1 Polymorphism

Consider the following partial definition of `class Faculty`:

```
class Faculty
{
protected:
    Student* students;
    int numStudents;
public:
    void GetStudents();
    void SortStudents();
    void CalculateStatistics();

    void ProcessStudents()
    {
        GetStudents();
        SortStudents();
        CalculateStatistics();
    }
};
```

Assume that we need to improve the sorting algorithm of the function `SortStudents()`. According to the open/closed principle, we should derive a new class from `Faculty` and overload its `SortStudents()` function as follows:

```
class EnhancedFaculty : public Faculty
{
public:
    void SortStudents();     // Contains modified implementation
};
```

Consider the following code:

```
EnhancedFaculty f;
f.SortStudents();          // Calls SortStudents() of class EnhancedFaculty
f.ProcessStudents();       // Calls SortStudents() of class Faculty (unexpected)
```

Class `EnhancedFaculty` does not behave as expected. The reason is that, when calling the function `ProcessStudents()` through an object of type `EnhancedFaculty`, the compiler will associate it with the base class function since it does not exist in the derived class. When the function call `SortStudents()` is encountered within the execution of `ProcessStudents()`, the compiler will execute the `SortStudents()` function of the base class, which is not consistent with our wish to replace it with the one in the derived class.

To achieve the desired behavior, the `virtual` modifier should precede the base function declaration:

```
1  class Faculty
2  {
3  protected:
4      Student* students;
5      int numStudents;
6  public:
7      void GetStudents();
8      virtual void SortStudents();
9      void CalculateStatistics();
10     void ProcessStudents()
11     {
12         GetStudents();
13         SortStudents();
14         CalculateStatistics();
15     }
16 };
17 class EnhancedFaculty : public Faculty
18 {
19 public:
20     void SortStudents();
21 };
```

Now, the desired behavior is achieved:

```
1  EnhancedFaculty f;
2  f.SortStudents();          // Calls SortStudents() of class EnhancedFaculty
3  f.ProcessStudents();       // Calls SortStudents() of class EnhancedFaculty
```

When calling the function `ProcessStudents()` through an object of type `EnhancedFaculty`, the compiler will associate it with the base class function since it does not exist in the derived class. When the function call `SortStudents()` is encountered within the execution of the function `ProcessStudents()`, the `virtual` modifier will direct the execution to check the object type during run-time and calls will execute the `SortStudents()` function of the derived class as needed. Only if it is not found in the derived class, the base class function will be executed.

The above behavior is called overriding the base class function `SortStudents()`. Calling the function based on the object type checked at run-time is called run-time polymorphism. In our lectures, we just call it polymorphism.

C++ allows a base class pointer to hold the address of an object from a derived class. Also, C++ allows a base class reference to refer to an object from a derived class. Combined with the run-time ability of the `virtual` modifier to redirect execution to the actual type of the calling object, the following examples illustrate the effect of the `virtual` modifier. Note that a virtual destructor is essential for any class containing `virtual` functions.

```
1  class A
2  {
3  public:
4      virtual void F();
5      virtual void G();
6      virtual ~A() {}       // Virtual destructor is essential
7  };
8  class B : public A
9  {
10 public:
11     void F();
12     void G();
13 };
14 class C : public B
15 {
16 public:
17     void F();
18     void H();
19 };
20
21 void TestPointer()
22 {
23     A* a = new C;     // A pointer to a base class can hold address of derived class
24     a->F();           // Go to A::F(), virtual redirects to C, so C::F() is called
25     a->G();           // Go to A::G(), virtual redirects to C, C::G() is not found,
26                       //   then go to the base class of C, so B::G() is called
27     a->H();         ✖ // Compiler error because A::H() does not exist
28     delete a;         // Go to A::~A(), virtual redirects to C, so C is destroyed
29 }
30
31 void TestReference()
32 {
33     C c;
34     A& a = c;      // A reference to a base class can refer to object of derived class
35     a.F();         // Go to A::F(), virtual redirects to C, so C::F() is called
36     a.G();         // Go to A::G(), virtual redirects to C, C::G() is not found,
37                    //   then go to the base class of C, so B::G() is called
38     a.H();       ✖ // Compiler error because A::H() does not exist
39 }
```

Suppose we need to create a game where the user (player) can select a hero at the beginning to play with. The game behavior is basically the same for all heroes, except for some minor details related to each hero type.

A first attempt is the following implementation:

```
1  enum HeroType{SuperMario, SuperMan, BatMan};
2
3  class Hero
4  {
5  protected:
6      HeroType type;
7
8  public:
9      Hero(HeroType t) {type=t;}
10     void Fire();
11     void Jump();
12
13     void Draw()
14     {
15         if(type==SuperMario) {/* Draw SuperMario*/}
16         else if(type==SuperMan) {/* Draw SuperMan*/}
17         else if(type==BatMan) {/* Draw BatMan*/}
18     }
19 };
20
21 class Game
22 {
23 protected:
24     Hero hero;
25
26 public:
27     Game(Hero h) : hero(h) {}
28     void Play() {hero.Draw(); hero.Fire(); hero.Jump();}
29 };
30
31 void TestGame()
32 {
33     Hero hero(SuperMan);
34     Game game(hero);
35     game.Play();
36 };
```

The problem with this approach is that if we decide to add a new hero to the game, such as `SpiderMan`, the functions of `class Hero` such as `Draw()` need to be modified, which violates the open/closed principle.

A second attempt is the following implementation:

```
class SuperMario    class SuperMan     class BatMan
{                   {                  {
public:             public:            public:
    void Fire();        void Fire();       void Fire();
    void Jump();        void Jump();       void Jump();
    void Draw();        void Draw();       void Draw();
};                  };                 };

class GameSuperMario
{
protected:
    SuperMario hero;
public:
    void Play() {hero.Draw(); hero.Fire(); hero.Jump();}
};

class GameSuperMan
{
protected:
    SuperMan hero;
public:
    void Play() {hero.Draw(); hero.Fire(); hero.Jump();}
};

class GameBatMan
{
protected:
    BatMan hero;
public:
    void Play() {hero.Draw(); hero.Fire(); hero.Jump();}
};

void TestGame()
{
    GameSuperMan game;
    game.Play();
};
```

The problem with this approach is the need to write separate Game class for each hero, which have very similar behaviors, which causes code duplication. Also, if we decide to add a new hero to the game, such as `SpiderMan`, a `class GameSpiderMan` needs to be defined, with most of the code duplicated from an existing Game class.

Polymorphism provides a better approach as follows:

```
1  class Hero
2  {
3  public:
4      virtual void Fire();
5      virtual void Jump();
6      virtual void Draw();
7      virtual ~Hero() {}
8  };
9  class SuperMario : public Hero
10 {
11 public:
12     void Fire();
13     void Jump();
14     void Draw();    // Draw SuperMario
15 };
16 class SuperMan : public Hero
17 {
18 public:
19     void Fire();
20     void Jump();
21     void Draw();    // Draw SuperMan
22 };
23 class BatMan : public Hero
24 {
25 public:
26     void Fire();
27     void Jump();
28     void Draw();    // Draw BatMan
29 };
30 class Game
31 {
32 protected:
33     Hero* hero;
34 public:
35     Game(Hero* h) {hero=h;}
36     void Play() {hero->Draw(); hero->Fire(); hero->Jump();}
37 };
38 void TestGame1()
39 {
40     Hero* hero = new SuperMan;
41     Game game(hero); game.Play();
42     delete hero;
43 };
```

```
void TestGame2()
{
    Hero* hero = new SuperMan;
    Game* game = new Game(hero);
    game->Play();
    delete game; delete hero;
};
```

It is also possible to define `class Game` as follows:

```
class Game
{
protected:
    Hero& hero;
public:
    Game(Hero& h) hero(h) {}
    void Play() {hero.Draw(); hero.Fire(); hero.Jump();}
};
void TestGame()
{
    SuperMan s
    Game game(s);
    game.Play();
};
```

2 The Liskov substitution principle

The Liskov substitution principle is one of the SOLID principles stated as follows:

"Modules that use pointers or references to a base class must be able to use objects of derived classes without knowing them."

In the previous example, since functions of `class Game` use a `Hero` pointer, it can use objects of derived classes, such as `SuperMan`, and call its functions, without including any declaration of `SuperMan` inside `class Game`.

Inheritance and polymorphism are powerful mechanisms to apply the Liskov substitution principle. One of the important guidelines of this principle is that derived classes should behave in a compatible or semantically similar way to the base class. For example, if the intent of the `Draw()` function in `class Hero` is to draw the hero, the `Draw()` function of a derived class such as `SuperMan` must draw that hero. It must be implemented, and it should not do a different thing than drawing the `SuperMan` object.