# LAB QUESTIONS

Q1. Create a base class Shape with a virtual function draw() that prints "Drawing Shape". Derive two classes, Circle and Rectangle, each overriding draw() to print "Drawing Circle" and "Drawing Rectangle", respectively. In the main function: Create Circle and Rectangle objects. Use a Shape\* pointer to call draw() on both objects to show polymorphic behavior. Create a version of Shape without the virtual keyword for draw() and repeat the experiment. Compare outputs to explain why virtual functions are needed. Use a Circle\* pointer to call draw() on a Circle object and compare with the base class pointer’s behavior.

#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 Rectangle : public Shape {

public:

void draw() override {

cout << "Drawing Rectangle" << endl;

}};

int main() {

Shape\* s[2];

Circle c;

Rectangle r;

s[0] = &c;

s[1] = &r;

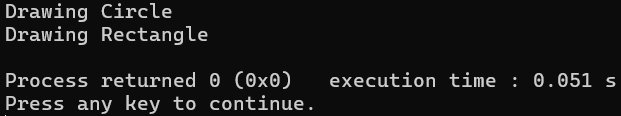
for(int i = 0; i < 2; i++) {

s[i]->draw(); // Runtime dispatch (dynamic binding)

}

return 0;

}



Now when virtual is not used,

#include<iostream>

using namespace std;

class Shape {

public:

void draw() {

cout << "Drawing Shape" << endl;

}};

class Circle : public Shape {

public:

void draw() {

cout << "Drawing Circle" << endl;

}};

class Rectangle : public Shape {

public:

void draw() {

cout << "Drawing Rectangle" << endl;

}};

int main() {

Shape\* s[2];

Circle c;

Rectangle r;

s[0] = &c;

s[1] = &r;

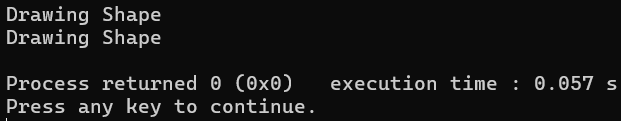
for(int i = 0; i < 2; i++) {

s[i]->draw(); // Compile-time dispatch (static binding)

}

return 0;

}



QN2. Create an abstract base class Animal with a pure virtual function speak() and a virtual destructor. Derive two classes, Dog and Cat, each implementing speak() to print "Dog barks" and "Cat meows", respectively. Include destructors in both derived classes that print "Dog destroyed" and "Cat destroyed". In the main function: Attempt to instantiate an Animal object (this should fail). Create Dog and Cat objects using Animal\* pointers and call speak(). Delete the objects through the Animal\* pointers and verify that derived class destructors are called. Modify the Animal destructor to be non-virtual, repeat the deletion, and observe the difference.

#include <iostream>

using namespace std;

class Animal {

public:

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

virtual ~Animal() { // Virtual destructor

cout << "Animal destroyed" << endl;

}};

class Dog : public Animal {

public:

void speak() override {

cout << "Dog barks" << endl;

}

~Dog() {

cout << "Dog destroyed" << endl;

}};

class Cat : public Animal {

public:

void speak() override {

cout << "Cat meows" << endl;

}

~Cat() {

cout << "Cat destroyed" << endl;

}};

int main() {

Animal\* a1 = new Dog();

Animal\* a2 = new Cat();

a1->speak(); // Output: Dog barks

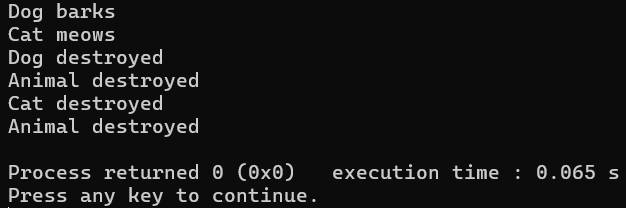
a2->speak(); // Output: Cat meows

delete a1;

delete a2;

return 0;

}



Using non virtual destructor

#include <iostream>

using namespace std;

class Animal {

public:

virtual void speak() = 0;

~Animal() {

cout << "Animal destroyed" << endl;

}};

class Dog : public Animal {

public:

void speak() override {

cout << "Dog barks" << endl;

}

~Dog() {

cout << "Dog destroyed" << endl;

}};

class Cat : public Animal {

public:

void speak() override {

cout << "Cat meows" << endl;

}

~Cat() {

cout << "Cat destroyed" << endl;

}};

int main() {

Animal\* a1 = new Dog();

Animal\* a2 = new Cat();

a1->speak();

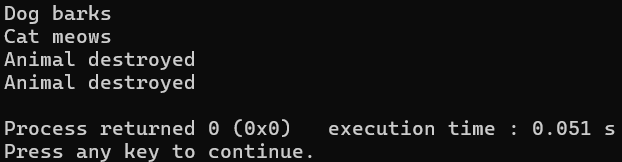
a2->speak();

delete a1;

delete a2;

return 0;

}



Q3. Create a base class Employee with a virtual function getRole() that returns a string "Employee". Derive two classes, Manager and Engineer, overriding getRole() to return "Manager" and "Engineer", respectively. In the main function: Create an array of Employee\* pointers to store Manager and Engineer objects. Iterate through the array to call getRole() for each object. Use dynamic\_cast to check if each pointer points to a Manager, and if so, print a bonus message (e.g., "Manager gets bonus"). Use typeid to print the actual type of each object.

#include <iostream>

#include <string>

#include <typeinfo>

using namespace std;

class Employee {

public:

virtual string getRole() {

return "Employee";

}

virtual ~Employee() {}

};

class Manager : public Employee {

public:

string getRole() override {

return "Manager";

}};

class Engineer : public Employee {

public:

string getRole() override {

return "Engineer";

}};

int main() {

const int size = 4;

Employee\* employees[size];

employees[0] = new Manager();

employees[1] = new Engineer();

employees[2] = new Manager();

employees[3] = new Engineer();

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

cout << "Role: " << employees[i]->getRole() << endl;

Manager\* m = dynamic\_cast<Manager\*>(employees[i]);

if (m != nullptr) {

cout << "Manager gets bonus!" << endl;

}

cout << "Actual type: " << typeid(\*employees[i]).name() << endl << endl;

}

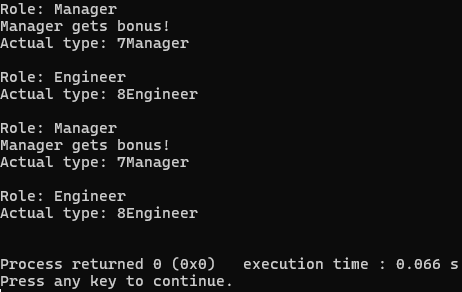
for (int i = 0; i < size; ++i) {

delete employees[i];

}

return 0;

}



Q4. Create a class Student with an integer id and a string name. In the main function: Create a Student object. Use reinterpret\_cast to treat the Student object as a char\* and print its memory address. Use reinterpret\_cast to convert an integer (e.g., 100) to a pointer type and print it.

#include <iostream>

#include <string>

using namespace std;

class Student {

public:

int id;

string name;

};

int main() {

Student s;

s.id = 1;

s.name = "Jun";

char\* ptr = reinterpret\_cast<char\*>(&s);

cout << "Memory address of Student object as char\*: " << static\_cast<void\*>(ptr) << endl;

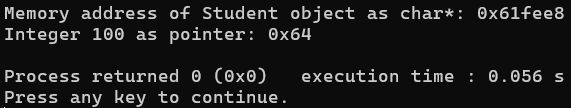
int number = 100;

void\* intAsPtr = reinterpret\_cast<void\*>(number);

cout << "Integer 100 as pointer: " << intAsPtr << endl;

return 0;

}



Q5. Create an abstract base class Vehicle with a pure virtual function operate() and a virtual destructor that prints "Vehicle destroyed". Derive two classes, Car and Truck, each implementing operate() to print distinct messages (e.g., "Car accelerates" and "Truck transports"). Include destructors in Car and Truck that print "Car destroyed" and "Truck destroyed", respectively. In the main function: Create Car and Truck objects. Use Vehicle\* pointers to call operate() on both objects. Use a Car\* pointer to call operate() on a Car object and compare with the base class pointer’s behavior. Modify a copy of the Vehicle class to make operate() non-virtual, repeat the calls using base class pointers, and observe the output differences. Create an array of Vehicle\* pointers to store Car and Truck objects, then iterate to call operate() for each. Attempt to instantiate a Vehicle object to confirm it cannot be created. Delete the objects via Vehicle\* pointers to verify derived class destructor calls. Test again with a non- virtual destructor in a separate version and note the difference. Use reinterpret\_cast to treat a Car object as a char\* and print its memory address, then cast an integer (e.g., 1000) to a pointer type and print it. Apply dynamic\_cast to check if each pointer in the array points to a Car, printing "Car identified" if successful. Use typeid to display the actual type of each object.

#include <iostream>

#include <string>

#include <typeinfo>

using namespace std;

class Vehicle {

public:

virtual void operate() = 0;

virtual ~Vehicle() {

cout << "Vehicle destroyed" << endl;

}};

class Car : public Vehicle {

public:

void operate() override {

cout << "Car accelerates" << endl;

}

~Car() {

cout << "Car destroyed" << endl;

}};

class Truck : public Vehicle {

public:

void operate() override {

cout << "Truck transports" << endl;

}

~Truck() {

cout << "Truck destroyed" << endl;

}};

int main() {

Car c;

Truck t;

Vehicle\* v1 = &c;

Vehicle\* v2 = &t;

cout << "Calling operate() via Vehicle\* pointers:" << endl;

v1->operate();

v2->operate();

cout << "\nCalling operate() via Car\* pointer:" << endl;

Car\* cptr = &c;

cptr->operate();

Vehicle\* vehicles[2] = { &c, &t };

cout << "\nIterating through Vehicle\* array:" << endl;

for (int i = 0; i < 2; i++) {

vehicles[i]->operate();

}

cout << "\nDeleting objects via Vehicle\* pointers:" << endl;

Vehicle\* vd1 = new Car();

Vehicle\* vd2 = new Truck();

delete vd1;

delete vd2;

cout << "\nUsing reinterpret\_cast:" << endl;

char\* carAsChar = reinterpret\_cast<char\*>(&c);

cout << "Car object as char\*: " << static\_cast<void\*>(carAsChar) << endl;

void\* intAsPtr = reinterpret\_cast<void\*>(1000);

cout << "Integer 1000 as pointer: " << intAsPtr << endl;

cout << "\nChecking dynamic\_cast for Car identification:" << endl;

for (int i = 0; i < 2; i++) {

Car\* carCheck = dynamic\_cast<Car\*>(vehicles[i]);

if (carCheck) {

cout << "Car identified" << endl;

}

else {

cout << "Not a Car" << endl;

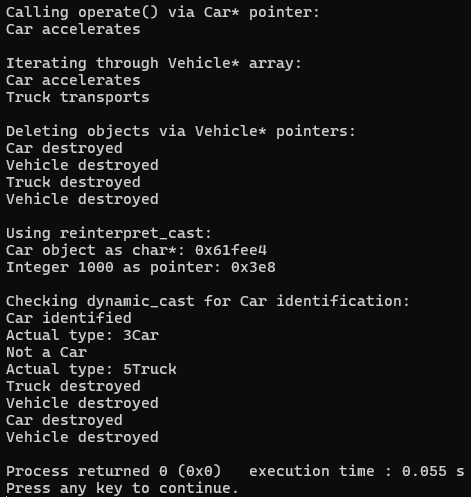
}

cout << "Actual type: " << typeid(\*vehicles[i]).name() << endl;

}

return 0;

}



# DISCUSSIONS

We discussed about the concepts of virtual function. We learnt the significance of the virtual function and also discussed the consequences for not using virtual function. Also the concept of virtual destructor, dynamic cast, reinterpret cast, typeid, etc were discussed in the session which helped to get the knowledge of these in developing a great program. We also got to see the errors and unsystematic program while using non-virtual function.

# CONCLUSIONS

Hence from the lab session, it demonstrated the necessity of virtual function in polymorphism which is the principle of OOP. Conclusively, we knew the importance of virtual function, virtual destructors and tools like dynamic\_cast, typeid, etc in developing the program. Overall, virtual functions are essential for designing robust and maintainable object-oriented programs.