TP C++ - 2

Thierry Géraud, Edwin Carlinet

Contents

1 Reminders

1.1 `class` vs `struct`

In the C++ language, there is few differences between a *class* and a *structure*: attributes and methods default visibility. In a class, the default visibility is *private*, whereas it is *public* in a structure.

1.2 Visibility

Visibility is a *really* important notion. Even if you have to write more lines, you ensure that nobody will modify your class attributes without using the correct way that *you* specified (*getter*/*setter*, etc.). Visibility allows you to keep control over your objects.

For instance, you may have an attribute that takes strange values at runtime. If the only way to modify this attribute is to use a *setter* (because its visibility is *private*), it will be very easy to debug: the error obviously comes from the *setter* implementation (or from the class itself). On the contrary, if this attribute is *public*, the attribute value can be modified everywhere in the code. Thus, localizing the bug could be very difficult!

2 Class notions

2.1 Constructor and initialization list

Yesterday we saw how to initialize objects. Object initialization takes place in the *constructor*. While it may be OK to write:

```
struct MyClass
{
  MyClass(int x) {
    x_ = x;
  }
  private:
    int x_;
};
```

it may become tricky is want to initialize a member trough its constructor, or if we want to call the base constructor in case of inheritance. Consider for example:

```
struct MyObject
{
  MyObject(int x);
};

struct MyClass
{
  MyClass(int x) {
    obj_ = x; // Invalid because MyObject must be initialized by its constructor
  }
  private:
    MyObject obj_;
};
```

We have to rely *initialization lists*.

Let's see it in practice, declare a class *Person* with the following attributes, and try to initialize them:

```
class Person
{
  Person()
  {
    // FIXME
  }
  private:
    const std::string name_;
    unsigned int age_;
};
```

You should have noticed that a problem occurs in your constructor with the following line:

```
name_ = name;
```

We cant assign a value to a constant variable, so the compiler will reject this code. But we need to set a value the first time! This is where we use an initialization list. Our constructors could be rewritten as:

```
Person::Person(const std::string& name, const std::string& date_of_birth)
    : name_(name)
{
    age_ = ...; // generate age from date\_of\_birth
}

Person::Person(const std::string& name, unsigned int age)
    : name_(name)
    , age_(age)
{}
```

Contrary to the previous affectations, instructions in the initialization list are done *before* the object is created, and specify how fields have to be initialized. Moreover, this initialization list allows to clarify the constructor code and remove useless initialization lines from the constructors body.

We must initialize constant attributes and constant references this way, but, of course, we can use an initialization list for other attributes. It is the only way to call the attributes' specific constructors, every attribute which is not in this list is implicitly initialized with its default constructor (so if an attribute has no default constructor, you must use an initialization list too).

Now, a last comment about the *order* of the attributes in the initialization list. Try to compile:

```
// This is the .hh file.
class ClassA
{
public:
    ClassA();

private:
    int a_;
    int b_;
};

// This is the .cc file.
ClassA::ClassA()
    : a_(42)
    , b_(51)
{}
```

OK, everything should compile and run smoothly. Now try to compile the following code (only the attributes order in the initialization list has changed):

```
// This is the .cc file.
ClassA::ClassA()
```

```
    : b_(51)
    , a_(42)
{}
```

If you use the *must-have* flags (`-Wall -Wextra -Werror -std=c++11 -pedantic`), your compiler should warn you about the initialization order. It is important to follow the definition of your class, because initialization is done in this order – *not* in the initialization list order.

Using an attribute that is not already initialized in the initialization list would be very hazardous for primitive data types (int, char, etc.).

2.2 Destructors

The destructor can be defined just like any constructor. That means that it has neither return value nor return type. However, it cannot take any argument. Its role is to free resources allocated for the object and to execute tasks bound to the destruction of the object.

It is automatically called at the end of the scope (if the object was allocated in the stack) or manually with the `delete` keyword if the object was dynamically allocated.

When an object (say Obj) is destroyed, the destructors of each of its object attributes are called automatically whether a destructor (for Obj) is defined or not.

To define a destructor, its like the constructor except that you must put a c̃haracter before. See this example:

```
# include <string>

class StringKeeper
{
public:
    StringKeeper(const std::string& str);
    ~StringKeeper();

private:
    std::string* my_string_;
};
```

And now the implementation:

```
StringKeeper::StringKeeper(const std::string& str)
{
    my_string_ = new std::string(str);
}

StringKeeper::~StringKeeper()
{
    delete my_string_;
}
```

Remember: pointers are nothing but memory addresses. They have no destructor. You have to define your own destructors to explicitly delete any class attribute which has been allocated manually with `new`.

2.3 Dynamic allocation and destruction of objects

2.3.1 New

`new` allows dynamic memory allocation on the heap. It is not always pleasant to write complex blocks for a simple variable allocation of type `T`:

```
T* p = (T*) malloc(sizeof(T));
if (p == NULL)
    // handle error
\end{c++}

The \texttt{new} way simplifies things:

\begin{minted}{c++}
T* p1 = new T; // Call to constructor with no arguments
T* p2 = new T(arg1, ..., argn) // Call to constructor with arguments
T* p_array = new T[20]; // Allocate an array of 20 elements calling default constructor
```

In C++, `new` is a keyword for allocation. Its also an operator, but dont put the cart before the horse! In fact, `new` automatically computes the type size, calls a *malloc*-like function and returns the right pointer.

Examples:

```
double* dp = new double;
double* table = new double[DIM];
StringKeeper* sk = new StringKeeper("C++ Rocks!");
```

`new` handles errors by throwing *exceptions*.

2.3.2 Delete

The `delete` operator returns memory allocated by `new` back to the heap. It must be called for every call to 'new' to avoid memory leaks.

```
T* p1 = new T; // Call to constructor with no arguments
T* p2 = new T(arg1, ..., argn) // Call to constructor with arguments
T* p_array = new T[20]; // Allocate an array of 20 elements calling default constructor
delete p1;
delete p2;
delete[] p_array;
```

Examples:

```
StringKeeper* sk = new StringKeeper("Keep me safe");
// Do something here.
delete sk;

double* table = new double[DIM];
// Do something here
delete [] table;
```

In contrast to the `C realloc`, it is not possible to directly reallocate memory allocated with `new`. To extend or reduce the size of a block, one must allocate a new block of adequate size, copy over the old memory, and delete the old block.

2.4 Exercise

Adapt the `MyStack` class implemented yesterday with an initialization list for constructors and automatic memory release when the stack gets out of scope.

Be sure to use to correctly use initialization where needed. In particular, test your code with `node_content_type = int&`.

Exemple of use:

```
#include "mystack.hh"
#include <iostream>

int main()
{
  MyStack s;
  int x = 0, y = 1;

  s.push(x);
  s.push(y);

  std::cout << s << std::endl;
  x = 2;
  std::cout << s << std::endl;
}
```

```
$ ./a.out
[1 0]
[1 2]
$ valgrind ./a.out
==2855== Memcheck, a memory error detector
==2855== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==2855== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==2855== Command: ./a.out
==2855==
[1 0]
[1 2]
==2855==
==2855== HEAP SUMMARY:
==2855==     in use at exit: 0 bytes in 0 blocks
==2855==   total heap usage: 2 allocs, 2 frees, 32 bytes allocated
==2855==
==2855== All heap blocks were freed -- no leaks are possible
```

3 Inheritance notions

3.1 Simple inheritance

One of the most important and interesting aspect of the object paradigm is inheritance. It enables the modeling of the "is a" relationship between two classes. Here are some examples from daily life:

- a circle *is a* geometric shape and *concrete*,
- a polygon *is a* geometric shape but *abstract*, (a polygon only factorizes some properties of some concrete classes),
- rectangle, square, triangle *are* polygons.

If we had transposed these examples in object-oriented programming, we would get the following relations:

- the `Circle` class inherits from the `Shape` class
- the `Polygon` class inherits from the `Shape` class

- the `Triangle, Square, Rectangle` classes inherit from the Polygon class

To be sure of the correctness of an inheritance, lets just imagine you are a 10-year-old child and think about this: *a circle is a shape*, obviously. But a drawing isnt a shape: it *has* some shapes drawn inside, which is completely different. That may seem obvious with such a simple example, but remember to always check that your inheritance models the "is a" relationship, and nothing else. Sometimes, a confusion between "is a" and "has a" can invalidate your whole model! Be careful. Also, in order to the inheritance take place, the sub-class must extend the base class. For exemple, a square is not a rectangle (even if it is true mathematically) because it adds contraints (the width and height must be equal). One important characteristic of inheritance is its *transitivity*: a square *is a* polygon, a polygon *is a* shape, *so*, a square *is a* shape. We just created a class hierarchy, depicted below. Arrows represent the inheritance.

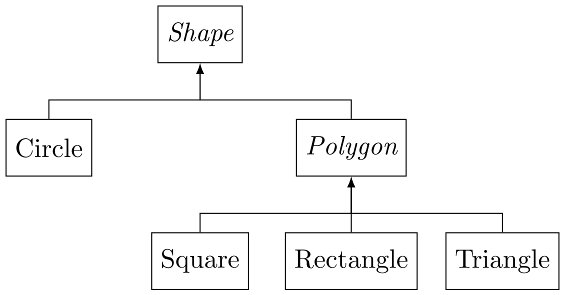

Figure 1: The "shape" hierarchy.

One of the main purposes of inheritance is *non-duplication* of code and data. Indeed, every shape has got a set of data which is independent from the shapes kind (for instance, center coordinates exist for circles, squares, etc.). For this reason, every field and method of the *base* class (here, `Shape`) belongs to the derived classes too (here, `Polygon`, `Square` and `Circle`).

Lets practice this with C++. The syntax to make a class inherit from another one uses the colon (':'), as in:

```
class Child: public Base
{
    // [...]
};
```

Yesterday, you saw the keywords *public* and *private*, which respectively define a field or method as visible from anybody or only from the class. A third visibility-related keyword exists: *protected*. Fields and methods defined as *protected* are visible from their class *and* its derived classes.

	Class	Subclasses	Others
public	✓	✓	✓
protected	✓	✓	×
private	✓	×	×

Figure 2: Quick summary about visibilaties

Example:
Lets define the Shape class as follows:

```
class Shape
{
public:
    Shape()
```

```
        : x_(0)
        , y_(0)
    {
        std::cout << "New shape created" << std::endl;
    }

    Shape(int x, int y)
        : x_(x)
        , y_(y)
    {
        std::cout << "New shape created (" << x_
                  << ", " << y_ << ")" << std::endl;
    }

    virtual ~Shape()
    {}

    void print() const
    {
        std::cout << "I am a shape located at ("
                  << x_ << ", " << y_ << ")" << std::endl;
    }

    void translate(int tx, int ty)
    {
        x_ += tx;
        y_ += ty;
    }

protected:
    int x_;
    int y_;
};
```

3.1.1 Exercise

Create a Circle class which:

- inherits from Shape
- gets a *radius_* field to hold the circles radius
- gets a default constructor which writes on the standard output the creation of a circle
- gets the following print method:

```
void print() const
{
    std::cout << "I am a circle located at (" << x_ << ", "
              << y_ << ") with a radius of " << radius_ << std::endl;
}
```

3.1.2 Call to parent constructor

To initialize an object, we call the constructor of the related class. However, this class can *inherit* from another class. Obviously, inserting the code of every parent constructor in each child class isn't an option!

Take back our Circle class and lets add some features:

```
class Circle
{
    // ...
    Circle(int x, int y, int radius);
    // FIXME
}
```

Two solutions remain:

- copy each fields initialization in the constructor that we just created
- call the Shape constructor that takes x and y as arguments, and initialize only the radius

Of course we prefer the second solution, so we get:

```
Circle::Circle(int x, int y, int radius)
    : Shape(x, y)
    , radius_ {radius}
{}
```

Lets go back to the (very strange!) following line: `Shape(x, y)`. It explicitly calls the Shape constructor, which expects two arguments of type `int`. Therefore, at the end of this call, our object will be initialized like a Shape object would. Note that, by default, the child constructor calls the parent constructor that expects no argument.

3.2 Inclusion Polymorphism

A circle is a shape, so the following instruction is correct:

```
Shape* p = new Circle(18, 42, 12);
```

Thus we can write:

```
p->translate(2, 4);
```

This is the inclusion polymorphism: we can put an object of class `B` in a variable of class `A` if `B` inherits from `A`.

3.2.1 Dynamic type and static type

Nevertheless, `p` being a pointer on a Shape, we cant write:

```
p->radius_set();
```

The compiler has no idea of what is behind `p`; it takes care only of its *static* type (the declared type), because its *dynamic* type (type of the actually referenced object) isnt known until runtime. For the compiler, its a type error: Shape doesnt have a `radius_set` method.

3.3 Visibility in inheritance

You already know the effect of different visibility modifiers (*public*, *protected*, *private*) on methods and attributes. However, you can also choose the visibility of the class which you inherit from! Most of the time we inherit with the *public* modifier, because otherwise the base class isn't accessible and therefore, you can't use inclusion polymorphism.

3.3.1 Example

```
class Shape
{
    ...
};

class Circle : Shape /* class inherits in private by default */
{
    ...
};

int main()
{
    // error: cannot cast 'Circle' to its private base class 'Shape'
    Shape* p = new Circle(18, 42, 12);
}
```

3.4 The virtual keyword

Note that this call doesnt do what we would like:

```
p->print();
```

Indeed, it prints `I am a shape located at (1, 1)`, although we would like it to print `I am a circle at (1, 1) with a radius of 1`.

To get that result, we must delay the `print` methods definition by introducing the `virtual` keyword (in the targeted methods):

```
// in shape.hh
virtual void print() const;
```

The `virtual` keyword tells the compiler to "go down" as far as possible in the class hierarchy of the callers dynamic type (dynamic dispatch). *Dynamic type* means type at *runtime*, regardless of the type your compiler sees (`Circle` instead of `Shape`, here).

3.5 Abstract classes: interface / contract notion

Until now, we were able to do the following:

```
Shape p(12, 12);
```

But this instantiation does not actually carry much sense: an object which is *only a shape* doesn't exist. If an object is of this type, it always has a more precise type: `Square`, `Circle`, etc. The same thing holds with animals: a "pure" animal doesnt exist, its just a concept which becomes reality through a dog, a rabbit, a cat, a pony...

C++ offers a mechanism to manage this constraint: *abstract* classes.

To make a class be abstract, we need to define at least one method without implementation, i.e. a *pure virtual method*.

In the case of `Shape`, this method may be `drawing`. This method must be present in derived classes (`Circle`, `Square`), so we make it *virtual*. To assert that we wont implement the method in the base class, we need to add "`= 0`" at the end of its prototype:

```
virtual void draw() const = 0;
```

Thus the previous example isnt possible anymore: now we cant instantiate an object of type Shape. A direct consequence of the Shape abstraction is that each class which inherits from it (and which we want to be able to instantiate) has to implement *all* pure virtual methods. Abstract classes are kind of a deal: “To get a shape, you must be able to draw it.”

In object-oriented programming, an interface is just a bunch of unimplemented methods. A class which implements an interface have to implement every method. However, in C++, you don’t have true interfaces; instead, programmers usually consider that an abstract class with only pure virtual methods is an interface (apart from constructors, destructors and some operators).

Usually, we end interface names with “-able”, since interfaces just state what a class “can” do. Here is a simple example:

```
class Sendable
{
    virtual void send() = 0;
};

class Message : public Sendable
{
    virtual void send();
};
```

By making Message inherit from Sendable, we ensure that Message implements a send method.

In our other example, we cant implement the draw method for Polygon: therefore Polygon is an abstract class itself. But since we know how to draw a Circle or a Square, these arent abstract classes.

3.6 Method definition

We would now like every shape to give its name (Circle, Square, etc.) when we call its print method, so we need to define the print method inherited from Shape in its derived classes. This feature is called *overriding*.

3.6.1 The override keyword

C++11 brings a useful keyword: override. You should use it (after the declaration of any overridden method) for two reasons:

- it helps the developer to know that the method is overridden
- the compiler can warn you if you actually didn’t override any existing virtual method

3.6.2 Example

```
class A
{
    virtual void foo();
};

class B : public A
{
    virtual void foo(int x); // Method hidding
};
```

Here, from an object obj of type B, calling obj->foo() yields an error saying that the call does match the signature. The method declared in B hides the one from the parent. Actually, the error is not explicit, and we clearly wanted the method foo to replace the one in A with the same arguments. We should have written:

```
class A
{
    virtual void foo();
};

class B : public A
{
    virtual void foo() override;
};
```

and force the compiler that ensure that we are really overidding the method and not hiding it.

3.6.3 Exercise

Add a print method (with the same signature as the one in the class Shape) in each derived class to achieve this result:

```
// in override.cc
#include "shape.hh"
#include "polygon.hh"
#include "square.hh"
#include "circle.hh"

int main()
{
    Square square(3, 3, 3);
    Circle circle(4, 4, 4);

    Shape* sq = &square;
    Shape* c = &circle;

    sq->print();
    c->print();
}
```

```
$ make
$ ./override
I am a square located at (3, 3) with a side of 3
I am a circle located at (4, 4) with a radius of 4
```

Keep in mind that we cant instantiate a Shape anymore.

3.6.4 Multiple inheritance

C++ offers a controversial possibility called *multiple inheritance*. As we can expect, its about making a class inherit from two or more classes. The derived class inherits all the fields and methods of all parent classes.

Lets imagine that we implement an OpenGL software. It might be interesting to separate a shape from its representation, so we could end with a GLDrawable class:

```
class GLDrawable
{
public:
    virtual void draw() = 0;

protected:
    int color_[3];
    int texture_;
};
```

Thus, the Square class and the GLDrawable class will give birth to a GLSquare class:

```
class GLSquare: public Square, public GLDrawable
{
public:
    virtual void draw() override
    {
        // Do *not* actually do this
        // OpenGL doesn't work like this anymore
        glBegin(GL_QUADS);
        glBindTexture(GL_TEXTURE_2D, texture_);
        glVertex2d(...);
        glEnd();
    }
};
```

Now here is one of the errors you could get dealing with multiple inheritance:

```
class A
{
public:
    void toto()
    {}
};

class B
{
public:
    void toto()
    {}
};

class C : public A, public B
{
};

int main()
{
    C c;
    c.toto();
};
```

```
$ g++ -o multiple multiple.cc
multiple.cc: In function 'int main()':
multiple.cc:29: error: request for member 'toto' is ambiguous
multiple.cc:13: error: candidates are: void B::toto()
multiple.cc:5: error:                  void A::toto()
```

This call is particularly ambiguous, because the method exists in both base classes.

A simple method to resolve this ambiguity is to *explicitly* specify that the class C will use the method toto from the inherited class A (and not from B):

```
class C : public A, public B
{
public:
    using A::toto;

    int test()
    {
        toto(); // A::toto() is called.
        B::toto(); // B::toto() is called.
    }
};

int main()
{
    C c;
    c.toto(); // A::toto() is called.
}
```

3.7 Exercice

The aim of this exercise is that you become familiar with inheritance.

3.7.1 Engine class

Write an Engine class. This class has a positive integer `fuel` attribute and the following methods:

- `Engine(int fuel)`: constructor.
- `bool start()` consumes one fuel unit. It sends back a boolean which indicates if there is enough fuel. This method displays (on the standard output): `Engine started with X fuel units`, where `X` is the number of available fuel units if the engine is able to start.
- `void use(int consumed)`: consumes consumed fuel units. Displays `Engine uses X fuel unit`, where `X` is the number of consumed fuel unit. If consumed is greater than the engine fuel unit number, then it consumes the maximum fuel units possible.
- `void stop() const`: displays `Stop Engine` on standard output.
- `void fill(int fuel)`: displays `Filled with X fuel units`, where `X` is the Engine fuel unit number.

3.7.2 Vehicle class

`Vehicle` is an abstract class which contains a `string` named `model`. This class contains two methods and a constructor:

- `Vehicle(const std::string& model)`
- `virtual bool start() = 0`
- `virtual void stop() const = 0`

3.7.3 MotorVehicle Class

`MotorVehicle` is a `Vehicle` subclass. This subclass has an `Engine` attribute, and the following methods:

- `MotorVehicle(const std::string& model, int fuel)`: constructor.

- `virtual bool start() override`: starts the vehicle motor.
- `virtual void stop() const override`: stops the vehicle motor.
- `void cruise(int fuel)`: uses the motor.
- `void fill(int fuel)`: fills the motor.

3.7.4 TwoWheelVehicle Class

Write a `TwoWheelVehicle` abstract class. This class has two methods:

- `virtual void changeFrontWheel() const = 0`
- `virtual void changeBackWheel() const = 0`

3.7.5 Motorcycle Class

Write a `Motorcycle` class which inherits from both `MotorVehicle` and `TwoWheelVehicle`. It has the following methods:

- `Motorcycle(const std::string& model)`: builds a model `Motorcycle` which holds 13 fuel units.
- `virtual void changeFrontWheel() const override`: displays `Change front wheel` on the standard output.
- `virtual void changeBackWheel() const override`: displays `Change back wheel` on the standard output.