Imperial College London

Software Engineering 2: Object Oriented Software Engineering

Week 6 – Polymorphism and virtual functions – Lab *

Sahbi Ben Ismail (s.ben-ismail@imperial.ac.uk)

More on inheritance

Analyse and test the following code.

```
1  #include <iostream>
2  #include <vector>
3
4  class Base {
5    public:
6      Base(): n_priv(0), n_prot(0) {}
7      Base(int n1, int n2) : n_priv(n1), n_prot(n2) {}
8
9      ~Base() { std::cout << "~Base(): n_priv=" << n_priv << ", ↩
           n_prot=" << n_prot << " is leaving" << std::endl; }
10
11     void f() { std::cout << "Base::f()" << std::endl; }
12     void display() { std::cout << "Base::display(): n_priv=" <<↩
           n_priv << ", n_prot=" << n_prot << std::endl; }
13     void call_all_functions() { std::cout << "Base::↩
           call_all_functions()" << std::endl; f(); f_priv(); ↩
           f_prot(); }
14
15   private:
16     int n_priv;
```

*Lab content (second exercise) originally written by Max Cattafi.

```
17      void f_priv() { std::cout << "Base::f_priv()" << std::endl;↩
            }
18
19    protected:
20      int n_prot;
21      void f_prot() { std::cout << "Base::f_prot()" << std::endl;↩
            }
22  };
23
24
25  class Derived : public Base {
26    public:
27      Derived(): Base(), n_base(0) {}
28      Derived(int n1, int n2, int n3): Base(n1, n2), n_base(n3) ↩
            {}
29
30      ~Derived() { std::cout << "~Derived(): n_base=" << n_base ↩
            << " is leaving" << std::endl; }
31
32      void f() { std::cout << "Derived::f()" << std::endl; }
33      void f(int n) { std::cout << "Derived::f(int n) with n=" <<↩
            n << std::endl; }
34
35      void display() { std::cout << "Derived::display(): " << std↩
            ::endl; Base::display(); std::cout << "n_base=" << ↩
            n_base << std::endl; }
36
37      void new_function() { std::cout << "Derived::new_function()↩
            " << std::endl; }
38
39    protected:
40      int n_base;
41  };
42
43  class DerivedAgain : public Derived {};
44
45  int main() {
46    //TEST 1: Base and Derived classes
47
48
49    //TEST 2: DerivedAgain class
50
51
52    //TEST 3: Vector of Base objects (intro to polymorphism)
53
54
55    //TEST 4: Vector of Base dynamic objects (intro to ↩
          polymorphism)
56
```

```
57
58    //TEST 5: Make f() a virtual member function in Base class, ↩
          and repeat TEST 3 and TEST 4
59    //virtual void f() { std::cout << "Base::f()" << std::endl; }
60
61  }
```

Complete the main function to perform tests in the given order:

- TEST 1: What is inherited and what is accessible in class `Derived`? What is the constructor calling order? What is the destructor calling order? Is it possible to *redefine* functions? Is it possible to *overload* functions?
- TEST 2: Yet another inheritance with `DerivedAgain`.
- TEST 3: Vector of `Base` objects
- TEST 4: Vector of `Base` dynamic objects
- TEST 5: TEST 3 and TEST 4 with f() as a *virtual* function in class `Base`.

Using polymorphism to draw shapes

The following code draws 2 lines into a .svg file. Analyse how it works in order to add the function `Draw()` to our `triangle` class, and call it in order to draw the triangle.

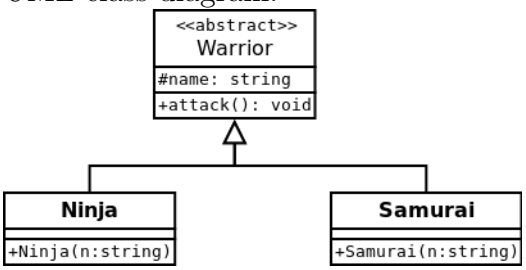
```
1  #include <iostream>
2
3  using namespace std;
4
5  void DrawLine(double x1, double x2, double y1, double y2);
6
7  int main() {
8      double width = 100, height = 100;
9      cout << "<svg width=\"" << width << "\" height=\"" << ↩
           height << "\" xmlns=\"http://www.w3.org/2000/svg\">" <<↩
            endl;
10     cout << "<style type=\"text/css\">line{stroke:black;stroke-↩
           width:1;stroke-opacity:0.5;stroke-linecap:round;}</↩
           style>" << endl;
11
12     DrawLine(10, 50, 10, 50);
```

```
13      DrawLine(25, 100, 50, 50);
14
15      cout << "</svg>" << endl;
16  }
17
18  void DrawLine(double x1, double x2, double y1, double y2) {
19      cout << "<line x1=\"" << x1 << "\" x2=\"" << x2 << "\" y1↩
            =\"" << y1 <<"\" y2=\"" << y2 << "\"/>" << endl;
20  }
```

After drawing a triangle, create an abstract class called `shape` with `Draw()` as a virtual function. Make triangle inherit from the class, and make another class called `square`, which that also inherits from `shape`. Make use of polymorphism by creating a vector of shapes constructed with triangles and squares, and calling `Draw()` on each of them.

Using polymorphism to play a mini combat game

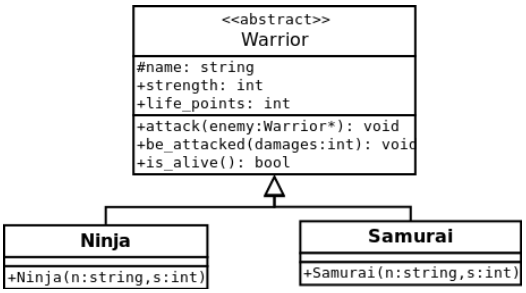
Write classes `Warrior`, `Ninja` and `Samurai` corresponding to the following UML class diagram.



Step 1: simple attacks

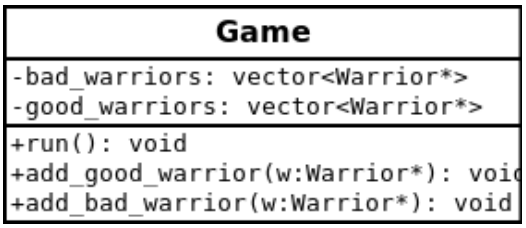
Like in the lecture notes, function attack simply displays a text on the screen.

Step 2: real attacks



A warrior has life points (a number from 0 to 100) and a strength (a number, initialised randomly between 1 to 100). When a warrior attacks another warrior, he causes damages (randomly between 1 and strength). A warrior is alive if he still has life points.

Step 3: good vs. bad warriors, the ultimate battle



Write class `Game` having two teams of warriors: good warriors and bad warriors (two vectors of pointers to Warrior). In the main function, create an object instance of Game, create objects instances of classes Ninja and Samurai, add them to the game. Run the game check which team won the battle. Before starting the battle, make sure the two teams have the same number of warriors. At each step of the battle, only one warrior of one team attacks one enemy from the opposite team. At the next step, one warrior of the attacked team will reply by attacking one enemy. Remember, we don't attack dead warriors!