DYNAMIC POLYMORPHISM

6

Chapter Outline

6.1 Introduction

My friend Narayana from Andhra Pradesh and I went for lunch. We took coupons for an Andhra and a Maharashtrian meal each for Narayana and me. We handed over the coupons to the waiter at the counter and got Andhra and Maharashtrian meals for ourselves. Based on the type of the coupon, the waiter served either the Andhra or the Maharashtrian meal.

```
Meal ServeMeal();        // Function
coupon1. ServeMeal();  // Returns Andhra meal as the coupon is for Andhra  meal.
coupon2. ServeMeal();  // Returns Maharashtrian meal as the coupon is for Maharashtrian meal.
```

It means though the same function is called by different entities (coupons here), functions associated with that coupon is called. It may seem mind-boggling. Have some patience and you will understand the magic of dynamic polymorphism in programming.

6.2 Dynamic Polymorphism: What, When and Why?

Let us write a program (Program 6.1) that displays information of cuboid/spherical metals as shown in Table 6.1.

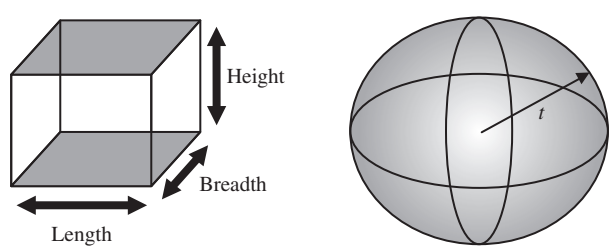

Table 6.1

Metal Name	Shape	Length (cm)	Breadth (cm)	Height (cm)	Mass (gm)	Accurate Density gm/cm^3
Silver	Cuboid	10	9.5	2.8	2793	10.5
Gold	Cuboid	7.1	5.8	9.1	89	19.3
Platinum	Cuboid	3.5	2.7	1.8	189	21.4
Silver	Spherical	21 (Radius)	Not Applicable	Not Applicable	38808	10.5
Gold	Spherical	42 (Radius)	Not Applicable	Not Applicable	5991955.2	19.3
Platinum	Spherical	63 (Radius)	Not Applicable	Not Applicable	189	21.4

Program 6.1

```
#include<iostream>
#include<string>
using namespace std;
class MetalShape
{
    protected:
      char  metalType[10];
      float mass;
      float accurate_density;
    public:
      MetalShape(char*n, float m,float a)
      {
          strcpy(metalType,n);
          mass=m;
          accurate_density=a;
      }
      void DisplayData()
      {
          cout<<"Metaltype:"<<metalType<<endl;
          cout<<"Mass:"<<mass<<endl;
          cout<<"Accurate_Density:"<<accurate_density<<endl<<endl;
      }
      ~MetalShape ()
      {
          cout<<"MetalShape Destructor Called"<<endl;
      }
};
```

```
class CuboidMetal: public MetalShape
{
    float length;
    float breadth;
    float height;
    public:
      CuboidMetal(char*n,float l,float b,float h, float m,float a):
        MetalShape(n,m,a)
      {
          length=l;
          breadth=b;
          height=h;
      }
      void DisplayData()
      {
          cout<<"Metaltype:"<<metalType<<endl;
          cout<<"Length"<<length<<endl;
          cout<<"Breadth"<<breadth<<endl;
          cout<<"Height"<<height<<endl;
          cout<<"Mass:"<<mass<<endl;
          cout<<"Accurate_Density:"<<accurate_density<<endl<<endl;
      }
      ~ CuboidMetal ()
      {
          cout<<"CuboidMetal Destructor Called"<<endl;
      }
};
class SphericalMetal : public MetalShape
{
    private:
      float radius;
      float getVolume()
      {
          float v=4/3.0*22/7.0*radius*radius*radius;
          return v;
      }
     public:
      SphericalMetal(char*n,float r, float m,float a):MetalShape(n,m,a)
      {
          radius=r;
      }
      void DisplayData()
      {
          cout<<"Metaltype:"<<metalType<<endl;
          cout<<"Mass:"<<mass<<endl;
          cout<<"Accurate_Density:"<<accurate_density<<endl;
```

```
            cout<<"Radius"<<radius<<endl<<endl;
        }
        ~ SphericalMetal ()
        {
            cout<<"SphericalMetal Destructor Called"<<endl;
        }
};
int main()
{
    CuboidMetal c1("Silver",10,9.5,2.8,2793,10.5);
    CuboidMetal c2("Gold",7.1,5.8,9.1,7103,19.3);
    CuboidMetal c3("Platinum",10,9.5,2.8,5400,21.4);

    SphericalMetal s1("Silver",21,407484,10.5);
    SphericalMetal s2("Gold",42,5991955.2,19.3);
    SphericalMetal s3("Platinum",63,22423262.4,21.4);

    c1.DisplayData();
    c2.DisplayData();
    c3.DisplayData();
    s1.DisplayData();
    s2.DisplayData();
    s3.DisplayData();
}
```

Output

```
Metaltype:Silver
Length10
Breadth9.5
Height2.8
Mass:2793
Accurate_Density:10.5

Metaltype:Gold
Length7.1
Breadth5.8
Height9.1
Mass:7103
Accurate_Density:19.3

Metaltype:Platinum
Length10
Breadth9.5
Height2.8
Mass:5400
Accurate_Density:21.4
```

```
Metaltype:Silver
Mass:407484
Accurate_Density:10.5
Radius21

Metaltype:Gold
Mass:5.99196e+006
Accurate_Density:19.3
Radius42

Metaltype:Platinum
Mass:2.24233e+007
Accurate_Density:21.4
Radius63
SphericalMetal Destructor Called
MetalShape Destructor Called
SphericalMetal Destructor Called
MetalShape Destructor Called
SphericalMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
```

Program 6.1 displays data for three cuboid metals and three spherical metals. The number of instructions written in the main function for each cuboid metal or spherical metal are 2, i.e.

```
CuboidMetal c1("Silver",10,9.5,2.8,2793,10.5);
c1.DisplayData();
```

If we want to display data of 1000 such cuboid metals or spherical metals each, we need to write $1000 \times 2 = 2000$ instructions for cuboid metal and spherical metal, respectively. Is it practicable? No. One can use an array to store the objects of the cuboid metal and the spherical metal as discussed in Chapter 2. The modified main function is as follows:

```
int main()
{
    CuboidMetal  c[3]={
                        CuboidMetal("Silver",10,9.5,2.8,2793,10.5),
                        CuboidMetal("Gold",7.1,5.8,9.1,7103,19.3),
                        CuboidMetal("Platinum",10,9.5,2.8,5400,21.4)
                      };
```

```
    SphericalMetal s[3]={
                                SphericalMetal("Silver",21,407484,10.5),
                                SphericalMetal("Gold",42,5991955.2,19.3),
                                SphericalMetal("Platinum",63,22423262.4,21.4)
                              };
    for(int i=0;i<3;i++)
      c[i].DisplayData();
    for(int i=0;i<3;i++)
      s[i]. DisplayData();
}
```

The number of instructions in the main function are lesser than that in Program 6.1. But this is not the optimized solution because it has for loop for every class – cuboid metal and spherical metal in this scenario. If we have 100 classes then we will require 100 such loops which is not practical. Can we have a single for loop that can perform this job? Yes. A base class pointer can contain the address of a derived class object. Hence, instead of creating an array of individual objects, let us create an array of the base class pointer. The modified program can be written as in Program 6.2.

Program 6.2

```
// Classes remains same as defined in Program 6.1
int main()
{
    CuboidMetal c1("Silver",10,9.5,2.8,2793,10.5);
    CuboidMetal c2("Gold",7.1,5.8,9.1,7103,19.3);
    CuboidMetal c3("Platinum",10,9.5,2.8,5400,21.4);

    SphericalMetal s1("Silver",21,407484,10.5);
    SphericalMetal s2("Gold",42,5991955.2,19.3);
    SphericalMetal s3("Platinum",63,22423262.4,21.4);

    MetalShape* m[6]={&c1,&c2,&c3,&s1,&s2,&s3};
    for(int i=0;i<6;i++)
      m[i]->DisplayData();
}
```

Output

```
Metaltype:Silver
Mass:2793
Accurate_Density:10.5

Metaltype:Gold
Mass:7103
Accurate_Density:19.3
```

```
Metaltype:Platinum
Mass:5400
Accurate_Density:21.4

Metaltype:Silver
Mass:407484
Accurate_Density:10.5

Metaltype:Gold
Mass:5.99196e+006
Accurate_Density:19.3

Metaltype:Platinum
Mass:2.24233e+007
Accurate_Density:21.4

SphericalMetal Destructor Called
MetalShape Destructor Called
SphericalMetal Destructor Called
MetalShape Destructor Called
SphericalMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
```

```
m[0]->DisplayData(); // Calls function of MetalShape class as m[0] is type of MetalShape.
m[1]->DisplayData(); // Calls function of MetalShape class as m[1] is type of MetalShape.
m[2]->DisplayData(); // Calls function of MetalShape class as m[2] is type of MetalShape.
m[3]->DisplayData(); // Calls function of MetalShape class as m[3] is type of MetalShape.
m[4]->DisplayData(); // Calls function of MetalShape class as m[4] is type of MetalShape.
m[5]->DisplayData(); // Calls function of MetalShape class as m[5] is type of MetalShape.
```

The program compiles and successfully executes. However, the output is incorrect. **When the call is made to** m[i]->DisplayData() function, DisplayData **function defined in the base class, i.e.,** MetalShape **gets called**. It is expected that it should call the derived class DisplayData function, i.e., of CuboidMetal or SphericalMetal class. It does not happen because the function call to DisplayData is resolved by the type of pointer – base class pointer MetalShape – and not by the address of the object that the base class pointer contains.

Is there any way by which a base class pointer can invoke a derived class function (instead of a base class function) if it contains the address of a derived class object?

Yes. If the function is declared virtual, a function is called on the basis of the address that the base class pointer contains and not the type of the base class pointer. If the base class

pointer contains the address of a base class object, it calls the base class function. If the base class pointer contains the address of a derived class object, it calls a derived class function. The modified correct program is as in Program 6.3.

Program 6.3

```
#include<iostream>
using namespace std;
class MetalShape
{
    protected:
      char  metalType[10];
      float mass;
      float accurate_density;
    public:
      MetalShape(char*n, float m,float a)
      {
          strcpy(metalType,n);
          mass=m;
          accurate_density=a;
      }
      virtual void DisplayData()
      {
          cout<<"Metaltype:"<<metalType<<endl;
          cout<<"Mass:"<<mass<<endl;
          cout<<"Accurate_Density:"<<accurate_density<<endl<<endl;
      }
      ~MetalShape ()
      {
          cout<<"MetalShape Destructor Called"<<endl;
      }
};
class CuboidMetal: public  MetalShape
{
    float length;
    float breadth;
    float height;
    public:
      CuboidMetal(char*n,float l,float b,float h, float m,float a):
        MetalShape(n,m,a)
      {
          length=l;
          breadth=b;
          height=h;
      }
```

```
    void DisplayData()
    {
        cout<<"Metaltype:"<<metalType<<endl;
        cout<<"Length"<<length<<endl;
        cout<<"Breadth"<<breadth<<endl;
        cout<<"Height"<<height<<endl;
        cout<<"Mass:"<<mass<<endl;
        cout<<"Accurate_Density:"<<accurate_density<<endl<<endl;
    }
    ~ CuboidMetal ()
    {
        cout<<"CuboidMetal Destructor Called"<<endl;
    }
};
class SphericalMetal: public MetalShape
{
  private:
    float radius;
    float getVolume()
    {
        float v=4/3.0*22/7.0*radius*radius*radius;
        return v;
    }
  public:
    SphericalMetal(char*n,float r, float m,float a):MetalShape(n,m,a)
    {
        radius=r;
    }
    void DisplayData()
    {
        cout<<"Metaltype:"<<metalType<<endl;
        cout<<"Mass:"<<mass<<endl;
        cout<<"Accurate_Density:"<<accurate_density<<endl;
        cout<<"Radius"<<radius<<endl<<endl;
    }
    ~ SphericalMetal ()
    {
        cout<<"SphericalMetal Destructor Called"<<endl;
    }
};
int main()
{
  CuboidMetal c1("Silver",10,9.5,2.8,2793,10.5);
  CuboidMetal c2("Gold",7.1,5.8,9.1,7103,19.3);
  CuboidMetal c3("Platinum",10,9.5,2.8,5400,21.4);
```

```
    SphericalMetal s1("Silver",21,407484,10.5);
    SphericalMetal s2("Gold",42,5991955.2,19.3);
    SphericalMetal s3("Platinum",63,22423262.4,21.4);

    MetalShape* m[6]={&c1,&c2,&c3,&s1,&s2,&s3};
    for(int i=0;i<6;i++)
      m[i]->DisplayData();
}
```

Output

```
Metaltype:Silver
Length10
Breadth9.5
Height2.8
Mass:2793
Accurate_Density:10.5

Metaltype:Gold
Length7.1
Breadth5.8
Height9.1
Mass:7103
Accurate_Density:19.3

Metaltype:Platinum
Length10
Breadth9.5
Height2.8
Mass:5400
Accurate_Density:21.4

Metaltype:Silver
Mass:407484
Accurate_Density:10.5
Radius21

Metaltype:Gold
Mass:5.99196e+006
Accurate_Density:19.3
Radius42

Metaltype:Platinum
Mass:2.24233e+007
Accurate_Density:21.4
Radius63
```

```
SphericalMetal Destructor Called
MetalShape Destructor Called
SphericalMetal Destructor Called
MetalShape Destructor Called
SphericalMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
```

```
m[0]->DisplayData(); // Calls function of CuboidMetal class as m[0] is address of CuboidMetal object.
m[1]->DisplayData(); // Calls function of CuboidMetal class as m[1] is address of CuboidMetal object.
m[2]->DisplayData(); // Calls function of CuboidMetal class as m[2] is address of CuboidMetal object.
m[3]->DisplayData(); // Calls function of SphericalMetal class as m[3] is address of SphericalMetal object.
m[4]->DisplayData(); // Calls function of SphericalMetal class as m[4] is address of SphericalMetal object.
m[5]->DisplayData(); // Calls function of SphericalMetal class as m[5] is address of SphericalMetal object.
```

DisplayData function is known as an overridden function because it is implemented in the base class (MetalShape) as well as in the derived class (CuboidMetal). When a base class pointer calls an overridden function, the overridden function of the respective class is called on the basis of the address of the object that the base class pointer points to instead of always calling the base class function. This is achieved by declaring overridden function as a virtual function in the base class. This process is also known as late binding or runtime polymorphism.

Virtual functions works as follows. The compiler creates a v-table for each class that has at least one virtual function. In case of the aforementioned scenario, the compiler creates a separate v-table for MetalShape and CuboidMetal class, respectively. v-table contains the addresses of virtual functions in that class, i.e., v-table of MetalShape contains the physical address of the DisplayData function where the compiled implementation (binary code) of DisplayData function is stored. The v-table of CuboidMetal contains the physical address of the DisplayData function where the compiled implementation of the DisplayData function is stored. When the object of a class that has at least one virtual function is created, the compiler implicitly creates a virtual pointer or v-pointer or vptr that points to the v-table of that class. For example, during the resolution of m[3]->DisplayData(); m[3] has a virtual pointer that points to a v-table of SphericalMetal class from where the address of DisplayData function is fetched.

Program 6.3 can be compacted by calling the DisplayData function of the base class (MetalShape) from DisplayData function defined in CuboidMetal or SphericalMetal as follows:

```
void DisplayData()
{
    cout<<"Length"<<length<<endl;
    cout<<"Breadth"<<breadth<<endl;
```

```
    cout<<"Height"<<height<<endl;
    MetalShape::DisplayData();
}
```

You can modify the DisplayData function of SphericalMetal class and execute Program 6.3.

6.3 Virtual Destructor

Do you feel that there is any memory leak in Program 6.3? You may say that all objects of CuboidMetal and SphericalMetal class are created on stack; hence, there is no memory leak because when an object created on stack goes out of scope, the destructor of these objects are automatically called. When m[0],m[1],m[2],m[3],m[4],m[5] goes out of scope, the destructor of the base class is called. **According to C++ standard, deleting a derived class object through a base class pointer is an undefined behaviour.** Hence, we cannot draw any conclusion about a memory leak in Program 6.3 because this leads to undefined behaviour. This can be corrected by declaring the destructor of a base class virtual.

```
virtual ~MetalShape ()
{
   cout<< "MetalShape Destructor Called" << endl;
}
```

When a base class pointer of the class with a virtual destructor is deleted, it first calls the destructor of the derived class in a hierarchy and then the base class destructor is called at the end. Object construction happens from a base class to a derived class whereas object destruction takes place from a derived class to a base class. Is there any memory leak in the following main function? Refer to CuboidMetal class defined in Program 6.3.

```
int main()
{
    MetalShape*  c1= new CuboidMetal("Silver",10,9.5,2.8,2793,10.5);
    c1->DisplayData();
    delete c1;  // Attempt to delete derived class object through base class pointer
}
```

Considering that deletion of a derived class object through a base class pointer is undefined behaviour, the destructor in CuboidMetal class must be declared virtual. This is also applicable to SphericalMetal class. When the destructor of a CuboidMetal or SphericalMetal class is declared as virtual, the destructor of CuboidMetal or SphericalMetal class is called when an attempt to delete the object of CuboidMetal or SphericalMetal class is done through a base class pointer.

6.4 Pure Virtual Function

Let us implement the virtual function mechanism for Program 5.7. Here, the DeterminePurity function of derived classes is called. Hence, we should declare the DeterminePurity function as virtual in a base class and create an array of the base class pointer – array of MetalShape*. However,

the base class – MetalShape does not contain the DeterminePurity function. Hence, we need to define the empty DeterminePurity function as shown in Program 6.4 and declare it as virtual.

Program 6.4

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

class MetalShape
{
    protected:
      char  metalType[10];
      float mass;
      float accurate_density;
      float getDensity(float vol)
      {
          float d=mass/vol;
          return d;
      }
    public:
      MetalShape(char*n, float m,float a)
      {
          strcpy(metalType,n);
          mass=m;
          accurate_density=a;
      }
      virtual void DeterminePurity()  // Empty function in base class to use virtual function
                                      // mechanism
      {
      }
};
class CuboidMetal: public  MetalShape
{
    float length;
    float breadth;
    float height;
    float getVolume()
    {
        float v=length*breadth*height;
        return v;
    }
    public:
      CuboidMetal(char*n,float l,float b,float h, float m,float a):MetalShape(n,m,a)
      {
```

```
            length=l;
            breadth=b;
            height=h;
        }
        void DeterminePurity()
        {
            float v=getVolume();
            float d=getDensity(v);

            if(d==accurate_density)
               cout<<"Metal "<< metalType<<" is pure."<<endl;
            else
               cout<<"Metal "<< metalType<<" is not pure."<<endl;
        }
};
class SphericalMetal : public MetalShape
{
    private:
       float radius;
       float getVolume()
       {
           float v=4/3.0*22/7.0*radius*radius*radius;
           return v;
       }
    public:
       SphericalMetal(char*n,float r, float m,float a):MetalShape(n,m,a)
       {
           radius=r;
       }
       void DeterminePurity()
       {
           float v=getVolume();
           float d=getDensity(v);

           if(d== accurate_density)
              cout<<"Metal "<< metalType<<" is pure."<<endl;
           else
              cout<<"Metal "<< metalType<<" is not pure."<<endl;
       }
};
int main()
{
    CuboidMetal c1("Silver",10,9.5,2.8,2793,10.5);
    CuboidMetal c2("Gold",7.1,5.8,9.1,7103,19.3);
    CuboidMetal c3("Platinum",10,9.5,2.8,5400,21.4);
```

```
    SphericalMetal s1("Silver",21,407484,10.5);
    SphericalMetal s2("Gold",42,5991955.2,19.3);
    SphericalMetal s3("Platinum",63,22423262.4,21.4);

    MetalShape* ptr[6]={&c1,&c2,&c3,&s1,&s2,&s3};
    for(int i=0;i<6;i++)
      ptr[i]->DeterminePurity();
}
```

Output

```
Metal Silver is pure.
Metal Gold is not pure.
Metal Platinum is not pure.
Metal Silver is pure.
Metal Gold is pure.
Metal Platinum is pure.
```

Program 6.4 works successfully. However, one thought may linger in your mind: why should one provide the DeterminePurity function in base class MetalShape with empty body although the object of MetalShape class does not exist practically? Yes, you are right. It is not a good idea to create an object of MetalShape because metal shape does not exist physically. MetalShape represents an abstraction of a family of classes and common features of derived classes such as CuboidMetal and SphericalMetal. If someone asks you to show a metal shape physically, you will show a cuboid, spherical, cube metals and alike, but you will not be able to show a metal shape. It is true that every subclass of MetalShape should implement functions such as DeterminePurity().

Is there any way by which a base class can force a derived class to implement certain function/functions?

Yes. This can be achieved by declaring function as pure virtual function. The syntax of pure virtual function is as follows:

```
virtual void DeterminePurity = 0;
```

A class that contains at least one pure virtual function is known as an abstract class.

MetalShape class is an abstract class. One cannot create an object of an abstract class. Program 6.5 is the modified version of Program 6.4.

Program 6.5

```
#include<iostream>
#include<string>
using namespace std;
class MetalShape
{
   protected:
```

```
    char  metalType[10];
    float mass;
    float accurate_density;
    float getDensity(float vol)
    {
        float d=mass/vol;
        return d;
    }
  public:
    MetalShape(char*n, float m,float a)
    {
        strcpy(metalType,n);
        mass=m;
        accurate_density=a;
    }
    virtual void DeterminePurity()=0;
};
class CuboidMetal: public  MetalShape
{
  float length;
  float breadth;
  float height;
  float getVolume()
  {
     float v=length*breadth*height;
     return v;
  }
  public:
    CuboidMetal(char*n,float l,float b,float h, float m,float a):MetalShape(n,m,a)
    {
        length=l;
        breadth=b;
        height=h;
    }
    void DeterminePurity()
    {
        float v=getVolume();
        float d=getDensity(v);

        if(d==accurate_density)
           cout<<"Metal "<< metalType<<" is pure."<<endl;
        else
           cout<<"Metal "<< metalType<<" is not pure."<<endl;
    }
};
```

```
class SphericalMetal : public MetalShape
{
    private:
      float radius;
      float getVolume()
      {
          float v=4/3.0*22/7.0*radius*radius*radius;
          return v;
      }
    public:
      SphericalMetal(char*n,float r, float m,float a):MetalShape(n,m,a)
      {
          radius=r;
      }
      void DeterminePurity()
      {
          float v=getVolume();
          float d=getDensity(v);

          if(d== accurate_density)
             cout<<"Metal "<< metalType<<" is pure."<<endl;
          else
             cout<<"Metal "<< metalType<<" is not pure."<<endl;
      }
};
int main()
{
    CuboidMetal c1("Silver",10,9.5,2.8,2793,10.5);
    CuboidMetal c2("Gold",7.1,5.8,9.1,7103,19.3);
    CuboidMetal c3("Platinum",10,9.5,2.8,5400,21.4);

    SphericalMetal s1("Silver",21,407484,10.5);
    SphericalMetal s2("Gold",42,5991955.2,19.3);
    SphericalMetal s3("Platinum",63,22423262.4,21.4);

    MetalShape* ptr[6]={&c1,&c2,&c3,&s1,&s2,&s3};
    for(int i=0;i<6;i++)
        ptr[i]->DeterminePurity();
}
```

Output

```
Metal Silver is pure.
Metal Gold is not pure.
Metal Platinum is not pure.
Metal Silver is pure.
Metal Gold is pure.
Metal Platinum is pure.
```

FAQs

6.1 Is it mandatory to implement DeterminePurity function of MetalShape class if one wants to derive a new class such as CubicalMetal from MetalShape class?

Ans. Yes, it is mandatory. Otherwise, you will get a compilation error, while compiling that newly derived class.

6.2 Is it possible to create an object of an abstract class?

Ans. No. It is a compilation error because metal does not exist physically but various metals shape do.

6.3 A constructor is used to create an object. One cannot instantiate (create) an object of an abstract class. Then, why does an abstract class have a constructor as shown in MetalShape class?

Ans. When a derived class object is created, it is necessary to call a base class constructor to initialize attributes in the base class. MetalShape class initializes mass, metalType and accurate_density for all its derived classes.

6.4 Can an abstract class have a destructor?

Ans. Yes. An abstract class has a destructor which is called at the end of the destruction of every derived class object. It is necessary to release the memory allocated on heap during object construction by using new/malloc in the destructor.

6.5 Is it good practice to declare an abstract class destructor as virtual?

Ans. Yes. When the pointer of an abstract class is deleted, object should be destroyed from a derived to abstract class hierarchy. A virtual destructor plays an important role in releasing memory when dynamic memory allocation is performed in any of its derived classes.

6.6 xyz function is declared virtual in a base class. Will it be inherited in its derived class?

Ans. Yes. It is not mandatory to implement the virtual function. If it is not implemented, the immediate derived class's function is inherited.

It is possible to define pure virtual destructor. If you declare the destructor as pure virtual, it is mandatory for derived classes to implement it. The destructor can be declared as pure virtual inside a class definition and its implementation can be provided outside the class as in Program 6.6.

Program 6.3 can be written using the pure virtual destructor concept as in Program 6.6. For simplicity, SphericalMetal class objects are not created in this program. You can try to implement the virtual function mechanism to SphericalMeta class as an exercise.

Program 6.6

```
#include<iostream>
#include<string>
using namespace std;
class MetalShape
{
   protected:
     char metalType[10];
```

```
    float mass;
    float accurate_density;
  public:
    MetalShape(char*n, float m,float a)
    {
       strcpy(metalType,n);
       mass=m;
       accurate_density=a;
    }
    virtual void DisplayData()
    {
       cout<<"Metaltype:"<<metalType<<endl;
       cout<<"Mass:"<<mass<<endl;
       cout<<"Accurate_Density:"<<accurate_density<<endl;
    }
    virtual ~MetalShape ()=0;
};
MetalShape:: ~MetalShape ()
{
  cout<<"MetalShape Destructor Called"<<endl;
}
class CuboidMetal: public MetalShape
{
  float length;
  float breadth;
  float height;
public:
  CuboidMetal(char*n,float l,float b,float h, float m,float a):
    MetalShape(n,m,a)
  {
     length=l;
     breadth=b;
     height=h;
  }
  void DisplayData()
  {
     cout<<"Metaltype:"<<metalType<<endl;
     cout<<"Length"<<length<<endl;
     cout<<"Breadth"<<breadth<<endl;
     cout<<"Height"<<height<<endl;
     cout<<"Mass:"<<mass<<endl;
     cout<<"Accurate_Density:"<<accurate_density<<endl;
  }
  ~ CuboidMetal ()
  {
```

```
        cout<<"CuboidMetal Destructor Called"<<endl;
    }
};
int main()
{
    CuboidMetal c1("Silver",10,9.5,2.8,2793,10.5);
    CuboidMetal c2("Gold",7.1,5.8,9.1,7103,19.3);
    CuboidMetal c3("Platinum",10,9.5,2.8,5400,21.4);
    MetalShape* m[ 3 ]={ &c1,&c2,&c3};
    for(int i=0;i<3;i++)
      m[i]->DisplayData();
}
```

Output

```
Metaltype:Silver
Length10
Breadth9.5
Height2.8
Mass:2793
Accurate_Density:10.5

Metaltype:Gold
Length7.1
Breadth5.8
Height9.1
Mass:7103
Accurate_Density:19.3

Metaltype:Platinum
Length10
Breadth9.5
Height2.8
Mass:5400
Accurate_Density:21.4

CuboidMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
CuboidMetal Destructor Called
MetalShape Destructor Called
```

6.5 Upcasting, Downcasting and Object Slicing

We have studied that runtime polymorphism is achieved by storing the address of a derived class object into a base class pointer and calling the overridden function. For example, in case

of classes MetalShape,CuboidMetal defined in Program 6.3, we can implement runtime polymorphism as follows:

```
MetalShape* ptr=new CuboidMetal( );  // Storing address of derived class object into base class pointer
                                     // is upcasting.
ptr->DeterminePurity();              // Call to overridden function
```

When the address of a derived class object is stored into a base class pointer, it is known as upcasting. When the address of a base class object is assigned to a derived class pointer, downcasting occurs.

```
CuboidMetal* ptr=new MetalShape();  // This will not compile.
```

When a derived class object is assigned to a base class object, the base class portion of the derived class object is copied into the base class object and the derived class portion of the derived class object is sliced off. This is known as object slicing. For example,

```
MetalShape m("Platinum",5400,21.4);  // As MetalShape class in 6.1 program is not abstract, we can
                                     // create object.
CuboidMetal c("Silver",10,9.5,2.8,2793,10.5);
m = c;  // Object slicing happens here
```

The contents metalType ("Silver"), mass (10) and accurate_density(9.5) of object c are copied as attributes of object m, while the remaining part of object c, i.e., length(2.8), breadth(2793) and height(10.5) are sliced off.

6.6 Guess the Output

1.

```
#include<iostream>
using namespace std;
class A
{
    int a;
    public:
    virtual void display(){cout<<"A"; }
};
int main()
{
    cout<<sizeof(A);
}
```

Output

8

When any class contains at least one virtual function, it internally creates virtual pointer vptr that points to the virtual table which contains the addresses of virtual functions. Hence, the size of the object of class A is equal to the size of integer plus size of virtual pointer vptr (int). Hence, the size is 8.

2.

```
#include<iostream>
using namespace std;
class A
{
    public:
    virtual void display(){cout<<"A"; }
    void show(){cout<<"show A"; }
};
class B: public A
{
    public:
    void display(){cout<<"B"; }
    virtual void show(){cout<<"show B"; }
};
class C: public B
{
    public:
    void display(){cout<<"C"; }
    void show(){cout<<"show C"; }
};
int main()
{
     A* a1=new C();
     a1->show();
     delete a1;
}
```

Output

```
Show A
```

The address of object C is stored in the pointer of base class A. The virtual table of class A does not contain the address of show function which is defined ordinary in case of class A. Hence, the function call to show is resolved at compile time and show method defined in class A is called.

3.

```
//Refer class A ,B and C defined in 2.
int main()
{
    B* b1=new C();
    b1->show();
    delete b1;
}
```

Output

```
Show C
```

The virtual table of class B contains address of show function. Hence, the aforementioned function call gets resolved at runtime.

4.

```
#include<iostream>
using namespace std;
class A
{
    public:
    virtual void display()
    {
        cout<<"A \n";
    }
};
class B: public A
{
    public:
    B ( )
    {
        display();
    }
   virtual void display()
    {
       cout<<"B \n";
   }
};
int main()
{
    B b1;
}
```

Output

```
B
```

5.

```
#include<iostream>
using namespace std;
class A
{
    public:
    virtual void display()
    {
        cout<<"A \n";
    }
```

```
};
class B: public A
{
    public:
    virtual void display()
    {
        cout<<"B \n";
    }
};
int main()
{
    B b1;
    A & a1=b1;  // Use of reference
    a1.display();
}
```

Output

```
B
```

It is possible to implement dynamic polymorphism using references as shown in the aforementioned program. But as an array of references is not possible, writing the impact code is difficult using references and virtual functions.

6.

```
#include<iostream>
using namespace std;
class A
{
    public:
    virtual void display()
    {
        cout<<"A \n";
    }
    virtual void display(int a)
    {
        cout<<"A:"<<a;
    }
};
class B: public A
{
    public:
      void display()
    {
        cout<<"B \n";
    }
```

```
        void display(int a)
    {
        cout<<"B:"<<a;
    }
};
int main()
{
    A* a1=new B();
    a1->display( 9 );
    delete a1;
}
```

Output

B:9

It is possible to overload and override the same function. For example, display function is overloaded in class A and class B and overridden in a class hierarchy.

7.

```
#include<iostream>
using namespace std;
class A
{
    int x;
  public:
    A(int x1) {x=x1;}
    virtual void fun(){cout<<x;}
    ~A(){ };  // non-virtual destructor in base class containing virtual function
};
class B: public A
{
    int y;
   public:
    B(int x1,int y1):A(x1) {y=y1;}
    virtual void fun(){cout<<y;}
    ~B(){ };
};
int main()
{
    B b2(3,4);
    A* a2= &b2;
    a2->fun();
}
```

Output

4

Undefined behaviour

When object a2 goes out of scope, the derived class object b2 on stack should be automatically reclaimed (deleted) which will not happen as the derived class destructor will not be called. Hence, one should declare the destructor of base class – class A – as virtual. When a base class contains at least one virtual or pure virtual function, it must declare the destructor virtual. This avoids undefined behaviour which may be caused when an attempt is made to delete the derived class object through a base class pointer. Let me give one practical example. One should carry an umbrella in the rainy season even though it may not rain as no one can guarantee that it will not rain. Similarly, there is no guarantee of undefined behaviour if an attempt is made to delete the derived class object through a base class pointer.

8.

```
#include<iostream>
using namespace std;
class A
{
    int x;
  public:
    A(int x1) {x=x1;}
    virtual void fun(){cout<<x;}
    ~A(){ };  // non-virtual destructor in base class containing virtual function
};
class B: private A
{
    int y;
  public:
    B(int x1,int y1):A(x1) {y=y1;}
    virtual void fun(){cout<<y;}
    ~B(){ };
};
int main()
{
    B b2(3,4);
    A* a2= &b2;
    a2->fun();
}
```

Output

Compilation error

6.7 Points to Remember

(i) A virtual function must be a member of a class. It cannot be defined at a global level.
(ii) A virtual function cannot be static.
(iii) virtual functions are accessed by using object pointers or even the object can access it directly.

(iv) A virtual function in a base class must be defined even though it may not be used.
(v) The prototype of the base class version of a virtual function and all derived class versions must be identical. If functions with the same name have different prototype, C++ considers them as overloaded functions and the virtual function mechanism is ignored.
(vi) A base pointer can point to any type of derived object, the reverse is not true. For example, we cannot directly use a pointer to a derived class to access an object of the base type.
(vii) If a virtual function is defined in the base class, it is not mandatory to redefine it in the derived class. In such a case, a base class virtual function is called. For example, D class is derived from B class. In this scenario, the virtual function defined in B class is not implemented in D class. However, the function in B class is declared virtual because a new class D1 can be added in future which requires its implementation.
(viii) When any function is declared virtual in a class, the same function defined in all its derived classes becomes virtual by default.
(ix) It is not possible to declare a constructor as virtual.
(x) A virtual function cannot be inline because a call to inline function is resolved at compile time and virtual functions are implemented at runtime. However, modern compilers allow to declare inline function as virtual.
(xi) It is possible to use switch case statements instead of dynamic polymorphism. One can understand the type of the object and make an appropriate function call. This is discussed in Chapter 11. But use of dynamic polymorphism helps to write a more compact code.
(xii) A disadvantage of using virtual functions is that they require more instructions for execution and hence are slower. Another disadvantage is that it is difficult to understand the function call i.e. made which makes maintenance difficult.
(xiii) It is possible to force derived classes to implement particular functions by declaring those functions pure virtual in a base class.
(xiv) If you declare a function virtual in a base class, it automatically becomes virtual in all its derived classes. virtual keyword is optional for functions defined in the derived classes which are already declared virtual in a base class.
(xv) Abstract class can have virtual functions apart from pure virtual function or functions.
(xvi) A class which allows its object to be created is called as a concrete class.

6.8 Deep Knowledge Section

Q1. Does pure virtual function have implementation?

Ans. Yes. As shown in Program 6.7, it is possible to define pure virtual function outside the class definition.

Program 6.7

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

```
class A
{
    public:
    virtual void display(){cout<<"A"; }
    virtual void show()=0;
};
void A:: show(){cout<<"show A\n"; }
class B: public A
{
    public:
    void display(){cout<<"B"; }
    virtual void show()
    {
       A::show();  // Pure virtual function may be called.
       cout<<"show B";
    }
};
int main()
{
    B* b1=new B();
    b1->show();
}
```

Output

```
show A
show B
```

Q2. Guess the output of the following program?

```
#include<iostream>
using namespace std;
class A
{
    public:
    virtual void display(){cout<<"Santosh Gavali"; }
};
class B: public A
{
    private:
    void display(){cout<<"Vijay Virkar"; }
};
int main()
{
    A* ptr=new B();
    ptr->display();
}
```

Output

```
Vijay Virkar
```

Function display is declared public in a base class A, the pointer of which contains the address of derived class object B. At compile time, pointer ptr calls the function display of a base class. This call is resolved at runtime as per the virtual function mechanism discussed earlier. Hence, it calls the function in class B.

Q3. Why can a constructor not be virtual? Is it possible to implement a function that can simulate a virtual constructor?

Ans. A constructor cannot be virtual because the constructor requires the exact type of the object i.e. to be created and the constructor interacts with a memory management routine differently than ordinary functions. However, it is possible to implement a virtual function that can create the object of base class or derived class at runtime.

Program 6.8

```
#include<iostream>
using namespace std;
class A
{
    int x;
    public:
    A() {x=5;}
    virtual A* fun(){return new A();}
    virtual void show(){cout<<x; }
    virtual ~ A(){ };
};
class B: public A
{
    int y;
    public:
    B():A() {y=10;}
    virtual B* fun(){return new B();}
    virtual void show(){cout<<y<<endl; }
    virtual ~ B(){ };
};
int main()
{
    A* a1=new B();
    A* b1=a1->fun();  // Creates object of class B at runtime
    b1->show();
    A* a2=new A();
    A* b2=a2->fun();  // Creates object of class A at runtime
    b2->show();
    delete a1,b1,a2,b2;
}
```

Output

```
10
5
```

Q4. Guess the output of the following program?

```
#include<iostream>
using namespace std;
class A
{
    public:
    void display(){cout<<"A"; }
};
// class B is abstract though base class A is not abstract..Strange design
class B: public A
{
     public:
     void display(){cout<<"B";}
     virtual void show()=0;
};
class C: public B
{
     public:
     void display(){cout<<"C";}
     virtual void show(){cout<<"Showing C";}
};
int main()
{
     B* b=new C();
     b->show();
}
```

Output

```
Showing C
```

This means that it is possible that a base class can be a concrete class whereas a derived class is an abstract class. In the aforementioned program, class A is concrete class, while class B is abstract class.

Exercises

A. State True or False.

1. Virtual constructor is available in C++.
2. Polymorphism is nothing but one name, multiple forms.
3. Dynamic binding means late binding.
4. We can create an object of an abstract class.
5. Static binding and dynamic polymorphism are the same.
6. Virtual functions must be static.
7. Abstract class can contain a constructor.
8. Destructors can be virtual.

9. Virtual functions' call up is maintained by the compiler via lookup tables.
10. If your class has at least one virtual function, you should make a destructor for this class virtual.
11. A pure virtual function can have a definition.
12. Virtual function can have empty implementation.
13. Assigning the address of a derived class object to a base class pointer is downcasting.
14. Assigning a base class object to a derived class object is object slicing.

B. Guess the Output of the Following Programs:

1.
```
#include<iostream>
using namespace std;
class A
{
    virtual void display(){cout<<"Pramodini Deshmukh"; }
};
class B :public A
{
    void display(){cout<<"Vaibhav Choramale" ;}
};
int main()
{
    A* a=new B();
    a->display();
}
```

2.
```
#include<iostream>
using namespace std;
class A
{
    public:
    virtual void display(){cout<<"Sangram Kendre"; }
};
class B :public A
{
    private:
    void display(){cout<<"Govind Kendre" ;}
};
int main()
{
    A* a=new B();
    a->display() ;
}
```

3.
```
#include<iostream>
using namespace std;
class A
{
    public:
    void display(){cout<<"A"; }
};
class B :public A
{
    public:
    virtual void display(){cout<<"B" ;}
};
int main()
{
    A* a=new A();
    a->display();
}
```

4.
```
#include<iostream>
using namespace std;
class A
{
    public:
    A()
    {
        cout<<"A Constructor \n";
    }
    void Greet()
    {
        cout<<"A says hi \n";
    }
    virtual void Talk()
    {
        cout<<"A says to discuss \n";
    }
};
class B: public A
```

```
{
    public:
    B()
    {
        cout<<"B constructor\n";
    }
    void Talk()
    {
        cout<<"B says to discuss \n";
    }
};
int main()
{
    B b1;
    b1.Talk();
    b1.Greet();
}
```

5.
```
// Refer class A and class B from above question iv.
int main()
{
    A* a1=new B();
    a1->Talk();
    a1->Greet();
}
```

6.
```
#include<iostream>
using namespace std;
class A
{
    public:
    virtual void display()=0;
    void show(){cout<<"Prem";}
};
int main()
{
    A a1;
    a1.show();
}
```

C. What is meant by dynamic polymorphism? Explain with a suitable example.

D. Why is virtual function used?

E. Why is there need for pure virtual function? What is its declaration?

F. What is the need of a virtual destructor?

G. Why C++ does not directly support virtual constructor?

H. Define the Following Terms:

1. Dynamic polymorphism
2. Virtual function
3. Abstract class
4. Virtual destructor
5. Pure virtual function
6. Object slicing
7. Upcasting
8. Downcasting
9. Object slicing
10. Pure virtual destructor
11. Concrete class

Answers

A. True/False

1. False	2. True	3. True	4. False
5. False	6. False	7. True	8. True
9. True	10. True	11. True	12. True
13. False	14. False		

B. Output of Programs

1. Compilation error as display function is declared private in class A.
2. Govind Kendre. Refer deep knowledge section question.
3. A. Address of object of class A is stored in a pointer of class A. Hence, it calls function defined in base class A.
4. A constructor
   A says to discuss
   A says hi
5. A constructor
   B constructor
   B says to discuss
   A says hi
6. Compilation error as it is not possible to create object of an abstract class.