Virtual base classes

To prevent our programs to inherit multiple copies of a base class we can declare this base class as a virtual base class.
Example:

```
#include <iostream>
class base
{
        public:
        int i;
};
class deriv1:virtual public base
{
        public:
        int j;
};
class deriv2:virtual public base
{
        public:
        int k;
};
class deriv3:public deriv1,public deriv2
{
        public:
        int sum;
};
void main()
{
        deriv3 object;
        object.i=10;                    //….it’s ok?? But without virtual base??
        object.j=20;
        object.k=30;
        object.sum=object.i+object.j+object.k;
        cout<<object.i<<" + "<<object.j<<" + "<<object.k<<" is "<<object.sum<<endl;
}
```

If the derivation of “deriv1” and of “deriv2” is not virtual, then we get errors of ambiguity at line 3 in the function main.

Virtual functions

A virtual function stays virtual no matter how many times is inherited.
When a class inherits another class methods, it can happen that the class member names to be in conflict.

Example:

```
#include <iostream>
class base
{
	public:
	virtual void funct()
	{
		cout<<" The function from the base class "<<endl;
	}
};
class deriv1:public base
{
	public:
	void funct()
	{
		cout<<" The function from the class deriv1 "<<endl;
	}
};
class deriv2:public deriv1
{
	public:
	void funct()
	{
		cout<<"The function from the class deriv2 "<<endl;
	}
};

void main()
{
	base *p, b;
	deriv1 d1;
	deriv2 d2;
	p=&b;	//indicating???
	p->funct();	//which of the functions will be called?
	p=&d1;
	p->funct();	//??
	p=&d2;
	p->funct();	//??
}
```

What happens if the function funct( ) from the base class is not virtual???

Pure virtual functions

A pure virtual function is a function that is declared in the base class and that requires its own implementation of it in the derived class.

Prototype: *virtual data_type function_name(parameters)=0;*

Example:

```
#include <iostream>
class base
{
	public:
	virtual void funct()
	{
		cout<<" The base class "<<endl;
	}
	virtual void functv(void)=0;
};
class deriv:public base
{
	public:
	void funct()
	{
		cout<<" :) "<<endl;
	}
	virtual void functv(void)
	{
		cout<<"pure virtual function "<<endl;
	}
};
void main()
{
	base *p=new deriv;
	p->funct();
	p->functv();
}
```

Abstract classes

When a class contains at least one pure virtual function, then the class is called *abstract.*
C++ will not allow you to create a variable whose type is an abstract class.
A class which contains only pure virtual member functions is called *interface*.

Polymorphism

Is the ability of an object to take many forms. This is possible in C++ by using virtual functions. The same pointer can indicate to different classes in order to perform different operations.

Example:

```
#include <iostream>
#include <string>
class base
{
	public:
	virtual int add(int a,int b)
	{
```

```
            cout<<" add from the base class  "<<endl;
            return (a+b);
          }
         virtual int subtract(int a,int b)
          {
           cout<<" subtract from the base class "<<endl;
            return (a-b);
          }
          virtual int multip(int a,int b)
          {
           cout<<" multiplication from the base class "<<endl;
            return (a*b);
          }
};
class deriv1:public base
{
int multip(int a,int b)
{
        cout<<" multiplication from the class deriv1"<<endl;
        return (a*b);
}
};
class deriv2:public base
{
        int subtract(int a, int b)
        {
                cout<<" subtraction from the class deriv2"<<endl;
                return (abs(a-b));
        }
};

void main()
{
        base *p=new deriv1;
    cout<<p->add(300,299)<<" "<<p->subtract(799,200)<<" "<<p->multip(599,1)<<endl;
        p=new deriv2;
    cout<<p->add(222,111)<<" "<<p-> subtract (18,288)<<" "<<p-> multip (398,2)<<endl;
}
```

Class hierarchies

Exercises:

1. Implement a class hierarchies which allows working with geometrical shapes. Requests:

-Define an abstract class Shape. A shape is characterized by a name and has a method GetName( ) in order to obtain the name. Also, the class Shape contains an abstract method GetArea( ) which will return its area;

- Define a class Circle, derived from the class Shape. The class Circle will have a constructor receiving the name of the circle, the circle center coordinates and the radius. The class Circle will have a method to overload the method GetArea() from the class Shape;

- Define another abstract class PolygonalShape, derived from the class Shape. The class PolygonalShape class will define an interface for working with shapes made of points and segments. The PolygonalShape class will have an abstract method called GetPointsCount() in order to read the number of points of the figure. The class PolygonalShape will overload the indexing operator []. You have to declare as a virtual function operator[] in the class PolygonalShape;

- Define a class Triangle, derived from the class PolygonalShape. The Triangle class will have a constructor receiving the name of the triangle and the coordinates of three points of the triangle. The Triangle class overloads GetArea() and GetPointsCount() and the operator [];

-Define a class Rectangle, derived from the class PolygonalShape. The Rectangle class will have a constructor receiving the name and the coordinates of two opposite points (corners) of the rectangle. The Rectangle class overloads the methods GetArea() , GetPointsCount() and the operator [];

- To work with points define a class Point that will hold the coordinates of a point. The Point class will have a constructor that will get the coordinates of a point. Built in the class Point two methods GetX() and GetY() in order to read the coordinates. Add also the method DistanceTo(Point& other) which will return the distance from one point to another point. Use the class Point class in the classes Circle, Triangle and Rectangle (composition relationship);

The function main is:

```
int main()
{
Circle c("C", 2.5, 2.5, 2.5);
Triangle t("ABC", 0, 0, 5, 0, 0, 5);
Rectangle r("PQRS", 0, 0, 5, 5);
int i;
cout << "------ SHAPES ------" << endl;
Shape* shapes[3];
shapes[0] = &c;
shapes[1] = &t;
shapes[2] = &r;
for (i=0; i < 3; i++)
cout << shapes[i]->GetName() << ": area=" << shapes[i]-
>GetArea() << endl;
cout << "------ POLYGONAL SHAPES ------" << endl;
PolygonalShape* polyShapes[2];
polyShapes[0] = &t;
polyShapes[1] = &r;
for (i=0; i < 2; i++)
{
cout << polyShapes[i]->GetName() << ": area=" << polyShapes[i]-
```

```
>GetArea() <<
", pointsCount=" << polyShapes[i]->GetPointsCount() << ",
points:";
for (int j=0; j < polyShapes[i]->GetPointsCount(); j++)
cout << " " << (*polyShapes[i])[j];
cout << endl;
}
return 0;
   }
```

The output should be:

```
---=== SHAPES ===---
C: area=45.3416
ABC: area=12.5
PQRS: area=35.3553

---=== POLYGONAL SHAPES ===---
ABC: area=12.5, pointsCount = 3, points: (0, 0) (5, 0) (0, 5)
PQRS: area=35.3553, pointsCount  = 4, points: (0, 0) (0, 5) (5, 5) (5, 0)
```

Hint

1) When implementing the indexing operator (*operator[]*) for the *Triangle* and *Rectangle* classes, you must handle the situation when an invalid index is passed to the operator. For example for a *Triangle* it only makes sense to pass 0, 1 or 2 as index value, since we have only three points in a triangle. Also for a *Rectangle* it only makes sense to pass 0, 1, 2 or 3 as an index value, since we have only four points in a *Rectangle*. Define a static member of type *Point* in the *PolygonalShape* class. Call this member BAD_POINT and initialize it with whatever values you consider (for example a point with negative coordinates). Then, when overloading *operator[]*, if the index value is invalid, just return a reference to BAD_POINT. Make the BAD_POINT member protected, so that it can be accessed from the derived classes.

2) Before starting the implementation, create the UML class diagram for an overall structure of the system.

The class hierarchy: (the UML diagram)