**Assignment -4**

1.What is polymorphism in C++ and why is it important?

Ans-Polymorphism means **"**one thing behaving in different ways."

In C++, it allows you to use one function or method name to perform different tasks, depending on the object or input.

Why is it useful?

* Makes your code clean and flexible.
* Same function name, less confusion.
* Easy to extend and manage program.

2. Explain the concept of compile-time (static) polymorphism with examples.

Ans-It means the function to be called is decided at compile time, not at runtime.

- It’s called static because everything is fixed when the code is compiled.

Example-#include <iostream>

using namespace std;

class Math {

public:

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

double add(double a, double b) { return a + b; }

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

};

int main() {

Math m;

cout << "Sum (int): " << m.add(2, 3) << endl;

cout << "Sum (double): " << m.add(2.5, 3.7) << endl;

cout << "Sum (three ints): " << m.add(1, 2, 3) << endl;

return 0;

}

**Output:**

Sum (int): 5

Sum (double): 6.2

3. Describe the concept of runtime (dynamic) polymorphism with examples.

Ans-Runtime polymorphism means one function behaves differently depending on theobject that is calling it — and this decision is made while the program is running, not while compiling.

Example-#include <iostream>

using namespace std;

class Shape {

public:

virtual void draw() { cout << "Drawing Shape" << endl; }

};

class Circle : public Shape {

public:

void draw() override { cout << "Drawing Circle" << endl; }

};

class Square : public Shape {

public:

void draw() override { cout << "Drawing Square" << endl; }

};

int main() {

Shape \*s;

Circle c;

Square sq;

s = &c; s->draw();

s = &sq; s->draw();

return 0;

}

**Output:**

Drawing Circle

Drawing Squa

4. What is the difference between static and dynamic polymorphism?

Ans-

| **Feature** | **Static Polymorphism** | **Dynamic Polymorphism** |
| --- | --- | --- |
| Also known as | Compile-time Polymorphism | Runtime Polymorphism |
| Decision made at | Compile time | Runtime |
| Achievd by | Function Overloading, Operator Overloading | Virtual Functions + Function Overriding |
| Flexibility | Less flexible | More flexible |
| Speed | Faster (no runtime check) | Slightly slower (uses virtual table) |
| Example | void show(int) and void show(string) | Base class pointer calls derived method |
| Uses virtual keyword? | No | Yes |

5. How is polymorphism implemented in C++?

Ans-Polymorphism means one function behaves differently based on the context.  
In C++, polymorphism is implemented in two main ways:

. Compile-time Polymorphism (Static)

Implemented using:

* Function Overloading
* Operator Overloading

Function Overloading Example:

cpp

#include <iostream>

using namespace std;

class Greet {

public:

void sayHello() {

cout << "Hello!" << endl;

}

void sayHello(string name) {

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

}

};

int main() {

Greet g;

g.sayHello(); // Hello!

g.sayHello("Pooja"); // Hello, Pooja!

return 0;

}

🔹 **Operator Overloading Example**:

cpp

CopyEdit

class Complex {

public:

int real, imag;

Complex(int r, int i) : real(r), imag(i) {}

// Operator + overloading

Complex operator+(const Complex& c) {

return Complex(real + c.real, imag + c.imag);

}

};

**2. Runtime Polymorphism (Dynamic)**

Implemented using:

-Inheritance

**-**Virtual Functions

-Function Overriding

**Virtual Function Example**:

cpp

#include <iostream>

using namespace std;

class Animal {

public:

virtual void sound() {

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

}

};

class Dog : public Animal {

public:

void sound() override {

cout << "Dog barks" << endl;

}

};

int main() {

Animal\* a;

Dog d;

a = &d;

a->sound(); // Output: Dog barks

return 0;

6. What are pointers in C++ and how do they work?

Ans-A pointer is a variable that stores the memory address of another variable.

A pointer "points to" the memory location of a variable.

cpp

#include <iostream>

using namespace std;

int main() {

int num = 10;

int\* ptr = &num; // pointer stores the address of num

cout << "Value of num: " << num << endl; // 10

cout << "Address of num: " << &num << endl; // e.g. 0x61ff08

cout << "Value stored in ptr: " << ptr << endl; // same as &num

cout << "Value pointed by ptr: " << \*ptr << endl; // 10

return 0;

}

7. Explain the syntax for declaring and initializing pointers.

Ans-A pointer is a variable that stores the memory address of another variable.

**Syntax for Declaring a Pointer**

cpp

CopyEdit

datatype\* pointer\_name;

* datatype → type of the variable the pointer will point to
* \* → tells the compiler it's a pointer
* pointer\_name → name of the pointer

**Example:**

cpp

int\* ptr; // pointer to int

float\* fptr; // pointer to float

char\* cptr; // pointer to char

You can also write: int \*ptr; (Both int\* ptr; and int \*ptr; are valid)

Syntax for Initializing a Pointer

You usually initialize a pointer with the address of a variable using the & operator.

**Example:**

cpp

int num = 10;

int\* ptr = &num; // ptr now points to num

8. How do you access the value pointed to by a pointer?

Ans- **1. Declare a normal variable:**

cpp

int num = 10;

**2. Create a pointer to that variable:**

cpp

int\* ptr = &num;

* &num → gets the address of num
* ptr → now stores that address

**3. Access the value using \* (dereference operator):**

cpp

cout << \*ptr; // Output: 10

* \*ptr means “go to the address stored in ptr and get the value”

**Simple Example:**

cpp

#include <iostream>

using namespace std;

int main() {

int num = 25;

int\* ptr = &num;

cout << "Address stored in ptr: " << ptr << endl;

cout << "Value pointed to by ptr: " << \*ptr << endl;

return 0;

9. Describe the concept of pointer arithmetic.

Ans-Pointer arithmetic means performing operations (like +, -, ++, --) on pointers to move through memory.

It’s mostly usedwith arrays and dynamic memory to access elements.

10. What are the common pitfalls when using pointers?

Ans-

11. How are pointers used with objects in C++?

Ans-In C++, pointers can be used with objects just like they are used with variables of basic types. But since objects belong to a class, using pointers with them lets you dynamically create and manage objects, access their members, and pass them around efficiently.

12. Explain the process of dynamically allocating objects using pointers.

Ans-**Step-by-Step Process**

**1. Declare a Pointer to the Class**

You start by declaring a pointer that will point to an object of your class.

cpp

ClassName\* ptr;

**2. Use new to Allocate Memory**

The new keyword creates an object in heap memory and returns the address.

cpp

ptr = new ClassName();

Now, ptr holds the address of the object you just created.

**3. Access Members Using -> Operator**

You use -> to access the methods and variables of the object.

cpp

ptr->someFunction();

ptr->someVariable = value;

**4. Free Memory with delete**

When you're done with the object, release the memory to avoid memory leaks.

Cpp

delete ptr

13.Provide an example of accessing object members using pointers.

Ans-#include <iostream>

using namespace std;

class Student {

public:

string name;

int roll;

void display() {

cout << "Name: " << name << ", Roll: " << roll << endl;

}

};

int main() {

// Step 1: Create an object normally

Student s1; // Step 2: Assign values

s1.name = "Pooja";

s1.roll = 101;

// Step 3: Create a pointer to the object

Student\* ptr = &s1;

// Step 4: Access members using pointer

cout << "Using pointer:" << endl;

cout << "Name: " << ptr->name << endl; // accessing data member

cout << "Roll: " << ptr->roll << endl; // accessing data member

ptr->display(); // calling member function

return 0;

}

Output:Using pointer:

Name: Pooja

Roll: 101

Name: Pooja, Roll: 101

14.What is the difference between a pointer to an object and a reference to an object?

Ans-

| **Feature** | **Pointer** | **Reference** |
| --- | --- | --- |
| Can be null | Yes | No (must refer to something) |
| Can be reassigned | Yes (can point to other objects) | No (fixed once assigned) |
| Syntax | Uses \* and -> | Uses & and . |
| Memory | Takes separate memory | Alias (no extra memory) |
| Initialization | Can be uninitialized | Must be initialized immediately |
|  |  |  |
|  |  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |

Top of Form

Bottom of Form

15. How do you release dynamically allocated objects in C++?Top of Form

Ans-In C++, dynamically allocated objects (those created using new) must be explicitly released using the delete or delete[] operators to avoid memory leaks.

int\* ptr = new int; // dynamically allocate an integer

\*ptr = 10;

// release the memory

delete ptr;

ptr = nullptr; // optional but good practice

16. What is the this pointer in C++ and what is its significance?

Ans-The this pointer in C++ is a special pointer available inside all non-static memberfunctions of a class. It points to the object that invokedthe function**.**

Bottom of Form

Top of Form

Bottom of Form

Top of Form

Bottom of Form

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

17. How is the this pointer used in member functions?

* Ans-The this pointer is an implicit pointer available inside non-static memberfunctions.
* It points tothe object that invoked the member function.

So, when you write obj.display(), inside the display() function, this points to &obj.

18. Explain how the this pointer can be used to return the current object.

Ans-In C++, the this pointer is an implicit pointer available inside all non-static memberfunctions. It points to the current object — the object that invoked the function.

Using this to return the current object:

When you want a member function to return the current object, especially to allow chaining of function calls, you can use the this pointer.

19. What is a virtual function in C++ and why is it used?

Ans-A virtual function in C++ is a member function in a base class that you expect to be overridden in derived classes. It allows for runtime polymorphism, meaning the function that gets called is determined at runtime, not compile-time.

class Base {

public:

virtual void show() {

std::cout << "Base class show()" << std::endl;

}

};

class Derived : public Base {

public:

void show() override {

std::cout << "Derived class show()" << std::endl;

}

};

1. To achieve runtime polymorphism – the correct function is called based on the object type at runtime.
2. To override base class behavior in derived classes.
3. Useful in situations where you use base class pointers or references to point toderived class objects**.**

20. Describe the syntax for declaring a virtual function.

Ans-**Syntax for Declaring a Virtual Function:**

cpp

class Base {

public:

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

};

**Example:**

cpp

CopyEdit

class Animal {

public:

virtual void sound() { // virtual function

std::cout << "Animal makes sound" << std::endl;

}

};

**Explanation:**

* virtual: keyword to declare the function as virtual.
* void: return type (can be any valid type).
* sound(): function name.
* {}: function body (optional in base class if it’s a pure virtual function).

21. Explain the concept of a vtable (virtual table) and its role in virtual functions.

Ans-A vtable (virtual table) is a hidden lookup table created by the compiler to support runtime polymorphism when virtual functions are used.

It stores pointers to virtualfunctions of a class. Each class with virtual functions has its own vtable, and each object of that class has a hidden pointer (called the vptr) pointing to that table.

22. What is a pure virtual function and how is it declared?

Ans-A pure virtual function is a virtual function that must be overridden in a derived class. It has no definition in the base class — just a declaration.

Declaring at least one pure virtual function makes the class an abstract class, meaning you cannot create objects of that class.

**Syntax:**

cpp

class ClassName {

public:

virtual return\_type function\_name(parameters) = 0;

};

The = 0 part tells the compiler that this is a **pure** virtual function.

**Example:**

cpp

class Shape {

public:

virtual void draw() = 0; // Pure virtual function

};

**Usage with Derived Class:**

cpp

class Circle : public Shape {

public:

void draw() override {

std::cout << "Drawing a circle" << std::endl;

}

};

Why Use Pure Virtual Functions?

* To define a common interface for all derived classes.
* To enforce overriding in child classes.
* To create abstract base classes that act like templates or blueprints.

23. Provide an example of a class with pure virtual functions.

Ans-#include <iostream>

using namespace std;// Abstract base class

class Shape {

public: // Pure virtual function

virtual void draw() = 0;

virtual double area() = 0;

};

24. What are the implications of having pure virtual functions in a class?

Ans-

25. How is polymorphism implemented using inheritance and virtual functions?

Ans-Polymorphism means "many forms" — in OOP, it means the same function name can behave differently for different objects.

Implementation Stepsin C++:

class Animal {

public:

virtual void sound() {

std::cout << "Animal makes a sound\n";

}

};

class Dog : public Animal {

public:

void sound() override {

std::cout << "Dog barks\n";

}

};

class Cat : public Animal {

public:

void sound() override {

std::cout << "Cat meows\n";

}

};

Animal\* a;

Dog d;

Cat c;

a = &d;

a->sound(); // Output: Dog barks

a = &c;

a->sound(); // Output: Cat meows

26. Provide an example of implementing polymorphism with base and derived classes?

Ans-#include <iostream>

using namespace std;

// Base class

class Animal {

public:

// Virtual function

virtual void sound() {

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

}

};

// Derived class 1

class Dog : public Animal {

public:

void sound() override {

cout << "Dog barks" << endl;

}

};

// Derived class 2

class Cat : public Animal {

public:

void sound() override {

cout << "Cat meows" << endl;

}

};

// Main function

int main() {

Animal\* a; // Base class pointer

Dog d;

Cat c;

a = &d;

a->sound(); // Output: Dog barks

a = &c;

a->sound(); // Output: Cat meows

return 0;

}

27. Explain the concept of late binding in the context of polymorphism.

Ans-Late Binding (also called dynamic binding or runtime binding) is when the function to be called is determined at runtime instead of at compile time.

**In the Context of Polymorphism:**

Late binding happens when:

1. You have a base class pointer or reference.
2. You call a virtual function using that pointer.
3. The actual derived class’s version of the function is called at runtime.

**Example:**

cpp

#include <iostream>

using namespace std;

class Animal {

public:

virtual void sound() { // virtual = late binding enabled

cout << "Animal sound" << endl;

}

};

class Dog : public Animal {

public:

void sound() override {

cout << "Dog barks" << endl;

}

};

int main() {

Animal\* a;

Dog d;

a = &d;

a->sound(); // Late binding: decides at runtime to call Dog's sound()

return 0;

}

28. How does the compiler manage polymorphism in C++?

Ans-C++ manages runtime polymorphism mainly using a mechanism called the vtable (virtual table) and vptr (virtual pointer).

Let’s walk through it step by step

**1. Virtual Table (vtable)**

* For every class that has virtual functions, the compiler creates a hidden structure called a vtable.
* It’s basically a lookup table of function pointers.
* Each entry in the vtable points to the most derived version of the virtual function.

**2. Virtual Pointer (vptr)**

* Each object of a class with virtual functions contains a hidden pointer called a vptr.
* This vptr points to the vtable of the class that actually created

cpp

class Animal {

public:

virtual void sound() {

cout << "Animal sound" << endl;

}

};

class Dog : public Animal {

public:

void sound() override {

cout << "Dog barks" << endl;

}

};

Int main(){

Animal\* a = new Dog();

a->sound();

}

29. What is an abstract class in C++?

Ans-An abstract class is a class that cannot be instantiated directly and is meant to be used as a base class.

It contains at least one pure virtual function.

30. How do abstract classes differ from regular classes?

Ans-

| **Feature** | **Abstract Class** | **Regular Class** |
| --- | --- | --- |
| **Instantiation** | Cannot create objects directly | Can create objects directly |
| **Contains Pure Virtual Function?** | Yes (at least one) | No pure virtual function required |
| **Purpose** | Meant to be a base/interface class | Can be used directly |
| **Inheritance Use** | Used for polymorphism and inheritance | Can be used standalone or as base |
| **Function Implementation** | May have both implemented and pure virtual functions | Usually has all functions defined |

31. Explain the role of abstract methods in abstract classes.

Ans-An abstract class has at least one pure virtual function and cannot be instantiated.

**Detailed Explanation**

* **Abstract Method**: Pure virtual function (= 0).
* **Usage**: Defines a contract for derived classes.

**Example Programs**

1. **Simple Abstract Class**

cpp

#include <iostream>

using namespace std;

class Game {

public:

virtual void play() = 0;

};

class Chess : public Game {

public:

void play() override { cout << "Playing Chess" << endl; }

};

int main() {

Game \*g = new Chess();

g->play();

delete g;

return 0;

}

**Output:**

text

Playing Chess

Top of Form

Bottom of Form

Top of Form

Bottom of Form

Top of Form

Bottom of Form32. Provide an example of defining and using an abstract class.

Ans-#include <iostream>

using namespace std;

// Abstract Class

class Shape {

public:

// Abstract method (pure virtual function)

virtual void draw() = 0;

// Regular method

void display() {

cout << "This is a shape." << endl;

}

};

// Derived class 1

class Circle : public Shape {

public:

void draw() override {

cout << "Drawing a Circle" << endl;

}

};

// Derived class 2

class Rectangle : public Shape {

public:

void draw() override {

cout << "Drawing a Rectangle" << endl;

}

};

int main() {

Shape\* s; // Base class pointer

Circle c;

Rectangle r;

s = &c;

s->draw(); // Output: Drawing a Circle

s->display(); // Output: This is a shape.

s = &r;

s->draw(); // Output: Drawing a Rectangle

s->display(); // Output: This is a shape.

return 0;

}

33. What are the benefits of using abstract classes in C++?

1. Ans-**Define a Blueprint**  
   Abstract classes help you create a common blueprint for your derived classes. All the derived classes must follow this blueprint.

*Example:*  
 If Shape is abstract with a method draw(), every derived class like Circle or Rectangle must implement draw().

1. **Easy to Use Polymorphism**  
   Abstract classes allow **polymorphism**, so you can use base class pointers and call different functions depending on the object type.

*Example:*  
You can store a Circle or Rectangle in a Shape\* pointer, and the correct draw() function will be called at runtime.

3.**Forces Derived Classes to Follow Rules**  
Abstract classes **force** derived classes to implement important functions. No function is missing!

*Example:*  
If you have an abstract class Animal with a speak() method, every derived class (like Dog, Cat) **must** implement the speak() function.

4.**Code Reusability**  
Abstract classes let you reuse common code in the base class, while specific behavior is added in derived classes.

5.**Organized and Easy to Expand**  
Abstract classes make your code organized and easy to add more features. You can always add more derived classes without changing existing.

34. What is exception handling in C++ and why is it important.

Ans**-Exception Handling in C++**

It’s a way to handle runtime errors without crashing your program. You use:

* try block: Code that might throw an error.
* throw statement: Used to raise the error.
* catch block: Used to handle the error.

Why it’s Important:

* Prevents crashes: Handle errors without stopping the program.
* Keeps code clean: Separates error-handling from normal code.
* Improves debugging: Helps you find and fix errors easily.

35. Describe the syntax for throwing and catching exceptions in C++.

Ans-

36. Explain the concept of try, catch, and throw blocks.

Ans-

Top of Form

Bottom of Form

Top of Form

Bottom of Form

37. What is the role of the catch block in exception handling.

1. Ans-**Catches Exceptions**  
   The catch block is used to **catch** exceptions that are thrown by the throw statement in the try block.  
   It allows the program to **respond to errors** without crashing, by handling the issue appropriately.
2. **Handles Errors**  
   When an exception is thrown, control is transferred to the corresponding catch block. The catch block contains code that **handles the error** (e.g., logging, displaying a message, recovering from the error).
3. **Specifies Exception Type**  
   The catch block can specify the type of exception it is designed to catch. If an exception of that type is thrown, the catch block executes; otherwise, it skips that block and moves on.
4. **Prevents Program Crash**  
   Without a catch block, any unhandled exceptions would cause the program to terminate abruptly. The catch block ensures that the program can **recover** or at least exit gracefully.

38. Provide an example of handling multiple exceptions in C++.

Ans-#include <iostream>

using namespace std;

int divide(int a, int b) {

if (b == 0) {

throw "Error: Division by zero!"; // Division by zero exception

}

return a / b;

}

void checkNegative(int number) {

if (number < 0) {

throw "Error: Negative number not allowed!"; // Negative number exception

}

}

int main() {

try {

int num1 = 10;

int num2 = 0;

int num3 = -5;

// Attempting to divide by zero

int result = divide(num1, num2);

cout << "Division result: " << result << endl;

// Checking for negative number

checkNegative(num3);

cout << "Valid number: " << num3 << endl;

}

catch (const char\* e) { // Catch division by zero exception

cout << "Caught exception: " << e << endl;

}

catch (const char\* e) { // Catch negative number exception

cout << "Caught exception: " << e << endl;

}

return 0;

}

39. How does the throw keyword work in exception handling?

Ans-**How the throw Keyword Works:**

In C++, the throw keyword is used to raise or **"**throw**"** an exception when an error or exceptional condition is detected. It is a way of transferring control from the point where the error occurred to a catch block that can handle the exception.

#include <iostream>

using namespace std;

void checkDivision(int denominator) {

if (denominator == 0) {

throw "Error: Division by zero!"; // Throwing exception

}

cout << "Division is possible!" << endl;

}

int main() {

try {

int a = 10, b = 0;

checkDivision(b); // Will throw exception because b = 0

}

catch (const char\* msg) { // Catching exception

cout << "Caught exception: " << msg << endl;

}

return 0;

}

40. What is the purpose of the finally block in exception handling?

Ans-In exception handling, the finally block is used to define code that should always run, regardless of whether an exception was thrown or not. It is typically used for cleanup tasks such as closing files, releasing resources, or closing database connections.

Even if an exception occurs in the try block and is caught by the except block, the code in the finally block will still execute. This ensures that important finalization tasks are not skipped, regardless of the program's flow.

For example:

Cpp

41. How do you create custom exception classes in C++?

Ans-In C++, you can create custom exception classes by inheriting from the standard exception class (std::exception) or any of its derived classes. By creating a custom exception class, you can include additional information (like error codes or custom messages) to make the exception more informative.

Here's how to create a custom exception class in C++:

42. What are templates in C++ and why are they useful?

Ans-Templates in C++ are a feature that allows you to define generic functions and classes. With templates, you can write code that works with any data type, providing flexibility and reusability. Instead of writing the same code for different data types (e.g., int, float, or custom types), templates allow you to write the code once and use it with different types.

There are two main types of templates in C++:

1. **Function Templates**: Used to define generic functions.
2. **Class Templates**: Used to define generic classes.

**Why are Templates Useful?**

1. **Code Reusability**: Templates allow you to write generic functions and classes that can be reused with different data types. This reduces the need to duplicate code.
2. **Type Safety**: Templates provide type checking at compile-time, ensuring that operations are performed on compatible data types.
3. **Flexibility**: With templates, you can create more flexible code that can work with any data type, including user-defined types.
4. **Improved Maintainability**: Templates simplify maintenance because there is only one version of the function or class, reducing the complexity and amount of code.
5. **Performance**: Templates allow for the generation of highly optimized code for different data types without runtime overhead.

**Function Template Example**

cpp

#include <iostream>

using namespace std;

// Function template to find the maximum of two values

template <typename T>

T getMax(T a, T b) {

return (a > b) ? a : b;

}

int main() {

int intResult = getMax(3, 7); // Calls template with int

double doubleResult = getMax(5.6, 3.2); // Calls template with double

cout << "Max of 3 and 7: " << intResult << endl;

cout << "Max of 5.6 and 3.2: " << doubleResult << endl;

return 0;

}

cout << getMax(3, 7) << endl; // Calls template with int

cout << getMax(5.6, 3.2) << endl; // Calls template with double

cout << getMax("apple", "banana") << endl; // Calls specialized template for string

return 0;}

43. Describe the syntax for defining a function template.

Ans-The general syntax for defining a **function template** is:

cpp

template <typename T>

return\_type function\_name(T parameter1, T parameter2, ...) {

// function body using type T

}

**Keywords:**

* template – Introduces the template definition.
* typename or class – Used to declare a generic type (both are interchangeable here).
* T – A placeholder for the data type (can be any valid identifier like T, U, etc.).

You can also use multiple type parameters like template <typename T1, typename T2>

**Example: Template to Find the Maximum of Two Values**

cpp

#include <iostream>

using namespace std;

template <typename T>

T getMax(T a, T b) {

return (a > b) ? a : b;

}

int main() {

cout << "Max of 10 and 20: " << getMax(10, 20) << endl;

cout << "Max of 3.5 and 2.1: " << getMax(3.5, 2.1) << endl;

return 0;

}

44. Provide an example of a function template that performs a generic operation.

Ans-#include <iostream>

using namespace std;

// Function template to swap two values

template <typename T>

void swapValues(T &a, T &b) {

T temp = a;

a = b;

b = temp;

}

int main() {

int x = 10, y = 20;

cout << "Before swap (int): x = " << x << ", y = " << y << endl;

swapValues(x, y);

cout << "After swap (int): x = " << x << ", y = " << y << endl;

double p = 3.14, q = 2.71;

cout << "Before swap (double): p = " << p << ", q = " << q << endl;

swapValues(p, q);

cout << "After swap (double): p = " << p << ", q = " << q << endl;

return 0;

}

45. What is a class template and how is it different from a function template?

Ans-A class template allows you to create a generic class — a blueprint for classes that can work with any data type.

Just like function templates make functions flexible with data types, class templates make entire classes flexible.

Difference Between Class Template and Function Template**:**

| **Feature** | **Function Template** | **Class Template** |
| --- | --- | --- |
| Purpose | Makes a function generic | Makes a whole class generic |
| Scope of T | Only within the function | For the entire class (data members + methods) |
| Complexity | Usually simpler | More complex, can involve multiple functions |
| Use Case Example | swap(), getMax() | Pair<T>, Stack<T>, Vector<T> |

46. Explain the syntax for defining a class template.

Ans-#include <iostream>

using namespace std;

template <typename T>

T add(T num1, T num2) {

return (num1 + num2);

}

int main() {

int result1;

double result2;

// calling with int parameters

result1 = add<int>(2, 3);

cout << "2 + 3 = " << result1 << endl;

// calling with double parameters

result2 = add<double>(2.2, 3.3);

cout << "2.2 + 3.3 = " << result2 << endl;

return 0;

}

47. Provide an example of a class template that implements a generic data structure.

Ans-#include <iostream>

using namespace std;

// Class template

template <class T>

class Number {

private:

// Variable of type T

T num;

public:

Number(T n) : num(n) {} // constructor

T getNum() {

return num;

}

};

int main() {

// create object with int type

Number<int> numberInt(7);

// create object with double type

Number<double> numberDouble(7.7);

cout << "int Number = " << numberInt.getNum() << endl;

cout << "double Number = " << numberDouble.getNum() << endl;

return 0;

}

48. How do you instantiate a template class in C++?

Ans-instantiation means creating an object of a template class by specifying the actual datatype you want to use in place of the template parameter.

template <typename T>

class Box {

private:

T value;

public:

Box(T val) : value(val) {}

void show() {

cout << "Value: " << value << endl;

}

};

int main() {

Box<int> intBox(42); // T is int

Box<string> strBox("Pooja"); // T is string

intBox.show(); // Output: Value: 42

strBox.show(); // Output: Value: Pooja

return 0;

}

49. What are the advantages of using templates over traditional class inheritance?

Ans-

| **Feature** | **Templates (Generic Programming)** | **Inheritance (OOP)** |
| --- | --- | --- |
| **1. Code Reusability** | Allows writing a single class/function for any type. | Also reusable, but for specific base/derived relationships. |
| **2. Type Flexibility** | Works with any data type (int, float, custom types). | Tied to a specific type hierarchy. |
| **3. Compile-Time Resolution** | Template code is generated and checked at compile time (faster, no virtual overhead). | Uses runtime polymorphism, which adds slight overhead. |
| **4. Performance** | No virtual functions = better performance. | Uses vtables for polymorphism = small performance cost. |
| **5. No Base Class Required** | Works without needing a common base class. | Requires base class for polymorphism. |
| **6. Type Safety** | Type checking is done at compile-time. | Also type-safe, but can be looser with base class pointers. |

50. How do templates promote code reusability in C++?

Ans-Code reusability means writing code once and using it multiple times for different data types or scenarios — without rewriting the same logic again and again.