**Assignment 4: C++ Theory Questions**

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

Answer: Polymorphism in C++ allows objects of different classes to be treated as objects of a common base class. It enables a single interface to represent different underlying forms (data types).

Importance:

* Enhances code flexibility and reusability.
* Allows functions to work with objects of various types without knowing their exact type.
* Supports extensibility, making it easier to add new classes without modifying existing code.

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

Answer: Compile-time polymorphism is achieved when the function or operator to be called is resolved during compilation. It is implemented using function overloading and operator overloading.

Example:

#include <iostream>

using namespace std;

class Calculator {

public:

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

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

};

int main() {

Calculator calc;

cout << calc.add(5, 10) << endl; // Calls int version

cout << calc.add(5.5, 10.5) << endl; // Calls double version

return 0;

}

Here, the compiler decides which add function to call based on the argument types.

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

Answer: Runtime polymorphism is achieved when the function to be called is resolved during program execution. It is implemented using virtual functions and inheritance.

Example:

#include <iostream>

using namespace std;

class Animal {

public:

virtual void sound() { cout << "Some sound" << endl; } // Virtual function

};

class Dog : public Animal {

public:

void sound() override { cout << "Woof" << endl; }

};

int main() {

Animal\* animal = new Dog(); // Base class pointer to derived class object

animal->sound(); // Calls Dog's sound() at runtime

delete animal;

return 0;

}

Here, the function call is resolved at runtime based on the actual object type.

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

Answer:

| **Feature** | **Static Polymorphism** | **Dynamic Polymorphism** |
| --- | --- | --- |
| **Binding Time** | Compile-time | Runtime |
| **Achieved Using** | Function Overloading, Operator Overloading, Templates | Virtual Functions, Inheritance |
| **Function Selection** | Based on arguments/data types (at compile time) | Based on object type (at runtime) |
| **Flexibility** | Less flexible | More flexible |
| **Performance** | Faster (no runtime overhead) | Slower (uses virtual table mechanism) |
| **Inheritance Required** | Not necessary | Necessary (uses base and derived classes) |
| **Example** | void show(int); and void show(double); | virtual void speak(); overridden in subclass |

5. How is polymorphism implemented in C++?

Answer: Polymorphism in C++ is implemented using:

Static Polymorphism: Function overloading, operator overloading, and templates.

Dynamic Polymorphism: Inheritance with virtual functions, where a base class pointer or reference can point to derived class objects, and the appropriate function is called at runtime.

Virtual Functions: Declared with the virtual keyword to enable dynamic

binding.

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

Answer: A pointer is a variable that stores the memory address of another variable. Pointers allow direct manipulation of memory, enabling efficient data handling.

How they work:

* A pointer is declared with a type (e.g., int\*) to indicate the type of data it points to.
* It holds the address of a variable, which can be accessed or modified using the pointer.

7. Explain the syntax for declaring and initializing pointers.

Answer: Syntax:

type\* pointerName; // Declaration

pointerName = &variable; // Initialization

Example:

int x = 10;

int\* ptr; // Declaration

ptr = &x; // Initialization with address of x

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

Answer: The value pointed to by a pointer is accessed using the dereference operator \*.

Example:

#include <iostream>

using namespace std;

int main() {

int x = 10;

int\* ptr = &x;

cout << \*ptr << endl; // Accesses value (10)

return 0;

}

9. Describe the concept of pointer arithmetic.

Answer: Pointer arithmetic refers to performing arithmetic operations (e.g., increment, decrement) on pointers. It adjusts the pointer's address based on the size of the data type it points to. Example:

#include <iostream>

using namespace std;

int main() {

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

int\* ptr = arr;

cout << \*ptr << endl; // Prints 10

ptr++; // Moves to next int (4 bytes forward)

cout << \*ptr << endl; // Prints 20

return 0;

}

10. What are the common pitfalls when using pointers?

Answer: the common pitfalls when using pointers are:

1. Dangling Pointers: Pointing to memory that has been freed or deleted.
2. Null Pointer Dereference: Accessing a pointer that is nullptr.
3. Memory Leaks: Forgetting to deallocate dynamically allocated memory.
4. Uninitialized Pointers: Using a pointer before assigning it a valid address.
5. Incorrect Pointer Arithmetic: Moving a pointer beyond valid memory boundaries.

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

Answer: Pointers can point to objects, allowing dynamic allocation and manipulation of objects. They are used to access object members (data or functions) using the arrow operator ->. Example:

#include <iostream>

using namespace std;

class MyClass {

public:

void display() { cout << "Hello" << endl; }

};

int main() {

MyClass\* obj = new MyClass();

obj->display(); // Access member function

delete obj;

return 0;

}

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

Answer: Dynamic allocation is done using the new operator, which allocates memory on the heap and returns a pointer to it. Steps:

1. Declare a pointer: ClassName\* ptr;.
2. Allocate memory: ptr = new ClassName;.
3. Use the object via the pointer.
4. Deallocate memory using delete ptr;.

Example of it is given in question 11

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

Answer:

#include <iostream>

using namespace std;

class Student {

public:

string name = "John";

void display() { cout << "Name: " << name << endl; }

};

int main() {

Student\* s = new Student();

cout << s->name << endl; // Access data member

s->display(); // Access member function

delete s;

return 0;

}

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

Answer:

| **Feature** | **Pointer to Object** | **Reference to Object** |
| --- | --- | --- |
| **Definition** | Stores the **memory address** of an object | An **alias (another name)** for an object |
| **Declaration** | ClassName\* ptr = &obj; | ClassName& ref = obj; |
| **Can be null?** | Yes (ptr = nullptr;) | No (must be initialized when declared) |
| **Can be reassigned?** | Yes (can point to another object) | No (once set, refers to the same object) |
| **Syntax for access** | Use -> or \* (e.g., ptr->fun()) | Use dot . like normal object (ref.fun()) |
| **Memory** | Takes separate memory (stores address) | Does not take extra memory (just an alias) |
| **Use in Functions** | Used to pass address (can be modified) | Used to pass by reference |
| **Default Behavior** | Can be uninitialized | Must be initialized immediately |

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

Answer: Use the delete operator to deallocate memory allocated with new. For arrays, use delete[].

Example:

ClassName\* ptr = new ClassName();

delete ptr; // Release single object

int\* arr = new int[10];

delete[] arr; // Release array

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

Answer: The this pointer is a special pointer available in non-static member functions of a class. It points to the object that invoked the member function. Significance:

* Resolves ambiguity between member variables and function parameters with the same name.
* Enables returning the current object for method chaining.

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

Answer: The this pointer is used to access the current object’s members explicitly. Example:

#include <iostream>

using namespace std;

class MyClass {

int x;

public:

void setX(int x) { this->x = x; } // Resolves naming conflict

void display() { cout << this->x << endl; }

};

int main() {

MyClass obj;

obj.setX(10);

obj.display();

return 0;

}

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

Answer: The this pointer is used to return \*this from a member function, enabling method chaining.

Example:

#include <iostream>

using namespace std;

class MyClass {

int x;

public:

MyClass& setX(int x) {

this->x = x;

return \*this; // Return current object

}

void display() { cout << x << endl; }

};

int main() {

MyClass obj;

obj.setX(10).display(); // Chaining

return 0;

}

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

Answer: A virtual function is a member function declared with the virtual keyword in a base class, allowing it to be overridden in derived classes. It is used to Enables runtime polymorphism, ensuring the correct derived class function is called via a base class pointer or reference.

20. Describe the syntax for declaring a virtual function.

Answer: Syntax:

virtual returnType functionName(parameters) { code }

Example:

class Base {

public:

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

};

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

Answer: A vtable is a table of function pointers created by the compiler for classes with virtual functions. Each object of the class contains a pointer (vptr) to the vtable. Its role is that at runtime, the vtable is used to resolve which virtual function to call based on the actual object type, enabling dynamic polymorphism.

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

Answer: A pure virtual function is a virtual function with no implementation in the base class, declared by assigning it to = 0.

Syntax:

virtual returnType functionName(parameters) = 0;

Example:

class Base {

public:

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

};

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

Answer:

#include <iostream>

using namespace std;

class Shape {

public:

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

};

class Circle : public Shape {

public:

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

};

int main() {

Shape\* shape = new Circle();

shape->draw();

delete shape;

return 0;

}

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

Answer: the implications of having pure virtual functions in a class are:

* The class becomes an abstract class and cannot be instantiated.
* Derived classes must override all pure virtual functions to be concrete (instantiable).
* Ensures that derived classes provide specific implementations for the abstract interface.

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

Answer: Polymorphism is implemented using:

Inheritance – A derived class inherits from a base class.

Virtual functions – A function in the base class is marked virtual and overridden in the derived class.

Base class pointer/reference – When it points to a derived class object, the overridden function is called at runtime, enabling runtime polymorphism.

Example:

cpp

Copy

Edit

class Animal {

public:

virtual void sound() { cout << "Animal sound"; }

};

class Dog : public Animal {

public:

void sound() override { cout << "Dog barks"; }

};

Animal\* a = new Dog();

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

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

Answer:

#include <iostream>

using namespace std;

class Vehicle {

public:

virtual void drive() { cout << "Driving vehicle" << endl; }

};

class Car : public Vehicle {

public:

void drive() override { cout << "Driving car" << endl; }

};

class Bike : public Vehicle {

public:

void drive() override { cout << "Riding bike" << endl; }

};

int main() {

Vehicle\* v1 = new Car();

Vehicle\* v2 = new Bike();

v1->drive(); // Calls Car's drive()

v2->drive(); // Calls Bike's drive()

delete v1;

delete v2;

return 0; }

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

Answer: Late binding (dynamic binding) refers to resolving function calls at runtime rather than compile time. It occurs with virtual functions, where the function to call is determined by the actual object type pointed to by a base class pointer or reference.

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

Ans Static Polymorphism: The compiler resolves function calls at compile time based on function signatures (overloading) or template instantiation.

Dynamic Polymorphism: The compiler creates a vtable for classes with virtual functions. At runtime, the vptr in the object is used to look up the correct function in the vtable.

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

Answer: An abstract class is a class that cannot be instantiated and is designed to be inherited. It contains at least one pure virtual function.

30. How do abstract classes differ from regular classes?

Ans Here`s the difference:

| **Feature** | **Abstract Class** | **Regular Class** |
| --- | --- | --- |
| **Definition** | A class that contains **at least one pure virtual function** | A class with **no pure virtual functions** |
| **Object Creation** | **Cannot** create objects | **Can** create objects |
| **Purpose** | Used as a **base class only** | Can be used directly or as a base class |
| **Function Implementation** | May have both implemented and unimplemented functions | All functions are implemented |
| **Example** | virtual void show() = 0; | void display() { } |

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

Answer: Abstract methods (pure virtual functions) define an interface that derived classes must implement. They ensure that all derived classes provide specific functionality, enforcing a contract for the class hierarchy.

32. Provide an example of defining and using an abstract class.

Answer:

#include <iostream>

using namespace std;

class Animal {

public:

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

};

class Cat : public Animal {

public:

void makeSound() override { cout << "Meow" << endl; }

};

int main() {

Animal\* animal = new Cat();

animal->makeSound();

delete animal;

return 0;

}

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

Answer: Benefits are:

* Enforce a common interface for derived classes.
* Promote code reusability and modularity.

Support polymorphism, allowing flexible and extensible designs.

* Prevent instantiation of incomplete base classes.

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

Answer: Exception handling is a mechanism to handle runtime errors gracefully, preventing program crashes. It uses try, catch, and throw to manage errors. Importance:

* Improves program reliability by handling errors.
* Separates error-handling code from normal code.
* Prevents abrupt termination of the program.

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

Answer: Syntax:

try {

// Code that might throw an exception

throw value; // Throw exception

}

catch (type variable) {

// Handle exception

}

Example:

try {

throw 42; // Throw integer

}

catch (int e) {

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

}

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

Ans try: Contains code that might throw an exception.

throw: Signals an exception by throwing a value (e.g., throw 42;).

catch: Catches and handles the thrown exception based on its type.

Flow: If an exception is thrown in the try block, control transfers to the matching catch block.

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

Answer: The catch block handles exceptions thrown in the try block. It specifies the type of exception it can handle and contains code to process the error (e.g., logging, recovery).

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

Answer:

#include <iostream>

using namespace std;

int main() {

try {

int choice = 2;

if (choice == 1) throw 42; // Integer exception

if (choice == 2) throw string("Error"); // String exception

}

catch (int e) {

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

}

catch (string e) {

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

}

catch (...) { // Catch-all for any other type

cout << "Unknown exception" << endl;

}

return 0;

}

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

Answer: The throw keyword is used to signal an exception by throwing a value (e.g., an integer, string, or object). It transfers control to the nearest catch block that matches the thrown value’s type.

Example: throw 42; throws an integer exception.

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

Answer: C++ does not have a finally block like some other languages (e.g., Java). Instead, cleanup tasks (e.g., releasing resources) are typically handled using RAII (Resource Acquisition Is Initialization) with objects like smart pointers or destructors.

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

Answer: Custom exception classes are created by defining a class, often inheriting from std::exception.

Example:

#include <iostream>

#include <exception>

using namespace std;

class MyException : public exception {

public:

const char\* what() const noexcept override {

return "Custom Exception";

}

};

int main() {

try {

throw MyException();

}

catch (const MyException& e) {

cout << "Caught: " << e.what() << endl;

}

return 0;

}

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

Answer: Templates allow writing generic code that works with any data type. They are used to define functions or classes that are type-independent. It is useful because:

* Promote code reusability.
* Reduce code duplication.
* Enable type-safe generic programming.

43. Describe the syntax for defining a function template.

Ans Syntax:

template <typename T>

returnType functionName(parameters) {

// Code using T

}

Example:

template <typename T>

T max(T a, T b) {

return (a > b) ? a : b;

}

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

Answer:

#include <iostream>

using namespace std;

template <typename T>

T add(T a, T b) {

return a + b;

}

int main() {

cout << add(5, 10) << endl; // Works with int

cout << add(5.5, 10.5) << endl; // Works with double

return 0;

}

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

Answer: A **class template** allows you to create a class that works with any data type, while a **function template** allows the same for functions.

| **Feature** | **Class Template** | **Function Template** |
| --- | --- | --- |
| **Definition** | Template used to create generic classes | Template used to create generic functions |
| **Syntax** | template <typename T> class ClassName { }; | template <typename T> T func(T a) { } |
| **Use** | For generic data structures (e.g., Stack, List) | For generic operations (e.g., swap, add) |
| **Example** | template<class T> class Box { T data; }; | template<class T> T max(T a, T b) { return a>b?a:b; } |

46. Explain the syntax for defining a class template.

Ans A class template is defined using the template keyword followed by a template parameter, usually typename or class.

Syntax:

template <class T>

class ClassName {

public:

T data;

void display() {

cout << data;

}

};

* template <class T>: Declares a template with type T.
* T can be used as a data type inside the class.

template <typename T>

class ClassName {

public:

// Members using T

};

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

Answer:

#include <iostream>

using namespace std;

template <typename T>

class Stack {

T arr[100];

int top;

public:

Stack() : top(-1) {}

void push(T value) { arr[++top] = value; }

T pop() { return arr[top--]; }

};

int main() {

Stack<int> s;

s.push(10);

s.push(20);

cout << s.pop() << endl; // Prints 20

return 0;

}

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

Ans A template class is instantiated by specifying the data type in angle brackets.

Example:

Stack<int> intStack; // Instantiates Stack for int

Stack<double> doubleStack; // Instantiates Stack for double

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

Answer:

Type Safety: Templates generate type-specific code at compile time, avoiding runtime type errors.

Code Reusability: Templates work with any data type without requiring a class hierarchy.

Performance: Templates avoid the overhead of virtual functions used in inheritance-based polymorphism.

Flexibility: Templates can be used for unrelated types, unlike inheritance, which requires a base-derived relationship.

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

Answer: Templates allow writing generic functions and classes that work with any data type, eliminating the need to duplicate code for different types. For example, a single template function like max<T> can handle int, double, or custom types, reducing code redundancy and maintenance effort.