Lab Sheet 7
Understanding the Concept of Virtual function, Virtual base class and

RTTI

Virtual Function

The overridden function in the derived class can be invoked by means of a base class pointer if the function is declared virtual in the base class. Suppose a virtual function get( ) is defined in the base class Base and again it is defined in the derived class Derived.

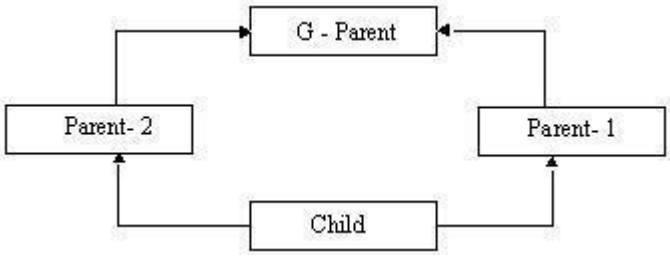
We can use the base class pointer to invoke the get( ) function of the derived class.

```
Derived d;

Base *b;

b=&d;

b-> get( ) //it calls the get ( ) function of the derived

class.
```

Virtual Destructors

When a base class pointer that is pointing to a derived class object is deleted, destructor of the derived class as well as destructors of all its base classes is invoked, if the destructor in the base class is declared as virtual.

Virtual Base Class



In this type of inheritance there may be ambiguity in the members of the derived class child because it is derived from two base classes, which are again derived from the same base class. Hence to avoid this ambiguity the class G – parent can be made virtual.

Runtime Type Information (RTTI)

The runtime type information is one of the features of C++ that exhibit runtime polymorphic behavior. In C++ we can find the type information of an object at runtime and change the type of the object at runtime. The operators dynamic_cast and typeid are used for runtime type information.

For example if Animal is a polymorphic base class and Dog and Cat are derived classes of base class Animal then

```
Animal *anmp;
Dog dg;

Cat ct;
anmp=&dg;

cout<< typeid(*anmp).name();
```

displays the information of the object pointed by anm pointer Similarly

```
Cat *cpt;

cpt=dynamic_cast<Cat*>(panm);
```

The down cast is successful if panm is holding the address of objects of class Cat.

Exercises

1. Write a program to create a class shape with functions to find area of the shapes and display the name of the shape and other essential component of the class. Create derived classes circle, rectangle and trapezoid each having overridden functions area and display. Write a suitable program to illustrate virtual functions and virtual destructor.

```
#include <iostream>
#include <cstring>
#define pi 3.1415
using namespace std;
class shape
{
 protected:
  string sname;
  float sarea;
 public:
  shape()
  {
    sname = "shape";
    sarea = 0;
  }
  shape(float a, string n="shape")
  {
    sname = n;
    sarea = a;
  }
  virtual float area()
  {
    return sarea;
  }
  string name()
  {
    cout << "Shape " << sname << endl;
    return sname;
  }
  virtual ~shape()
  {
    cout << "Destructor of Shape " << endl;
  }
};
class circle:public shape
{
 protected:
  float radius;
 public:
  circle(int r, string n = "circle")
  {
```

```
    radius = r;
    sname = n;
  }
  float area()
  {
    sarea = pi * radius * radius;
    return shape::area();
  }
  string name()
  {
    cout << "Circle " << sname << endl;
    return sname;
  }
   ~circle()
  {
    cout << "Circle destructor" << endl;
  }

};
class rectangle:public shape
{
 protected:
  float length, breadth;
 public:
  rectangle(float l, float b, string
n="rectangle"):length(l),breadth(b)
  {
    sname = n;
  }
  float area()
  {
    sarea = length*breadth;
    return shape::area();
  }
  string name()
  {
    cout << "Rectangle " << sname << endl;
    return sname;
  }
   ~rectangle()
  {
```

```
    cout << "Rectangle destructor" << endl;
  }
};
class trapezoid:public shape
{
 protected:
  float paralleside[2];
  float nonparallelside[2];
 public:
  trapezoid(float a1, float a2, float b1, float b2,
string n= "Trapezoid")
  {
    paralleside[0] = a1;
    paralleside[1] = a2;
    nonparallelside[0] = b1;
    nonparallelside[1] = b2;
    sname = n;
  }
  float area()
  {
    sarea =
(paralleside[0]+paralleside[1])/2.0*(nonparallelside[0
]+nonparallelside[1])/2.0;
    return shape::area();
  }
  string name()
  {
    cout << "Trapezoid " << sname << endl;
    return sname;
  }
   ~trapezoid()
  {
    cout << "Trapezoid destructor" << endl;
  }
};
int main()
{
  shape *sh;
  sh = new circle(4,"ball");
  sh->name();
  cout << sh->area() << endl;
```

```
  delete(sh);
  sh = new trapezoid(200,400, 100 , 100,"fancy
stadium");;
  sh->name();
  cout << sh->area() << endl;
  delete(sh);
  sh = new rectangle(240,240,"ground");;
  sh->name();
  cout << sh->area() << endl;
  delete(sh);
  return 0;
}
```

2. Create a class Person and two derived classes Employee, and Student, inherited from class Person. Now create a class Manager which is derived from two base classes Employee and Student. Show the use of the virtual base class.

```
#include <iostream>
#include <cstring>
#define pi 3.1415
using namespace std;
class shape
{
 protected:
  string sname;
  float sarea;
 public:
  shape()
  {
    sname = "shape";
    sarea = 0;
  }
  shape(float a, string n="shape")
  {
    sname = n;
    sarea = a;
  }
  virtual float area()
```

```
  {
    return sarea;
  }
  string name()
  {
    cout << "Shape " << sname << endl;
    return sname;
  }
  virtual ~shape()
  {
    cout << "Destructor of Shape " << endl;
  }
};
class circle:public shape
{
 protected:
  float radius;
 public:
  circle(int r, string n = "circle")
  {
    radius = r;
    sname = n;
  }
  float area()
  {
    sarea = pi * radius * radius;
    return shape::area();
  }
  string name()
  {
    cout << "Circle " << sname << endl;
    return sname;
  }
   ~circle()
  {
    cout << "Circle destructor" << endl;
  }

};
class rectangle:public shape
```

```
{
 protected:
  float length, breadth;
 public:
  rectangle(float      l,      float      b,      string
n="rectangle"):length(l),breadth(b)
  {
    sname = n;
  }
  float area()
  {
    sarea = length*breadth;
    return shape::area();
  }
  string name()
  {
    cout << "Rectangle " << sname << endl;
    return sname;
  }
   ~rectangle()
  {
    cout << "Rectangle destructor" << endl;
  }
};
class trapezoid:public shape
{
 protected:
  float paralleside[2];
  float nonparallelside[2];
 public:
  trapezoid(float  a1,  float  a2,  float  b1,  float  b2,
string n= "Trapezoid")
  {
    paralleside[0] = a1;
    paralleside[1] = a2;
    nonparallelside[0] = b1;
    nonparallelside[1] = b2;
    sname = n;
  }
  float area()
```

```
  {
    sarea                                                    =
(paralleside[0]+paralleside[1])/2.0*(nonparallelside[0]
+nonparallelside[1])/2.0;
    return shape::area();
  }
  string name()
  {
    cout << "Trapezoid " << sname << endl;
    return sname;
  }
   ~trapezoid()
  {
    cout << "Trapezoid destructor" << endl;
  }
};
int main()
{
  shape *sh;
  sh = new circle(4,"ball");
  sh->name();
  cout << sh->area() << endl;
  delete(sh);
  sh   =   new   trapezoid(200,400,   100   ,   100,"fancy
stadium");;
  sh->name();
  cout << sh->area() << endl;
  delete(sh);
  sh = new rectangle(240,240,"ground");;
  sh->name();
  cout << sh->area() << endl;
  delete(sh);
  return 0;
}
```

3. Write a program with Student as abstract class and create derive classes Engineering, Medicine and Science from base class Student. Create the objects of the derived classes and process them and access them using array of pointer of type base

```
#include <iostream>
#include <cstring>
using namespace std;
class student
{
 private:
 protected:
  string name;
  int rank;
 public:
  virtual string getname() = 0;
  virtual int getrank() = 0;
};
class engineering : public student
{
 private:
 public:
  engineering(string n,int r){
    name= n;
    rank=r;
  }
  string getname()
  {
    return name;
  }
  int getrank()
  {
    return rank;
  }
};
class medicine: public student
{
 private:
 public:
  medicine(string n,int r){
    name=n;
    rank=r;
  }
  string getname()
```

```
  {
    return name;
  }
  int getrank()
  {
    return rank;
  }
};
class science : public student
{
 private:
 public:
  science(string n, int r){
    name = n;
    rank = r;
  }
  string getname()
  {
    return name;
  }
  int getrank()
  {
    return rank;
  }
};

int main()
{
  student* st[3];
  st[0] = new engineering("Abcd", 3);
  st[1] = new medicine("Efgh",1);
  st[2] = new science("Ijkl",2);
  cout << "Student of various field" << endl;
  for (int i = 0; i < 3; ++i)
  {
    cout << "Name " << st[i]->getname() << endl;
    cout << "Rank " << st[i]->getrank() << endl;
  }
  return 0;
}
```

4. Create a polymorphic class Vehicle and create other derived classes Bus, Car and Bike from Vehicle. With this program illustrate RTTI by the use of dynamic_cast and typeid operators.

```
#include <iostream>
#include <cstring>
#include <typeinfo>
using namespace std;
class vehicle
{
 private:
 protected:
  string registration;
  int noofwheels;
 public:
  vehicle(string r, int n)
  {
    registration = r;
    noofwheels = n;
  }
  string getregistration()
  {
    cout << "Vehicle getRegistratin called" << endl;
    return registration;
  }
};
class bus : public vehicle
{
 private:
 public:
  bus(string r):vehicle(r,4){};
  string getregistration()
  {
    cout << "Bus getRegistratin called" << endl;
    return registration;
  }
};
```

```
class car : public vehicle
{
 private:
 public:
  car(string r):vehicle(r,4){};
  string getregistration()
  {
    cout << "Car getRegistratin called" << endl;
    return registration;
  }
};
class bike : public vehicle
{
 private:
 public:
  bike(string r):vehicle(r,2){};
  string getregistration()
  {
    cout << "Bike getRegistratin called" << endl;
    return registration;
  }
};

int main()
{
  vehicle *vlist[3];
  bus *bs = new bus("1");
  car *c = new car("1");
  bike *b = new bike("1");
  vlist[0] = dynamic_cast<vehicle *>(bs);
  vlist[1] = dynamic_cast<vehicle *>(c);
  vlist[2] = dynamic_cast<vehicle *>(b);
  for(int i = 0; i < 3 ; i++)
  {
    cout << typeid(*vlist[i]).name() << endl;
    cout << vlist[i]->getregistration() << endl;
  }
  cout << typeid(*bs).name() << endl;
  cout << typeid(*c).name() << endl;
  cout << typeid(*b).name() << endl;
```

```
  return 0;
}
```