**1. Introduction to C++**

**1. Key Differences Between Procedural Programming and Object-Oriented Programming (OOP)**

| **Feature** | **Procedural Programming (POP)** | **Object-Oriented Programming (OOP)** |
| --- | --- | --- |
| **Approach** | Follows a top-down approach | Follows a bottom-up approach |
| **Focus** | Focuses on procedures (functions) | Focuses on objects and classes |
| **Data Handling** | Data is exposed and can be accessed by any function | Data is encapsulated and accessed only through methods |
| **Reusability** | Limited code reusability | Promotes code reusability through inheritance and polymorphism |
| **Security** | Less secure due to global data access | More secure due to data hiding and encapsulation |
| **Examples** | C, Pascal | C++, Java, Python (with OOP) |

**2. Main Advantages of OOP Over POP**

1. **Encapsulation:**
   * Keeps data safe by bundling it with methods that operate on it.
   * Prevents unauthorized access using access specifiers (private, public, protected).
2. **Inheritance:**
   * Enables code reuse.
   * Allows new classes to inherit properties and behavior from existing classes.
3. **Polymorphism:**
   * Allows functions or methods to behave differently based on input.
   * Supports method overloading and overriding.
4. **Abstraction:**
   * Hides complex implementation details and shows only essential features.
   * Simplifies code management and enhances modularity.
5. **Modularity:**
   * Code is organized into objects or classes, making it easier to maintain and scale.
6. **Security:**
   * Data hiding protects the internal object state from unintended modification.

**3. Steps Involved in Setting Up a C++ Development Environment**

**Step 1: Install a Compiler**

* Install GCC (GNU Compiler Collection) or MSVC (Microsoft Visual C++).
* For Windows: [MinGW](http://mingw-w64.org) or TDM-GCC.

**Step 2: Choose an IDE or Text Editor**

* IDEs: Code::Blocks, Visual Studio, Eclipse CDT, CLion
* Text Editors: VS Code, Sublime Text, Atom

**Step 3: Configure Compiler Path (if needed)**

* Set the environment variable (e.g., add compiler path to system PATH).

**Step 4: Write Your First Program**

* Save a .cpp file with a basic program:

#include <iostream>

using namespace std;

int main() {

cout << "Hello, World!" << endl;

return 0;

}

**Step 5: Compile and Run**

* Using Terminal/Command Prompt:

g++ yourfile.cpp -o output

./output # on Windows: output.exe

**Optional Tools:**

* Debuggers like GDB.
* Build tools like CMake or Makefiles for larger projects.

**4. Main Input/Output Operations in C++ (with Examples)**

C++ uses **iostream** for input/output operations:

**Input: cin**

Used to take input from the user.

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age;

cout << "You entered: " << age << endl;

return 0;

}

**Output: cout**

Used to print output to the console.

cout << "Hello, World!" << endl;

**Chaining Input/Output:**

int a, b;

cin >> a >> b; // input two values

cout << "Sum: " << (a + b) << endl; // output result

**File I/O (Advanced)**

#include <fstream>

using namespace std;

int main() {

ofstream outFile("example.txt");

outFile << "Writing to a file.\n";

outFile.close();

ifstream inFile("example.txt");

string line;

while (getline(inFile, line)) {

cout << line << endl;

}

inFile.close();

return 0;

}

**2. Variables, Data Types, and Operators**

**1. Different Data Types in C++ (With Examples)**

C++ supports several **built-in** and **user-defined** data types:

**A. Fundamental/Built-in Data Types**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| int | Integer values | int age = 25; |
| float | Floating-point numbers | float pi = 3.14f; |
| double | Double-precision float | double g = 9.80665; |
| char | Single character | char grade = 'A'; |
| bool | Boolean value | bool isReady = true; |
| void | No value / return type | void sayHello() {} |

**B. Derived Data Types**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| Array | Collection of same data type | int nums[5] = {1,2,3,4,5}; |
| Pointer | Stores address of variable | int\* ptr = &age; |
| Reference | Another name for variable | int &ref = age; |
| Function | Code block that returns a type | int sum(int a, int b) { return a + b; } |

**C. User-Defined Data Types**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| struct | Group of variables | struct Car { string brand; int year; }; |
| class | Object-oriented structure | class Student { public: string name; }; |
| union | Memory-sharing structure | union Data { int i; float f; }; |
| enum | Named set of constants | enum Day {Mon, Tue, Wed}; |

**2. Implicit vs Explicit Type Conversion in C++**

**Implicit Type Conversion (Type Promotion)**

* Done **automatically by the compiler**.
* Happens when mixed types are used in an expression.

int x = 10;

float y = 5.5;

float result = x + y; // int x is implicitly converted to float

**Explicit Type Conversion (Type Casting)**

* Done **manually by the programmer**.
* Syntax: (new\_type)value

double pi = 3.14159;

int whole = (int)pi; // Explicit cast: result = 3

**3. Different Types of Operators in C++ (With Examples)**

| **Operator Type** | **Example** | **Description** |
| --- | --- | --- |
| **Arithmetic** | +, -, \*, /, % | Performs math operations |
|  | int sum = a + b; |  |
| **Relational** | ==, !=, <, >, <=, >= | Compares values |
|  | if (a != b) |  |
| **Logical** | &&, ` |  |
|  | if (a > 0 && b > 0) |  |
| **Assignment** | =, +=, -=, \*=, /=, %= | Assigns values |
|  | x += 5; // x = x + 5 |  |
| **Increment/Decrement** | ++, -- | Increases or decreases value |
|  | x++, --y |  |
| **Bitwise** | &, ` | , ^, ~, <<, >>` |
|  | a & b, a << 1 |  |
| **Conditional (Ternary)** | condition ? true\_value : false\_value; | Shortcut for if-else |
|  | int min = (a < b) ? a : b; |  |
| **Sizeof** | sizeof(type) | Returns memory size |
|  | sizeof(int) |  |
| **Scope Resolution** | :: | Access global variable or class members |
|  | ::x |  |

**4. Constants and Literals in C++**

**Constants:**

* A **constant** is a value that **doesn't change** during program execution.
* Declared using const keyword.

const int MAX\_USERS = 100;

// MAX\_USERS cannot be modified later

**Literals:**

* **Literals** are the **fixed values** used directly in code.
* They can be of different types:

| **Type** | **Example** |
| --- | --- |
| Integer | 42, 0, -7 |
| Float | 3.14, 2.0f |
| Character | 'A', '1' |
| String | "Hello" |
| Boolean | true, false |

**Purpose of Constants and Literals:**

* **Improves code readability**.
* **Avoids accidental modification**.
* **Makes maintenance easier** (e.g., change in one place updates the whole program).

**3. Control Flow Statements**

**1. Conditional Statements in C++**

Conditional statements let you **make decisions** in your code based on certain conditions.

**A. if and if-else Statements**

* Used to execute code **only if** a condition is true.

int age = 18;

if (age >= 18) {

cout << "You are eligible to vote." << endl;

} else {

cout << "You are not eligible to vote." << endl;

}

You can also use else if for multiple conditions:

if (score >= 90) {

cout << "Grade: A";

} else if (score >= 75) {

cout << "Grade: B";

} else {

cout << "Grade: C";

}

**B. switch Statement**

* Used to execute one block of code among many options, based on a **single variable's value**.

int day = 3;

switch(day) {

case 1:

cout << "Monday";

break;

case 2:

cout << "Tuesday";

break;

case 3:

cout << "Wednesday";

break;

default:

cout << "Invalid day";

}

* break prevents **fall-through** to the next case.
* default runs if no case matches.

**2. Difference Between for, while, and do-while Loops**

| **Loop Type** | **When to Use** | **Syntax** | **Key Point** |
| --- | --- | --- | --- |
| For | When number of iterations is known | for (init; cond; update) | Pre-test loop |
| While | When condition must be checked before loop | while (condition) | Pre-test loop |
| do-while | When loop should run at least once | do { } while (condition); | Post-test loop |

**Examples:**

**for loop:**

for (int i = 0; i < 5; i++) {

cout << i << " ";

}

**while loop:**

int i = 0;

while (i < 5) {

cout << i << " ";

i++;

}

**do-while loop:**

int i = 0;

do {

cout << i << " ";

i++;

} while (i < 5);

**3. break and continue Statements in Loops**

**break: Exits the loop immediately.**

for (int i = 1; i <= 10; i++) {

if (i == 5)

break;

cout << i << " ";

}

// Output: 1 2 3 4

**continue: Skips the current iteration and moves to the next one.**

for (int i = 1; i <= 5; i++) {

if (i == 3)

continue;

cout << i << " ";

}

// Output: 1 2 4 5

**4. Nested Control Structures**

A **nested control structure** is when one control structure (like a loop or if-statement) is placed inside another.

**Example: Nested if inside a for loop**

for (int i = 1; i <= 3; i++) {

if (i % 2 == 0) {

cout << i << " is even\n";

} else {

cout << i << " is odd\n";

}

}

**4. Functions and Scope**

**1. What is a Function in C++?**

A **function** is a block of code designed to perform a specific task. Functions make programs **modular**, **reusable**, and easier to maintain.

* **Three Key Parts of a Function:**

**A. Declaration (Function Prototype)**

* Tells the compiler about the function’s name, return type, and parameters **before** its actual definition.
* Usually placed above main() or in a header file.

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

**B. Definition**

* Actual code that performs the task.

int add(int a, int b) {

return a + b;

}

**C. Calling**

* Executes the function from another function like main().

int main() {

int result = add(5, 3); // Calling

cout << "Sum = " << result;

return 0;

}

**2. Scope of Variables in C++**

**Scope** defines where a variable can be accessed in a program.

**🔹 Local Scope**

* Declared **inside a function or block**.
* Accessible **only within** that function or block.

void show() {

int x = 10; // Local variable

cout << x;

}

**🔹 Global Scope**

* Declared **outside all functions**.
* Accessible from **anywhere** in the file (after declaration).

int x = 100; // Global variable

int main() {

cout << x; // Accessible here

}

**➕ Tip: If a local variable has the same name as a global one, local takes precedence in that scope.**

**3. Recursion in C++**

**Recursion** is when a function calls **itself** to solve smaller parts of a problem.

**🔹 Example: Factorial Using Recursion**

int factorial(int n) {

if (n <= 1)

return 1; // Base case

else

return n \* factorial(n - 1); // Recursive call

}

int main() {

cout << "Factorial of 5 = " << factorial(5);

return 0;

}

// Output: 120

**➕ Key Components of Recursion:**

* **Base Case** – condition to stop recursion.
* **Recursive Case** – function calls itself.

**4. Function Prototypes 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 its parameters

**🔹 Why Use Function Prototypes?**

* Allows calling functions **before their definitions** in the code.
* Helps the compiler check for correct usage (number and type of arguments).

**🔹 Syntax:**

return\_type function\_name(type1, type2, ...);

**🔹 Example:**

#include <iostream>

using namespace std;

int multiply(int, int); // Function prototype

int main() {

cout << multiply(4, 5); // Can call before function is defined

return 0;

}

int multiply(int a, int b) { // Function definition

return a \* b;

}

**5. Arrays and Strings**

**1. What are arrays in C++? Difference between single-dimensional and multidimensional arrays**

**Array in C++:**

An **array** is a collection of elements of the same data type stored in **contiguous memory locations**. Instead of declaring multiple variables, you can use an array to group them under one name and use an index to access each element.

**Single-dimensional array (1D):**

A single row of elements. It's like a list.

int numbers[5] = {10, 20, 30, 40, 50};

* Access elements: numbers[0], numbers[1], etc.

**Multidimensional array (2D or more):**

An array of arrays. Most common is a **2D array** (like a matrix).

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

* Access elements: matrix[0][1] gives 2, matrix[1][2] gives 6.

**2. String handling in C++ with examples**

C++ supports string handling in **two main ways**:

* Using **C-style strings** (character arrays)
* Using **C++ string class** (part of STL - Standard Template Library)

**C-style string:**

char name[] = "Alice";

* Use #include <cstring> for functions like strlen, strcpy, strcmp.

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char str1[] = "Hello";

char str2[] = "World";

strcat(str1, str2); // Not safe without enough space in str1

cout << str1 << endl;

}

**C++ string class:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string greeting = "Hello";

string name = "Alice";

string message = greeting + ", " + name + "!";

cout << message << endl;

}

**3. How are arrays initialized in C++? Provide examples**

**1D Array Initialization:**

int arr1[5] = {1, 2, 3, 4, 5}; // Full initialization

int arr2[5] = {10, 20}; // Remaining values set to 0: {10, 20, 0, 0, 0}

int arr3[] = {100, 200, 300}; // Compiler infers size (3)

**2D Array Initialization:**

int matrix1[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

int matrix2[2][3] = {1, 2, 3, 4}; // Remaining: {1,2,3}, {4,0,0}

**4. String operations and functions in C++**

**Using C++ string class:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string s1 = "Hello";

string s2 = "World";

cout << s1 + " " + s2 << endl; // Concatenation

cout << s1.length() << endl; // Length of string

cout << s1.substr(1, 3) << endl; // Substring: "ell"

cout << s1.find("lo") << endl; // Find position: 3

cout << (s1 == s2) << endl; // Comparison: 0 (false)

}

**Common string functions:**

| **Function** | **Description** |
| --- | --- |
| length() or size() | Returns length of string |
| substr(pos, len) | Returns substring |
| find(substring) | Returns index of first occurrence |
| compare(str) | Lexicographical comparison |
| append(str) or + | Appends to string |
|  |  |

**6. Introduction to Object-Oriented Programming**

**1. Key Concepts of Object-Oriented Programming (OOP)**

OOP is a programming paradigm based on the concept of **objects**. It helps in organizing code for **reusability, scalability**, and **maintainability**. The four key principles of OOP are:

| **Concept** | **Description** |
| --- | --- |
| **Encapsulation** | Hiding the internal details of an object and only exposing what's necessary |
| **Abstraction** | Representing essential features without showing background details |
| **Inheritance** | Mechanism for one class to acquire properties of another |
| **Polymorphism** | Ability of a function or object to behave differently in different contexts |

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

**Class:**

A **class** is a user-defined data type that contains data members (variables) and member functions (methods).

**Object:**

An **object** is an instance of a class. It holds the actual data and can access the class's functions.

**✅ Example:**

#include <iostream>

using namespace std;

class Car {

public:

string brand;

int year;

void start() {

cout << brand << " is starting." << endl;

}

};

int main() {

Car myCar; // Creating an object

myCar.brand = "BMW";

myCar.year = 2023;

myCar.start(); // Accessing method

}

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

**Inheritance** allows a class (child/derived class) to acquire the properties and behaviors of another class (parent/base class). It promotes **code reuse**.

**Types of inheritance in C++:**

* Single
* Multiple
* Multilevel
* Hierarchical
* Hybrid

**✅ Example: Single Inheritance**

#include <iostream>

using namespace std;

// Base class

class Animal {

public:

void speak() {

cout << "Animal makes a sound" << endl;

}

};

// Derived class

class Dog : public Animal {

public:

void bark() {

cout << "Dog barks" << endl;

}

};

int main() {

Dog d;

d.speak(); // Inherited from Animal

d.bark(); // Defined in Dog

}

**4. What is encapsulation in C++? How is it achieved in classes?**

**Encapsulation** is the process of **binding data and functions** that manipulate the data into a single unit (class) and **restricting access** to some components.

**How it’s achieved:**

* By using **access specifiers**: private, protected, public
* Data members are usually kept **private**, and accessed through **public methods** (getters/setters).

**✅ Example:**

#include <iostream>

using namespace std;

class Employee {

private:

int salary; // Private data

public:

void setSalary(int s) {

salary = s;

}

int getSalary() {

return salary;

}

};

int main() {

Employee e;

e.setSalary(50000);

cout << "Salary: " << e.getSalary() << endl;

}