Modern C++ - Inheritance

20. Inheritance

In this chapter we discuss the inheritance in C++. We suppose that the reader knows the theory of inheritance, so in this chapter we concentrate on the implementation of inhertitance.

Generalisation and specialization

Let suppose that we want to implement a software for British MOT-like test – an annual test of vehicle safety, roadworthiness aspects (see wikipedia). Naturally, our example is just a simplification of the real tests and serves only demonstrational purposes.

The *vehicles* to test are *cars*, *buses*, and *trucks* and each kind of vehicle requires different test details.

We start to collect the features of the different vehicles. We recognise soon that there are common attributes of all vehicles: _ plate_, *horsepower* and *owner*. However, there are also feautures typical only for a specific type of vehicle: CO *emission* for cars, *length* and number of max passanger *persons* for busses, and total *weight* and number of *axles* for trucks. The problem is that some of the attributes valid only for some kind of objects, e.g. axles are not interesting for buses.

Let start it. Of course, we program in object-oriented way, so we create a separate class for all kind of vehicles. There will be a constructor and the mot() function for the exam itself. This is the first attempt:

```
 1 class Owner { /* .. */ };
 2 class Car
 3 {
 4 public:
 5   Car( std::string pl, int hpw, Owner own, double em ); // constructor
 6   bool mot() const;         // the exam itself
 7   std::string plate() const { _plate; }
 8   // Further Car interface ...
 9 private:
10   std::string _plate;  // licence plate
11   int         _hpw;    // hosepower
12   Owner       _owner;  // the owner
```

```
13   double      _emis;   // CO emission
14 };
```

```
 1 class Bus
 2 {
 3 public:
 4   Bus( std::string pl, int hpw, Owner own, int pers); // constructor
 5   bool mot() const;         // the exam itself
 6   std::string plate() const { _plate; }
 7   // Further Bus interface ...
 8 private:
 9   std::string _plate;   // licence plate
10   int         _hpw;     // hosepower
11   Owner       _owner;   // the owner
12   int         _persons; // max #persons to carry
13 };
```

```
 1 class Truck
 2 {
 3 public:
 4   Truck( std::string pl, int hpw, Owner own, int w, int axl); //
constructor
 5   bool mot() const;         // the exam itself
 6   std::string plate() const { _plate; }
 7   // Further Truck interface ...
 8 private:
 9   std::string _plate;  // licence plate
10   int         _hpw;    // hosepower
11   Owner       _owner;  // the owner
12   int         _weight; // total weight (kg)
13   int         _axl;    // axles
14 };
```

The problem is that the common attributes for all vehicles (licence plate, horsepower, owner) now are distributed into three classes. This is not good when we are thinking about software maintenance. In case of any change, we have to modify (consistently!) the code at minimum three different places. An other probem, that since cars, busses, trucks belong to different classes we are in trouble when we want to implement common functions or want to store them in a common container.

The other way is to create a common class for all vehicles. There will be all attributes for all kind of vehicles here and the overloaded constructors are responsible to set the

attributes *in use* for the actual kind of vehicle.

```
 1 class Vehicles
 2 {
 3 public:
 4   Vehicle( std::string pl, int hpw, Owner own, double emission);   // Car
 5   Vehicle( std::string pl, int hpw, Owner own, int persons);       // Bus
 6   Vehicle( std::string pl, int hpw, Owner own, int w, int axl); // Truck
 7   bool mot() const;           // the test itself
 8   // Further Vehicle interface ...
 9   std::string plate() const { _plate; }
10 private:
11   std::string _plate;  // licence plate
12   // Further Vehicle attributes ...
13 };
```

One problem is that attributes not used in a specific vehicle will be unitialized or have some extremal value (like null pointer or similar). This is a serious source of mistakes.

The major problem is, however, that for the MOT test some of the attributes are interesting only for some specific kind of objects, e.g. axles are not interesting for buses but for trucks only. So we can write something like this:

```
 1 // bad solution
 2 bool Vehicle::mot() const
 3 {
 4   if ( this object is a Car )
 5     // evaluate Car-specific tests
 6   else if ( this object is a Bus )
 7     // evaluate Bus-specific tests
 8   else if ( this object is a Truck )
 9     // evaluate Truck-specific tests
10 }
```

Although, there are techniques to identify the actual class of an object (we can use an *enumerator value* set in the constructor or the *typeid* operator), this solution is also a *maintenance nightmare*. Every time we add a new type of vehicle to the system, we have to find all the type selections in the code, modify and, and of course have to re-test everything.

Perhaps it would be better to mix the previous solutions: we create separate classes for cars, buses, trucks, but we still want to keep common attributes and methods in one

separate, well-defined class.

```
1 class Car
2 {
3 public:
4   // Interface for cars
5 private:
6   Vehicle _veh; // common attributes for all vehicles
7   // Car specific parts
8 };
9 class Truck
10 {
11 public:
12   // Interface for trucks
13 private:
14   Vehicle _veh; // common attributes for all vehicles
15   // Truck specific parts
16 };
17 // etc...
```

The problem is, that the Vehicle is not part of the Car/Truck interface, e.g. calling **c.plate()** on a **c Car** object is error, since the Vehicle functions does not exist automatically on Car, Bus, Truck classes. We should solve this manually:

```
1 class Car
2 {
3 public:
4   // ...
5   // Lots of boilerplate code for forwarding Vehicle interface to Car
6   std::string plate() { return _veh._plate; }
7 private:
8   Vehicle _veh; // common attributes for all vehicles
9   // Car specific parts
10 };
```

Inheritance

Inheritance provides the solution: while the *derived class* containes all attributes *inherited* from the *base class* (plus its own specific ones), the derived class is also a *subclass* of the base, i.e. all operations defined on the base is also available on the

derived class.

Liskov Substitutional Principle (LSP)

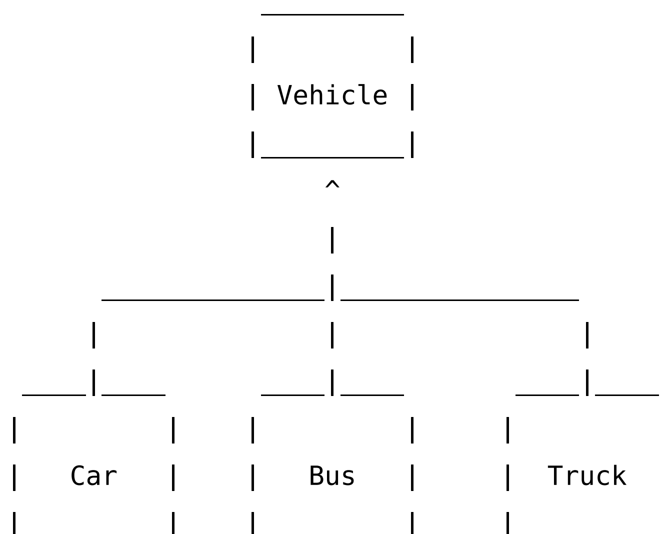
Barbara Liskov in a 1987 conference keynote address entitled *Data abstraction and hierarchy* formulated the *semantical* connection between a *subtype* and its *supertype* in the following way (The work was common with *Jeannette Wing*):

*If **S** is a subtype of **T**, then objects of type **T** may be replaced with objects of type **S** (i.e., objects of type **S** may substitute objects of type **T**) without altering any of the desirable properties of that program*.

Let aware, that the LSP formulates a *semantical* connection. Object-oriented languages try to implement this connection with various synatactical rules.

Inheritance hierarchy

Based on object oriented design, we will define **Vehicle** *base class* to collect the common properties of all vehicles, and separate *derived classes* for cars, busses and trucks.



One can extend this hierarchy with a new derived class any time without disturbing the already existing (and well tested) other components.

The base class is the **Vehicle** class. For simplicity, we now give only the header file.

```
1 // vehicle.h
2 #ifndef VEHICLE_H
3 #define VEHICLE_H
4
5 #include <string>
```

```
 6 #include "owner.h"  // for Owner type
 7
 8 class Vehicles
 9 {
10 public:
11   Vehicle( std::string pl, int hpw, Owner own);   // constructor
12   bool mot() const;                               // the test
13   std::string plate() const { _plate; }
14   // Further Vehicle interface ...
15 private:
16   std::string _plate;  // licence plate
17   int         _hpw;    // horsepower
18   Owner       _owner;  // owner of car
19   // Further Vehicle attributes ...
20 };
21 #endif // VEHICLE_H
```

Dervived classes are adding new attributes and *extending* the base class interface. However, the derived class constructor is responsible to provide the data for the constructor of the base class.

```
 1 #ifndef CAR_H
 2 #define CAR_H
 3
 4 #include <string>
 5 #include "vehicle.h"
 6
 7 class Car : public Vehicle
 8 {
 9 public:
10   Car( std::string pl, int hpw, Owner own, double emis)
11             : Vehicle( pl, hpw, own), _emis(emis) {}
12   double emission () const { return _emis; }
13   // the test for Car
14   bool mot() const { return Vehicle::mot() && _emis < 130; /* grams/km */
}
15 private:
16   double _emis; // gram/km CO2 emission
17 };
18 #endif /* CAR_H */
```

```
 1 // bus.h
 2 #ifndef BUS_H
 3 #define BUS_H
 4
 5 #include <string>
 6 #include "vehicle.h"
 7
 8 class Bus : public Vehicle
 9 {
10 public:
11   Bus( std::string pl, int hpw, Owner own, int len, int pers)
12             : Vehicle( pl, hpw, own), _length(len), _persons(pers) {}
13   int length() const { return _length; }
14   int person() const { return _persons; }
15   // the test for Bus
16   bool mot() const { return Vehicle::mot() && _persons/_length < 4; }
17 private:
18   int _length;  // total length
19   int _persons; // max persons carried
20 };
21 #endif /* BUS_H */
```

```
 1 // truck.h
 2 #ifndef TRUCK_H
 3 #define TRUCK_H
 4
 5 #include <string>
 6 #include "vehicle.h"
 7
 8 class Truck : public Vehicle
 9 {
10 public:
11   Truck( std::string pl, int hpw, Owner own, int tw, int al)
12             : Vehicle( pl, hpw, own), _totalWeight(tw), _axles(al) {}
13   int totalWeight() const { return _totalWeight; }
14   int axles() const { return _axles; }
15   // the test for Truck
16   bool mot() const { return Vehicle::mot() && _totalWeight/_axles < 5000.;
}
17 private:
```

```
18   int _totalWeight;  // kg
19   int _axles;        // number of
20 };
21 #endif /* TRUCK_H */
```

Inheritance in C++

While in most object-oriented programming languages there is only one kind of inheritance – the interface extension, e.g. *extend* in Java – in C++ there are three different inheritance.

1. Public

This is the *classical* inheritance. The derived class’s public interface *extends* the base class’s interface. This is similar to inheritance in Java. All public methods of the base class is callable on the derived class. Also, *upcast* works: i.e. the objects of the derived class are converted to objects of the base class.

2. Protected

The public interface of the base class is visible in all derived classes for the derived class methods only. I.e. the method of a derived class can see the base class publics, but the outer word can not. The protected inheritance is *transitive* in the way that *all* derived will see the base interface in any step of inheritance.

3. Private

The public part of the base class is visible inside the methods of the immediate derived class, but are not visible outside of that derived class (including further derivations). This is *inheritance for implementation*, used when we do not want to extend the base class interface, but need its members (phyically) or its methods. It is almost the same that we have the base class object as an attribute (but wecan override the base class virtual functions). A typical case for private inheritance is to forbid copying of the object before C++11 using private inheritance from **boost::noncopyable**.

The kind of inheritance is marked by the appropriate keyword after the colon denoting the inheritance. When the keyword is missing the inheritance is *public* for **struct** and *private* for **class**.

Naturally, most cases we use public inheritance to express subtyping. Private inheritance happens for much specific cases for implementational purposes. Protected inheritance is used in rare cases.

Constructor and destructor

In the derived classes we have to provide the parameters for the base class constructors:

```
1 Truck( std::string pl, int hpw, Owner own, int tw, int al)
2           : Vehicle( pl, hpw, own), _totalWeight(tw), _axles(al) {}
```

The construction order of subobjects is (in recursive way):

1. base classes (in order of declaration)
2. attrinutes (in order of declaration)
3. run of the constructor body

The destruction order is the reverse.

When a base class or any attributes of the derived class has a constructor or destructor, the compiler automatically creates a constructor ordestructor for the derived class which calls the same method of the base(s).

When the programmer defines a constructor or destructor for the derived class the compiler weaves the base class method call to the defined special member function.

Copying of the objects (when possible) is done by default copying the base with its own (perhaps user defined) copy operation. Otherwise, the programmer can provide a user defined copy constructor or assignment operator which should call the similar copy operation on the base.

```
 1 Derived( const Derived& rhs) : Base(rhs) // copy constructor
 2 {
 3   // copy of derived specific part
 4 }
 5 Derived &operator=( const Derived& rhs) // assignment
 6 {
 7   Base::operator=(rhs);
 8   // copy of derived specific part
 9   return *this;
10 }
```

Move operations (since C++11) work in similar way, but the programmer should aware of the *if it has a name* rule.