Lab Questions:

1. 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’sbehavior.

```
#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;
   Circle C;
   Rectangle R;
```

```
Drawing Circle:
Drawing Rectangle:
```

```
S= &C;

S->draw();

S= &R;

S->draw();

return 0;

}
```

2. 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.

```
Dog Woofs
Cat Meows
Cat Destroyed
Animal Destroyed
Dog Destroyed
Animal Destroyed
```

```
#include<iostream>

using namespace std;

class Animal{

public:

    virtual void speak() const = 0;

     ~Animal(){

    cout<<"Animal Destroyed"<<endl;

    }

};

class Dog:public Animal{

public:

    void speak()const override{

    cout<<"Dog Woofs"<<endl;

    }

     ~Dog(){

     cout<<"Dog Destroyed"<<endl;

     }
```

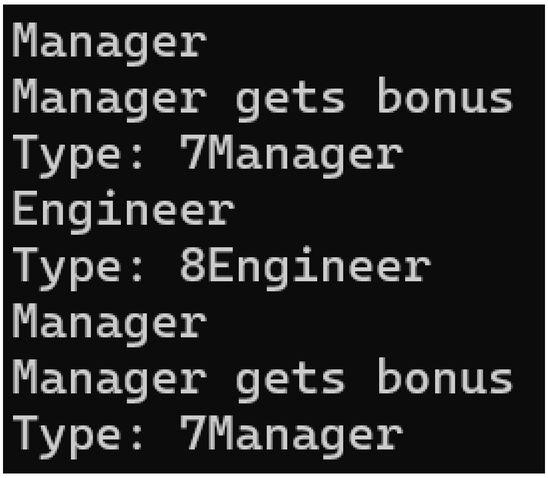
```
};
class Cat:public Animal{
public:
    void speak()const override{
    cout<<"Cat Meows"<<endl;
    }
     ~Cat(){
     cout<<"Cat Destroyed"<<endl;
     }
};
int main(){
 Animal* A;
 Dog D;
 Cat C;
 A=&D;
 A->speak();
 A=&C;
 A->speak();
 //delete A;
 return 0;
}
```

```
};
```

3. 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() {
    vector<Employee*> employees;
```



```
    employees.push_back(new Manager());
    employees.push_back(new Engineer());
    employees.push_back(new Manager());
    for (Employee* e : employees) {
        cout << e->getRole() << endl;
        if (dynamic_cast<Manager*>(e)) {
            cout << "Manager gets bonus" << endl;
        }
        cout << "Type: " << typeid(*e).name() << endl;
    }
    for (Employee* e : employees) {
        delete e;
    }
    return 0;
}
```

4. 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 = "Alice";
    char* ptr = reinterpret_cast<char*>(&s);
    cout << "Student object as char*: " << static_cast<void*>(ptr) << endl;
    uintptr_t value = 100;
    int* intPtr = reinterpret_cast<int*>(value);
    cout << "Integer value 100 as pointer (not dereferenced): " << intPtr << endl;
    return 0;
}
```

```
Student object as char*: 0x61fe00
Integer value 100 as pointer (not dereferenced): 0x64
```

5. 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 <typeinfo>
using namespace std;
class Vehicle {
public:
```

```cpp
    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 carObj;
    Truck truckObj;

    Vehicle* v1 = &carObj;
    Vehicle* v2 = &truckObj;
    v1->operate();
    v2->operate();
    Car* cPtr = &carObj;
```

```
Car accelerates
Truck transports
Car accelerates
Car accelerates
Truck transports
Car identified
Type: 3Car
Type: 5Truck
Car destroyed
Vehicle destroyed
Truck destroyed
Vehicle destroyed
Car object memory address as char*: 0x61fdd8
Integer 1000 as pointer: 0x3e8
Car destroyed
Vehicle destroyed
Truck destroyed
Vehicle destroyed
Car destroyed
Vehicle destroyed
```

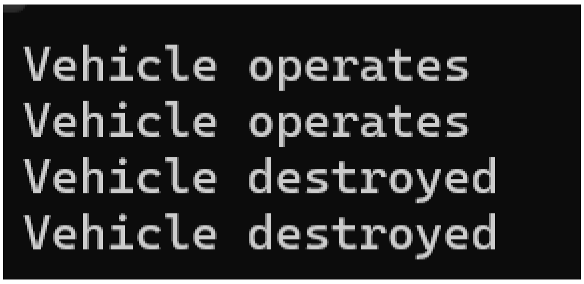
```
    cPtr->operate();
    Vehicle* vehicles[2];
    vehicles[0] = new Car();
    vehicles[1] = new Truck();
    for (int i = 0; i < 2; ++i) {
        vehicles[i]->operate();
    }
    for (int i = 0; i < 2; ++i) {
        if (dynamic_cast<Car*>(vehicles[i])) {
            cout << "Car identified" << endl;
        }
        cout << "Type: " << typeid(*vehicles[i]).name() << endl;
    }
    for (int i = 0; i < 2; ++i) {
        delete vehicles[i];
    }
    Car tempCar;
    char* rawPtr = reinterpret_cast<char*>(&tempCar);
    cout << "Car object memory address as char*: " << static_cast<void*>(rawPtr) << endl;

    int val = 1000;
    void* voidPtr = reinterpret_cast<void*>(val);
    cout << "Integer 1000 as pointer: " << voidPtr << endl;
    // Vehicle v; // Uncommenting this line will cause a compilation error
    return 0;
}
```

Now, version with non-virtual operate() and destructor, to observe the difference:

```
#include <iostream>
#include <typeinfo>
using namespace std;
```

```
class Vehicle {
public:
    void operate() {
        cout << "Vehicle operates" << endl;
    }
    ~Vehicle() {
        cout << "Vehicle destroyed" << endl;
    }
};
class Car : public Vehicle {
public:
    void operate() {
        cout << "Car accelerates" << endl;
    }
    ~Car() {
        cout << "Car destroyed" << endl;
    }
};
class Truck : public Vehicle {
public:
    void operate() {
        cout << "Truck transports" << endl;
    }
    ~Truck() {
        cout << "Truck destroyed" << endl;
    }
};
int main() {
    Vehicle* vehicles[2];
    vehicles[0] = new Car();
    vehicles[1] = new Truck();
    for (int i = 0; i < 2; ++i) {
```

```
        vehicles[i]->operate();
    }
    for (int i = 0; i < 2; ++i) {
        delete vehicles[i];
    }
    return 0;
}
```



Key Observations:

- With virtual operate() and destructor, correct function and destructor from the derived class are called.
- With non-virtual operate(), base class function is called regardless of actual object.
- With non-virtual destructor, only the base class destructor runs — leading to potential memory/resource leaks.

Discussions:

In this lab session, we have to be careful about the use of pointers for implementing virtual function, pure virtual function and abstract class. The use of syntax should be clear and clean so that no error occurs during the implementation of the program.

Conclusions:

And hence, we successfully implemented the concept of virtual function, pure virtual function and abstract class in C++.