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 apply to *"friends"*.

原则上，类的private和protected成员不能从声明它们的类的外部访问。然而，这条规则不适用于“朋友”。

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

友元是用friend关键字声明的函数或类。

A non-member function can access the private and protected members of a class if it is declared a *friend* of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword `friend`:

非成员函数可以访问类的private和protected成员，如果它被声明为该类的友元。这是通过在类中包含这个外部函数的声明，并在其前面加上关键字friend来实现的:

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

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}
```

```
int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

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

duplicate函数是Rectangle类的友元。因此，函数复制能够访问不同类型矩形对象的成员宽度和高度(它们是私有的)。但是请注意，无论是在duplicate的声明中，还是在以后在main中使用它时，函数duplicate都被认为是Rectangle类的成员。它不是!它只是访问其私有和受保护的成员，而不是成员。

Typical use cases of friend functions are operations that are conducted between two different classes accessing private or protected members of both.

友元函数的典型用例是在访问私有或受保护成员的两个不同类之间执行的操作。

Friend classes

Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class:

与友元函数类似，友元类是其成员可以访问另一个类的private或protected成员的类:

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

class Square;

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

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
```

```
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

In this example, class `Rectangle` is a friend of class `Square` allowing `Rectangle`'s member functions to access private and protected members of `Square`. More concretely, `Rectangle` accesses the member variable `Square::side`, which describes the side of the square.

在本例中，类Rectangle是类Square的友元，允许矩形的成员函数访问Square的私有和受保护的成员。更具体地说，Rectangle访问成员变量Square::side，它描述了正方形的边长。

There is something else new in this example: at the beginning of the program, there is an empty declaration of class `Square`. This is necessary because class `Rectangle` uses `Square` (as a parameter in member `convert`), and `Square` uses `Rectangle` (declaring it a friend).

这个示例中还有一些新内容:在程序的开头，类Square的声明是空的。这是必要的，因为类Rectangle使用Square(作为成员转换中的参数)，而Square使用Rectangle(将它声明为友元)。

Friendships are never corresponded unless specified: In our example, `Rectangle` is considered a friend class by `Square`, but Square is not considered a friend by `Rectangle`. Therefore, the member functions of `Rectangle` can access the protected and private members of `Square` but not the other way around. Of course, `Square` could also be declared friend of `Rectangle`, if needed, granting such an access.

在我们的例子中，Rectangle被Square认为是一个朋友类，但是Square不被Rectangle认为是一个朋友。因此，Rectangle的成员函数可以访问Square的受保护成员和私有成员，而不能访问Square的受保护成员和私有成员。当然，如果需要，Square也可以声明为Rectangle的友元，并授予这样的访问权限。

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

友谊的另一个特性是它们是不能传递的:朋友的朋友不会被认为是朋友，除非有明确的说明。

Inheritance between classes

Classes in C++ can be extended, creating new classes which retain characteristics of the base class. This process, known as inheritance, involves a *base class* and a *derived class*: The *derived class* inherits the members of the *base class*, on top of which it can add its own members.

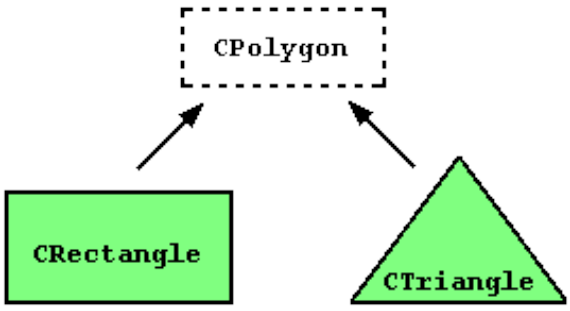
c++中的类可以扩展，创建保留基类特征的新类。这个过程称为继承，涉及基类和派生类:派生类继承基类的成员，在基类之上添加自己的成员。

For example, let's imagine a series of classes to describe two kinds of polygons: rectangles and triangles. These two polygons have certain common properties, such as the values needed to calculate their areas: they both can be described simply with a height and a width (or base).

例如，让我们想象一系列类来描述两种多边形:矩形和三角形。这两个多边形具有某些共同属性，例如计算它们的面积所需的值:它们都可以简单地用高度和宽度(或基础)来描述。

This could be represented in the world of classes with a class `Polygon` from which we would derive the two other ones: `Rectangle` and `Triangle`:

这可以用一个类Polygon来表示，我们可以从中派生出另外两个类:Rectangle和Triangle:



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

Polygon类将包含两种类型的多边形都通用的成员。在我们的例子中:宽度和高度。矩形和三角形是它的派生类，具有不同于一种多边形的特定特征。

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 a class from it with another member called `B`, the derived class will contain both member `A` and member `B`.

派生自其他类的类继承基类的所有可访问成员。这意味着，如果基类包含成员a，而我们派生了一个名为B的成员，派生类将同时包含成员a和成员B。

The inheritance relationship of two classes is declared in the derived class. Derived classes definitions use the following syntax:

两个类的继承关系在派生类中声明。派生类定义使用以下语法:

```
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` or `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.

其中derived_class_name是派生类的名称，base_class_name是它所基于的类的名称。公共访问说明符可以被任何其他访问说明符(受保护的或私有的)所替代。此访问说明符限制从基类继承的成员的最易访问级别:具有更易访问级别的成员将继承此级别，而具有相同或限制性更强访问级别的成员将其限制性级别保留在派生类中。

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

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

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

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

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

The objects of the classes `Rectangle` and `Triangle` each contain members inherited from `Polygon`. These are: `width`, `height` and `set_values`.

Rectangle和Triangle类的对象都包含继承自Polygon的成员。它们是:width, height和set_values。

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

Polygon类中使用的受保护访问说明符类似于private。事实上，它唯一的区别在于继承:当一个类继承另一个类时，派生类的成员可以访问从基类继承的受保护成员，但不能访问它的私有成员。

By declaring `width` and `height` as `protected` instead of `private`, these members are also accessible from the derived classes `Rectangle` and `Triangle`, instead of just from members of `Polygon`. If they were public, they could be accessed just from anywhere.

通过将width和height声明为protected而不是private，这些成员也可以从派生类Rectangle和Triangle中访问，而不仅仅是从Polygon的成员中。如果它们是公开的，就可以从任何地方访问它们。

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

我们可以根据函数的访问方式来总结不同的访问类型:

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

Where "not members" represents any access from outside the class, such as from `main`, from another class or from a function.

其中“非成员”表示来自类外部的任何访问，例如来自main、来自另一个类或来自函数。

In the example above, the members inherited by `Rectangle` and `Triangle` have the same access permissions as they had in their base class `Polygon`:

在上面的例子中，Rectangle和Triangle继承的成员拥有与它们在基类Polygon中拥有的相同的访问权限:

```
Polygon::width           // protected access
Rectangle::width         // protected access

Polygon::set_values()    // public access
Rectangle::set_values()  // public access
```

This is because the inheritance relation has been declared using the `public` keyword on each of the derived classes:

这是因为继承关系是在每个派生类上使用public关键字声明的:

```
class Rectangle: public Polygon { /* ... */ }
```

This `public` keyword after the colon (`:`) denotes the most accessible level the members inherited from the class that follows it (in this case `Polygon`) will have from the derived class (in this case `Rectangle`). 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.

冒号(:)后面的public关键字表示从其后的类(在本例中为Polygon)继承的成员从派生类(在本例中为Rectangle)继承的最易访问的级别。由于public是最容易访问的级别，通过指定这个关键字，派生类将继承基类中具有相同级别的所有成员。

With `protected`, all public members of the base class are inherited as `protected` in the derived class. Conversely, if the most restricting access level is specified (`private`), all the base class members are inherited as `private`.

使用protected，基类的所有公共成员都在派生类中作为protected对象继承。相反，如果指定了最严格的访问级别(private)，则所有基类成员都继承为private。

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

例如，如果daughter是派生自mother的类，则定义为:

```
class Daughter: protected Mother;
```

This would set `protected` as the less restrictive 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` from declaring its own public members. That *less restrictive access level* is only set for the members inherited from `Mother`.

这将为它从母体继承的Daughter的成员设置受保护的访问级别。也就是说，所有在《母亲》中公开的成员都将在《女儿》中受到保护。当然，这并不会限制“女儿”宣布自己的公众成员。这种限制较少的访问级别只设置为从母亲继承的成员。

If no access level is specified for the inheritance, the compiler assumes private for classes declared with keyword `class` and public for those declared with `struct`.

如果没有为继承指定访问级别，编译器将假定用关键字class声明的类为private，用struct声明的类为public。

Actually, most use cases of inheritance in C++ should use public inheritance. When other access levels are needed for base classes, they can usually be better represented as member variables instead.

实际上，c++中的大多数继承用例都应该使用公共继承。当基类需要其他访问级别时，通常可以用成员变量来表示它们。

What is inherited from the base class?

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

原则上，公共派生类继承了对基类每个成员的访问权，除了:

- its constructors and its destructor

  它的构造函数和析构函数
- its assignment operator members (operator=)

  它的赋值操作符成员(operator=)
- its friends

  它的朋友
- its private members

  其私人成员

Even though access to the constructors and destructor of the base class is not inherited as such, they are automatically called by the constructors and destructor of the derived class.

即使基类的构造函数和析构函数的访问不是这样继承的，派生类的构造函数和析构函数会自动调用它们。

Unless otherwise specified, the constructors of a derived class calls the default constructor of its base classes (i.e., the constructor taking no arguments). Calling a different constructor of a base class is possible, using the same syntax used to initialize member variables in the initialization list:

除非另有说明，派生类的构造函数调用基类的默认构造函数(即不带参数的构造函数)。调用基类的不同构造函数是可能的，使用初始化列表中的成员变量的相同语法:

`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 kelly(0);
  Son bud(0);

  return 0;
}
```

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 due to the different constructor declarations of `Daughter` and `Son`:

注意在创建新的子对象时调用哪个Mother的构造函数，以及在它是子对象时调用哪个Mother的构造函数之间的区别。区别在于子和子的构造函数声明不同:

```
Daughter (int a)          // nothing specified: call default constructor
Son (int a) : Mother (a)  // constructor specified: call this specific constructor
```

Multiple inheritance

A class may inherit from more than one class by simply specifying more base classes, separated by commas, in the list of a class's base classes (i.e., after the colon). For example, if the program had a specific class to print on screen called `Output`, and we wanted our classes `Rectangle` and `Triangle` to also inherit its members in addition to those of `Polygon` we could write:

一个类可以从多个类继承，只要在类的基类列表中指定多个基类，并用逗号分隔(例如，冒号后面)。例如，如果程序有一个名为Output的特定类要在屏幕上打印，而我们希望我们的类Rectangle和Triangle除了继承Polygon的成员外还继承它们的成员，我们可以这样写:

```
class Rectangle: public Polygon, public Output;
class Triangle: public Polygon, public Output;
```

Here is the complete example:

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

class Polygon {
  protected:
    int width, height;
  public:
```

```
    Polygon (int a, int b) : width(a), height(b) {}
};

class Output {
  public:
    static void print (int i);
};

void Output::print (int i) {
  cout << i << '\n';
}

class Rectangle: public Polygon, public Output {
  public:
    Rectangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height; }
};

class Triangle: public Polygon, public Output {
  public:
    Triangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height/2; }
};

int main () {
  Rectangle rect (4,5);
  Triangle trgl (4,5);
  rect.print (rect.area());
  Triangle::print (trgl.area());
  return 0;
}
```