8 Friend classes and functions. Nested classes

The C++ language allows, in addition to the inheritance and composition relations, also other methods by which a class can accesses the members of other classes. These methods are ***friend classes*** and ***nested classes.*** Unlike composition and inheritance, in the case of friend classes or nested classes the access to class members cannot be done directly, but through some objects of the respective classes.

8.1 Friend functions and classes

It is necessary in some cases that functions that uses objects of a certain classes to refer its private members (`private` or `protected`). For example, in the case when the *Point* class is defined as:

```
class Point {
protected:
  double x, y;
Point(double a, double b): x(a), y(b) { }
};
```

a function which determines the distance between two points would need the reference to the members *x* and *y* of the object from the *Point* class:

```
double Dist(Point& c, Point& b) {
  double d = sqrt((a.x-b.x)*(a.x-b.x) +
      (a.y-b.y)*(a.y-b.y);
  return d;
}
```

A solution for this problem would be as the *Dist* function to be defined as a ***friend*** function of the *Point* class, which has access to its *private* members;

```
class Point {
friend double Dist(Point& c, Point& b);
protected:
  double x, y;
public:
  Point(double a, double b) : x(a), y(b) { }
};
```

An example of using the function *Dist* is:

```
void Processing() {
  Point p1(1, 1), p(2, 2);
  double d = dist(p1, p2);
  // ...
}
```

Remarks:

1) The declaration of a ***friend function*** is made in a class where the function is friend and not in the function declaration;
2) The declaration of a ***friend function*** will be always visible outside the class, regardless of the place where it is declared in the class (the *friend* function does not belong to the class to which it is friend; it is exterior to the class).
3) The ***friend function*** has access at all the members of the friend class, indifferently if they are public or not.
4) A ***friend function*** cannot have directly access to the members of its friend class, but only by the instances of this class. For example, in the body of *Dist* function the members *x* and *y* can not be directly accessed by its name, so the following statement is incorrect:

```
d = sqrt(x*x + y*y);
```

The declaration of a ***friend function*** is done by specifying the keyword `friend` before the function declaration.

For adding new `friend` function to a certain class, the respective class declaration must be modified. It results that the declaration of the `friend` functions must be realized in the design stage of the developing cycle of a software application.

Usually the *friend* functions of a class could be also defined as member functions of the respective class, case when they can have direct access to the class members. For example, the *Dist* function can be defined as member function to the class *Point*:

```
class Point {
protected:
  double x, y;
public:
  Point(int a, int b) : x(a), y(b) { }
  Double Dist(Point& b) {
    double d = sqrt((x-b.x)*(x-b.x)+(y-b.y)*(y-b.y);
    return d;
  }
};

void Processing() {
  Point p1(1, 1), p2(2, 4);
  double d1 = p1.Dist(p2);
  double d2 = p2.Dist(p1); //the same thing
  // ...
}
```

When it is used the member function variant, such a function can be called as a method of an object of the respective class. The difference between the use of *Dist* function as member function and as *friend* function in this example is a kind of asymmetry in the case of member function. In general, when a certain function is regarded as an implementation of a certain operation with two operands (a binary operator), the method of *friend* function can be more natural, because the method of member function needs only one explicit parameter (the other operand is the hidden pointer this to the current object).

There are situations when is desired as a *certain member function of a class* to be friend of another class.

Example 8.1. The class c*ontorA* contains a pointer to an object of a class *A* and a function which counts the reference number to this object.

```
class A;

class contorA {
  A *a;
public:
  contorA(int k = 0);
  int increment();
};

class A {
  int n;
public:
  A(int k = 0) { n = k; }
friend int contorA::increment();
};

int contorA::increment() { return a->n++; }
contorA::contorA(int k) { a = new A; }

int main() {
  contorA c;
  cout << c.increment() << endl;
  cout << c.increment() << endl;
  return 0;
}
```

Observation. The *increment* and constructor functions of the class *contorA* have not been defined inline, because *A* has not been completely defined.

The *increment* function from *contorA* class must increment the private member *n* of the *a* object, which belongs to the class *A*. From this reason the function has been declared as *friend* function in the *A* class. The function was prefixed with the class name to which it belongs.

If it desired that *more member functions of a certain class A* to have access to the private members of a class *B*, the whole class *A* can be declare as a ***friend class*** of the *B* class. In this case the declaration contains only the `friend` keyword, followed by the class declaration which is friend.

From previous example, the class *contorA* can be declared as a ***friend class*** of the *A* class.

```
class A;

class contorA {
  A *a;
public:
  contorA(int k = 0);
  int increment();
};

class A {
  int n;
public:
friend class contorA;
  A(int k = 0) { n = k; }
};
```

Remark. The relation of friendship it is not a biunivocal relation, in the way that the declaration:

```
friend class contorA;
```

does not mean that the members of the class *A* have access to the private members of the class *contorA*.

An intuitive example of using *friend* classes is the rewriting the *node* and *list* classes from the definition of a simple linear linked list. Because the functions from the *list* class access the private member *next* of *node* class, the *list* class can be declared a friend class of *node*.

```
class list;

class node {
friend class list;
  int val;
  node* next;
public:
  node(int v, node* p = 0) { val = v; next = p; }
  ~node() { next =0; }
  void Add(int v) {
    node* q = new node(v);
    next = q;
  }
  int Val() const { return val; }
  void Print() const { cout << val << endl; }
};
class list {
  node* first;
  void Delete();
  void Copy(node* p);
public:
  list() { first = 0; }
  list(list& l) { first = 0; Copy(l.first); }
  ~list() { Delete(); first = 0; }
  //adds an element at the end of the list
  void AddLast(int v);
  void Print() const {
    for (node* p=first; p; p=p->next)
      p->Print();
  }
  int VidList() const { return first == 0; }
};

void list::AddLast(int v) {
  if (!first)
    first = new node(v);
  else {
    for (node* q=first; q->next; q=q->next);
    q->Add(v);
  }
}

void list::Copy(node* p) {
  first = 0;
```

```
    for (node* q=p; q; q=q->next)
      AddLast(q->val);
  }

  void list::Delete() {
    node *p = first, *q;
    while (p) {
      q = p;
      p = p->next;
      delete q;
    }
  }

  int main() {
    list l;
    l.AddLast(7);
    l.AddLast(5);
    l.AddLast(9);
    l.Print();
    return 0;
  }
```

Remark. There is a distinction between a class friend to another class and a derived class from another class. For example:

```
  class D1 {
    // ...
  };

  class B1 {
  friend class D1;
    // ...
  };

  class B {
    // ...
  };

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

  void Processing() {
    D1 d1;
    D d;
    // ...
  }
```

The object *d* of the class *D* contains as members all the members of the class *B*, at which are added the supplementary members owned by class *D*. The object *d1* of the class *D1* contains (unlike the *d* object), only the members owned by class *D1*, not the ones of class *B1* (the declaration `friend class D1;`). So, the member functions of the class *D1* can access private members of the class *B1* only by using the objects of *B1*.

8.2 Nested classes

The presence of several *friend* classes or functions in a class hierarchy denotes an inefficient design of the hierarchy. In these cases is wanted a hierarchy design which minimizes the appearance of *friend* functions and classes: the redefinition of *friend* functions as member functions, and the redefinition of *friend* classes as ***nested classes.***

This observation is justified because the *friend* functions and classes do not represent a specific characteristic of a pure object-oriented language, but only a compromise for the pragmatism of developing applications.

Example 8.3. The class *list* can be defined in a pure object-oriented style as follows (the implementations of the functions from the class *list* are identical as in the previous example):

```
class list {
  struct node {
    int val;
    node* next;
    node(int v, node*p = 0): val(v), next(p) { }
    ~node() { next = 0; }
    void Add(int v) {
      node* q = new node(v);
      next = q;
    }
    void Print() const { cout << val << endl; }
  };
  node *first;
  void Delete();
  void Copy(node* p);
public:
  list() { first = 0; }
  list(list& l) { first = 0; Copy(l.first); }
  ~list() { Delete(); first = 0; }
  void Add(int v);    //adds an elem. At the end of the //list
  void Print() const {
    for (node* p=first; p; p=p->next)
      p->Print();
  }
  int VidList() const { return first == 0; }
};
```

The class *node* from the previous example is defined inside the *list* class, in its *private* section.

A ***nested class*** defined inside another class can be considered as a member definition of the respective class and can be defined in any part of the respective class: in the private, protected or public section. Its accessibility depends of the class section in which it was defined:

- a class defined in the `public` section of a class is visible outside the respective class;
- a class defined in the `protected` section is visible only in the classes derived from the respective class;
- a class defined in the `private` section is visible only inside the class to which it belongs.

The accessibility of members of a *nested* class respects the general rules of accessibility of the class members. In conclusion, the private members from a class *B*, nested in a class *A*, cannot be accessed in the class *A*, regardless of the section where the class *B* has been defined. In the case when it is desired as the whole class *A* to have access to the private members of the class *B*, or the class *B* to have access at the private members of the class *A*, the classes can be defined as *friend*. The declaration of *friend* class for a *nested* class can precede or succeed the class definition.

Example 8.4. The classes *B* and *C* are defined inside the class *A*, so each of them have access to the private members of the other classes.

```
class A {
  int n;

  class B {
  friend class A;
    int k;
  public:
    B(int n = 0): k(n) { }
    int K() const { return k; }
  };
  friend class B;

public:
  class C {
  friend class A;
    int l;
  public:
    C(int n = 0): l(n) { }
    int L() const { return l; }
  };
  friend class C;

  A(int a): n(a) { }
  int N() const { return n; }

  // ...
};
```

The accessibility of a *nested* class concerns only the class name regarded as a data type and not its members. The access to the members of the nested class can be realized by an instance object, as in the case of the *friend* class.

In the previous example of the *list* class, the *node* class is not directly accessed by the member functions of the *list* class; for this it is used a data member, *first*, which is a pointer to the *node* class.

The implementation of functions (that are not *inline*) of a nested class can be made outside the class where it is defined as nested, by using the resolution operator.

For example, in the case when the *Add* function from the *node* class is be defined as follows:

```
class list {
  struct node {
    int val;
    node* next;
    node(int v, node*p = 0): val(v), next(p) { }
    ~node() { next = 0; }
    void Add(int v);
    void Print() const { cout << val << endl; }
  };
  node *first;
  // ...
};
```

Then its implementation in the outside of the *list* class can be:

```
void list::node::Add(int v) {
  node* q = new node(v);
  next = q;
}
```

In the case when a *nested* class has static data, their access can be realized also with the resolution operator.

Example 8.5. The example with classes *A*, *B* and *C* is used again, by using the static data and functions:

```
class A {
  int n;
  static int v;

  class B {
  friend class A;
    int k;
    static int v;
  public:
    B(int n = 0): k(n) { }
    int K() const { return k; }
    static void SetV(int n) {
      B::v = n;
    }
    static void SetAV(int n) {
      A::v = n;
    }
  };
  friend class B;

public:

  class C {
  friend class A;
    int l;
    static int v;
  public:
    C(int n = 0): l(n) { }
    int L() const { return l; }
    static void SetV(int n) {
      C::v = n;
    }
    static void SetAV(int n) {
      A::v = n;
    }
  };
  friend class C;

  A(int a): n(a) { }
  int N() const { return n; }
  static void SetV(int a) { v = a; }

  // ...
};

int A::v = 0;
int A::B::v = 0;
int A::C::v = 0;

int main() {
  A::C::SetV(1);
  A::C::SetAV(3);
  A::SetV(0);
  // ...
}
```

In the case when a class has several *nested* classes, the definition order is, in general, not restricted. The exceptions are those *nested* classes which are dependent one to each other, case when they must be declared before the definition.

The *enumerations*, even they do not represent classes, are data types and they can be defined inside other classes. The enumeration name and also the elements values can be used in classes inside which they have been defined, and in the case when they are defined in a public section, they can be used also outside the classes.

Example 8.6. The class *Clock* class allows the display of the current hour of a clock for different predefined time zones: *LondonHour*, *ParisHour*, *BucharestHour*, *MoskowHour*. A clock is considered to be fixed at the *Bucharest* hour.

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

class clock {
  int hour, min, sec;
public:
  enum HourDisplay {
    LondonHour,
    ParisHour,
    BucharestHour,
    MoskowHour
  };
  clock(int o = 0, int m = 0, int s = 0):
      hour(o), min(m), sec(s) { }
  void DisplayHour(HourDisplay h) {
    cout << "hour " << hour + h - BucharestHour;
    cout << ": min " << min;
    cout << ": sec " << sec << endl;
  }
};

int main() {
  clock c(14, 20, 50);
  c.DisplayHour(clock::BucharestHour);
  c.DisplayHour(clock::ParisHour);
  c.DisplayHour(clock::LondonHour);
  c.DisplayHour(clock::MoskowHour);
  return 0;
}
```