Friendship and inheritance

Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect *friends*.

Friends are functions or classes declared with the `friend` keyword.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword `friend`:

```
1  // friend functions
2  #include <iostream>
3  using namespace std;
4
5  class CRectangle {
6      int width, height;
7    public:
8      void set_values (int, int);
9      int area () {return (width * height);}
10     friend CRectangle duplicate (CRectangle);
11 };
12
13 void CRectangle::set_values (int a, int b) {
14   width = a;
15   height = b;
16 }
17
18 CRectangle duplicate (CRectangle rectparam)
19 {
20   CRectangle rectres;
21   rectres.width = rectparam.width*2;
22   rectres.height = rectparam.height*2;
23   return (rectres);
24 }
25
26 int main () {
27   CRectangle rect, rectb;
28   rect.set_values (2,3);
29   rectb = duplicate (rect);
30   cout << rectb.area();
31   return 0;
32 }
```

```
24
```

The `duplicate` function is a friend of `CRectangle`. From within that function we have been able to access the members `width` and `height` of different objects of type `CRectangle`, which are private members. Notice that neither in the declaration of `duplicate()` nor in its later use in `main()` have we considered `duplicate` a member of class `CRectangle`. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate `duplicate()` within the class `CRectangle`.

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 first class access to the protected and private members of the second one.

```
1  // friend class
2  #include <iostream>
3  using namespace std;
4
```

```
16
```

```
5  class CSquare;
6
7  class CRectangle {
8      int width, height;
9    public:
10     int area ()
11       {return (width * height);}
12     void convert (CSquare a);
13 };
14
15 class CSquare {
16   private:
17     int side;
18   public:
19     void set_side (int a)
20       {side=a;}
21     friend class CRectangle;
22 };
23
24 void CRectangle::convert (CSquare a) {
25   width = a.side;
26   height = a.side;
27 }
28
29 int main () {
30   CSquare sqr;
31   CRectangle rect;
32   sqr.set_side(4);
33   rect.convert(sqr);
34   cout << rect.area();
35   return 0;
36 }
```

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`.

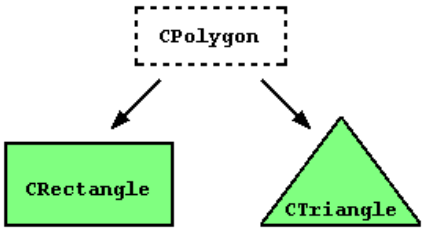
Consider that friendships are not corresponded if we do not explicitly specify so. In our example, `CRectangle` is considered as a friend class by `CSquare`, but `CRectangle` does not consider `CSquare` to be a friend, so `CRectangle` can access the protected and private members of `CSquare` but not 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.

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 limits the most accessible level for the members inherited from the base class: The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

```
1  // derived classes
2  #include <iostream>
3  using namespace std;
4
5  class CPolygon {
6    protected:
7      int width, height;
8    public:
9      void set_values (int a, int b)
10       { width=a; height=b;}
11   };
12
13 class CRectangle: public CPolygon {
14   public:
15     int area ()
16       { return (width * height); }
17   };
18
19 class CTriangle: public CPolygon {
20   public:
21     int area ()
22       { return (width * height / 2); }
23   };
24
25 int main () {
26   CRectangle rect;
27   CTriangle trgl;
28   rect.set_values (4,5);
29   trgl.set_values (4,5);
30   cout << rect.area() << endl;
31   cout << trgl.area() << endl;
32   return 0;
33 }
```

```
20
10
```

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

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 from 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`:

```
1  CPolygon::width           // protected access
2  CRectangle::width         // protected access
3
4  CPolygon::set_values()    // public access
5  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 most accessible level the members inherited from the class that follows it (in this case `CPolygon`) will have. 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`.

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:

```
// 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;
```

```
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
```

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:

```
// 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;
}
```

```
20
10
```

Previous: **Classes (II)** Index Next: **Polymorphism**