LECTURE-18

INHERITANCE.

18.1 FRIEND CLASSES

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that second class access to the protected and private members of the first one.

Program 18.1: Friend's classes

```
// friend class
#include <iostream>
using namespace std;

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (CSquare a);
};

class CSquare {
  private:
    int side;
  public:
    void set_side (int a)
      {side=a;}
    friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
      }
```

```
/* OUTPUT
  16
*/
```

In this example, we have declared CRectangle as a friend of CSquare so that CRectangle member functions could have access to the protected and private members of CSquare, more concretely to CSquare::side, which describes the side width of the square.

You may also see something new at the beginning of the program: an empty declaration of class CSquare. This is necessary because within the declaration of CRectangle we refer to CSquare (as a parameter in convert()). The definition of CSquare is included later, so if we did not include a previous empty declaration for CSquare this class would not be visible from within the definition of CRectangle.

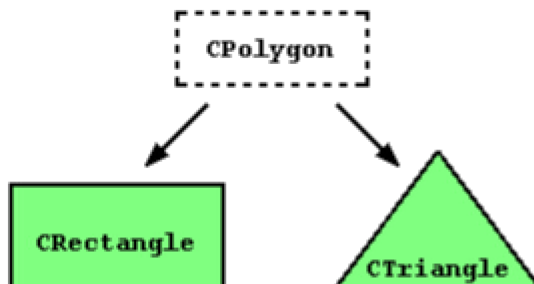
Consider that friendships are not corresponded if we do not explicitly specify so. In our example, CRectangle is considered as a friend class by CSquare, but CRectangle does not consider CSquare to be a friend, so CRectangle can access the protected and private members of CSquare but not the reverse way. Of course, we could have declared also CSquare as friend of CRectangle if we wanted to.

Another property of friendships is that they are *not transitive*: The friend of a friend is not considered to be a friend unless explicitly specified.

18.2 INHERITANCE BETWEEN CLASSES

A key feature of C++ classes is *inheritance*. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own. For example, we are going to suppose that we want to declare a series of classes that describe polygons like our CRectangle, or like CTriangle. They have certain common properties, such as both can be described by means of only two sides: height and base.

This could be represented in the world of classes with a class CPolygon from which we would derive the two other ones: CRectangle and CTriangle.



The class CPolygon would contain members that are common for both types of polygon. In our case: width and height. And CRectangle and CTriangle would be its derived classes, with specific features that are different from one type of polygon to the other.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive it to another class with another member called B, the derived class will contain both members A and B.

In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers protected and private. This access specifier describes the minimum access level for the members that are inherited from the base class.

Program 18.2 : Example of Derived classes

```
// derived classes
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
```

}

/* 20 10*/

The objects of the classes CRectangle and CTriangle each contain members inherited from CPolygon. These are : width, height and set_values().

Program 18.3 Single inheritance

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

class basic_info{
private:
	char name[20];
	long int rollno;
	char sex;
public:
	void getdata();
	void display();
};//end of main class

class physical_fit : public basic_info
{
private:
	float height;
	float weight;
public:
	void getdata();
	void display();
};//end of derived class

void basic_info :: getdata()
{
	cout<<" Enter a name:\n";
	cin>> name;
	cout<<" ERoll No:\n";
	cin>> rollno;
	cout<<" sex:\n";
	cin>> sex;
}

void basic_info :: display()
{
	cout<< name <<"      ";
	cout<<rollno<<"      ";
	cin>> rollno;
	cout<< sex <<"     ";
	}
void physical_fit :: getdata()
{
	basic_info :: getdata();
	cout<<" Height?:\n";
	cin>> height;
```

```
        cout<<" weight?:\n";
        cin>> weight;
        cout<<" sex:\n";
        cin>> sex;
}

void physical_fit :: display()
{
        basic_info :: display();
        cout<<setprecision(2);
        cout<< height <<"     ";
        cout<<weight<<"     ";
//      cout<<sex<<"     ";  cannot eccess sex being a private member
        }

void main()
{
        physical_fit a;
        cout <<"Enter the following information:\n";
        a.getdata();
        cout <<"_______________________________________________\n";
        cout<< "   Name      Roll No         Sex       ht         Wt   \n";
       cout <<"_______________________________________________\n";
               a.display();
         cout <<endl;
         cout <<"_______________________________________________\n";

        }
```

The protected access specifier is similar to private. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the **protected members inherited from the base class, but not its private members.**

Since we wanted width and height to be accessible by the members of the derived classes CRectangle and CTriangle and not only by members of CPolygon, we have used **protected** access instead of private.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

where "not members" represent any access from outside the class, such as from main(), from another class or from a function.

In our example, the members inherited by CRectangle and CTriangle have the same access permissions as they had in their base class CPolygon:

```
CPolygon::width              // protected access
CRectangle::width            // protected access

CPolygon::set_values()       // public access
CRectangle::set_values()     // public access
```

This is because we have used the public keyword to define the inheritance relationship on each of the derived classes:

```
class CRectangle: public CPolygon { ... }
```

This public keyword after the colon (:) denotes the ***minimum access level*** for all the members inherited from the class that follows it (in this case CPolygon). Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

If we specify a more restrictive access level like protected, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: private, all the base class members are inherited as private.

For example, if daughter was a class derived from mother that we defined as:

```
class daughter: protected mother;
```

This would set protected as the maximum access level for the members of daughter that it inherited from mother. That is, all members that were public in mother would become protected in daughter. Of course, this would not restrict daughter to declare its own public members. That maximum access level is only set for the members inherited from mother.

If we do not explicitly specify any access level for the inheritance, the compiler assumes private for classes declared with class keyword and public for those declared with struct.

18.3 What is inherited from the base class?

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

```
derived_constructor_name (parameters) : base_constructor_name
(parameters) {...}
```

For example:

Program 18.4: `// constructors and derived classes`

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

class mother {
  public:
    mother ()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};

class daughter : public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
  public:
    son (int a) : mother (a)
      { cout << "son: int parameter\n\n"; }
};

int main () {
  daughter cynthia (0);
  son daniel(0);

  return 0;

}
```

/* OUTPUT

```
mother: no parameters
daughter: int parameter
```

```
mother: int parameter
son: int parameter  */
```

Notice the difference between which mother's constructor is called when a new daughter object is created and which when it is a son object. The difference is because the constructor declaration of daughter and son:

```
daughter (int a)            // nothing specified: call default
son (int a) : mother (a)    // constructor specified: call this
```

18.4 MULTIPLE INHERITANCE

In C++ it is perfectly possible that a class inherits members from more than one class. This is done by simply separating the different base classes with commas in the derived class declaration. For example, if we had a specific class to print on screen (COutput) and we wanted our classes CRectangle and CTriangle to also inherit its members in addition to those of CPolygon we could write:

```
class CRectangle: public CPolygon, public COutput;
class CTriangle: public CPolygon, public COutput;
```

here is the complete example :

Program 18.5 : Multiple inheritance

```
// multiple inheritance
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class COutput {
  public:
    void output (int i);
  };

void COutput::output (int i) {
  cout << i << endl;
  }

class CRectangle: public CPolygon, public COutput {
  public:
    int area ()
```

```
      { return (width * height); }
  };

class CTriangle: public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  rect.output (rect.area());
  trgl.output (trgl.area());
  return 0;

}
```