LAB ASSIGNMENTS

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:

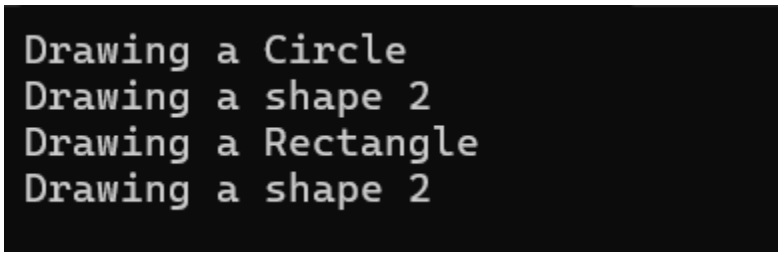
- Create Circle and Rectangle objects.
- Use a Shape* pointer to call draw() on both objects to show polymorphic behavior.
- Create a version of Shape without the virtual keyword for draw() and repeat the experiment. Compare outputs to explain why virtual functions are needed.
- Use a Circle* pointer to call draw() on a Circle object and compare with the base class pointer's behavior.

Source Code:

```
#include <iostream>

using namespace std;

class Shape{

   public:

  virtual void Draw(){

      cout << "Drawing a shape" << endl;

   }

   void Draw2(){

      cout << "Drawing a shape 2" << endl;

   }

};

class Circle : public Shape{

   public:

   void Draw() override{

      cout << "Drawing a Circle" << endl;

   }

   void Draw2() {

      cout << "Drawing a Circle 2" << endl;
```

```
    }
};
class Rectangle : public Shape{
    public:
    void Draw() override{
        cout << "Drawing a Rectangle" << endl;
    }
    void Draw2() {
        cout << "Drawing a Rectangle 2" << endl;
    }
};
int main() {
    Shape *shape;
    shape = new Circle();
    shape->Draw();
    shape->Draw2();
    shape = new Rectangle();
    shape->Draw();
    shape->Draw2();
    return 0;
}
```

Output:



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:

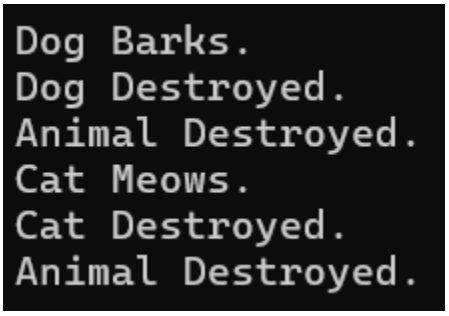
- Attempt to instantiate an Animal object (this should fail).
- Create Dog and Cat objects using Animal* pointers and call speak().
- Delete the objects through the Animal* pointers and verify that derived class destructors are called.
- Modify the Animal destructor to be non-virtual, repeat the deletion, and observe the difference.

Source Code:

```
#include <iostream>

using namespace std;

class Animal{

   public:

   virtual void speak() = 0;

   virtual~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 *animal;
    animal = new Dog();
    animal->speak();
    delete animal;
    animal = new Cat();
    animal->speak();
    delete animal;
    return 0;
}
```

Output:



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.

Source Code:

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Employee{
   public:.
   virtual void getRole(){
      cout<<"Employee"<<endl;
   }
};
class Manager : public Employee{
   public:
   void getRole() override{
      cout<<"Manager"<<endl;
   }
};
class Engineer : public Employee{
   public:
   void getRole() override{
      cout<<"Engineer"<<endl;
   }
```

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

Output:

```
Manager
Manager gets bonus
Actual type: 7Manager
Engineer
Actual type: 8Engineer
```

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.

Source Code:

```
#include <iostream>
#include <string>
using namespace std;
class Student {
public:
    int id;
    string name;
};
int main() {
    Student s;
    s.id = 1;
    s.name = "Suraj";
    char* addr = reinterpret_cast<char*>(&s);
    cout << "Student object memory address: " << static_cast<void*>(addr) << endl;
    int num = 100;
    void* intPtr = reinterpret_cast<void*>(num);
    cout << "Integer 100 as pointer: " << intPtr << endl;
    return 0;
}
```

Output:

```
Student object memory address: 0x61fee8
Integer 100 as pointer: 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.

Source Code:

```
#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 << endl;
cout << "Calling operate() via Car* pointer:" << endl;
Car* cptr = &c;
cptr->operate();
Vehicle* vehicles[2] = { &c, &t };
cout << endl;
cout << "Iterating through Vehicle* array:" << endl;
for (int i = 0; i < 2; i++) {
    vehicles[i]->operate();
}
cout << "Deleting objects via Vehicle* pointers:" << endl;
Vehicle* vd1 = new Car();
Vehicle* vd2 = new Truck();
delete vd1;
delete vd2;
cout << "Using reinterpret_cast:" << endl;
char* carAsChar = reinterpret_cast<char*>(&c);
cout << "Car object as char*: " << static_cast<void*>(carAsChar) << endl;
void* intAsPtr = reinterpret_cast<void*>(reinterpret_cast<void*>(1000));
cout << "Integer 1000 as pointer: " << intAsPtr << endl;
cout << "Checking 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;
}
```

Output:

```
Calling operate() via Vehicle* pointers:
Car accelerates.
Truck transports.
Calling operate() via Car* pointer:
Car accelerates.
Iterating through Vehicle* array:
Car accelerates.
Truck transports.
Deleting objects via Vehicle* pointers:
Car destroyed.
Vehicle destroyed.
Truck destroyed.
Vehicle destroyed.
Using reinterpret_cast:
Car object as char*: 0x61fee4
Integer 1000 as pointer: 0x3e8
Checking dynamic_cast for Car identification:
Car identified.
Actual type: 3Car
Not a Car.
Actual type: 5Truck
Truck destroyed.
Vehicle destroyed.
Car destroyed.
Vehicle destroyed.
```

DISCUSSION

In this lab we were able to understand the core concept of virtual functions where we learnt about overriding functions, pure and virtual functions. We also learnt about dynamic cast, typeid and typeinfo got information on interpreting cast and many more. It was very difficult to understand these concepts but with the help of online mediums on the internet I was able to understand these concepts.

CONCLUSION

From this lab, we can conclude that the use of virtual functions enhances code maintainability, reduces the need for rewriting code, and facilitates polymorphism which is a core concept in OOP.