Bài 9: **Kế thừa** (***Inheritance***)

Khái niệm

- Để quản lý nhân sự của công ty, ta có thể định nghĩa các lớp tương ứng với các vị trí làm việc của công ty:

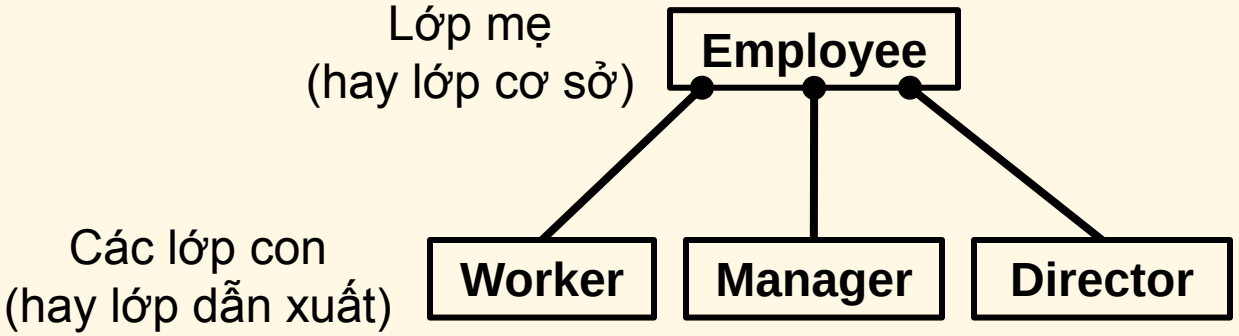
```
class Worker {
private:
   string name;
   float salary;
   int level;
public:
   string getName() {...}
   void pay() {...}
   void doWork() {...}
   ...
};
```

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

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

- Cả 3 lớp trên đều có những biến và hàm giống hệt nhau về nội dung → tạo ra một lớp Employee chứa các thông tin chung đó để sử dụng lại
  - Sử dụng lại code
  - Giảm số code cần viết
  - Dễ bảo trì, sửa đổi về sau
  - Rõ ràng hơn về mặt logic trong thiết kế chương trình

Khái niệm *(tiếp)*



- Hai hướng thừa kế:
  - Cụ thể hoá: lớp con là một trường hợp riêng của lớp mẹ (như ví dụ trên)
  - Tổng quát hoá: mở rộng lớp mẹ (vd: Point2D thêm biến z để thành Point3D)
- Kế thừa cho phép các lớp con sử dụng các biến và phương thức của lớp mẹ như của nó, trừ các biến và phương thức private
- Kế thừa với public và private:
  - public: các thành phần public của lớp mẹ vẫn là public trong lớp con
  - private: toàn bộ các thành phần của lớp mẹ trở thành private của lớp con

Kế thừa public

```
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; // lỗi
  }
};

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

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

- Các thành phần public của lớp mẹ vẫn là public trong lớp con
- Lớp con chuyển kiểu được thành lớp mẹ, nhưng ngược lại không được

Kế thừa private

```
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); // lỗi
s.getTail();      // lỗi
```

- Tất cả các thành phần của lớp mẹ đều trở thành private của lớp con

Thành phần protected

- Ngoài public và private, còn có các thành phần protected: có thể được sử dụng bởi các phương thức trong lớp dẫn xuất từ nó, nhưng không sử dụng được từ ngoài các lớp đó

```
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";     // lỗi
cout << w.getSalary(); // lỗi
```

Tổng kết các kiểu kế thừa

		Kiểu kế thừa		
		private	**protected**	**public**
Phạm vi	**private**	(không)	(không)	(không)
	protected	private	protected	protected
	public	private	protected	public

- Cột: các kiểu kế thừa
- Hàng: phạm vi các biến/phương thức thành phần trong lớp mẹ

- Kết quả: phạm vi các biến/phương thức trong lớp dẫn xuất

Constructor và destructor trong kế thừa

- Constructor và destructor không được các lớp con thừa kế
- Mỗi constructor của lớp dẫn xuất phải gọi một constructor của lớp mẹ, nếu không sẽ được ngầm hiểu là gọi constructor mặc định

```
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:
  // sai: Eagle() {...}
  Eagle(): Bird(true) {...}
};
```

- Destructor của các lớp sẽ được gọi tự động theo thứ tự ngược từ lớp dẫn xuất tới lớp cơ sở
  - `~Dog()` → `~Pet()`
  - `~Eagle()` → `~Bird()`

Gọi cons của lớp mẹ trong cons của lớp con

- Không thể gọi cons của lớp mẹ trong cons của lớp con như hàm, mà phải gọi ở danh sách khởi tạo
  -
```
class Point3D: private Point2D {
protected:
  float z;
public:
   Point3D(): Point2D(0., 0.), z(0.)  // đúng
      { ... }
   Point3D(double x, double y, double z)
      // gọi cons mặc định Point2D()
   {
     Point2D(x, y); // sai: tạo đối tượng Point2D tạm
     this->z = z;
   };
   ...
};
```

Phương thức ảo (virtual method)

- Là phương thức được khai báo ở lớp mẹ, nhưng có thể được định nghĩa lại (thay thế) ở các lớp dẫn xuất

```
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\
};
```

bắt buộc

có thể bỏ

```
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();
```

Kết quả chạy:

```
Circle::erase
Circle::draw
Shape::erase
Shape::draw
Shape::erase
Circle::draw
Shape::erase
Circle::draw
Shape::erase
Circle::draw
Shape::erase
Shape::draw
Shape::erase
Circle::draw
Shape::erase
Circle::draw
```

Lớp trừu tượng (abstract class)

- Phương thức ảo thuần tuý (pure virtual method): là phương thức được khai báo nhưng chưa được định nghĩa → cần được định nghĩa trong các lớp dẫn xuất
- Lớp trừu tượng là lớp có phương thức ảo thuần tuý
  - Không thể tạo được đối tượng từ lớp trừu tượng

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

Shape p;          // lỗi

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

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

Tính đa hình (polymorphism)

- Thừa kế và định nghĩa các hàm ảo giúp quản lý đối tượng dễ dàng hơn: có thể gọi đúng phương thức mà không cần quan tâm tới lớp thực sự của nó là gì (trong C phải dùng switch hoặc con trỏ hàm)

```
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();
// ...

// Thế này không được:
// Pet p2[2] = { Dog(), Cat() };
// ...
```

Kết quả chạy:

```
gruh
miao
miao
```

Destructor ảo

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

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

ClassB* b = new ClassB;
ClassA* a = (ClassA*)new ClassB;

delete b;  // ~ClassB, ~ClassA
delete a;  // ~ClassA
```

```
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;  // ~ClassB, ~ClassA
```

- Nên luôn khai báo destructor ảo nếu không có gì đặc biệt

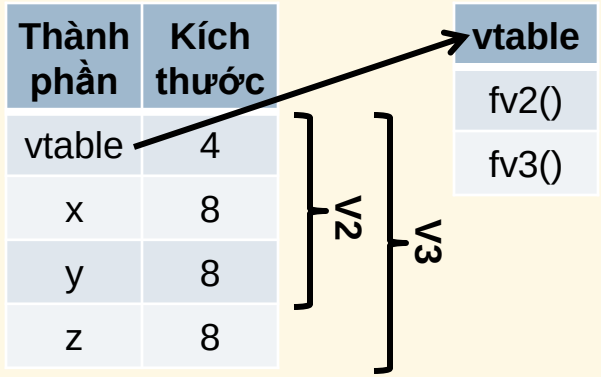
Biểu diễn trong bộ nhớ

```
#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);
```

Kết quả chạy:

```
1245000 20
1245000 28
1245004 1245012 1245020
```



- Dữ liệu static không nằm trong đối tượng
- Nếu lớp có phương thức ảo, thêm một con trỏ (vtable) tới một bảng các phương thức ảo → phương thức ảo tương tự như con trỏ hàm
- Dữ liệu của lớp con sẽ được nối tiếp vào sau dữ liệu của lớp mẹ
- Chú ý việc chỉnh biên dữ liệu (data alignment)

Đa kế thừa (kế thừa nhiều lớp)

- C++ cho phép một lớp có thể kế thừa từ nhiều lớp khác nhau

```
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();   // lỗi
p.turnFMOn();
p.call("0912345678");
```

Thành phần trùng tên

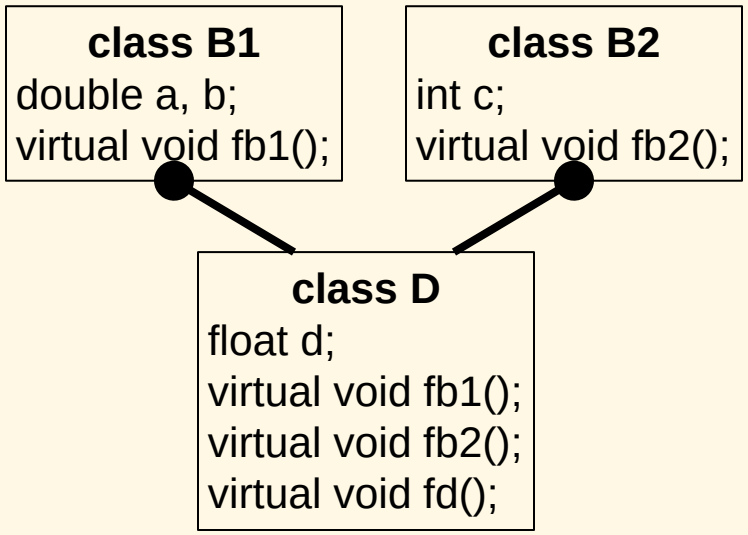
```
class Legged {
public:
  void move() { ... }
};

class Winged {
public:
  void move() { ... }
};

class Pigeon: public Legged,
  public Winged {
  ...
};

Pigeon p1;
p1.move();      // lỗi
```

```
p1.Legged::move();     // Legged
p1.Winged::move();     // Winged
((Legged&)p1).move(); // Legged
((Winged&)p1).move(); // Winged


class Penguin: public Legged,
  public Winged {
public:
  void move() { Legged::move(); }
  ...
};

Penguin p2;
p2.move();             // Penguin
((Legged&)p2).move(); // Legged
((Winged&)p2).move(); // Winged
```

- Đa kế thừa có thể khiến chương trình trở nên rất phức tạp và khó kiểm soát các biến/phương thức thành phần → chỉ nên sử dụng khi thực sự cần thiết

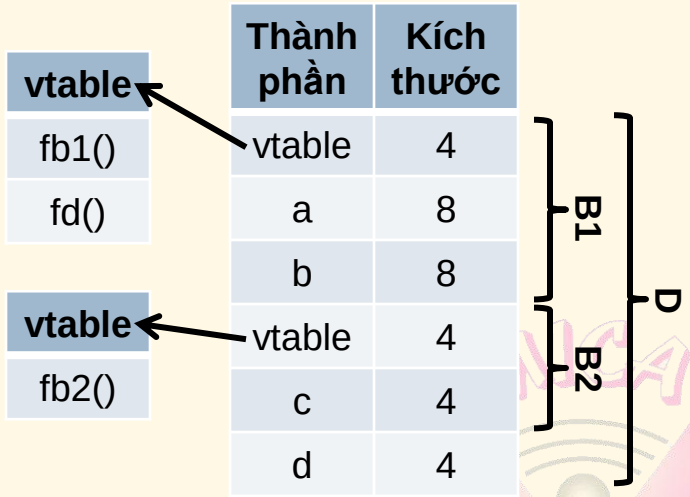
Biểu diễn đa kế thừa trong bộ nhớ

```
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));
```



Kết quả chạy:

```
1244996 32
1244996 20
1245016 8
```



- Các thành phần của các lớp cơ sở nằm nối tiếp nhau trong bộ nhớ
- Lớp kế thừa ảo: tự tìm hiểu thêm

Bài tập

1. Định nghĩa kiểu struct Shape trong C rồi viết các hàm draw(), area() tuỳ theo dạng hình: tròn, vuông, chữ nhật. Dùng hai cách làm: dùng switch, con trỏ hàm. So sánh với cách làm trong C++.
2. Với điều kiện nào thì có thể lưu một đối tượng ra file rồi đọc lại trong lần chạy sau như dưới đây? Giải thích và chạy thử.
   - lần chạy trước: `fwrite((void*)&obj, 1, sizeof(obj), file);`
   - lần chạy sau: `fread((void*)&obj, 1, sizeof(obj), file);`
3. Viết các lớp Shape (trừu tượng) và Circle, Square, Rectangle, Ellipse, Sphere. Hãy thiết kế việc kế thừa sao cho hợp lý.
4. Hoàn tất các lớp Employee, Worker, Manager, Director và viết một chương trình thử.
5. Mở rộng và sửa bài tập trên:
   - Thêm lớp Company chứa toàn bộ các nhân viên
   - Thêm quan hệ về công việc giữa các nhân viên. VD: mỗi Worker có 1 Manager,...
6. Viết các lớp B1, B2 và D trong phần đa kế thừa rồi kiểm tra kích thước các kiểu và địa chỉ các thành phần so với địa chỉ của đối tượng.