6. Objects composition

An important problem in the case of programming paradigm is that referring to ***code reusing***. In the case of designing a new class, it is simpler and more efficient the use of certain already created, implemented, and tested classes, than writing from scratch the corresponding code.

There are usually two main methods for reusing an already existent code: ***object composition*** and ***class inheritance***. In the case of the first method, objects of certain compound classes are composed of one or more objects belonging to other classes, and this imposes for the declaration of the compound class that certain data members to be objects from other classes.

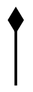
The code reusing by objects composition is not a method specific for object-oriented programming, even if it is a very used in this paradigm. On the contrary, the method of class inheritance is specific to this paradigm and it is based on a class hierarchy mechanism from where some classes are derived from another classes.

An important difference between the two methods is that of the utilization mode of the existent classes. In the case of inheritance it is generally desired to use their interface, the inheritance relation being in most cases a *refining* relation: a derived class that inherits a base class adds to the base class more specific information. In the case of composition, the relation used is one of the *membership* relation. A compound class uses the properties of the component classes, without inheriting their interfaces.

6.1 Defining compound classes and using composite objects

*The **composition*** is a relation of class ***association***, where objects of the compound class must manage the life cycle of their component objects, like the creation operation and their destruction.

In the UML language the composition relation is graphical represented as:



where the diamond is related to the compound class.

Example: The class diagram of the following three classes is presented in Figure 6.1.

```
class A;
class B;
class X {
  A a;
  B b;
  // ...
};
```

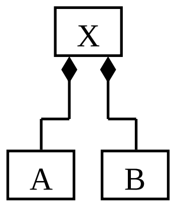


Figure 6.1. A compound class

The objects which are data members in a compound class can be private (or protected) or public members. The object components from a compound class are, usually, private (or protected), because in the compound class is used the functionality of the component classes, and not their interfaces.

When the objects members of a compound class are public members they can be direct referred by the selection operator.

Example 6.1:

```
class A {
public:
  int n;
  A();
  // ...
};

class B {
public:
  double z;
  void f();
  B();
  // ...
};

class X {
  char c;
public:
  A a;
  B b;
  X();
  // ...
};

// ...
void Processing () {
  X x;
  x.a.n = 3;
  x.b.z = 4.5;
  x.b.f();
};
```

When the component objects are private members into a compound class these members cannot be direct accessed by using of the selection operator. Moreover, the compound class must contain its own functions for accessing the members of the component objects.

These access functions of the compound class can, eventually, have the same name as the functions of the component classes. But in the case when different component classes contain functions with the same name, the corresponded access functions from the compound class must be distinct.

Example 6.2:

```
class A {
 int n;
public:
  int N() const { return n; }
  void g() { cout << ” g function in A class” << endl; }
  A(int k = 0) { n = k; }
  // ...
};

class B {
  double z;
public:
  double Z() const { return z; }
  void f() { cout << ” f function in B class” << endl; }
  void g() { cout << ” g function in B class” << endl; }
  B(double k=0) { z = k; }
  // ...
};

class X {
  char c;
  A a;
  B b;
public:
  void f() { b.f(); }
  void ga() { a.g(); }
  void gb() { b.g(); }
  X();
  // ...
};

void Processing () {
  X x;
  x.f();
  x.ga();
  x.gb();
  // ...
}
```

6.2 Creation and destruction of composite objects. The constructor initializer list

If a class contains objects of other classes as data members, the constructors and the destructor of the compound class must manages the calling of the appropriate constructors and the destructors of the component classes, in order to initialize the sub-object of the composite object, and to destroy these sub-objects respectively.

The C++ language allows this operation with the help of the *initialization list of members*, which is called ***constructor initializer list***. A constructor initializer list is specified between the constructor header and its body. The list is preceded by the ':' character and the list elements are separated by comma.

Each element from the initialization list is associated to a class member and it is specified by its member name and arguments that will be passed to its constructor. For example, for the class *X* from the previous example, the class constructor can be described as:

```
X::X(): a(5), b(5.3), c('x')
{
  cout << " X class constructor";
}
```

The constructor call for the objects *b* and *c* can be seen as the following definitions:

```
A a(5); B b(5.5);
```

The name of the component classes must not be specified in the constructor initializer list; because there cannot exists two member objects with the same name in the compound class, the compiler can determine from the names of these sub-objects the classes that the sub-objects belong to. It follows that for every element from the constructor initializer list it is called a constructor of the class that the respective sub-object belongs to.

Remarks:

1. The constructor initializer list can appear in the inline constructors, and also in the ones implemented outside of the class declaration, as in the previous example.
2. In the constructor initializer list can appear, by extension, elements for those members which belong to a predefined data type. This extension allows a consistent syntax which treats initializer variables as a pseudo-constructor. In the previous example, the element *c('x')* is seen as the initialization *c='x'*.

The ***pseudo-constructor*** notion is used in the C++ language also for initialization of variables belonging to a predefined type outside a certain class, as in the following sequence:

```
int n(3);
int* p = new int(7);
```

A good programming style imposes as the constructor initialization list to have the number of elements equal to the number of data members of the compound class, ensuring thus a safe initialization of the data members before the execution of the statements from the body of the class constructor. There are cases when the explicit call of the constructors is not necessary in the initialization list. For example, if a sub-object has an implicit constructor or a constructor with default values for parameters and it is desired its call (and not the call of another constructor), the corresponding initializer element from the initialization list can be omitted.

As specified, the object destructors are called in the moment of their cleanup. The call order of destructors is always reverse of the order of the constructors call.

It results from here some important effects in the case of composite objects:

1. The destructors for sub-objects are called after the destructor call of the composite object. If a sub-object is also composed from other objects, this process is recursively performed.
2. The order of constructors call for the initialization of the members of a composite object is not necessary the order in which the members appear in the initialization list; the constructors and pseudo-constructors are called in the order in which the members appear in the class declaration. For example, if a compound class has two constructors in which the order of the component appearance in the initialization list is different, this fact cannot determine the correct order of destructors call (because the class can have maximum one destructor with only one order of destructor calls). As consequence the compound class constructors will call the constructors of the members always in the same order (the appearance members in the class declaration).
3. The destructors of the sub-objects are called in the reverse order of the declaration of these sub-objects in the compound class.

Example 6.3:

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

class A1 {
  int p;
public:
  A1(int k = 0) {
    p = k;
    cout << ”A1 class constructor” << endl;
  }
  ~A1() { cout << ”A1 class destructor” << endl; }
};

class A2 {
  int q;
public:
  A2(int k = 0) {
    q = k;
    cout << ” A2 class constructor” << endl;
  }
```

```
    ~A2() { cout << ”A2 class destructor” << endl; }
   };

  class A {
    A1 a1;
    A2 a2;
  public:
    A(int i = 0, int j = 0): a1(i), a2(j)
    {
      cout << ”A class constructor” << endl;
    }
    ~A() { cout << ”A class destructor” << endl; }
   };

  class B {
    double z;
  public:
    B(double k = 0) {
      z = k;
      cout << ”B class constructor” << endl;
    }
    ~B() { cout << ”B class destructor” << endl; }
   };

  class X {
    A a;
    B b;
  public:
    X(int i = 0, int j = 0, double k = 0): a(i, j), b(k)
    {
      cout << ”X class constructor” << endl;
    }
    ~X() { cout << ”X class destructor” << endl; }
   };

  int main() {
    cout << “main function starts” << endl;
    {
      X x(7,9,3.33);
      cout << ”x object was created” << endl;
    }
    cout << ”main function ends” << endl;
    return 0;
  }
```

The program output is the following:

```
main function starts
A1 class constructor
A2 class constructor
A class constructor
```

```
B class constructor
X class constructor
x object was created
X class destructor
B class destructor
A class destructor
A2 class destructor
A1 class destructor
main function ends.
```