Lesson 9: Inheritance
To manage the personnel of a company, we need to define classes corresponding to posts as follows:

```cpp
class Worker {
    private:
        string name;
        float salary;
        int level;
    public:
        string getName() {...}
        void pay() {...}
        void doWork() {...}
};
```

```cpp
class Manager {
    private:
        string name;
        float salary;
        int dept;
    public:
        string getName() {...}
        void pay() {...}
        void doWork() {...}
};
```

```cpp
class Director {
    private:
        string name;
        float salary;
    public:
        string getName() {...}
        void pay() {...}
        void doWork() {...}
};
```

All 3 above classes have variables and methods that are exactly the same → define an Employee class for those things and then reuse

- Reuse code
- Reduce written code
- Facilitate the maintenance, modifications in the future
- Clarify the logic in the software design
Two types of inheritance:

- Specialize: child class is a particular case of parent (like above example)
- Generalize: child class extends parent (ex: add z to Point2D to get Point3D)

Inheritance allows child classes using variables and methods of parent like they are their members, except private members

Public and private inheritance:

- public: public members of parent are still public of children
- private: all members of parent become private of children
Public inheritance

class Employee {
private:
    string name;
    float salary;
public:
    ...
    string getName() {...}
    void pay() {...}
};

class Worker : public Employee {
private:
    int level;
public:
    ...
    void doWork() {...}
};

void show() {
    cout << getName() << salary; // error
}

Worker w;
w.getName();
w.doWork();
w.pay();
w.salary = 10; // error
w.show();

Employee e = w; // OK
Worker w2 = e; // error
Worker w3 = (Worker)e; // error

- Public members of parent class are still public of child one
- Objects of child class can be casted as parent class, but not inverse
Private inheritance

class LinkedList {
private:
    ...
public:
    void insertTail(int x) { ... }
    void insertHead(int x) { ... }
    void deleteHead() { ... }
    void deleteTail() { ... }
    int getHead() { ... }
    int getTail() { ... }
    ...
};

class Stack : private LinkedList {
public:
    void push(int x) { insertHead(x); }
    int pop() {
        int x = getHead();
        deleteHead();
        return x;
    }
    ...}

Stack s;
s.push(10);
s.push(20);
s.pop();
s.insertTail(30); // error
s.getTail(); // error

- All members of parent class become private of child class
Protected members

- Apart public and private, there are also protected members: accessible from methods of child classes, but not from outside of those classes

```cpp
class Employee {
protected:
    string name;
    float rate;
    int hours;

    int getSalary()
    { return rate*hours; }

public:
    void setName(const char* s)
    { name = s; }
    string getName()
    { return name; }
    void pay() { ... }
    ...
};

class Worker: public Employee {
public:
    void doWork() { ... }
    void print() {
        cout << "Ten: " << name
             << "Luong: " << getSalary();
    }
    ...
};

Worker w;
w.doWork();
w.pay();
w.print();
w.name = "NV Tung";    // error
cout << w.getSalary(); // error
```

Summary of inheritance scopes

<table>
<thead>
<tr>
<th>Scope</th>
<th>Inheritance type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>private</td>
</tr>
<tr>
<td>private</td>
<td>(n/a)</td>
</tr>
<tr>
<td>protected</td>
<td>private</td>
</tr>
<tr>
<td>public</td>
<td>private</td>
</tr>
</tbody>
</table>

- Columns: inheritance types
- Rows: scopes of member variables/methods of parent class
- Results: scopes of member variables/methods of derived class
Constructors and destructors in inheritance

- Constructors and destructors are not inherited by child classes
- Each constructor of child class must call one constructor of parent, or implicitly the default constructor is called if exists

```cpp
class Pet {
public:
    Pet() {...}
    Pet(string name) {...}
};

class Dog: public Pet {
public:
    Dog() {...} // Pet()
    Dog(string name): Pet(name) {...}
};

class Bird {
public:
    Bird(bool canFly) {...}
};

class Eagle: public Bird {
public:
    // wrong: Eagle() {...}
    Eagle(): Bird(true) {...}
};
```

- Destructors are called automatically in the reverse order of inheritance
  - `~Dog() → ~Pet()`
  - `~Eagle() → ~Bird()`
Calling cons of parent from cons of children

- Impossible to call constructors of parent class from constructors of child classes like functions, but via the so-called initialization list

```cpp
class Point3D: private Point2D {
    protected:
        double z;
    public:
        Point3D(): Point2D(0., 0.), z(0.) // correct
            { ... }
        Point3D(double x, double y, double z)
            // implicitly call default cons Point2D()
            {
                Point2D(x, y); // incorrect: creation of a temporary instance of Point2D
                this->z = z;
            }
        ...}
};
```
Virtual methods

- Are methods declared in parent class, but can be redefined (or replaced) in derived classes

```cpp
class Shape {
public:
    virtual void draw() {
        cout << "Shape::draw\n";
    }
    void erase() {
        cout << "Shape::erase\n";
    }
    void redraw() {
        erase(); draw();
    }
};

class Circle: public Shape {
public:
    virtual void draw() {
        cout << "Circle::draw\n";
    }
    void erase() {
        cout << "Circle::erase\n";
    }
};

void main() {
    Circle c;
    Shape s1 = c;
    Shape& s2 = c;
    Shape* s3 = &c;

    c.erase();
    c.draw();
    s1.erase(); s1.draw();
    s2.erase(); s2.draw();
    s3->erase(); s3->draw();
    c.redraw();
    s1.redraw();
    s2.redraw();
    s3->redraw();
}
```

Result:
- Circle::erase
- Circle::draw
- Shape::erase
- Shape::draw
- Circle::draw
- Shape::erase
- Circle::draw
- Shape::erase
- Circle::draw
- Circle::draw
- Shape::erase
- Circle::draw
Abstract classes

- Pure virtual methods: are methods declared but not implemented → need to be implemented in the derived classes
- Abstract classes are classes possessing a least one pure virtual method
  - Impossible to instantiate abstract classes

```cpp
class Shape {
public:
    virtual void draw() = 0;
    virtual void erase() = 0;
    virtual void area() = 0;
    void redraw() { ... }
};

class Circle: public Shape {
public:
    ... 
    virtual void draw() { ... }
    virtual void erase() { ... }
}
```

```cpp
virtual void area() { ... }

Shape p; // error
Circle c;
Shape p2 = c; // error
Shape& p3 = c; // OK
Shape* p4 = &c; // OK

void func(Shape s) {...} // error
void func(Shape& s) {...} // OK
void func(Shape* s) {...} // OK
```
Polymorphism

- Inheritance and virtual methods facilitate the work with objects: possible to call the right method without need to know the real class of object (compared to using “switch” and function pointers in C)

- One of the 3 pillars of OOP: encapsulation, inheritance, polymorphism

```cpp
class Pet {
public:
    virtual void say() = 0;
};

class Cat: public Pet {
public:
    virtual void say()
    {
        cout << "miao\n";
    }
};

class Dog: public Pet {
public:
    virtual void say()
    {
        cout << "gruh\n";
    }
};

Pet* p[3] = {
    new Dog(), new Cat(), new Cat() 
};

for (int i=0; i<3; i++)
    p[i]->say();

// This is not correct:

Result:
gruh
miao
miao
Virtual destructors

```cpp
class ClassA {
public:
    ClassA() { ... }
    virtual ~ClassA() { ... }
};

class ClassB: public ClassA {
public:
    ClassB() { ... }
    virtual ~ClassB() { ... }
};

ClassB* b = new ClassB;
ClassA* a = (ClassA*)new ClassB;
delete b; // ~ClassB, ~ClassA
delete a; // ~ClassA
```

- Should declare destructors as virtual in almost every cases
Expression in memory

```cpp
#pragma pack(1)

class V2 {
public:
    double x, y;
    static int i;
    void f2();
    virtual void fv2();
};

class V3: public V2 {
public:
    double z;
    void f3();
    virtual void fv2();
    virtual void fv3();
};

V3 v3;
V2& v2 = v3;

printf("%d %d\n", &v2, sizeof(v2));
printf("%d %d\n", &v3, sizeof(v3));
printf("%d %d %d\n", &v3.x, &v3.y, &v3.z);
```

### Result:
- `1245000 20`
- `1245000 28`
- `1245004 1245012 1245020`

<table>
<thead>
<tr>
<th>Member</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>vtable</td>
<td>4</td>
</tr>
<tr>
<td>x</td>
<td>8</td>
</tr>
<tr>
<td>y</td>
<td>8</td>
</tr>
<tr>
<td>z</td>
<td>8</td>
</tr>
</tbody>
</table>

- Static variables not included in object memory
- If class has virtual methods, there is a pointer (vtable) to a table of virtual methods → virtual methods are similar to function pointers
- Data of child class is appended next to data of parent class
- Pay attention to data alignment
Multiple inheritance

- C++ allows a class inheriting from multiple classes

```cpp
class Camera {
public:
    void takePicture();
    ...
};

class FMDevice {
public:
    void turnOn();
    void turnOff();
    void setFreq(float f);
    ...
};

class Phone {
public:
    void call(string num);
    ...
};

class CellPhone:
    public Camera,
    protected FMDevice,
    public Phone
{
public:
    void turnFMOn();
    void turnFMOff();
    void setFMFreq(float f);
    ...
};

CellPhone p;
p.takePicture();
p.turnOn();  // error
p.turnFMOn();
p.call("0912345678");
```
Members of the same name

- Multiple inheritance makes class hierarchy become complicated and hard to control member variables/methods → only use when really needed.
Expression of multiple inheritance in memory

class B1 {...}
class B2 {...}
class D: public B1, public B2 {...}

D d;
B1 & b1 = d;
B2 & b2 = d;
printf("%d %d\n", &d, sizeof(d));
printf("%d %d\n", &b1, sizeof(b1));
printf("%d %d\n", &b2, sizeof(b2));

Result:
1244996 32
1244996 20
1245016 8

- Members of base classes stay next to each others in memory
- Virtual inheritance: self-study
Problems

1. Define a Shape structure **in C** then write draw(), area() functions to for circles, squares, rectangles. Do this in two ways: using “switch”, using function pointers. Compare with the solution in C++.

2. Is it safe to save an object to file then restored later in the following manner? Try to run program and explain.
   - saving: fwrite((void*)&obj, 1, sizeof(obj), file);
   - restoring: fread((void*)&obj, 1, sizeof(obj), file);

3. Define an abstract class Shape and derive Circle, Square, Rectangle, Ellipse, Sphere classes. Design the hierarchy for your convenience.

4. Complete Employee, Worker, Manager, Director classes and write a demo program.

5. Extend problem 4 by:
   - adding Company class that hold information of all staffs
   - adding relationship to staffs. Ex: each Worker has a Manager,...

6. Write B1, B2 and D classes in the slides of multiple inheritance, then check for the size and address of member variables compared to those of the object.