7. Class hierarchies

The ***inheritance*** represents a distinctive element of the object-oriented programming paradigm. Because it allows for a certain class to inherit the data and member functions of other classes, it results from here the possibility of existence of class hierarchies.

An inheritance is also called a ***derivation*** operation: a class *A*, which inherits the members of another class *B*, it is said to be ***derived*** from the class *B*, and the class *A* is said a ***base*** class for *B*. In the UML language the inheritance relation is denoted as:

the arrow being oriented to the base class.

Example. The classes *P* and Q inherit the class *X*. The class diagram is presented in the Figure 7.1.

```
class X {
  // ...
};

class P: X {
  // ...
};

class Q: X {
  // ...
};
```

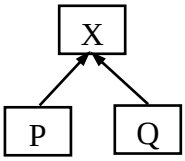

Figure 7.1. An inheritance example.

Remark: When a class is derived only from one class we have ***simple inheritance***; when a class inherits several classes we have ***multiple inheritance***.

7.1 Classes derivation

The syntax for deriving a class from one or more base classes was presented in the paragraph 3.1. The inheritance operation is specified in the class declaration, immediately after the name of the current class, by listing all base classes.

So, we can build classes hierarchy based on the inheritance relation. When simple inheritance is used, a class hierarchy has a tree structure, all classes from the hierarchy being derived from a single base class. Thus single inheritance represents a simple and safe manner of designing an application.

In the case when it is used multiple inheritance, the hierarchy structure represents an oriented graph, the base classes that generate the respective hierarchy being those vertices which do not have any input edge.

Remark. Unions can not take part in a class hierarchy, because they cannot be base classes, and neither derived classes.

Example 7.1.

```
struct Point {
  double x,y;
  void SetCoord (double a, double b) {
    x = a;
    y = b;
  }
};

struct Circle: Point {
  double r;
  void SetRadius (double a) { r = a; }
};
```

It is observed that in a derived class are specified only supplementary members that are added to the base class.

An object from a derived class is regarded by the compiler as having all the members specified in the derived class, at which is added a ***hidden object*** which is an instance of the base class. In this way the inheritance represents a special type of composition relation. For the previous example, an instance object of the *Circle* class has the following structure, as presented in Figure 7.2.

In the following sequence:

```
Point p1;
p1.SetCoord(1, 0);
Circle c1;
c1.SetCoord(7, 9);
c1.SetRadius(2);
```

the object *c1* contains a hidden object of the *Point* class, which must not be specified by its name (it is not needed a supplementary selection operator). In this way, the members of the class *Point* can be directly accessed, as in the case they are declared in the *Circle* class.

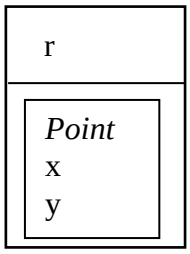

Figure 7.2. The memory image of an object of the class *Circle*

The inheritance of a base class in a derived class can modify the access type of the members from the base class. A base class can be inherited in three ways: `public`, `protected`, and `private`. The keyword which specifies the inheritance type of the base class precedes the name of the base class; in the case when this keyword is absent, the inheritance is by default of `public` type for *struct* and `private` for *class*.

The access at the members of the base class can be sometimes restricted in the derived class, but never more restrictively, as seen in the next table:

The access type of a member in the base class	*Inheritance type*	*The access type of the member after inheritance*
Public Private Protected	Public	Public *Inaccessible* Protected
Public Private Protected	Private	Private *Inaccessible* Private
Public Private Protected	Protected	Protected *Inaccesible* Protected

It is observed the fact that the `private` members of a base class are inaccessible in the derived class, whatever is the inheritance type. The others two access types are kept unaltered in the derived classes in the case of `public` type inheritance or modified at the `private` (or `protected`) type in the case of `private` (or `protected`) type inheritance.

It is also observed from the previous table the role of `protected` access type of a member from a base class: to allow its use in derived classes, but to not allow the access to it from the exterior of the base class.

Example 7.2.

```
class A {
  int p;
public:
  int q;
protected:
  int r;
  // ...
};

class A1: A {
  int x;
public:
  void f1();
  // ...
};

class A2: public A {
  int y;
public:
  void f2();
  // ...
};
```

In the implementation of *f1* and *f2* functions can be direct used the members *q* and *r* from the *A* class, but in the class *A1* the members *q* and *r* are not accessible outside (for example, there can not be written a sequence like:

```
A1 a1;
int k = a1.q;
```

while in the class *A2* the member *q* is `public` (the next sequence is correct):

```
A2 a2;
int k = a2.q;
```

Remark. If it is desired as certain public members from the base class that become private by a inheritance of `private` type, to become public in the derived class, their name must be specified after the key word `public` (only the name, not its associated data type). For example, if it is desired as in the class *A1*, the member *q* to be become again public, the declaration of the *A1* class must be:

```
class A1: A {
  int x;
public:
  A::q;
  void f1();
  // ...
};
```

In this way, the *q* member is visible also outside the *A1* class.

7.2 Constructors and destructors in the class hierarchies

The constructors and destructors are the only functions of a base class which cannot be inherited in a derived class (excepting the *assignment operator*, which will be discussed later). This thing is natural, because the operations of creation and destruction of these objects are owned by the respective class and they cannot be passed down in a class hierarchy.

Because an object of derived class is similar to a composed object, which contains a hidden sub-object of the base class, the constructor of the object of the derived class must call first the constructor of the base class. The call of the constructor of the hidden object can be realized explicitly in the constructor initializer list of the derived class, or it can be generated by default by the compiler in the case when the base class does not have declared constructors.

The constructor call of the base class in the constructor initializer list of the derived class is relized by using the name of the base class, because the corresponding hidden object of the base class does not have an explicit name.

Example 7.3.

```
class Point {
protected:
  double x, y;
public:
  void SetCoord(double a, double b) {
    x = a;
    y = b;
  }
  Point(double a = 0, double b = 0) {
    SetCoord (a, b);
  }
  double X() const { return x; }
  double Y() const { return y; }
};

class Circle: public Point {
  double r;
public:
  void SetR(double a) { r = 0; }
  Circle(double a = 0, double b = 0, double c = 1):
      Point(a,b), r(c) { }
  Circle(Circle &c): Point(c.x, c.y), r(c.r) {}
};
```

It is known that a constructor for a class can be automatically generated by the compiler, in the case when the respective class does not have a declared constructor. There is an exception, in the case when a class is derived from one or more base classes, and all these base classes have explicit constructors which contain parameters: in this case the derived class must contain an explicit constructor that explicit calls the constructors of the base classes that have parameters.

The destructors of the objects of the derived classes call by default the destructors of the objects of the derived classes in reverse order of the constructors call order. For example, for an object *o* of a class *D* derived from a class *B*, the destructors call order is:

- destructor of the *o* object;
- destructor of class *B*;
- destructors of supplementary members of class *D*;
- destructors of members from the class *B*;

Remark. In the case of multiple inheritance, the calling order of the constructors of the objects from the base classes is the order of their declaration in the derived class, while the calling of destructors is performed in reverse order.

Example 7.4. A macrodefinition is used for classes definition:

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

#define CLASS(ID) class ID {\
  public:\
    ID(int){cout<<"Class Constructor  "<<#ID<<endl;}\
    ~ID(){cout<<"Class Destructor  "<<#ID<<endl;}\
  };

CLASS(B1);
CLASS(B2);
CLASS(M1);
CLASS(M2);

class D: public B1, B2 {
  M1 m1;
  M2 m2;
public:
  D(int): m1(10), m2(20), B1(30), B2(40) {
    cout << "Constructor class D" << endl;
  }
  ~D() { cout << "Destructor class D" << endl; }
};

int main() {
  D d(0);
  // ...
}
```

The output of the program is the following:

```
class Constructor B1
class Constructor B2
class Constructor M1
class Constructor M2
class Constructor D
class Destructor D
class Destructor M2
class Destructor M1
class Destructor B2
class Destructor B1
```

When a derived class does not have an explicit defined *copy-constructor*, it will be automatically generated by the compiler. It calls the copy-constructors of the base classes, followed, if necessary, by the copy-constructors (or pseudo-constructors) of the members of the class.

Example 7.5.

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

class Base {
  int n;
public:
  Base(int i): n(i) {
    cout << ”Base(int i)” << endl;
  }
  Base(const Base& b): n(b.n) {
    cout << ”Base(const Base& b)” << endl;
  }
  Base(): n(0) { cout << ”Base()” << endl; }
  void Print() const {
    cout << ”Base; n=‟ << n << endl;
  }
};

class Member {
  int n;
public:
  Member(int i): n(i) {
    cout << ”Member(int i)” << endl;
  }
  Member(const Member& m): n(m.n) {
    cout << ”Member(const Member& m)” << endl;
  }
  void Print() const {
    cout << ”Member; n=‟ << n << endl;
  }
};
```

```
class Derived: public Base {
  int n;
  Member m;
public:
  Derived(int i): Base(i), n(i), m(i) {
    cout << ”Derived(int i)” << endl;
  }
  void Print() const {
      cout << ”Derived; n=“ << n << endl;
      Base::Print();
      m.Print();
  }
};

int main() {
  Derived o1(7);
  cout << ” copy-constructor call: “ << endl;
  Derived o2 = o1;
  cout << ”Values in o2: ” << endl;
  o2.Print();
  return 0;
}
```

Program output:

```
Base(int i)
Member(int i)
Derived(int i)
Copy-constructor call:
Base(const Base& b)
Member(const Member& m)
Values in o2:
Derived; n=7
Base; n=7
Member; n=7
```

The copy-constructors of the classes *Base* and *Member* have been implicitly called (lines 5 and 6), and also the pseudo-constructor for copying the member *n* (in line 10, the value of member *n* of the object *o2* is also 7, as the value for *n* from the object *o1*).

In the case when a copy-constructor is explicitly defined in a derived class, it must call explicitly (in the constructor initializer list) the copy-constructor of the base class.

Example of a wrong copy-constructor of the class *Derived*:

```
Derived(const Derived& d): n(d.n), m(d.m) { }
```

In this case the last three lines of program output are:

```
Derived; n=7
```

```
Base; n=0
Member; n=7
```

Because there is not an explicit call for a constructor of the class *Base*, the compiler will insert a default constructor for this.

A correct copy-constructor example of the class *Derived*:

```
Derived(const Derived& d): Base(d), n(d.n), m(d.m) { }
```

The program output will be the same as in the last example.

Remark. The copy-constructor parameter of the class *Base* is a reference to an object of the derived class, which is a correct operation in the case of public inheritance (*Derived* class is a subtype of the *Base* class). In the case of private inheritance, probably it is indicated that such a copy-constructor to be default called by the compiler.

7.3 Public and private inheritance

The using of `public` or `private` inheritance type has a distinct significance in designing and developing an application.

A. Public inheritance

The `public` inheritance is related especially to the *design* phase of an application, in the case when a derived certain represents conceptually a specialization of the base class. For example, the *student* class is a specialization of the *person* class, because every student is a person. In the literature this relation type is called a "is-a" relation.

This type of inheritance supposes that the derived class inherits both the interface and the implementation of the base class; in this way an object of the derived class can be used instead of an object from the base class.

Example 7.6. Let us consider the classes *Point* and *Circle* from the example 7.3. The function *Distance* determines the distances between two points:

```
double Distance (Point& p1, Point& p2) {
  double d = sqrt((p1.X()-p2.X())*(p1.X()-p2.X())+
                  (p1.Y()-p2.Y())*(p1.Y()-p2.Y()));
  return d;
}
```

The next sequence is correct and displays the values 1 and 4:

```
Point p1, p2;
p1.SetCoord(1, 1);
p2.SetCoord(0, 0);
Circle c1, c2;
c1.SetCoord(7, 7);
c2.SetCoord(3, 3);
double d1 = Distance(p1, p2);
double d2 = Distance(c1, c2);
cout << d1 << d2 << endl;
// ...
```

An important propriety of the inheritance relation is the fact that a class that is public derived from a base class is treated as a ***subtype*** of that base class. In this way all instances of the derived class are compatible with the objects of the base class, and they can be used instead of them. The following sequence is correct:

```
Point p2;
p2 = c1;
```

Because a public derived class inherits both the declaration and the implementation of the base class, in the derived class some of the members from the base class can be redefined. The redefinition of a member supposes the hiding in the derived class of the member from the base class with the same name.

Example 7.7.

```
class A {
public:
  void f() const {
    cout << ”f in class A” << endl;
  }
};

class B: public A {
public:
  void f() const {
    cout << ”f in class B” << endl;
  }
};

int main (){
  A a;
  a.f();        //f from the class A
  B b;
  b.f();        //f from the class B
  b.A::f();     //f from the class A
  // ...
}
```

When it is desired the use of the hidden member, this member must be prefixed by the name of the base class.

Remark. Usually the redefinition of the members from a base class in a derived class is not indicated, because it supposes an error of designing of the class hierarchy. Instead of the redefinition, the ***virtual functions*** can be used.

Example 7.8. There are birds that do not fly, even if the great majority of them flies; for example the penguin. The next example represents a wrong design of the classes hierarchy:

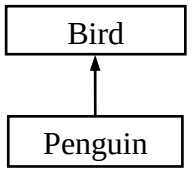

Figure 7.3. A class hierarchy

```
class Bird {
public:
  void flies () const {
    cout << ”bird flies” << endl;
  }
  // ...
};

class Penguin: public Bird {
  // ...
};

// ...
Bird eagle;
Penguin penguin;
eagle.flies();        //correct
penguin.flies();      //error!!
```

A way for modify the above code is the redefinition of the *flies* function in the *penguin* class:

```
class Penguin: public Bird {
public:
  void flies () const {
    cout << ”bird does not flies” << endl;
  }
  // ...
};

// ...
Penguin penguin;
penguin.flies();      //correct
```

A better solution is a correct design of the class hierarchy that must distinguish between the two bird categories, as presented in Figure 7.4.

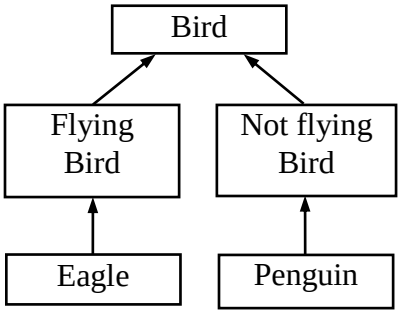

Figure 7.4. Re-design of the class hierarchy from the Figure 7.3

```
class FlyingBird: public Bird {
public:
  void flies () const {
    cout << ”bird flies” << endl;
  }
  // ...
};

class NotflyingBird: public Bird {
public:
  void flies () const {
    cout << ”bird does not fly” << endl;
  }
// ...
};

class Eagle: public FlyingBird {
  // ...
};

class Penguin: public NotflyingBird {
  //...
};

// ...
Eagle eagle;
Penguin penguin;
Eagle.flies();
```

B. Private inheritance

The private inheritance is specific to the *developing* phase of an application, when there exists certain class hierarchies already defined.

In the case of `private` inheritance, all the members of the base class become `private` members in the derived class, which make them not available outside of the derived class (with the exception of the members `private` from the base class, which become inaccessible in the derived class). So, a derived class inherits only the implementation of the base class, and not its interface. From this point of view, this kind of relation is called in the literature as "an_implementation_of" relation.

In the case of `private` inheritance an object from the derived class is not converted by the compiler to a base class object (the derived class does not represent a subtype of the base class).

Example 7.9. A new perspective of the previous defined classes *Point* and *Circle*.

```
class Point {
protected:
  double x, y;
public:
  void SetCoord (double a, double b) {
    x = a;
    x = b;
  }
  Point (double a = 0, double b = 0) {
    SetCoord(a, b);
  }
  double X() const { return x; }
  double Y() const { return y; }
};

class Circle: Point {
public:
  double r;
  void SetRadius (double a = 1) { r = a; }
  Circle (double a = 0, double b = 0, double c = 1):
         Point(a,b),r(c) { }
};

double Distance(Point p1, Point p2) {
  double d = sqrt((p1.X()-p2.X())*(p1.X()-p2.X())+
                (p1.Y()-p2.Y())*(p1.Y()-p2.Y()));
  return d;
}

// ...
point p1(0, 0), p2(1, 1);
circle c1(3, 3, 1), c2(7, 7, 2);
double d1 = Distance(p1, p2);   //correct
double d2 = Distance(c1, c2);   //error!!!
```

Remark. The same type of relation "is_an_implementation_of" is also represented by the objects composition, because for a sub-object of a compound class is inherited only the implementation of

the class to which the sub-object belongs, and not its interface. Usually it is indicated the use of objects composition in this case, every time it is possible. The using of private inheritance is necessary only in the cases when the base class contains protected members that otherwise cannot be used in the derived class.

Example 7.10. Let us suppose the writing of a class *OrderedList*, which stores the elements of a list in an increasing order by using the already defined *list* class. The method used to keep the list sorted is to insert each element of the list in a corresponding position, so that the list to remain ordered also after insertion (the method of sorting by insertion). There are two variants, having the *list* class, defined in chapter 5: (a) deriving the class *OrderedList* from the class *list*, or (b) defining the class *OrderedList* as a composed class which contains a sub-object of the *list* type. The public derivation is not indicated in this case, because in this way the interface of the *list* class will be inherited also in the class *OrderedList* and the function *Add* of the class *list* will become visible; this will make as an ordered list to become unordered by calling the function *Add* that adds each element as the last element of the list.

```
class OrderedList: list {
public:
  OrderedList();
  ~ OrderedList();
  void AddOrdered (int);
  void Print();
};
```

C. Type conversion between base classes and derived classes in the case of public inheritance

An important property of the inheritance relation is that a public derived class from a base class is treated as a subtype of the base class. In this way the objects of the derived classes are compatible with the objects of the base class and they *can appear in an assignment statement*. For the previous example of the *Circle* and *Point* classes, the next assignment statement is correct:

```
Point p2;
Circle c1;
p2 = c1;
```

The assignment operator will be discussed later, but for the previous example it is considered that it copies bit by bit the data members of the object from the right part of assignment, in the object from the left part.

Remarks:

1. The assignment operation copies only the members of the base class;
2. The assignment can work only in a direction: there can not be assigned an object of a base class to an object of a derived class. For example, the next instruction is wrong:

```
c1 = p2;
```

The rule of assignment compatibility is extended also to pointers and object references. For example, the next sequence is correct:

```
class B {
  // ...
};

class A1: public B {
  // ...
};

class A2: public B {
  // ...
};

// ...
A1 a1;
A2 a2;
B *pb;
pb = &a1;
pb = &a2;
```

In the previous example there exists a default conversion type from a pointer to a derived class to a pointer to the base class. Such a conversion, from a pointer (or reference) to a derived class to a pointer (or reference) to the base class is called ***upcasting.*** This name comes from the default conversion of a type (*cast*) and from the direction of the conversion in the class hierarchy (*up*): from a bottom hierarchy class to a upper class.

The ***upcasting*** operation is frequently used in the applications that use class hierarchies, allowing a uniform treatment of the objects from such a hierarchy by using pointers to the base class.

For example, in the case of public inheritance between *Point* and *Circle* classes, the use of *Distance* function is based on upcasting:

```
double Distance (Point& p1, Point& p2) {
  // ...
}

Circle c1(7, 3), c2(2, 2);
// upcasting from Circle& to Point&
double d = Distance(c1, c2);
```

7.4 Multiple inheritance

Not all the languages which support the object-oriented programming paradigm accept ***multiple inheritance.*** For example, the Smalltalk language, which was the reference language until the appearance of C++ language does not accepts this type of inheritance. Smalltalk is called a *pure* object-oriented language, which has a predefined class hierarchy representing a tree with only one

root called *Object*. Every class defined by the programmer must be derived from Object or from a class derived from *Object*.

The C++ language is not a *pure* object-oriented language (it is a hybrid language), but it has the advantage of allowing the creation of classes and classes hierarchies, independent of a certain predefined hierarchy.

The multiple inheritance appears in the case when a certain derived class inherit (`public` or `private`) from more than one base classes. An example of multiple inheritance exists in the predefined classes for Input/Output operations (defined in the *iostream* header file). The diagram of this hierarchy is presented in the Figure 7.5.

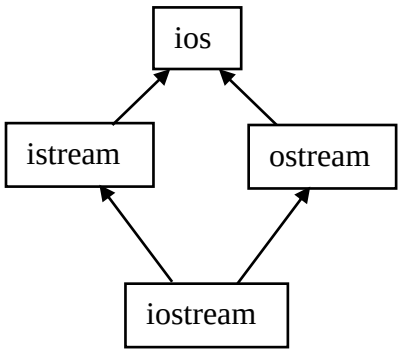

Figure 7.5. The iostream class hierarchy

There are two important problems which can appear in the case of using multiple inheritance: the ***duplication of hidden objects***, and the ***existence of some members with the same name*** in the different base classes.

The duplication of hidden objects can appear in the case of hierarchies that has a structure as presented in the previous figure (called also the diamond hierarchy type, because its shape). Because an object of a derived class has a hidden sub-object of the base class, for a hierarchy as:

```
class B { /* ... */ };

class M1: public B {
  int a;
  // ...
};

class M2: public B {
  int b;
  // ...
};

class MI: public M1, public M2 {
  int m;
  // ...
};
```

an object of the class *MI* has the following structure, as presented in Figure 7.6.

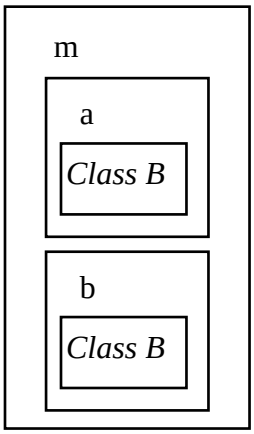

Figure 7.6. The structure of the memory allocated to an object of the class *MI*.

It is observed that there are two identical hidden sub-objects, corresponding to the same class *B*. This fact produces an *additional* amount of memory, which can on its turn to induce ambiguities in an application which it is not correct designed.

The main problem of multiple inheritance is generated by the case in which several base classes have a member with the same name: how this member will be used in the derived class?

Example 7.11.

```
class B1 {
public:
  int a;
  // ...
};

class B2 {
public:
  double a;
  // ...
};

class D: public B1, public B2 {
  // ...
};

void Processing () {
  D d;
  d.a = 5;      // error!!
  // ...
}
```

In the previous example, the statement *d.a=5;* generates an error due to the confusion which appears at the member selection.

In this case there exist two common variants of avoiding the confusion:

a) the explicit use of a certain member by using the resolution operator;
b) the redefinition in the base class of a member with the same name;

In the first case the necessity of writing correct code is exclusively the task of the programmers who uses a class hierarchy already designed. For example, the statement for the previous example that generates the error could be written as follows:

```
d.B1::a = 5;
```

or

```
d.B2::a = 5;
```

The second case represents a problem for the programmers who design the class hierarchy.

Example 7.12. The next example represents a class hierarchy that defines a class *CircleText* thate allows displaying a text in the circle (the implementations of member functions that are not *inline* were not specified):

```
class Point {
protected:
  int x, y;
public:
  Point(int a, int b): x(a), y(b) { }
  int X() const { return x; }
  int Y() const { return y; }
};

// graphic point
class GPoint: public Point {
protected:
  int visible;
public:
  GPoint(int a, int b): Point(a, b), visible(1) { }
  void Show();       //displays a graphic point on the screen
  void Hide() { visible = 0; }
  int IsVisible() const { return visible; }
  void Translate(int a, int b) { x = a; y = b; }
};

class Circle: public GPoint {
protected:
  int r;
public:
  Circle(int a, int b, int c): GPoint(a, b), r(c) { }
  void Show();       //draw a circle
};
```

```
//displays a message on the screen in a rectangle
class Message: public Point {
public:
  char *msg;          //message
  int l, L;           //rectangle dimensions
  Message(int a, int b, int c, int d, char *m):
    Point(a, b), l(c), L(d), msg(m) { }
  void Show();        //displays the message
};

class CircleText: Circle, Message {
public:
  CircleText(int a, int b, int r, char *m):
    Circle(a, b, r), Message(a, b, r, r, m) { }
  void Show()   //displays the circle with the message inside
  {
    Circle::Show();
    Message::Show();
  }
};

int main() {
  // ...
  CircleTexT c1(250, 100, 25, ”circle C1”);
  c1.Show();
  // ...
}
```

It is observed the fact that the function *Show* was inherited in the class *CircleText* and it hides the functions with the same name from the classes *Circle* and *Message* (on which it uses with the help of the resolution operator).

In the previous class hierarchy, the multiple inheritance it is not absolutely necessary. A more real alternative can be the class *CircleText* as a composed class which contains inside two private objects, instances of the *Circle* and *Message* classes:

```
class CircleText {
  Circle circle;
  Message message;
public:
  CircleText(int a, int b, int r, char *m):
    circle(a, b, r), message(a, b, r, r, m) { }
  void Show()         //displays the message in the circle
  {
    circle.Show();
    message.Show();
  }
};
```

The necessity of using multiple inheritance can appear when a derived class needs the control over the base classes (by using the virtual functions mechanism, for example).

Another example is referring to the necessity of using some class hierarchies for which the sources cannot be modified (there are available only libraries and header files). An example which will be later discussed in the 10th chapter, considers the using of a container without the using of the `template` mechanism. Let us suppose that we have a class *Container*, which stores the pointers to a list of abstract objects from the *Container* class. In order to be able to use the *Container* class in an application for storing objects belonging to another class, *Figure*, where the application cannot have access, a new class, *DerivedFigure*, can be created by inheriting both the *Object* and *Figure* classes (as presented in Figure 7.7).

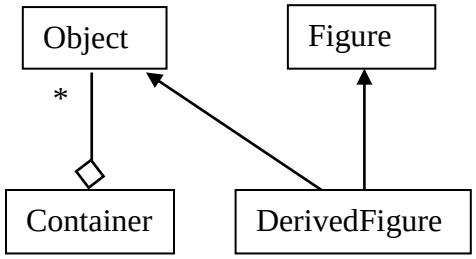

Figure 7.7. Multiple inheritance

In this way, by using the upcasting mechanism, the container can store also references to the objects of the class *DerivedFigure*.