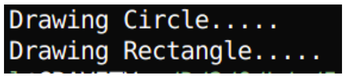
1. Create a base class Shape with a virtual function draw() that prints "Drawing Shape". Derive two classes, Circle and Rectangle, each overriding draw() to print "Drawing Circle" and "Drawing Rectangle", respectively, in he main function.

```
#include <iostream>
#include <string>
using namespace std;
class Shape {
public:
    virtual void draw() {
        cout << "Drawing Shape....." << endl;
    }
    virtual ~Shape() {} // always good to have virtual destructor in base
};
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* shapePtr;
    shapePtr = new Circle();
    shapePtr->draw();
    delete shapePtr;
    shapePtr = new Rectangle();
    shapePtr->draw();
    delete shapePtr;
    return 0;
}
```

OUTPUT



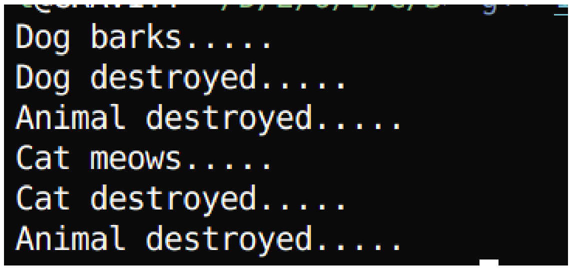
```
Drawing Circle.....
Drawing Rectangle.....
```

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 difference**

```
#include <iostream>
#include <string>
using namespace std;
class Animal {
public:
    virtual void Speak() {
        cout << "Animal is speaking....." << endl;
    }
    virtual ~Animal() {
        cout << "Animal destroyed....." << endl;
    } // always good to have virtual destructor in base
};
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* animalPtr;
    animalPtr = new Dog();
    animalPtr->Speak();
    delete animalPtr;
    animalPtr = new Cat();
    animalPtr->Speak();
    delete animalPtr;
    return 0;
}
```

OUTPUT



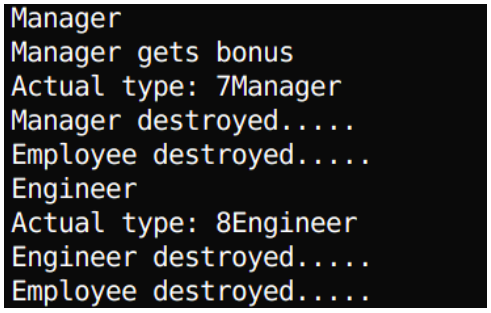
```
Dog barks.....
Dog destroyed.....
Animal destroyed.....
Cat meows.....
Cat destroyed.....
Animal destroyed.....
```

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.

- **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 he actual type of object**

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

```
class Employee {
public:
   virtual string getRole() {
      return "Employee";
   }

   virtual ~Employee() {
      cout << "Employee destroyed....." << endl;
   }
};
class Manager : public Employee {
public:
   string getRole() override {
      return "Manager";
   }
   ~Manager() {
      cout << "Manager destroyed....." << endl;
   }
};
class Engineer : public Employee {
public:
   string getRole() override {
      return "Engineer";
   }

   ~Engineer() {
      cout << "Engineer destroyed....." << endl;
   }
};
int main() {
   Employee* employeePtr[2];
   employeePtr[0] = new Manager();
   employeePtr[1] = new Engineer();
   for (int i = 0; i < 2; ++i) {
      cout << employeePtr[i]->getRole() << endl;
      if (dynamic_cast<Manager*>(employeePtr[i])) {
         cout << "Manager gets bonus" << endl;
      }
      cout << "Actual type: " << typeid(*employeePtr[i]).name() << endl;
      delete employeePtr[i];
   }
   return 0;
}
```
OUTPUT



```
Manager
Manager gets bonus
Actual type: 7Manager
Manager destroyed.....
Employee destroyed.....
Engineer
Actual type: 8Engineer
Engineer destroyed.....
Employee destroyed.....
```

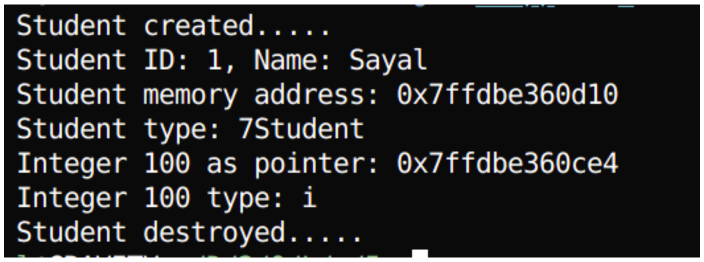
4. Create a class Student with an integer id and a string name. In the main function:

- **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>
#include <typeinfo>
using namespace std;
```

```
class Student {
public:
    int id;
    string name;
    Student(int id, string name) : id(id), name(name) {
        cout << "Student created....." << endl;
    }
    ~Student() {
        cout << "Student destroyed....." << endl;
    }
    void print() {
        cout << "Student ID: " << id << ", Name: " << name << endl;
    }
};
int main() {
    Student student(1, "Sayal");
    student.print();
    cout << "Student memory address: " << &student << endl;
    cout << "Student type: " << typeid(student).name() << endl;

    int intValue = 100;
    cout << "Integer 100 as pointer: " << &intValue << endl;
    cout << "Integer 100 type: " << typeid(intValue).name() << endl;

    return 0;
}
```

OUTPUT



```
Student created.....
Student ID: 1, Name: Sayal
Student memory address: 0x7ffdbe360d10
Student type: 7Student
Integer 100 as pointer: 0x7ffdbe360ce4
Integer 100 type: i
Student destroyed.....
```

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:

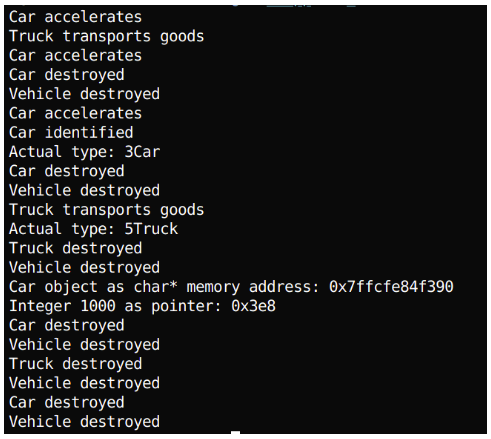
- **Create Car and Truck objects. Use Vehicle* pointers to call operate() on both objects.**
- **Use a Car* pointer to call operate() on a Car object and compare with the base class pointer's behavior.**
- **Modify a copy of the Vehicle class to make operate() non-virtual, repeat the calls using base 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:
   virtual void operate() {
      cout << "Vehicle operates" << endl;
   }
   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() {
   Vehicle* car = new Car();
   Vehicle* truck = new Truck();
   car->operate();
   truck->operate();

   char* carMA = reinterpret_cast<char*>(car);
   cout << "Memory address of Car: " << carMA << endl;
   int* intP = reinterpret_cast<int*>(1000);
   cout << "Casted integer to pointer: " << intP << endl;
   if (dynamic_cast<Car*>(car)) {
      cout << "Car identified" << endl;
   }
   if (dynamic_cast<Car*>(truck)) {
      cout << "Car identified" << endl;
   }
   cout << "Type of car: " << typeid(car).name() << endl;
```

```
    cout << "Type of truck: " << typeid(truck).name() << endl;

    delete car;
    delete truck;
    return 0;
}
```

OUTPUT



```
Car accelerates
Truck transports goods
Car accelerates
Car destroyed
Vehicle destroyed
Car accelerates
Car identified
Actual type: 3Car
Car destroyed
Vehicle destroyed
Truck transports goods
Actual type: 5Truck
Truck destroyed
Vehicle destroyed
Car object as char* memory address: 0x7ffcfe84f390
Integer 1000 as pointer: 0x3e8
Car destroyed
Vehicle destroyed
Truck destroyed
Vehicle destroyed
Car destroyed
Vehicle destroyed
```

DISCUSSION

This lab demonstrates the concept of virtual functions and runtime polymorphism through five C++ programs. The first program shows how virtual functions enable dynamic method resolution using base class pointers. The second introduces pure virtual functions and highlights the importance of virtual destructors to ensure proper cleanup of derived class objects. The third program uses dynamic_cast and typeid to safely identify and handle derived types at runtime. The fourth explores low-level casting with reinterpret_cast, displaying object memory addresses and type conversions. Finally, the fifth program combines all these concepts—virtual behavior, type checking, and casting—reinforcing the role of virtual functions in flexible and maintainable object-oriented designs.

CONCLUSION

This lab effectively demonstrates the concept and application of virtual functions in C++. Through practical examples, we observed how polymorphism enables runtime method resolution, and why virtual destructors are critical in polymorphic base classes. Additionally, the use of dynamic_cast, typeid, and reinterpret_cast provided deeper insight into C++'s runtime type system and memory handling. Mastery of these tools is essential for designing robust, extensible, and maintainable object-oriented systems in C++.