ASSIGNMENT=4

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

**Ans: Polymorphism** in C++ means **"many forms"** — it allows the same function name, operator, or interface to behave **differently based on context**.

It is a core concept of **Object-Oriented Programming (OOP)** that enables **flexibility**, **extensibility**, and **code reuse**.

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Compile-time** | Resolved at compile time | Function Overloading, Operator Overloading |
| **Run-time** | Resolved at run time | Function Overriding via Virtual Functions |

1.Function Overloading (Compile-time):

#include <iostream>

using namespace std;

class Printer {

public:

void print(int num) {

cout << "Printing int: " << num << endl;

}

void print(string text) {

cout << "Printing string: " << text << endl;

}

};

int main() {

Printer p;

p.print(42);

p.print("Hello");

return 0;

}

2. Function Overriding (Run-time Polymorphism):

#include <iostream>

using namespace std;

class Animal {

public:

virtual void speak() {

cout << "Animal speaks" << endl;

}

};

class Dog : public Animal {

public:

void speak() override {

cout << "Dog barks" << endl;

}

};

int main() {

Animal\* a;

Dog d;

a = &d;

a->speak(); // Outputs "Dog barks" because of run-time polymorphism

return 0;

}

| **Benefit** | **Description** |
| --- | --- |
| **Flexibility** | Allows the same interface to be used for different data types or behaviors. |
| **Code Reuse** | Avoids code duplication using common interfaces |
| **Extensibility** | Easy to add new classes/functions without changing existing code |
| **Simplifies Maintenance** | Code is easier to read, manage, and scale. |
|  |  |

Que.2: Explain the concept of compile time polymorphism(static) with example:

Ans: **Compile-time polymorphism**, also known as **static polymorphism**, is resolved **during compilation**. It allows multiple functions or operators to have the same name but behave differently based on their **parameter types or number**.

**🔹 Main Features:**

* Achieved using:
  1. **Function Overloading**
  2. **Operator Overloading**

| **Feature** | **Description** |
| --- | --- |
| **When resolved** | At compile time |
| **Techniques used** | Function overloading, operator overloading |
| **Advantage** | Increases readability and code reuse |
| **Limitation** | Less flexible than run-time polymorphism. |

Que.3: Describe the concept of runtime polymorphism with examples:

Ans: **Runtime polymorphism** occurs when the **function to be executed is determined at runtime**, not at compile time. It's typically achieved using **inheritance** and **virtual functions**.

| **Feature** | **Description** |
| --- | --- |
| **Virtual Function** | A function in the base class that can be overridden in derived classes |
| **Base Pointer** | Points to derived object |
| **Function Resolution** | Happens at **runtime**, using **vtable (virtual table)** mechanism |

**Advantages of Runtime Polymorphism**

| **Benefit** | **Description** |
| --- | --- |
| **Flexibility** | One interface, many implementations |
| **Extensibility** | Easy to add new classes without changing existing code |
| **Code Reuse** | Write general-purpose code using base class pointers. |

**Runtime polymorphism** is achieved using **virtual functions** and **inheritance**.

It allows **different behaviors** for the same function call based on the object type at runtime.

Key for **flexible and maintainable** object-oriented software.

Que.4: what is the difference between static and dynamic polymorphism:

Ans:

| **Feature** | **Static Polymorphism** | **Dynamic Polymorphism** |
| --- | --- | --- |
| **When Resolved** | At **compile time** | At **runtime** |
| **How Achieved** | Function overloading, Operator overloading | Virtual functions + Inheritance |
| **Performance** | **Faster** (no runtime overhead) | **Slightly slower** due to virtual table lookup |
| **Flexibility** | Less flexible | More flexible and extensible |
| **Function Binding** | **Early binding** | **Late binding** |
| **Use Case** | When behavior is known at compile time | When behavior varies based on object at runtime |
| **Example** | void add(int, int) and void add(double, double) | Overriding virtual void draw() in derived classes |

Que.5: How is polymorphism implemented in C++?

Ans: Polymorphism in C++ is implemented in **two main ways**:

1. **Compile-time polymorphism** (Static)
2. **Run-time polymorphism** (Dynamic)

**1. Compile-Time Polymorphism (Static)**

**Implemented Using:**

* **Function Overloading** – Same function name, different parameters.
* **Operator Overloading** – Redefining operators for user-defined types

**Example:** class Print {

public:

void show(int x) {

cout << "Integer: " << x << endl;

}

void show(string s) {

cout << "String: " << s << endl;

}

};

**2. Run-Time Polymorphism (Dynamic)**

Implemented Using:

* Inheritance
* Virtual Functions
* Base class pointers or references.

**Example:** class Animal {

public:

virtual void speak() {

cout << "Animal speaks" << endl;

}

};

class Dog : public Animal {

public:

void speak() override {

cout << "Dog barks" << endl;

}

};

int main() {

Animal\* a;

Dog d;

a = &d;

a->speak(); // Calls Dog's speak() due to virtual function

}

Que.6: What are pointers in C++ and how they worked?

Ans: In C++, a **pointer** is a variable that **stores the memory address** of another variable.

| **Term** | **Meaning** |
| --- | --- |
| & (Address-of) | Gets the memory address of a variable |
| \* (Dereference) | Accesses the value at a memory address |
| Pointer | Variable that holds an address |

#include <iostream>

using namespace std;

int main() {

int x = 10;

int\* ptr = &x; // Pointer to int, stores address of x

cout << "Value of x: " << x << endl;

cout << "Address of x: " << &x << endl;

cout << "Value stored in ptr (address): " << ptr << endl;

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

return 0;

}

* **Benefits of pointer:**

| **Benefit** | **Description** |
| --- | --- |
| **Efficiency** | Direct memory access |
| **Dynamic memory** | Needed for new / delete |
| **Function arguments** | Pass by reference using pointers |
| **Data structures** | Essential for linked lists, trees, etc. |
| Que.7: Explain the syntax for | Declaring and initializing the pointer: |

Ans: A **pointer** is declared using the \* operator and is initialized with the **address of a variable** using the & (address-of) operator.

**Syntax:** dataType\* pointerName;

**Declaration and initializing:**

int x = 10; // Regular integer variable

int\* ptr = &x; // Pointer to int, initialized with address of x.

**Printing and dereferencing:**

#include <iostream>

using namespace std;

int main() {

int x = 10;

int\* ptr = &x;

cout << "Value of x: " << x << endl;

cout << "Address of x: " << &x << endl;

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

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

return 0;

}

Que.8: How do you access the valued pointed to by a pointer?

Ans: To **access the value** that a pointer is pointing to, use the **dereference operator \***.

**Syntax:** \*pointerName.

Que.9: Describe the concept of pointer arithmetic:

Ans: **Pointer arithmetic** allows you to perform mathematical operations on pointers to navigate through memory, especially useful when working with **arrays**.

| **Operation** | **Meaning** |
| --- | --- |
| ptr + n | Moves the pointer forward by n elements (not bytes!) |
| ptr - n | Moves the pointer backward by n elements |
| ptr++, ++ptr | Advances to the next memory location |
| ptr--, --ptr | Moves to the previous memory location |
| ptr1 - ptr2 | Returns the number of elements between two pointers |
| ptr[n] | Same as \*(ptr + n) |

**Example:**

#include <iostream>

using namespace std;

int main() {

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

int\* ptr = arr; // Points to arr[0]

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

cout << "Value at ptr + 1: " << \*(ptr + 1) << endl; // 20

cout << "Value at ptr + 2: " << \*(ptr + 2) << endl; // 30

ptr++; // Move pointer to next element (arr[1])

cout << "After ptr++, value: " << \*ptr << endl; // 20

return 0;

}

Que.10: Explain the process of dynamically allocating objects with pointer:

Ans: In C++, you can **dynamically allocate memory** for objects using the new keyword. This allows you to create objects **at runtime**, rather than at compile time.

**Why use dynamic Allocation:**

| **Reason** | **Benefit** |
| --- | --- |
| Unknown size at compile time | Allocate memory based on user input/data |
| Lifetime control | Object exists until explicitly deleted |
| Flexibility | Create objects only when needed |

**Syntax:**

ClassName\* ptr = new ClassName();

**Example:**

#include <iostream>

using namespace std;

class Car {

public:

void drive() {

cout << "Driving the car!" << endl;

}

};

int main() {

Car\* myCar = new Car(); // Dynamically create Car object

myCar->drive(); // Access member using pointer

delete myCar; // Free allocated memory

return 0;

}

Que.11: Provide an example of accessing object members using pointers:

Ans: When you have a pointer to an object, you use the **arrow operator ->** to access its members (methods or variables).

#include <iostream>

using namespace std;

class Person {

public:

string name;

int age;

void display() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

// Dynamically allocate object

Person\* p = new Person();

// Access and assign member variables using ->

p->name = "Alice";

p->age = 30;

// Call member function using ->

p->display();

// Free the allocated memory

delete p;

return 0;

}

Que.12: Explain difference between pointer to an object and reference to an object:

| **Feature** | **Pointer to an Object** | **Reference to an Object** |
| --- | --- | --- |
| **Declaration Syntax** | ClassName\* ptr = &obj; | ClassName& ref = obj; |
| **Can be null?** | Yes, pointers can be nullptr. | No, a reference **must** refer to an object. |
| **Re-assignment** | Can be reassigned to point to another object. | Cannot be reassigned once initialized. |
| **Memory Allocation** | Pointers are variables that store addresses. | References are just aliases for objects. |
| **Dereferencing** | Use \*ptr to access the object’s members. | Use ref directly to access the object. |
| **Use of & (Address-of)** | &ptr gives the address of the pointer itself. | Reference is already an alias to the object. |
| **Use of \* (Dereference)** | Use \*ptr to dereference the pointer and access the object. | No dereferencing needed. |

Que.13: How is the this pointer used in member function:

Ans: In C++, **this** is a special pointer that points to the current object of a class. It is automatically passed to every non-static member function of the class.

The **this pointer** allows you to access the object itself from within its own member functions.

**🔹 Characteristics of the this Pointer**

* **Type**: The this pointer is of type ClassName\*, where ClassName is the name of the class.
* **Access**: It can be used to access members of the object inside the member function.
* **Implicit**: You cannot modify the this pointer (it's read-only). It implicitly points to the object that called the member function.
* **Non-static functions**: Only available in non-static member functions (since static methods don’t belong to any specific object).

#include <iostream>

using namespace std;

class Rectangle {

public:

int width, height;

Rectangle(int w, int h) {

width = w;

height = h;

}

// Member function using the 'this' pointer

void displayDimensions() {

// Accessing object members via 'this'

cout << "Width: " << this->width << ", Height: " << this->height << endl;

}

// Another example of returning the 'this' pointer

Rectangle\* getThis() {

return this; // Returning the 'this' pointer

}

};

int main() {

Rectangle rect(10, 5);

rect.displayDimensions(); // Calling member function

// Using 'this' pointer in a function

Rectangle\* ptr = rect.getThis();

cout << "Address of object: " << ptr << endl;

return 0;

}

Que.14: What is the virtual function in c++ and why it is used:

Ans: A **virtual function** in C++ is a member function that is **declared in the base class** and is **overridden in derived classes**. It allows for **runtime polymorphism**, meaning the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.

**🔹 Why Virtual Functions are Used**

* **Runtime Polymorphism**: Virtual functions enable **dynamic method dispatch** at runtime. When a function is called through a base class pointer or reference, the **correct function** is called according to the actual object type (not the base type).
* **Allows Overriding**: Derived classes can override base class functions, and when the function is called through a base class pointer, the overridden function of the derived class is executed.
* **Supports Inheritance**: They allow for more **flexible code** when you are dealing with **inheritance** and **base/derived relationships**.

**🔸 Syntax for Virtual Functions**

1. **Declare a virtual function in the base class** using the virtual keyword.
2. **Override** the function in the derived class.
3. Use a **base class pointer/reference** to call the function.

**Example:**

#include <iostream>

using namespace std;

class Animal {

public:

// Declare a virtual function

virtual void sound() {

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

}

};

class Dog : public Animal {

public:

// Override the virtual function

void sound() override {

cout << "Dog barks" << endl;

}

};

class Cat : public Animal {

public:

// Override the virtual function

void sound() override {

cout << "Cat meows" << endl;

}

};

int main() {

Animal\* animal1 = new Dog(); // Base class pointer, points to Dog object

Animal\* animal2 = new Cat(); // Base class pointer, points to Cat object

// Calls the overridden version of sound() based on actual object type

animal1->sound(); // Output: Dog barks

animal2->sound(); // Output: Cat meows

delete animal1; // Clean up dynamically allocated memory

delete animal2; // Clean up dynamically allocated memory

return 0;

}

Que.15: Explain the concept of virtual table in its role in virtual function:

Ans: A **virtual table (v-table)** is a table (usually an array of function pointers) maintained per class that has **virtual functions**. It maps virtual functions to their **most-derived implementations**.

**For each class with virtual functions**, the compiler generates a v-table.

Each object of such a class contains a **hidden pointer** (called vptr or vtable pointer) to the v-table of its class.

When a virtual function is called on an object (via a pointer/reference to the base class), the compiler:

* Uses the vptr to look up the function in the v-table,
* Then calls the correct function — potentially from a derived class.
* **Virtual table is created at compile time**, but the function resolution happens at **runtime**.
* Only classes with **at least one virtual function** have a v-table.
* Virtual functions enable **dynamic dispatch** through this mechanism.

Que.16: What is a pure virtual function and how it is declared?

Ans: A **pure virtual function** is a virtual function that **must be overridden** by derived classes. It has **no implementation in the base class**, making the class **abstract** — meaning you **cannot instantiate** it directly.

They are used to define **interfaces** or **abstract base classes** — where you want to specify that all derived classes **must implement** certain behavior.

**Declaration:**

class Shape {

public:

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

};

Que.17. Provide an example of a class with pure virtual function:

Ans: class Shape {

public:

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

};

class Circle : public Shape {

public:

void draw() override {

std::cout << "Drawing Circle\n";

}

};

int main() {

// Shape s; // Error: cannot instantiate abstract class

Circle c;

c.draw(); // Output: Drawing Circle

}

Que.18: What are the implications of pure virtual functions in a class:

Ans: Declaring one or more **pure virtual functions** in a class has several important implications in C++:

**1.Makes the Class Abstract:**

* Any class containing at least one pure virtual function becomes an **abstract class**.
* **Abstract classes cannot be instantiated directly.**

**2. Forces Derived Classes to Implement the Function**

* **Derived classes must provide an override for all inherited pure virtual functions.**
* **Otherwise, they also become abstract and can't be instantiated.**

**3. Defines an Interface**

* **Abstract classes with only pure virtual functions are often used as interfaces.**
* **This enables polymorphic behavior and allows for decoupling implementation from interface.**

**4. Enables Runtime Polymorphism**

* **Using base class pointers/references, you can call overridden methods in derived classes via virtual dispatch.**

**5. Provides Design Clarity**

* **Signals that a class is meant to serve as a base and should not be used directly.**
* **Encourages a clear, organized class hierarchy.**

Que.19: How is polymorphism implemented using inheritance and virtual functions:

Ans: **Polymorphism** in C++ allows the same interface (e.g., a base class pointer or reference) to be used for different underlying types (i.e., derived classes).  
This is achieved through **inheritance** and **virtual functions**, enabling **dynamic (runtime) polymorphism**.

**1.Inheritance Sets Up the Hierarchy**

* You create a **base class** and one or more **derived classes**.
* The derived classes **inherit** behavior from the base but can override it.

**2.Virtual Functions Enable Dynamic Dispatch**

* Marking a function virtual in the base class tells the compiler to **resolve calls at runtime**, not compile time.
* This ensures the **correct function for the actual object type** is called, even if accessed through a base pointer or reference.

**3.How the Compiler Implements It (Internals)**

* The compiler builds a **virtual table (v-table)** for each class with virtual functions.
* Each object stores a hidden pointer to its class’s v-table (called the **vptr**).
* When you call a virtual function:
  + The vptr is used to look up the correct function in the v-table,
  + The function is then called, based on the **actual type** of the object.

Que.20: Provide an example of polymorphism with base and derived class?

#include <iostream>

using namespace std;

// Base class

class Animal {

public:

virtual void speak() {

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

}

};

// Derived class 1

class Dog : public Animal {

public:

void speak() override {

cout << "Dog barks." << endl;

}

};

// Derived class 2

class Cat : public Animal {

public:

void speak() override {

cout << "Cat meows." << endl;

}

};

// Function that uses polymorphism

void makeAnimalSpeak(Animal\* a) {

a->speak(); // Runtime decision: which 'speak' to call

}

int main() {

Animal a;

Dog d;

Cat c;

makeAnimalSpeak(&a); // Output: Animal makes a sound.

makeAnimalSpeak(&d); // Output: Dog barks.

makeAnimalSpeak(&c); // Output: Cat meows.

return 0;

}

Que.21: Explain the concept of late binding in the context of polymorphism:

Ans: **Late binding**, also known as **dynamic binding**, is a concept where the **function to be called is determined at runtime**, rather than at compile time.

In the context of **polymorphism**, late binding allows a **base class pointer or reference** to invoke **derived class implementations** of virtual functions — even though the exact object type isn't known until the program runs.

* When a function is **not virtual**, the call is **early bound** (i.e., resolved at compile time).
* When a function is **virtual**, the call is **late bound** — resolved at **runtime** using the virtual table (v-table).

**Example:**

#include <iostream>

using namespace std;

class Animal {

public:

virtual void speak() {

cout << "Animal sound" << endl;

}

};

class Dog : public Animal {

public:

void speak() override {

cout << "Bark" << endl;

}

};

int main() {

Animal\* a = new Dog();

a->speak(); // Late binding: calls Dog::speak() at runtime

delete a;

}

**Explanation**:

* The compiler knows a is an Animal\*, but it defers the decision of which speak() to call.
* At runtime, it sees a actually points to a Dog, and calls Dog::speak().

Que.22: How does the compiler manage polymorphism in C++?

Ans: In C++, **polymorphism** (specifically **runtime polymorphism**) is implemented and managed by the compiler using a combination of:

* **Virtual functions**
* **V-tables (virtual tables)**
* **V-pointers (vptrs)**

**1. Virtual Table (V-Table)**

* For every class with **virtual functions**, the compiler creates a **v-table** — an internal table containing **function pointers** to the virtual functions.
* If a derived class **overrides** a virtual function, its v-table stores a pointer to the **overridden function**.

**🧷 2. V-Ptr (Virtual Table Pointer)**

* Each object of a class with virtual functions has a hidden pointer called the **vptr**.
* This vptr points to the **v-table of its class**, enabling the object to call the correct function at runtime.

**🕹️ 3. Dynamic Dispatch (Late Binding)**

* When a **virtual function is called** through a base class pointer/reference, the compiler:
  + Uses the **vptr** in the object to find the correct **v-table**.
  + Looks up the function’s address in the v-table.
  + Calls the appropriate function — even if it belongs to a derived class.

**Example:**

class Base {

public:

virtual void show() { cout << "Base\n"; }

};

class Derived : public Base {

public:

void show() override { cout << "Derived\n"; }

};

int main() {

Base\* b = new Derived();

b->show(); // Compiler uses vptr and v-table to call Derived::show()

}

Que.23: What is an abstract class in C++?

Ans: An **abstract class** in C++ is a class that **cannot be instantiated** on its own.  
It is designed to be a **base class** from which other classes are derived, and it typically defines a common interface using **pure virtual functions**.

**Example:**

class Shape {

public:

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

};

class Circle : public Shape {

public:

void draw() override {

std::cout << "Drawing Circle" << std::endl;

}

};

int main() {

// Shape s; // Error

Circle c; // OK

Shape\* ptr = &c; // Polymorphic usage

ptr->draw(); // Output: Drawing Circle

}

Que.24: How do abstract class differ from regular class:

Ans:

| **Feature** | **Abstract Class** | **Regular (Concrete) Class** |
| --- | --- | --- |
| **Pure Virtual Functions** | Must have **at least one** pure virtual function | May or may not have virtual functions |
| **Instantiation** | Cannot be instantiated directly | Can be instantiated |
| **Usage** | Used as a **base/interface** for derived classes | Used to create objects and define complete logic |
| **Polymorphism Support** | Supports polymorphism via base pointers/references | Also supports polymorphism if virtuals are used |
| **Purpose** | Define a **contract/interface** to be implemented | Define **full behavior** |
| **Requirement for Derived Class** | Must **override** pure virtual functions | No such requirement |

Que.25: Explain the role of abstract methods in abstract class?

Ans: In C++, **abstract methods** are called **pure virtual functions** — they have **no implementation** in the base class and **must be overridden** by derived classes. They are central to making a class abstract.

| **Role** | **Description** |
| --- | --- |
| **Define Interface** | Abstract methods specify the **expected behavior** that all derived classes must provide. |
| **Enable Polymorphism** | Allow base class pointers/references to call **derived class implementations** dynamically. |
| **Enforce Design Contracts** | Prevent instantiation unless all required behavior is implemented. |
| **Decouple Interface from Implementation** | Let multiple classes implement the same behavior differently (e.g., draw() in Circle, Rectangle, etc.). |

Abstract methods in abstract classes **force derived classes to provide specific behavior**.

They are the foundation for **runtime polymorphism** and **interface-based design**.

You cannot **instantiate** a class with abstract methods.

Que.26: Provide an example of defining and using an abstract class?

Ans: 1.**Defining the Abstract Class**

An **abstract class** defines a common interface but doesn't provide implementations for the methods. In this case, the draw() method is a **pure virtual function**, making the class abstract.

#include <iostream>

using namespace std;

// Abstract class

class Shape {

public:

// Pure virtual function (abstract method)

virtual void draw() = 0;

// Virtual destructor

virtual ~Shape() {}

};

**2.Derived Classes**

Now, let's define two classes, Circle and Rectangle, which **inherit from Shape** and provide their own implementations of draw().

class Circle : public Shape {

public:

void draw() override {

cout << "Drawing a Circle\n";

}

};

class Rectangle : public Shape {

public:

void draw() override {

cout << "Drawing a Rectangle\n";

}

};

Que.27: What are the benefits of using abstract classes in c++?

Ans: Using **abstract classes** in C++ provides several advantages, particularly when designing scalable, maintainable, and flexible systems.

**1. Encapsulation of Interface**

* **Abstract classes** allow you to define a common **interface** for different derived classes. This helps in **hiding implementation details** and focusing on what operations the classes can perform, rather than how they do it.
* It ensures that **derived classes follow a common structure** (i.e., they must implement specific methods).

**2.Polymorphism**

* **Abstract classes enable polymorphism**, allowing you to write code that can work with **objects of different types** through a common base class pointer or reference.
* This helps in writing **generic code** that can handle objects of any class that derives from the abstract class.

**3. Design Flexibility and Extensibility**

* Abstract classes promote extensible designs by defining abstract methods (pure virtual functions) that derived classes must implement. As your application grows, you can easily add new derived classes without modifying existing code.
* This allows for easier maintenance and scalability of the software.

**4. Code Reusability and Modularity**

* **Abstract classes allow you to write generic, reusable code that works with different derived types. You can implement functionality in base classes (common behavior) while leaving specific functionality to be provided by the derived classes.**
* **This leads to better modularity as it separates the interface and implementation, encouraging cleaner and more organized code.**

Que.28: What is an exception handling in c++ and why is it important?

Ans: **Exception handling** in C++ is a mechanism to manage **runtime errors** or **exceptions** that occur during the execution of a program. It allows a program to respond to unforeseen conditions (e.g., division by zero, file not found, memory allocation failure) without crashing.

C++ provides the try, throw, and catch keywords to handle exceptions:

* **try block**: Contains the code that may throw an exception.
* **throw statement**: Used to **throw** an exception when a problem is detected.
* **catch block**: Used to **catch** and handle the thrown exception.

**Example**:

#include <iostream>

#include <stdexcept> // For standard exception classes

using namespace std;

void divide(int a, int b) {

if (b == 0) {

throw runtime\_error("Division by zero error!"); // Throw an exception

}

cout << "Result: " << a / b << endl;

}

int main() {

try {

divide(10, 0); // This will throw an exception

} catch (const runtime\_error& e) { // Catch the exception

cout << "Exception caught: " << e.what() << endl;

}

return 0;

}

Que.29: Describe the syntax for throwing and catching exception in c++?

Ans: In C++, exception handling uses the throw, try, and catch keywords to handle runtime errors. Let's break down the syntax for **throwing** and **catching** exceptions.

**1.Throwing an Exception**

To **throw** an exception, you use the throw keyword followed by an **exception object**. This object can be of any type, but typically it's either a predefined exception class (like std::runtime\_error) or a custom class that inherits from std::exception.

**Syntax:** throw expression; // expression can be an exception object

**Example:** throw std::runtime\_error("Something went wrong!");

int error\_code = 404;

throw error\_code; // Throws an integer exception

**2.Catching an Exception**

You catch exceptions in a try block using the catch keyword. When an exception is thrown, the program jumps to the matching catch block.

**Syntax:**

try {

// Code that may throw an exception

} catch (ExceptionType& e) {

// Code to handle the exception}

**Example:**

try {

// Code that might throw an exception

throw std::runtime\_error("Runtime error occurred");

} catch (const std::runtime\_error& e) {

std::cout << "Caught runtime\_error: " << e.what() << std::endl;

} catch (const std::exception& e) {

std::cout << "Caught a generic exception: " << e.what() << std::endl;

}

Que.30: Explain the concept of try, catch and throw block?

Ans: Exception handling in C++ revolves around three main keywords: try, throw, and catch. These work together to detect, report, and manage runtime errors without crashing the program.