| ^ | Bitwise XOR | 5 ^ 3 → 6 |
| ~ | Bitwise NOT | ~5 → -6 |
| << | Left shift | 5 << 1 → 10 |
| >> | Right shift | 5 >> 1 → 2 |

---

**7. Conditional (Ternary) Operator**

Short form of an if-else statement.

Syntax:

```
condition ? expression1 : expression2;
```

Example:

```
int a = 10, b = 5;
int max = (a > b) ? a : b; // max = 10
```

---

**8. Scope Resolution Operator (::)**

Used to access global variables or class members.

Example:

```
int x = 10;
int main() {
   int x = 20;
   cout << ::x; // prints global x → 10
}
```

---

**9. Member Access Operators**

- . → Access object members.
- -> → Access members via pointer.

Example:

```
struct Point { int x, y; };
Point p = {1, 2};
Point* ptr = &p;
cout << p.x;    // using .
cout << ptr->y;  // using ->
```

---

**10. Other Special Operators**

- **sizeof** → Returns size in bytes.
- **typeid** → Returns type information.
- **new / delete** → Dynamic memory allocation/deallocation.
- **cast operators** (static_cast, dynamic_cast, etc.).

## Q8. Explain the purpose and use of constants and literals in C++.

**A.** In **C++**, *constants* and *literals* are ways to represent fixed values in a program, but they serve slightly different purposes.

---

**1. Constants in C++**

A **constant** is a variable whose value **cannot be changed** after it's initialized.
They help make programs safer, easier to read, and less error-prone.

**Purpose of constants**

- **Prevent accidental modification** of important values.
- Make code **more readable** by giving names to values.
- **Ease of maintenance** – change the value in one place, and it updates everywhere.
- Improve **type safety** (compiler errors if you try to modify).

---

**Ways to define constants**

1. **Using const keyword** (type-safe)

   const double PI = 3.14159;

   - Must be initialized at declaration.
   - Can be of any data type.

2. **Using #define macro** (preprocessor)

   #define PI 3.14159

   - No type-checking (simple text replacement).
   - Generally less safe than const.

3. **Using constexpr** (compile-time constants)

   constexpr int DAYS_IN_WEEK = 7;

   - Must be computable at compile-time.
   - Useful for array sizes, template parameters, etc.

---

**Example:**

```
#include <iostream>

using namespace std;


int main() {
   const double PI = 3.14159;
   double radius = 5;
   double area = PI * radius * radius;
```

```
    cout << "Area: " << area << endl;
    return 0;
}
```

---

**2. Literals in C++**

A **literal** is a fixed value **directly written in the code** (hard-coded).
They are the actual values assigned to variables or used in expressions.

**Types of Literals**

1. **Integer literals**

   ```
   10    // decimal
   012   // octal
   0xA   // hexadecimal
   0b1010 // binary (C++14)
   ```

2. **Floating-point literals**

   ```
   3.14
   2.5e3  // scientific notation → 2500
   ```

3. **Character literals**

   ```
   'A'
   '\n'   // newline escape sequence
   ```

4. **String literals**

   ```
   "Hello World"
   ```

5. **Boolean literals**

   ```
   true
   false
   ```

6. **nullptr literal**

   ```
   nullptr
   ```

---

**Example:**

```
#include <iostream>
```

```
using namespace std;


int main() {
   int age = 21;         // 21 is an integer literal
   double pi = 3.14159;   // 3.14159 is a floating-point literal
   char grade = 'A';      // 'A' is a character literal
   string name = "Zaid";  // "Zaid" is a string literal


   cout << "Age: " << age << ", PI: " << pi << ", Grade: " << grade
       << ", Name: " << name << endl;
   return 0;
}
```

# Control Flow Statements

**Q9. What are conditional statements in C++? Explain the if-else and switch statements.**

**A.** In **C++**, **conditional statements** allow a program to **make decisions** and execute different code based on whether a condition is **true** or **false**.

They are essential for controlling the program's flow.

---

**1. The if-else Statement**

The if-else statement checks a condition:

- If the condition is **true**, the if block runs.
- If the condition is **false**, the else block runs (if present).

**Syntax:**

```
if (condition) {
   // Code runs if condition is true
}
else {
   // Code runs if condition is false
}
```

**Example:**

```
#include <iostream>
using namespace std;

int main() {
   int age = 18;

   if (age >= 18) {
      cout << "You are eligible to vote.";
   }
   else {
      cout << "You are not eligible to vote.";
   }
   return 0;
}
```

---

**Nested if-else (multiple conditions)**

```
if (marks >= 90)
   cout << "Grade A";
else if (marks >= 75)
   cout << "Grade B";
else
   cout << "Grade C";
```

---

**2. The switch Statement**

The switch statement allows **multi-way branching** based on a variable's value.

- Works well when you have many possible constant values to check.
- Each possible value is handled in a **case** block.
- break is used to exit after a match.

- default is used if no case matches.

**Syntax:**

```
switch (expression) {
  case constant1:
    // Code for case 1
    break;
  case constant2:
    // Code for case 2
    break;
  ...
  default:
    // Code if no case matches
}
```

**Example:**

```
#include <iostream>
using namespace std;

int main() {
  int day = 3;

  switch (day) {
    case 1: cout << "Monday"; break;
    case 2: cout << "Tuesday"; break;
    case 3: cout << "Wednesday"; break;
    case 4: cout << "Thursday"; break;
    case 5: cout << "Friday"; break;
    case 6: cout << "Saturday"; break;
    case 7: cout << "Sunday"; break;
    default: cout << "Invalid day";
```

```
    }
    return 0;
}
```

### Q10. What is the difference between for, while, and do-while loops in C++?

**A.** In **C++**, for, while, and do-while are **looping constructs** that allow you to repeat a block of code multiple times.
They differ mainly in **syntax**, **when the condition is checked**, and **common usage**.

---

**1. for Loop**

- Best when **number of iterations is known** in advance.
- Initialization, condition, and increment/decrement are written in one line.

**Syntax:**

```
for (initialization; condition; update) {
    // loop body
}
```

**Example:**

```
for (int i = 1; i <= 5; i++) {
    cout << i << " ";
}
```

**Flow:** Initialization → Condition check → Loop body → Update → Condition check → ...

---

**2. while Loop**

- Best when **number of iterations is not known** beforehand.
- **Condition is checked before** entering the loop (entry-controlled loop).
- If the condition is false initially, the loop body may **not run at all**.

**Syntax:**

```
while (condition) {
```

```
    // loop body
}
```

**Example:**

```
int i = 1;
while (i <= 5) {
    cout << i << " ";
    i++;
}
```

---

**3. do-while Loop**

- Similar to while, but **condition is checked after** the loop body (exit-controlled loop).
- The loop body runs **at least once**, even if the condition is false initially.

**Syntax:**

```
do {
    // loop body
} while (condition);
```

**Example:**

```
int i = 1;
do {
    cout << i << " ";
    i++;
} while (i <= 5);
```

## Q10. What is the difference between for, while, and do-while loops in C++?

**A.** In **C++**, for, while, and do-while are **looping constructs** that allow you to repeat a block of code multiple times.
They differ mainly in **syntax**, **when the condition is checked**, and **common usage**.

---

**1. for Loop**

- Best when **number of iterations is known** in advance.
- Initialization, condition, and increment/decrement are written in one line.

**Syntax:**

```
for (initialization; condition; update) {
   // loop body
}
```

**Example:**

```
for (int i = 1; i <= 5; i++) {
   cout << i << " ";
}
```

**Flow:** Initialization → Condition check → Loop body → Update → Condition check → ...

---

**2. while Loop**

- Best when **number of iterations is not known** beforehand.
- **Condition is checked before** entering the loop (entry-controlled loop).
- If the condition is false initially, the loop body may **not run at all**.

**Syntax:**

```
while (condition) {
   // loop body
}
```

**Example:**

```
int i = 1;
while (i <= 5) {
   cout << i << " ";
   i++;
}
```

---

**3. do-while Loop**

- Similar to while, but **condition is checked after** the loop body (exit-controlled loop).
- The loop body runs **at least once**, even if the condition is false initially.

**Syntax:**

```
do {
   // loop body
} while (condition);
```

**Example:**

```
int i = 1;
do {
   cout << i << " ";
   i++;
} while (i <= 5);
```

### Q11. How are break and continue statements used in loops? Provide examples.

**A.** In **C++**, break and continue are **control flow statements** that alter the normal execution of loops (for, while, do-while) and switch statements.

---

**1. break Statement**

- **Purpose:** Immediately **terminates** the nearest enclosing loop or switch statement.
- **Effect:** Control jumps to the first statement **after** the loop or switch.

**Syntax:**

```
break;
```

**Example (using for loop):**

```
#include <iostream>
using namespace std;

int main() {
   for (int i = 1; i <= 10; i++) {
      if (i == 5) {
         break; // loop ends when i == 5
      }
      cout << i << " ";
   }
```

```
    return 0;
}
```

**Output:**

```
1 2 3 4
```

---

**2. continue Statement**

- **Purpose:** Skips the **rest of the loop body** for the current iteration and moves to the **next iteration**.
- **Effect:** Control jumps to:
    - **Update step** in for loop.
    - **Condition check** in while / do-while.

**Syntax:**

```
continue;
```

**Example (using for loop):**

```
#include <iostream>
using namespace std;

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue; // skip printing when i == 3
        }
        cout << i << " ";
    }
    return 0;
}
```

**Output:**

```
1 2 4 5
```

### Q12. Explain nested control structures with an example.

**A.** In **C++**, **nested control structures** occur when one control structure (like an if, for, while, or switch) is placed **inside** another.
They are used when solving problems that require **multiple levels of decision-making** or **repetitive tasks within repetitive tasks**.

---

**Types of nesting**

- **Nested if statements** – Decision inside another decision.
- **Loop inside another loop** – Often used for multi-dimensional data (e.g., matrices).
- **Mix of loops and conditionals** – Common in complex algorithms.

---

**Example: Nested for loop with if**

This program prints a multiplication table but skips results that are **even numbers**.

```
#include <iostream>
using namespace std;


int main() {
   for (int i = 1; i <= 5; i++) {            // Outer loop
      for (int j = 1; j <= 5; j++) {         // Inner loop
         int result = i * j;
         if (result % 2 == 0) {              // Nested if inside inner loop
            continue; // Skip even results
         }
         cout << i << " x " << j << " = " << result << endl;
      }
      cout << "----" << endl; // Separator after each row
   }
   return 0;
}
```

**How it works:**

1. **Outer loop (i)** runs for each row.
2. **Inner loop (j)** runs for each column.
3. **Nested if** checks if the product is even, and skips printing it.

4. Both loops are independent but work together to cover all combinations.

---

**Real-life analogy**

Think of **nested control structures** like:

- Outer loop = Teacher checking each class.
- Inner loop = Teacher checking each student **within that class**.
- Nested if = Teacher giving a warning only if a student is absent.

# Functions and Scope

**Q13. What is a function in C++? Explain the concept of function declaration, definition, and calling.**

**A.** In **C++**, a **function** is a **block of code** designed to perform a specific task.
It can be **called** whenever you need that task, instead of writing the same code repeatedly.

---

## Purpose of Functions

- **Reusability** → Write once, use multiple times.
- **Modularity** → Break large programs into smaller, manageable parts.
- **Readability** → Code becomes easier to understand.
- **Maintainability** → Easier to update/modify specific tasks.

---

## 1. Function Declaration (Prototype)

- Tells the compiler:
    - **Function name**
    - **Return type**
    - **Parameter types**
- Placed **before main()** or in a header file.
- Ends with a **semicolon**.

### Syntax:

```
return_type function_name(parameter_list);
```

**Example:**

```
int add(int a, int b); // Declaration
```

---

### 2. Function Definition

- Contains the **actual code** (body) of the function.
- Must match the **declaration** in return type, name, and parameters.

**Syntax:**

```
return_type function_name(parameter_list) {
    // function body
}
```

**Example:**

```
int add(int a, int b) {
    return a + b; // Function body
}
```

---

### 3. Function Calling

- Executes the function code.
- Provide required **arguments** (if any).

**Example:**

```
int result = add(5, 3); // Call function
```

---

### Complete Example

```
#include <iostream>
using namespace std;
// Function Declaration
int add(int a, int b);
```

```
int main() {
    int x = 5, y = 3;
    int sum = add(x, y); // Function Call
    cout << "Sum: " << sum;
    return 0;
}

// Function Definition
int add(int a, int b) {
    return a + b;
}
```

**Output:**

```
Sum: 8
```

**Q14. What is the scope of variables in C++? Differentiate between local and global scope.**

**A.** In **C++**, the **scope** of a variable refers to the **region of the program** where that variable can be accessed or used.
The scope determines the **lifetime** and **visibility** of a variable.

---

**Types of Scope**

**1. Local Scope**

- A variable declared **inside** a function, block ({}), or loop.
- Can **only** be accessed **within** that function/block.
- Created when the block is entered, destroyed when the block ends.

**Example:**

```
#include <iostream>
using namespace std;

int main() {
```

```
    int x = 10; // Local to main()
    {
        int y = 20; // Local to this block
        cout << x << " " << y << endl;
    }
    // cout << y; // ❌ Error: y is not accessible here
    return 0;
}
```

---

### 2. Global Scope

- A variable declared **outside** all functions.
- Accessible from **any function** in the program (unless shadowed by a local variable with the same name).
- Exists for the **entire duration** of the program.

**Example:**

```
#include <iostream>
using namespace std;
int g = 100; // Global variable

void display() {
    cout << "Global g: " << g << endl;
}

int main() {
    cout << "Access in main: " << g << endl;
    display(); // Accessible in another function
    return 0;
}
```

**Q15. Explain recursion in C++ with an example.**

**A.** In **C++**, **recursion** is a technique where a **function calls itself**—either directly or indirectly—to solve a problem.

It works by breaking a large problem into **smaller subproblems** of the same type until it reaches a **base case** (a condition where the function stops calling itself).

---

### Key Concepts in Recursion

1. **Base Case** → The stopping condition that prevents infinite recursion.
2. **Recursive Case** → The part of the function where it calls itself with a simpler/smaller input.

---

### General Syntax

```
return_type function_name(parameters) {
    if (base_condition) {
        // Stop recursion
        return some_value;
    } else {
        // Recursive call with smaller/simpler problem
        return function_name(modified_parameters);
    }
}
```

---

### Example 1: Factorial Using Recursion

The factorial of a number n is:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

And by definition:

$$0! = 1$$

### Code:

```
#include <iostream>
using namespace std;
```

```
int factorial(int n) {
    if (n == 0) // Base case
        return 1;
    else
        return n * factorial(n - 1); // Recursive case
}

int main() {
    int num = 5;
    cout << "Factorial of " << num << " is " << factorial(num);
    return 0;
}
```

**Output:**

```
Factorial of 5 is 120
```

**Advantages of Recursion**

- Simplifies problems that can be divided into smaller, similar problems.
- Makes code shorter and easier to read for certain algorithms (like tree traversal, backtracking).

---

**Disadvantages of Recursion**

- More memory usage (due to function call stack).
- Can be slower than iterative solutions.
- Risk of **stack overflow** if base case is missing or incorrect.

**Q16. What are function prototypes in C++? Why are they used?**

**A.** In **C++**, a **function prototype** is a **declaration** of a function that tells the compiler:

- The **function's name**
- Its **return type**

- The **number and types of parameters**

...but **does not** contain the function body.

---

### Syntax

```
return_type function_name(parameter_list);
```

Example:

```
int add(int, int);
```

---

### Purpose of Function Prototypes

1. **Inform the compiler before function usage**
   - Ensures the compiler knows how to handle the function call, even if the function definition appears **after** main().
2. **Type checking**
   - The compiler checks that the arguments in a function call **match** the declared parameter types.
3. **Code organization**
   - Allows separating **function declarations** (often in header files) from **definitions** (in .cpp files).

---

### Example Without Prototype (May Cause Error)

```
#include <iostream>

using namespace std;


int main() {
   cout << add(3, 4); // ❌ Compiler may complain: 'add' not declared
   return 0;
}


int add(int a, int b) { // Definition after main
   return a + b;
```

```
}
```

---

**Example With Prototype**

```
#include <iostream>
using namespace std;

// Function prototype
int add(int a, int b);

int main() {
    cout << "Sum: " << add(3, 4); // ✅ Works fine
    return 0;
}

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

**Output:**

```
Sum: 7
```

# Arrays and Strings

**Q17. What are arrays in C++? Explain the difference between single-dimensional and multi- dimensional arrays.**

**A.** In **C++**, an **array** is a **collection of elements of the same data type** stored in **contiguous memory locations**.
Each element is accessed using an **index** (also called a subscript), starting from 0.
Arrays are useful for storing and working with multiple values without creating separate variables for each.

---

### Declaration of an Array

```
data_type array_name[size];
```

- **data_type** → type of elements stored (int, float, char, etc.)
- **array_name** → name you choose for the array
- **size** → number of elements

Example:

```
int marks[5];   // array of 5 integers
```

---

### Single-Dimensional Array

- Represents a **list** of elements in a single row.
- Accessed using **one index**: array_name[index]
- **Memory layout:** sequential

**Example:**

```
#include <iostream>
using namespace std;

int main() {
   int marks[5] = {90, 85, 78, 92, 88};  // single-dimensional array

   for (int i = 0; i < 5; i++) {
      cout << "marks[" << i << "] = " << marks[i] << endl;
   }
   return 0;
}
```

**Output:**

```
marks[0] = 90
marks[1] = 85
marks[2] = 78
marks[3] = 92
```

marks[4] = 88

---

### Multi-Dimensional Array

- Represents **data in tables, matrices, or higher dimensions**.
- Accessed using **multiple indices**.
- The most common form is a **two-dimensional array** (rows and columns).

**Syntax:**

```
data_type array_name[rows][cols];
```

**Example:**

```
#include <iostream>
using namespace std;

int main() {
    int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} }; // 2 rows, 3 columns

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << "matrix[" << i << "][" << j << "] = " << matrix[i][j] << "  ";
        }
        cout << endl;
    }
    return 0;
}
```

**Output:**

```
matrix[0][0] = 1  matrix[0][1] = 2  matrix[0][2] = 3
matrix[1][0] = 4  matrix[1][1] = 5  matrix[1][2] = 6
```

---

### Key Differences Between Single & Multi-Dimensional Arrays

| Feature | Single-Dimensional Array | Multi-Dimensional Array |
|---|---|---|
| Structure | Stores data in a single row | Stores data in a table/matrix |
| Number of Indices | One index [i] | Two or more indices [i][j]... |
| Usage | Lists, sequential data | Tables, matrices, grids |
| Memory | Contiguous, linear structure | Contiguous but stored in row-major order |
| Example | int arr[5] | int arr[3][4] |

**Q18. Explain string handling in C++ with examples.**

**A.** In C++, **string handling** means working with sequences of characters—creating, modifying, and processing them.
C++ supports string handling in two main ways:

---

1. Using C-style strings (char arrays)

These are inherited from the C language and are essentially arrays of characters ending with a **null character** '\0'.

**Example:**

```
#include <iostream>
#include <cstring>  // for strlen, strcpy, strcat, strcmp
using namespace std;

int main() {
    char str1[20] = "Hello";
    char str2[20] = "World";

    // Length of string
    cout << "Length of str1: " << strlen(str1) << endl;

    // Copy string
    strcpy(str2, str1);
```

```
    cout << "After copying, str2: " << str2 << endl;

    // Concatenate
    strcat(str1, " C++");
    cout << "After concatenation: " << str1 << endl;

    // Compare strings
    if (strcmp(str1, str2) == 0)
        cout << "str1 and str2 are equal" << endl;
    else
        cout << "str1 and str2 are different" << endl;

    return 0;
}
```

**Key functions in <cstring>:**

- strlen() – Finds string length.
- strcpy() – Copies one string to another.
- strcat() – Concatenates two strings.
- strcmp() – Compares two strings.

---

### 2. Using std::string class (C++ way)

The <string> library provides the std::string class, which is more convenient and safer than C-style strings.

**Example:**

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s1 = "Hello";
```

```
    string s2 = "World";

    // Concatenation
    string s3 = s1 + " " + s2;
    cout << "Concatenated: " << s3 << endl;

    // Length of string
    cout << "Length of s3: " << s3.length() << endl;

    // Substring
    cout << "Substring (first 5 chars): " << s3.substr(0, 5) << endl;

    // Replace
    s3.replace(6, 5, "C++");
    cout << "After replace: " << s3 << endl;

    // Find
    size_t pos = s3.find("C++");
    if (pos != string::npos)
        cout << "Found 'C++' at position: " << pos << endl;

    return 0;
}
```

**Common std::string functions:**

- .length() / .size() – Returns number of characters.
- .append() / + – Concatenates strings.
- .substr(start, length) – Extracts substring.
- .find(substring) – Finds position of substring.
- .replace(pos, len, new_str) – Replaces part of string.

- .erase(pos, len) – Removes characters.

**Q19. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.**

**A.** In C++, **arrays** can be initialized in several ways, and the method depends on whether the array is **one-dimensional (1D)** or **two-dimensional (2D)**.

---

### 1. Initializing 1D Arrays

A **1D array** is just a list of elements of the same type stored in contiguous memory locations.

**Syntax:**

```
data_type array_name[size] = {value1, value2, ...};
```

**Examples:**

```
#include <iostream>
using namespace std;

int main() {
   // Method 1: Specify size and values
   int arr1[5] = {10, 20, 30, 40, 50};

   // Method 2: Let compiler calculate size
   int arr2[] = {1, 2, 3, 4};

   // Method 3: Partial initialization (remaining elements set to 0)
   int arr3[5] = {5, 10};

   // Method 4: Initialize all elements to 0
   int arr4[5] = {0};

   // Display arr1
   cout << "arr1 elements: ";
   for (int i = 0; i < 5; i++)
```

```
        cout << arr1[i] << " ";
    cout << endl;

    return 0;
}
```

---

## 2. Initializing 2D Arrays

A **2D array** is like a table of rows and columns.

**Syntax:**

```
data_type array_name[rows][cols] = {
    {value11, value12, ...},
    {value21, value22, ...},
    ...
};
```

**Examples:**

```
#include <iostream>
using namespace std;

int main() {
    // Method 1: Full initialization
    int matrix1[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    // Method 2: Row-by-row initialization
    int matrix2[2][3] = {1, 2, 3, 4, 5, 6};

    // Method 3: Partial initialization (missing values set to 0)
```

```
    int matrix3[2][3] = {
        {1, 2}, // Remaining element becomes 0
        {4}     // Remaining two elements become 0
    };

    // Display matrix1
    cout << "matrix1 elements:\n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix1[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

**Q20. Explain string operations and functions in C++.**

**A.** In C++, **string operations and functions** can be done in two main ways:

1. **C-style strings** (char arrays with <cstring> functions)
2. **std::string class** (modern C++ approach with <string> library)

---

### 1. C-Style String Operations (<cstring>)

C-style strings are arrays of characters ending with a null character ('\0').
They are manipulated using functions from the <cstring> header.

#### Common Functions:

| Function | Purpose |
|---|---|
| strlen(str) | Returns length (excluding '\0'). |
| strcpy(dest, src) | Copies src into dest. |

| Function | Purpose |
| --- | --- |
| strcat(dest, src) | Appends src to dest. |
| strcmp(str1, str2) | Compares strings (0 if equal). |
| strchr(str, ch) | Finds first occurrence of a character. |
| strstr(str, sub) | Finds first occurrence of a substring. |

---

### 2. std::string Operations (<string>)

The std::string class is easier and safer, with built-in operators and methods.

#### Common Operations:

| Operation | Example | Purpose |
| --- | --- | --- |
| Concatenation | s3 = s1 + s2; | Joins two strings. |
| Assignment | s2 = s1; | Copies one string to another. |
| Comparison | s1 == s2 | Checks if strings are equal. |
| Access char | s1[0] | Gets first character. |
| Append | s1.append("text"); | Adds text to end. |
| Length | s1.length() | Number of characters. |
| Substring | s1.substr(pos, len) | Extracts part of string. |
| Find | s1.find("word") | Finds position of substring. |
| Replace | s1.replace(pos, len, "new") | Replaces part of string. |
| Erase | s1.erase(pos, len) | Removes characters. |
| Clear | s1.clear() | Makes string empty. |

---

# Introduction to Object-Oriented Programming

**Q21. Research different types of internet connections (e.g., broadband, Fiber, satellite) and list their pros and cons**

**A.** Object-Oriented Programming (OOP) in C++ is a programming paradigm that organizes code around **objects** (real-world entities) rather than functions and logic.
It helps make programs more **modular, reusable, and easier to maintain**.

Here are the **key concepts** of OOP:

---

### 1. Class and Object

- **Class** → A blueprint or template that defines the properties (**data members**) and behaviors (**member functions**) of objects.
- **Object** → An instance of a class. It has its own data but shares the class's structure.

---

### 2. Encapsulation

- Wrapping data and functions into a single unit (class).
- Uses **access specifiers** (private, protected, public) to control access.
- Protects data from unintended modification.

---

### 3. Abstraction

- Showing only **essential details** and hiding the internal implementation.
- Achieved using **abstract classes** or **interfaces** (in C++ via pure virtual functions).

---

### 4. Inheritance

- Allows one class (**child/derived class**) to acquire properties and behaviors of another (**parent/base class**).
- Promotes **code reuse**.

**Types in C++:**

- Single, Multiple, Multilevel, Hierarchical, Hybrid.

---

### 5. Polymorphism

- Means **many forms** – the ability of a function or object to behave differently based on context.
- **Compile-time polymorphism** → Function overloading, operator overloading.
- **Run-time polymorphism** → Function overriding (with virtual functions).

---

### 6. Data Hiding

- Part of encapsulation.
- Restricts direct access to object's data to maintain integrity.

---

### 7. Message Passing

- Objects communicate with each other by sending and receiving messages (calling methods).

**Q22. What are classes and objects in C++? Provide an example.**

**A.** In C++, **classes** and **objects** are the foundation of **Object-Oriented Programming (OOP)**.

---

**Class**

- A **class** is a **blueprint** or **template** that defines the structure of objects.
- It contains:
  - **Data members** → variables that store object data.
  - **Member functions** → functions that operate on that data.

**Syntax:**

```
class ClassName {
  access_specifier:
    // data members
    // member functions
};
```

---

## Object

- An **object** is an **instance** of a class.
- It has:
    - Its own copy of data members.
    - Access to the class's functions.

---

**Example:**

```
#include <iostream>
using namespace std;

// Defining a class
class Car {
public:
    string brand;
    int speed;

    void drive() {
        cout << brand << " is driving at " << speed << " km/h" << endl;
    }
};

int main() {
    // Creating objects of the Car class
    Car car1;
    Car car2;

    // Assigning values to car1
    car1.brand = "Toyota";
```

```
    car1.speed = 120;

    // Assigning values to car2
    car2.brand = "BMW";
    car2.speed = 150;

    // Calling member functions
    car1.drive();
    car2.drive();

    return 0;
}
```

**Q23. What is inheritance in C++? Explain with an example.**

**A.** In C++, **inheritance** is an **Object-Oriented Programming** concept where one class (**derived class**) acquires the properties and behaviors (data members and member functions) of another class (**base class**).

It allows you to **reuse code** and create a natural **hierarchical relationship** between classes.

---

### Key Points

- **Base Class** → The class whose members are inherited.
- **Derived Class** → The class that inherits from the base class.
- Helps avoid code duplication.
- Supports different types: single, multiple, multilevel, hierarchical, hybrid.

---

### Syntax

```
class DerivedClass : access_specifier BaseClass {
    // additional members
};
```

**Access Specifiers:**

- public → Public members stay public; protected stay protected.
- protected → Public and protected members become protected.
- private → Public and protected members become private.

---

**1. Single Inheritance**

- One derived class inherits from **one base class**.
- Represents a simple "is-a" relationship.

---

**2. Multiple Inheritance**

- One derived class inherits from **more than one base class**.
- Can cause **ambiguity** if base classes have members with the same name (resolved using the **scope resolution operator**).

---

**3. Multilevel Inheritance**

- A derived class becomes the base class for another class (inheritance chain).

---

**4. Hierarchical Inheritance**

- **Multiple derived classes** inherit from **the same base class**.

---

**5. Hybrid Inheritance**

- A combination of two or more types of inheritance.
- Often involves **multiple + multilevel** inheritance.
- Can lead to **diamond problem** (resolved using **virtual inheritance**).

**Example:**

```
class Engine {
public:
   void startEngine() { cout << "Engine started\n"; }
};
```

```
class Vehicle : public virtual Engine {
public:
    void fuelUp() { cout << "Fueled up\n"; }
};

class Boat : public virtual Engine {
public:
    void sail() { cout << "Sailing...\n"; }
};

class AmphibiousVehicle : public Vehicle, public Boat {
public:
    void driveOnLand() { cout << "Driving on land\n"; }
};
```

**Q24. What is inheritance in C++? Explain with an example.**

**A.** In C++, **encapsulation** is the process of **bundling data (variables)** and **methods (functions)** that operate on that data into a single unit called a **class**, while **restricting direct access** to some of the object's components.

It is one of the **four main pillars of OOP** (along with inheritance, polymorphism, and abstraction).

---

### Key Points

- Protects data from accidental or unauthorized modification.
- Achieved using **access specifiers**:
    - private → Accessible only inside the class.
    - protected → Accessible inside the class and derived classes.
    - public → Accessible from anywhere.

---

### How Encapsulation is Achieved in Classes

1. **Keep data members private** → Hides the internal representation.
2. **Provide public getter and setter functions** → Controls access to data.

---

### Example

```
#include <iostream>
using namespace std;

class BankAccount {
private:
    double balance; // data is hidden

public:
    // Constructor
    BankAccount(double initialBalance) {
        if (initialBalance >= 0)
            balance = initialBalance;
        else
            balance = 0;
    }

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

    // Getter
    double getBalance() {
```

```
        return balance;
    }
};

int main() {
    BankAccount account(1000);

    account.deposit(500); // modify via method
    cout << "Balance: " << account.getBalance() << endl;

    // account.balance = 5000; ❌ Not allowed (private member)

    return 0;
}
```