Advanced Programming

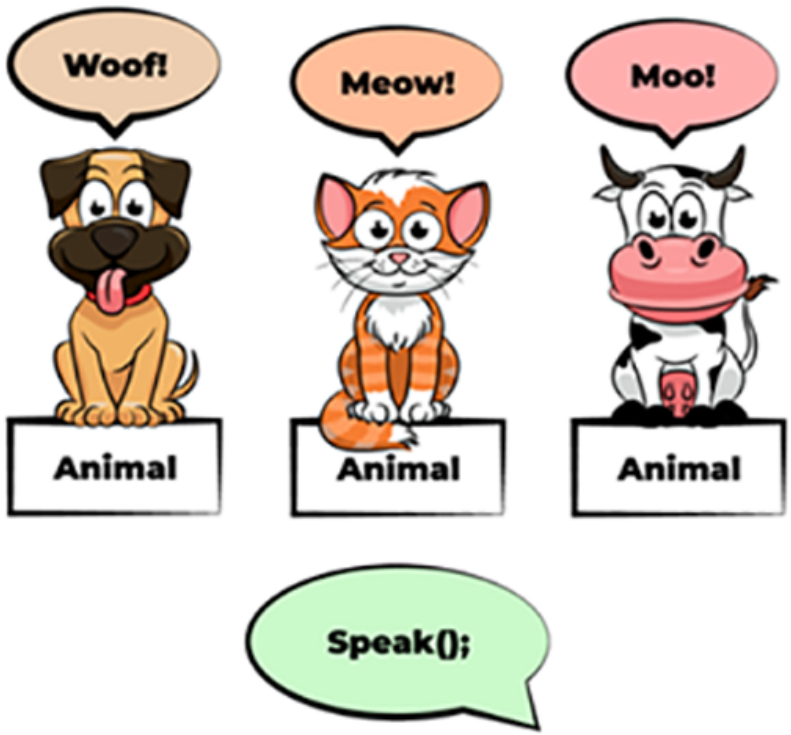
[Polymorphsim]

Zeinab Zali

References: (1) "C++ How to program" Deitel&Deitel, (2) "A Tour of C++" Bjarne Stroustrup, (3) Other useful learning pages such as geeksforgeeks and tutorialpoints

ECE Department, Isfahan University of Technology

Animals with different languages



Animals with different languages

- It is desired to have an array of Animal (base-class) objects
- We want to add new animals of different types (derived classes) to this array in **runtime**
- We expect to see the functionality of each type (derived class) for each different animal in the array, not the functionality of the generic Animal (base class)

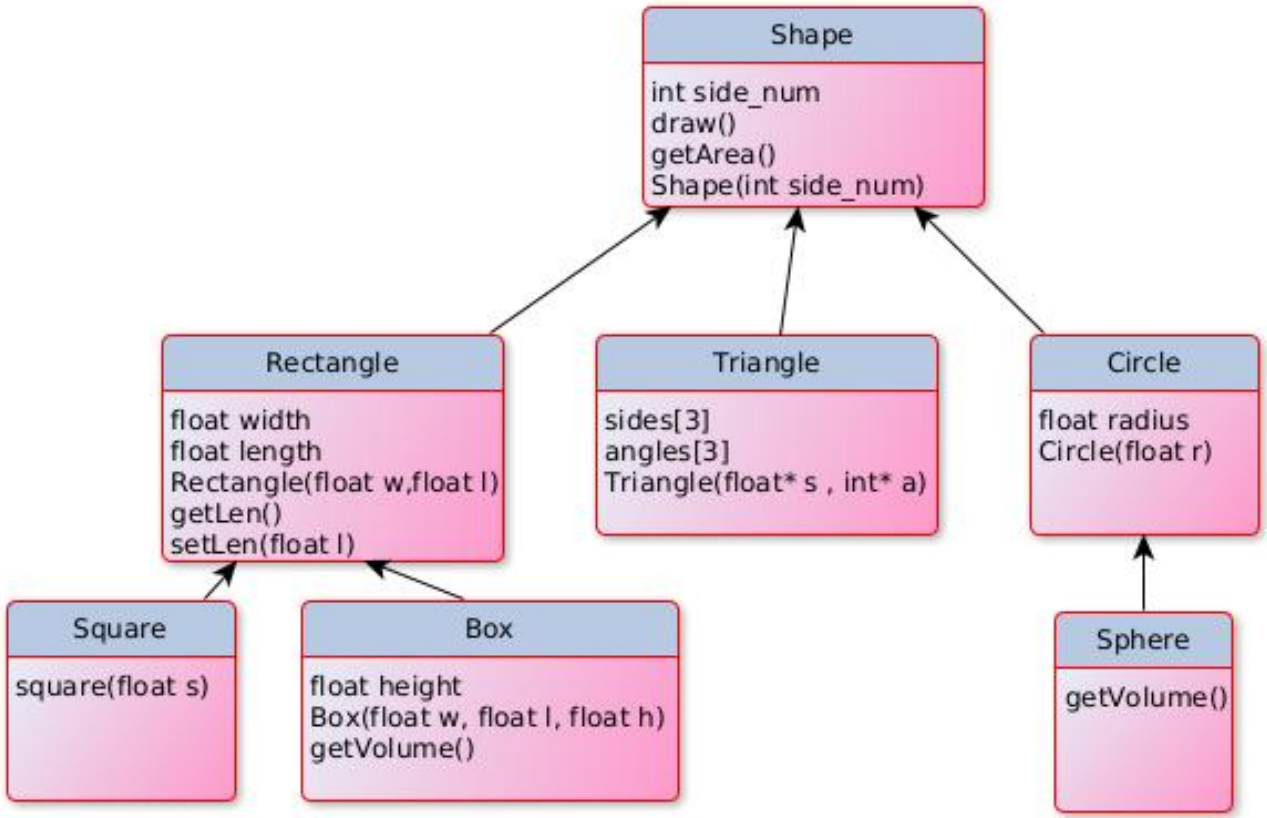
Let's Start with Some Steps

1. Assigning the address of derived-class to a base-class pointer (Ok)
2. Assigning the address of base-class to a derived-class pointer (Compile Error)
3. Invoking inherited member functions of base-class pointer that aimed to a derived-class (Ok, base-class version is called)
4. Invoking derived-class only members from a base-class pointer that aimed to a derived class (Compile Error)
5. Invoking derived class functionality from a base-class pointer (How? with polymorphism)

Converting a Derived Class Object to a Base-class

Upcasting is converting a derived-class reference or pointer to a base-class.

- For example assigning a rectangle object to a Shape pointer



Which Functionality is Observed?

- Assigning the address of a derived-class object to a base-class is valid, because each derived class object **is a** base class object too.
- Invoking a function via the base-class pointer invokes the base-class functionality in the derived-class object
- The type of the handle determines which function is called

```
1       Shape *s1;
2       Rectangle r1;
3       Shape *r2 = new Rectangle(10,20); //Ok, each
    rectangle is a shape
4       s1 = &r1; //Ok
5       cout << r2->getArea() << endl; //Shape getArea is
    called
6       cout << s1->getArea() << endl;
7       //cout << s1->getLen() << endl;  //Compile Error
```

Converting a Base-class Object to a Derived Class

Downcasting is an opposite process to the upcasting, which converts the base class's pointer or reference to the derived class's pointer or reference.

- For example assigning a Shape object to a Rectangle pointer

Converting a Base-class Object to a Derived Class

- Assigning the address of a base-class object to a derived-class pointer results in a compilation error.
- Because not every base-class object is a derived class one (not every Shape is necessarily a Rectangle)

```
1         Rectangle *r3;
2         r3= new Shape(4); //Compile error, a shape may not
      be a rectangle
3         Shape s2;
4         //r3 = s2;              //Compile error
```

Explicit Downcasting

- The compiler will allow access to derived-class-only members from a base-class pointer that's aimed at a derived-class object if we explicitly cast the base-class pointer to a derived-class pointer
- Downcasting is a potentially dangerous operation

```
1       r3 = (Rectangle*)&s2; //Ok, Explicit casting
2       cout<< r3->getLen()<<endl; //we can call getLen,
    but the len data of rectangle may not be initialized,
    so unknown value
3       r3->setLen(10);                 //valid. Assigning a
    value to a data mamber with member function of derived
    class
```

Virtual Functions

A virtual function is a member function which is declared within a base class and is redefined (overridden) by a derived class.

- If we do declare the base-class function as virtual , we can override that func- tion to enable **polymorphic behavior**
- Recall that the type of the handle determined which class's functionality to invoke
- With virtual functions, the type of the object—not the type of the handle used to invoke the object's member function—determines which version of a virtual function to invoke.

Virtual Functions

We can declare every member function (except constructors) of a base class as a virtual function, by adding a virtual keyword before the function.

```
1 class Shape{
2     protected:
3         const int sidesNum;
4     public:
5         Shape():sidesNum(4){

7         }
8         Shape(int s):sidesNum(s){
9             cout << "Shape Constructor\n";
10        }
11        virtual float getArea(){
12            cout << "Shape getArea()\n";
13            return 0;
14        }
```

Overriding Virtual Functions

- We can override a virtual function by redefining it in the derived class
- An overridden function in a derived class has the same signature and return type (i.e., prototype) as the function it overrides in its base class
- If we do not declare the base-class function as virtual , we can redefine that function, but it does not result in polymorphic behavior

```
1class Rectangle: public Shape{
2    public:
3         float getArea(){
4              cout << "Rectangle getArea()\n";
5              return width * len;
6         }
```

Dynamic Binding

Choosing the appropriate function to call at execution time (rather than at compile time) is known as **dynamic binding**

- If a program invokes a virtual function through a base-class pointer to a derived-class object (e.g., s1->getArea()) or a base-class reference to a derived-class object (e.g.,s3.getArea()), the program will choose the correct derived-class function dynamically (i.e., at execution time) based on the object type—not the pointer or reference type.

```
1        Shape *s1 = new Rectangle(10,20);
2        Shape *s2 = new Shape(4);
3        Rectangle r1(10,20);
4        Shape &s3 = r1;
5        cout << s1->getArea() << endl;   //rectangle getArea
6        cout << s3.getArea() << endl;    //rectangle getArea
7        cout << ((Rectangle*)s2)->getArea();//Shape getArea
```

Inheriting Virtual Characteristic

- Once a function is declared virtual , it remains virtual all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared virtual when a derived class overrides it.
- When a derived class chooses not to override a virtual function from its base class, the derived class simply inherits its base class's virtual function implementation.

```
1         Shape *s4 = new Box(10,20,30);
2         cout << s4->getArea() <<endl;    //Box getArea,
    recall Box is derived from Rectangle and getArea is
    also overriden in Box class
```

Dynamic Binding, Another Example

Here, we have an array of Shape pointers, but dynamically, fill it with different shapes with their own behaviors (polymorphic)

```
1     int type;
2     int num = 3;
3     Shape *figs[num];
4     for(int i=0;i<num;i++){
5         cout << "which shapes do you want to add?";
6         cout << "(1)Rectangle,(2)Circle, (3)Box?:\n";
7         cin >> type;
8         switch(type){
9             case 1: figs[i] = new Rectangle(10,10);break;
10            case 2: figs[i] = new Circle(10);break;
11            case 3: figs[i] = new Box(10,10,10);break;
12        }
13    }
14    for(int i=0;i<num;i++){
15        cout<< figs[i]->getArea() << endl; //getArea is
      called according to exact type of figs[i] (not Shape
      getArea)
16    }
```

Virtual Destructors

- If a derived-class object with a non-virtual destructor is destroyed by applying the delete operator to a base-class pointer to the object, the C++ standard specifies that the behavior is undefined.
- The simple solution to this problem is to create a public virtual destructor in the base class.
- If a base-class destructor is declared virtual, the destructors of any derived classes are also virtual.
- with virtual destructor in base class, if an object in the hierarchy is destroyed explicitly by applying the delete operator to a base-class pointer, the destructor for the appropriate class is called, based on the object to which the base-class pointer points.

default Keyword

For making the default destructor of a base class to a virtual destructor, you can use =default keyword.

- In C++11, you can tell the compiler to explicitly generate the default version of a default constructor, copy constructor, or destructor by following the special member function's prototype with = default

```
1        virtual ~Shape() = default;
```

Override Keyword

Error-Prevention Tip 12.1

To help prevent errors, apply C++11's `override` *keyword to the prototype of* every *derived-class function that overrides a base-class* `virtual` *function. This enables the compiler to check whether the base class has a* `virtual` *member function with the same signature. If not, the compiler generates an error. Not only does this ensure that you override the base-class function with the appropriate signature, it also prevents you from accidentally hiding a base-class function that has the same name and a different signature.*

```
1       float getArea() override{
2           cout << "Box getArea()\n";
3           return 2 * (Rectangle::getArea() +
4                   height * getWidth() +
5                   height * getLen());
6       }
```

```
1       //compile Error, because getVolume is not a virtual
    member function in base class
2       double getVolume() override{
3           cout << "Box getVolume()\n";
4           return height * Rectangle::getArea();
5       }
```

Final Keyword

In C++11, a base-class virtual function that's declared final in its prototype, cannot be overridden in any derived class

- this guarantees that the base class's final member function definition will be used by all base-class objects and by all objects of the base class's direct and indirect derived classes.
- Also,as of C++11, you can declare a class as final to prevent it from being used as a base class.
- Attempting to override a final member function or inherit from a final base class results in a compilation error.

```
1        virtual void f() final{ //making virtual f fuction
    final to avoid overriding it
2            cout <<"f in Rectangle\n";
3        }
```

```
1class Circle final: public Shape{ //making Circle final to
    avoid making a derived class from it
```

Pure Virtual Function

An **abstract class** is a template class of member functions and data members from which you never intend to instantiate any objects.

- An abstract class is a base class from which other classes can inherit
- abstract classes are incomplete, so derived classes must define the “missing pieces” before objects of these classes can be instantiated.
- A class is made abstract by declaring one or more of its virtual functions to be **“pure”**. A pure virtual function is specified by placing “= 0 ” in its declaration

Virtual vs Pure Virtual

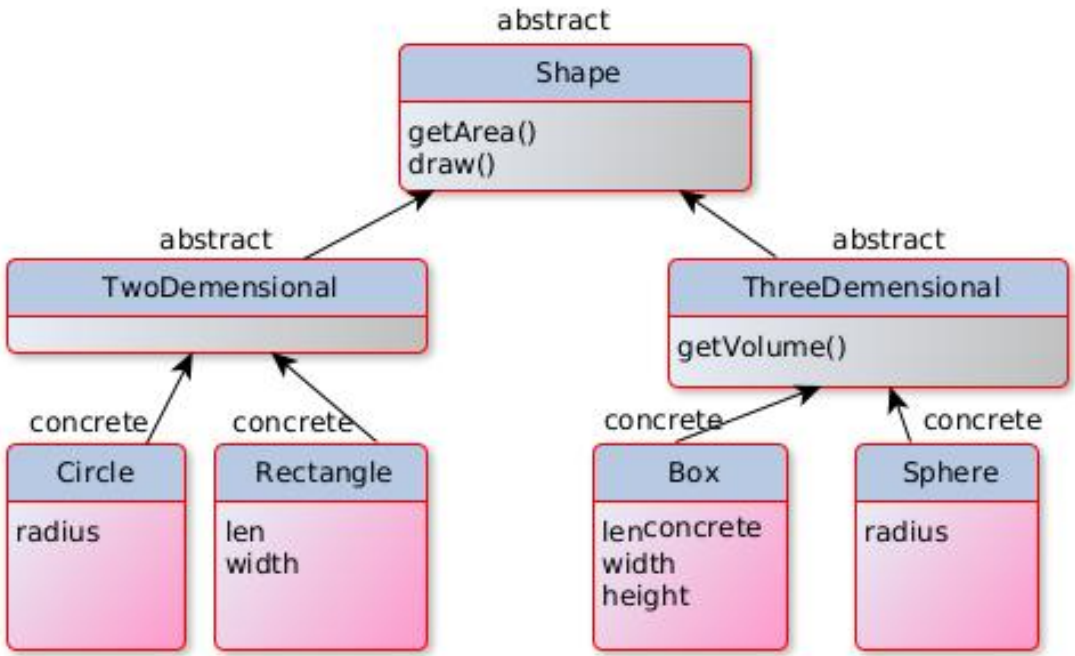
- Virtual function has an implementation and gives the derived class the option of overriding the function;
- A pure virtual function does not have an implementation and requires the derived class to override the function for that derived class to be concrete; otherwise the derived class remains abstract.
- Pure virtual functions are used when it does not make sense for the base class to have an implementation of a function, but you want to force all concrete derived classes to implement the function.

Abstract class and polymorphism

- Although we cannot instantiate objects of an abstract base class, we can use the abstract base class to declare pointers and references that can refer to objects of any concrete classes derived from the abstract class.
- Programs typically use such pointers and references to manipulate derived-class objects polymorphically.

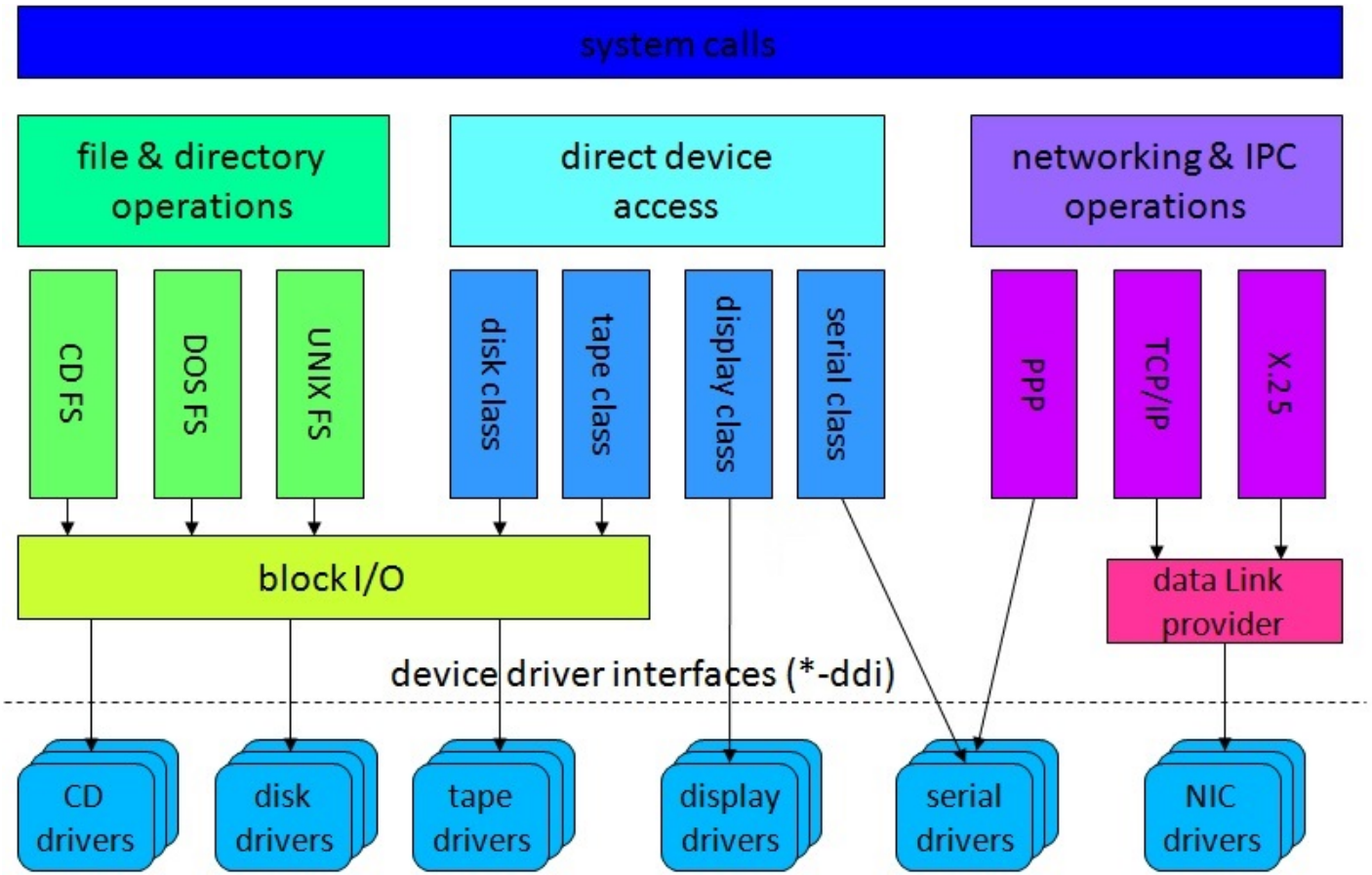
Abstract Class Example

- We need to be more specific before we can think of instantiating objects.
- Concrete classes provide the specifics that make it possible to instantiate objects.
  - For example, if someone tells you to “draw the two-dimensional shape,” what shape would you draw?



Abstract Class Applications (Device Drivers)

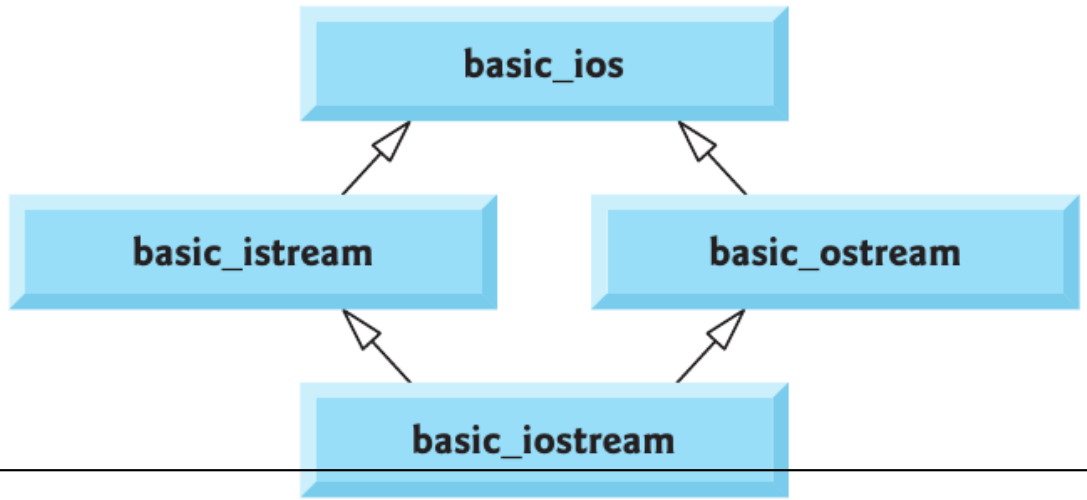
From the application view over the system call layer, we can call read/write function for all the devices with different types through the same prototype.



Problem Statement

The inheritance model described in this figure (for the example of multiple inheritance to form class basic_iostream) is referred to as **diamond inheritance**

- Class basic_iostream could contain two copies of the members of class basic_ios—one inherited via class basic_istream and one inherited via class basic_ostream.
- Such a situation would be ambiguous and would result in a compilation error, because the compiler would not know which version of the members from class basic_ios to use.



Example: Person base class

```
1 class Person{
2     static int idGen;
3 protected:
4     string name;
5     string family;
6     int id;
7 public:
8     Person():id(idGen++){
9         cout << "Person default constructor\n";
10    }
11    Person(string namestr, string familystr):name(namestr),
       family(familystr),id(idGen++){
12        cout << "Person constructor\n";
13    }
14    int getID(){
15        return id;
16    }
17    virtual void print()=0;

19 };
20 int Person::idGen = 1;
```

Example: Student derived from person

```
1 class Student: public Person{
2     static int idGen;
3 protected:
4     int stdID;
5 public:
6     Student():stdID(idGen++){
7         cout << "Student default constructor\n";
8     }
9     Student(string name, string family):Person(name, family
      ), stdID(idGen++){
10        cout << "Student constructor\n";
11    }
12    void print(){
13        cout<<"\nprint Student\n";
14        cout <<"Person ID: " << id <<endl;
15        cout <<"Person name: " <<name << ", family: "<<
      family<<endl;
16        cout << "Student ID: " <<stdID <<endl;

18    }
19 };
20 int Student::idGen = 1;
```

Example: Faculty derived from Person

```
1 class Faculty: public Person{
2     static int idGen;
3 protected:
4     int facID;
5 public:
6     Faculty():facID(idGen++){
7         cout << "Faculty default constructor\n";
8     }
9     Faculty(string name, string family):Person(name, family
      ), facID(idGen++){
10        cout << "Faculty constructor\n";
11    }
12    void print(){
13        cout<<"\nprint Faculty\n";
14        cout <<"Person ID: " << id <<endl;
15        cout <<"Person name: " <<name << ", family: "<<
      family<<endl;
16        cout << "Faculty ID: " <<facID <<endl;

18    }
19 };
20 int Faculty::idGen = 1;
```

Example: TA derived from both Student and Faculty

```
1class TA: public Student, public Faculty{
2     protected:
3     int exp;
4public:
5     TA():exp(0){
6          cout << "TA default constructor\n";
7     }
8     TA(string name, string family, int exp):Student(name,
     family),exp(exp){
9          cout << "TA constructor\n";
10    }
11    void print(){
12         cout<<"\nprint TA\n";
13         cout <<"Person ID: " << id <<endl;//compile error,
     reference to id is ambiguous
14         cout <<"Person name: " <<name << ", family: "<<
     family<<endl;//compile error
15         cout << "Student ID: " <<stdID <<", Faculty ID: "<<
      facID <<endl;
16         cout << "TA experience: "<< exp << endl;
17    }
18};
```

Example: instantiating some TAs

```
1     TA t1;
2     TA t2;
3     cout<<"t1 id: " <<t1.getID();//compile error, getID()
    is ambiguous, because of two versions from both Student
     and Faculty
4     cout<<"t1 person id: " <<t1.Student::getID() <<endl; //
    OK. Using scope resolution to solve the ambiguity
5     cout<<"t2 person id: " <<t2.Student::getID() <<endl;
```

```
~/Doc/IU/Te/AP/s/Ad/c/Polymorphism ./Diamond
Person default constructor
Student default constructor
Person constructor
Faculty constructor
TA constructor
Person default constructor
Student default constructor
Person constructor
Faculty constructor
TA constructor
t1 person id: 1
t2 person id: 3
```

Example: Instantiating Some TAs

- Calling getID() from a TA object results in an ambiguity, because getID() is inherited from Person twice in the TA object.
  - It results in compile error.
- Instantiating each object of TA results in executing Person constructor twice.
  - It results in generating wrong person ID.

Excercise: Instantiating Some TAs

```
1     TA(string name, string family, int exp):Student(name,
      family),exp(exp){
2           cout << "TA constructor\n";
3     }
```

- Execute the following code and analyse the result.
- Pay atention to TA constructor and its member initializer list.
- Try to explicitly call constructor of Faculty and analyze the execution result.
- Remove explicitly calling Student constructor and analyze the execution result.
- Try to explicitly call constructor of Person

```
1     TA t3("Ali", "Rahimi",3);
2     t3.Student::print();
3     t3.Faculty::print();
```

Polymorphic Behavior in Diamond

```
1     Person *p[3];
2     p[0] = new Student("Arezou","Rad");
3     p[1] = new Faculty("Ali","Amiri");
4     p[2] = new TA("Sahar","Sadeghi",2);//can not be
      instantiated, base class Person is ambiguous

6     for(int i=0;i<3;i++)
7         p[i]->print();
```

Solution: Virtual Inheritance

The problem of duplicate subobjects is resolved with **virtual inheritance**.

- When a base class is inherited as virtual , only one subobject will appear in the derived class—a process called virtual base-class inheritance.

```
1class Student: virtual public Person{
```

```
1class Faculty: virtual public Person{
```

Instantiating after Virtual Inheritance

Pay atention to constructors' calls

```
1     TA t1;
2     TA t2;
3     cout<<"t1 person id: "<<t1.getID()<<endl;//Ok, getID()
       is not ambiguous
4     cout<<"t2 person id: "<<t2.getID()<<endl;
```

```
~/Doc/IU/Te/AP/s/Ad/c/Polymorphism ./Diamond2
Person default constructor
Student default constructor
Faculty default constructor
TA default constructor
Person default constructor
Student default constructor
Faculty default constructor
TA default constructor
t1 person id: 1
t2 person id: 2
```

Polymorphic Behavior in Diamond

Now we can explicitly call Person constructor in the member initializer list of TA constructor

```
1     TA(string name, string family, int exp):Person(name,
      family),exp(exp){
2          cout << "TA constructor\n";
3     }
```

```
1     Person *p[3];
2     p[0] = new Student("Arezou","Rad");
3     p[1] = new Faculty("Ali","Amiri");
4     p[2] = new TA("Sahar","Sadeghi",2);//OK
5     for(int i=0;i<3;i++)
6          p[i]->print();
```

```
~/Doc/IU/Te/AP/s/Ad/c/Polymorphism ./Diamond2
Person constructor
Student constructor
Person constructor
Faculty constructor
Person constructor
Student default constructor
Faculty default constructor
TA constructor

print Student
Person ID: 1
Person name: Arezou, family: Rad
Student ID: 1

print Faculty
Person ID: 2
Person name: Ali, family: Amiri
Faculty ID: 1

print TA
Person ID: 3
Person name: Sahar, family: Sadeghi
Student ID: 2, Faculty ID: 2
TA experience: 2
```