**ASSINGMENT: 3**

**INTRODUCTION TO OOP PROGRAMMING:**

* **Introduction To C++:**

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

**1. Programming Paradigm**

* **Procedural Programming**: Follows a linear, step-by-step approach using procedures or functions.
* **OOP**: Organizes code around objects—instances of classes that encapsulate data and behavior.

**2. Structure**

* **Procedural:** Code is organized into functions and procedures.
* **OOP:** Code is organized into classes and objects.

**3. Data Handling**

* **Procedural**: Data is usually separate from functions; functions operate on external data.
* **OOP:** Data and functions are bundled together into objects.

**4. Key Concepts**

* **Procedural:** Focuses on procedure (function) calls, variables, and control structures (loops, conditionals).
* **OOP**: Uses encapsulation, inheritance, polymorphism, and abstraction.

**5. Code Reusability**

* **Procedural:** Code reuse is done via functions, but harder to manage for large systems.
* **OOP**: Encourages reuse through inheritance and polymorphism.

**6. Maintenance**

* **Procedural:** Can become hard to manage as complexity grows due to scattered logic.
* **OOP**: Easier to maintain and scale because of modular design using objects.

**7. Examples**

* **Procedural Languages**: C, Pascal, Fortran
* **OOP Languages**: Java, C++, Python (can do both)

1. **List and explain the main advantages of OOP over POP.**

 **Modularity** – Code is organized into objects, making it easier to manage.

 **Reusability** – Inheritance allows reuse of existing code.

 **Encapsulation** – Protects data by bundling it with related functions.

 **Maintainability** – Easier to update and scale due to object-based structure.

 **Abstraction** – Hides complexity and shows only essential features.

 **Polymorphism** – Allows one interface to be used for different implementations.

1. **Explain the steps involved in setting up a C++ development environment**

 **Install a C++ Compiler** – e.g., GCC, Clang, or MSVC.

 **Install an IDE or Code Editor** – e.g., Dev C++, VS Code, Code::Blocks, or Visual Studio.

 **Configure the IDE** – Set compiler path if needed (Dev C++ includes it by default).

 **Write a C++ Program** – Create and save a .cpp file.

 **Compile the Code** – Use the "Compile" option in the IDE.

 **Run the Program** – Use "Run" or "Compile & Run" to see the output.

1. **What are the main input/output operations in C++? Provide examples.**

* **Output Operation (std::cout):**
* Used to display data on the screen.
* cout << "Hello, World!" << endl;
* **Input Operation (std::cin)**
* Used to read user input.
* int num;
* cin >> num; // Input from user
* **Variables, Data Types, and Operators:**

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

**1. Basic Data Types**

* int – Integer values  
  *Example:* int age = 25;
* float, double – Decimal numbers  
  *Example:* float pi = 3.14;
* char – Single character  
  *Example:* char grade = 'A';
* bool – Boolean (true/false)  
  *Example:* bool isReady = true;

**2. Derived Data Types**

* Array – Collection of similar elements  
  *Example:* int nums[3] = {1, 2, 3};
* Pointer – Stores address of another variable  
  *Example:* int\* ptr = &age;
* Reference – Alias for a variable  
  *Example:* int& ref = age;
* Function – Reusable code block  
  *Example:* int add(int a, int b) { return a + b; }

**3. User-defined Data Types**

* struct, class – Custom data structures  
  *Example:* struct Person { string name; int age; };
* union – Shared memory for variables  
  *Example:* union Data { int i; float f; };
* enum – Named constants  
  *Example:* enum Color { RED, GREEN, BLUE };

**4. Void Type**

* void – No return value  
  *Example:* void show() { }

**5. Type Modifiers**

* signed, unsigned, short, long – Modify data range  
  *Example:* unsigned int x = 10;

1. **Explain the difference between implicit and explicit type conversion in C++.**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Implicit Type Conversion** | **Implicit Type Conversion** |
| **Also called** | Type Promotion / Coercion | Type Casting |
| **Who Performs It** | Compiler | Programmer |
| **Syntax** | Automatic | Manual using cast operator (e.g., (int)) |
| **Control** | No direct control | Full control over conversion |
| **Risk of Data Loss** | Low (safe conversions, like int → float) | Higher (e.g., float → int may lose precision) |
| **Purpose** | Convenience | Precision and control |
| **Example** | float b = 10; *(int to float automatically)* | int a = (int)3.14; *(float to int manually)* |

1. **What are the different types of operators in C++? Provide examples of each.**

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Description** | **Example** |
| **Arithmetic** | Basic math operations | +, -, \*, /, %, 🡪a+b |
| **Relational** | Compare two operations | ==, !=,>,<,🡪a>b |
| **Logical** | Logical decision | &&, **`** |
| **Assignment** | Assign values | =, +=, -= → a += 5 |
| **Increment/Decrement** | Increase or decrease value by 1 | ++, -- → a++, --b |
| **Bitwise** | Operate on bits | &, ` |
| **Ternary** | Shorthand for if-else | (a > b) ? a : b |
| **sizeof** | Get size of data type | sizeof(int) |
| **Scope Resolution (::)** | Access global or class members | ::x (global x) |

1. **Explain the purpose and use of constants and literals in C++.**

* Constants
* Fixed values that don’t change during program execution.

Declared using:

const keyword → const int max = 100;

#define macro → #define PI 3.14

* **Literals**

Actual constant values **used directly** in code.

| **Type** | **Example** |
| --- | --- |
| Integer | 10 |
| Float | 3.14 |
| Character | 'A' |
| String | "Hello" |
| Boolean | true, false |

* **Purpose**
* Prevents accidental changes
* Makes code **clear, safe**, and **maintainable**

**Control Flow Statements:**

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

* **Conditional statements** are used to perform **different actions based on different conditions** in a program. They control the **flow of execution** depending on whether a condition is true or false.
* **if-else Statement:**

Used to execute a block of code **only if a condition is true**. You can also use else or else if for alternative paths.

**Syntax**:

if (condition) {

// Executes if condition is true

} else {

// Executes if condition is false

}

* **switch statements :**

Used when you need to compare one variable against multiple constant values.

**Syntax:**

switch (expression) {

case value1:

// Code

break;

case value2:

// Code

break;

default:

// Code if no case matches

}

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **for Loop** | **while Loop** | **do-while Loop** |
| **Condition Check** | Before loop body | Before loop body | After loop body |
| **Executes at least once** | No | No | Yes |
| **Use Case** | Known number of iterations | Unknown but condition-based | Must run once, then check |
| **Syntax Complexity** | Compact | Slightly longer | Slightly longer |

1. **How are break and continue statements used in loops? Provide examples.**

* the break and continue statements are used to alter the flow of control inside loops. They help control the loop execution more precisely by either exiting or skipping certain iterations.

**Break Statement:**

* The break statement is used to **exit the loop completely**, regardless of the loop condition. When break is encountered, the loop stops immediately, and the program continues with the next statement after the loop.

Example:

#include <iostream>

using namespace std;

int main() {

for (int i = 0; i < 10; i++) {

if (i == 5) {

break; // Exit the loop when i equals 5 }

cout << i << " "; }

cout << "Loop exited";

return 0;

}

**Continue Statement:**

* The continue statement skips the current iteration of the loop and moves directly to the next iteration. It does not exit the loop but simply bypasses the remaining statements for that iteration.

**Example:**

#include <iostream>

using namespace std;

int main() {

for (int i = 0; i < 10; i++) {

if (i == 5) {

continue; // Skip the iteration when i equals 5

}

cout << i << " ";

}

cout << "\nLoop finished";

return 0;

}

1. **Explain nested control structures with an example**.

* Nested control structures are when one control structure (like if, for, while, etc.) is placed **inside another** to handle **complex logic**.
* **Common Types:**
* Nested if: One if inside another
* Nested loops: One loop inside another (for, while)
* Mixed nesting: e.g., if inside for

**i)Nested if:**

int num = 10;

if (num > 0) {

if (num % 2 == 0) {

cout << "Positive Even";

}

}

**ii)nested for:**

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

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

cout << "\* ";

}

cout << endl;

}

**Functions and Scope:**

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

* A function in C++ is a self-contained block of code that performs a specific task. It helps in reusing code, improving modularity, and making programs easier to manage.

**Function Declaration (Prototype):**

* Tells the compiler about the **function name**, **return type**, and **parameters**.
* Placed before the main() function or in a header file.

**Syntax:**

return\_type function\_name(parameter\_list);

**Function Definition:**

* Contains the **actual body** of the function.
* Specifies what the function does.

**Syntax:**

return\_type function\_name(parameter\_list) {

// function body

}

**Function Calling:**

* Invokes or **executes** the function from main() or another function.

**Syntax:**

function\_name(arguments);

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

* The scope of a variable in C++ refers to where in the program the variable can be accessed or used. It defines the lifetime and visibility of a variable.

**i)Local Variable:**

* Declared inside a function or block
* Accessible only within that block
* Destroyed when the block ends

**Example:**

void func() {

int x = 10; // Local

}

ii)**Global Variable:**

* Declared outside all functions
* Accessible throughout the entire program
* Exists for the entire program's lifetime

**Example:**

int x = 100; // Global

void func() {

cout << x;

}

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

* Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem. It's commonly used for problems that can be broken into smaller sub-problems of the same type.

**Key Components of Recursion:**

* **Base Case:** Stops the recursion to avoid infinite calls.
* **Recursive Case:** The function calls itself with a simpler or smaller input.
* **Example**:

#include <iostream>

using namespace std;

int factorial(int n) {

if (n == 0) // Base case

return 1;

else

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

}

int main() {

int num = 5;

cout << "Factorial of " << num << " is " << factorial(num);

return 0;

}

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

* function prototypes in C++ are declarations of functions that specify the function’s name, return type, and parameters before the actual definition.
* **Purpose :**
* Allow functions to be called before they are defined
* Enable type checking of arguments
* Support better code organization

**Example:**

int add(int, int); // function prototype

**Arrays and Strings:**

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

* An array in C++ is a collection of elements of the same data type stored in contiguous memory locations. Arrays are used to store multiple values in a single variable.

**i)Single-Dimensional Array:**

* A **single-dimensional array** is a **linear** collection of elements.

**Example:**

int numbers[5] = {1, 2, 3, 4, 5};

**ii)Multi-Dimensional Array:**

* A **multi-dimensional array** is an array of arrays (e.g., 2D, 3D).

**Example (2D array):**

int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};

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

* C++ provides **two main ways** to handle strings:

**1) C-style Strings:**

* Arrays of characters ending with a null character ('\0')
* Functions from the <cstring> library (e.g., strcpy, strlen, strcmp)

**Example:**

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char str1[20] = "Hello";

char str2[20];

strcpy(str2, str1); // Copy str1 to str2

cout << "Length: " << strlen(str1) << endl;

cout << "Copied String: " << str2 << endl;

return 0;

}

**2) C++ string Class:**

* More powerful and easier to use
* Comes from the <string> header
* Supports operations like concatenation, comparison, substring, etc.

**Example:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string str1 = "Hello";

string str2 = "World";

string str3 = str1 + " " + str2; // Concatenation

cout << "Combined String: " << str3 << endl;

cout << "Length: " << str3.length() << endl;

return 0;

}

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

* arrays can be initialized in several ways. Here's how you can initialize 1D and 2D arrays.
* **1D Array Initialization:**

// Declaration without initialization

int arr1[5];

// Declaration with initialization

int arr2[5] = {1, 2, 3, 4, 5};

// Partial initialization (rest will be 0)

int arr3[5] = {1, 2};

// Compiler determines size

int arr4[] = {10, 20, 30};

* **2D Array Initialization:**

int mat1[2][3]; // Uninitialized

int mat2[2][3] = {{1, 2, 3}, {4, 5, 6}}; // Fully initialized

int mat3[2][3] = {{1}, {4, 5}}; // Partially initialized

int mat4[2][3] = {1, 2, 3, 4, 5, 6}; // Flat list initialization

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

* **Using std::string (C++ Standard Library):**
* **Declare**: std::string s = "Hello";
* **Concatenate:** s += " World";
* **Length:** s.length() or s.size()
* **Access char**: s[0]
* **Substring:** s.substr(pos, len)
* **Find:** s.find("text")
* **Compare**: ==, !=, <, >
* **Insert**: s.insert(pos, "abc")
* **Erase**: s.erase(pos, len)
* **Replace**: s.replace(pos, len, "new")
* **C-style Strings (char[] with <cstring>):**
* **strlen(s)** – length
* **strcpy(dest, src)** – copy
* **strcat(dest, src)** – concatenate
* **strcmp(s1, s2)** – compare

**Introduction to Object-Oriented Programming:**

1. **Explain the key concepts of Object-Oriented Programming (OOP).**

**1. Class:** Blueprint for creating objects (defines data and functions).

**2. Object:** Instance of a class.

**3. Encapsulation:** Hides internal data; access through methods using private, public.

**4. Inheritance:** A class inherits properties/methods from another class.

**5. Polymorphism:** Same function behaves differently (via overloading or overriding).

**6. Abstraction:** Hides complex details; shows only essential features.

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

**Classes and Objects in C++:**

* A **class** is a user-defined **blueprint** for creating objects.
* An **object** is an **instance** of a class that holds actual data and can use class methods.
* **Class** defines structure.
* **Object** uses that structure with real data.

**Example:**

#include <iostream>

using namespace std;

// Class definition

class Person {

public:

string name;

void sayHello() {

cout << "Hello, my name is " << name << endl;

}

};

int main() {

Person p; // Create object

p.name = "Alice"; // Set data

p.sayHello(); // Call method

return 0;

}

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

* **Inheritance** is a feature in C++ that allows a class (**child/derived class**) to inherit properties and behaviors (members and methods) from another class (**parent/base class**).
* **Why Use Inheritance?**
* Code reuse
* Logical class hierarchy
* Extensibility
* **Types of Inheritance in C++**

1. Single (one base, one derived)
2. Multilevel
3. Multiple
4. Hierarchical
5. Hybrid

**Example:**

#include <iostream>

using namespace std;

// Base class

class Animal {

public:

void eat() {

cout << "This animal eats food." << endl;

}

};

// Derived class

class Dog : public Animal {

public:

void bark() {

cout << "The dog barks." << endl;

}

};

int main() {

Dog d;

d.eat(); // Inherited from Animal

d.bark(); // Defined in Dog

return 0;

}

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

* Encapsulation is the concept of hiding internal data and exposing only necessary parts through public methods. It helps protect data from unauthorized access and modification.
* **How It's Achieved in Classes:**

Encapsulation is achieved using access specifiers:

* private – members are accessible only within the class.
* public – members are accessible from outside the class.
* protected – used with inheritance.

Example:

#include <iostream>

using namespace std;

class Account {

private:

int balance; // Private: can't be accessed directly

public:

void setBalance(int b) {

if (b >= 0)

balance = b;

}

int getBalance() {

return balance;

}

};

int main() {

Account a;

a.setBalance(500); // Set value through public method

cout << a.getBalance() << endl; // Access value through public method

return 0;

}