

CHƯƠNG 4

Kế thừa

Nội dung

- Giới thiệu
- Kế thừa đơn
- Phạm vi truy xuất
- Khởi động và dọn dẹp

Giới thiệu

- Kế thừa là một đặc điểm của ngôn ngữ dùng để biểu diễn mối quan hệ đặc biệt giữa các lớp. Các lớp được trừu tượng hóa và tổ chức thành một sơ đồ phân cấp lớp.
- Kế thừa là một cơ chế trừu tượng hóa. Thủ tục và hàm là cơ chế trừu tượng hóa cho giải thuật, record và struct là trừu tượng hóa cho dữ liệu. Khái niệm lớp trong C++, kết hợp dữ liệu và thủ tục để được kiểu dữ liệu trừu tượng với giao diện độc lập với cài đặt và cho người sử dụng cảm giác thoải mái như kiểu dữ liệu có sẵn

Giới thiệu

- Kế thừa là một mức cao hơn của trừu tượng hóa, cung cấp một cơ chế gom chung các lớp có liên quan với nhau thành một mức khái quát hóa đặc trưng cho toàn bộ các lớp nói trên. Các lớp với các đặc điểm tương tự nhau có thể được tổ chức thành một sơ đồ phân cấp kế thừa. Lớp ở trên cùng là trừu tượng hóa của toàn bộ các lớp ở bên dưới nó.

Giới thiệu

- Quan hệ là 1: Kế thừa được sử dụng thông dụng nhất để biểu diễn quan hệ là một.
 - Một sinh viên là một người
 - Một hình tròn là một hình ellipse
 - Một tam giác là một đa giác
- Kế thừa tạo khả năng xây dựng lớp mới từ lớp đã có, trong đó hàm thành phần được thừa hưởng từ lớp cha. Trong C++, kế thừa còn định nghĩa sự tương thích, nhờ đó ta có cơ chế chuyển kiểu tự động.

Giới thiệu

- Kế thừa vừa có khả năng tạo cơ chế khái quát hoá vừa có khả năng chuyên biệt hoá.
- Kế thừa cho phép tổ chức các lớp chia sẻ mã chương trình chung nhờ vậy có thể dễ dàng sửa chữa, nâng cấp hệ thống.
- Kế thừa thường được dùng theo hai cách: (1) Phản ánh mối quan hệ giữa các lớp và (2) Chia sẻ chi tiết cài đặt của các lớp.

Giới thiệu

- Để phản ánh mối quan hệ giữa các lớp. Là công cụ để tổ chức và phân cấp lớp dựa vào sự chuyên biệt hóa, trong đó một vài hàm thành phần của lớp con là phiên bản hoàn thiện hoặc đặc biệt hoá của phiên bản ở lớp cha. Trong C++ mối quan hệ này thường được cài đặt sử dụng:
 - Kế thừa public.
 - Hàm thành phần là phương thức ảo

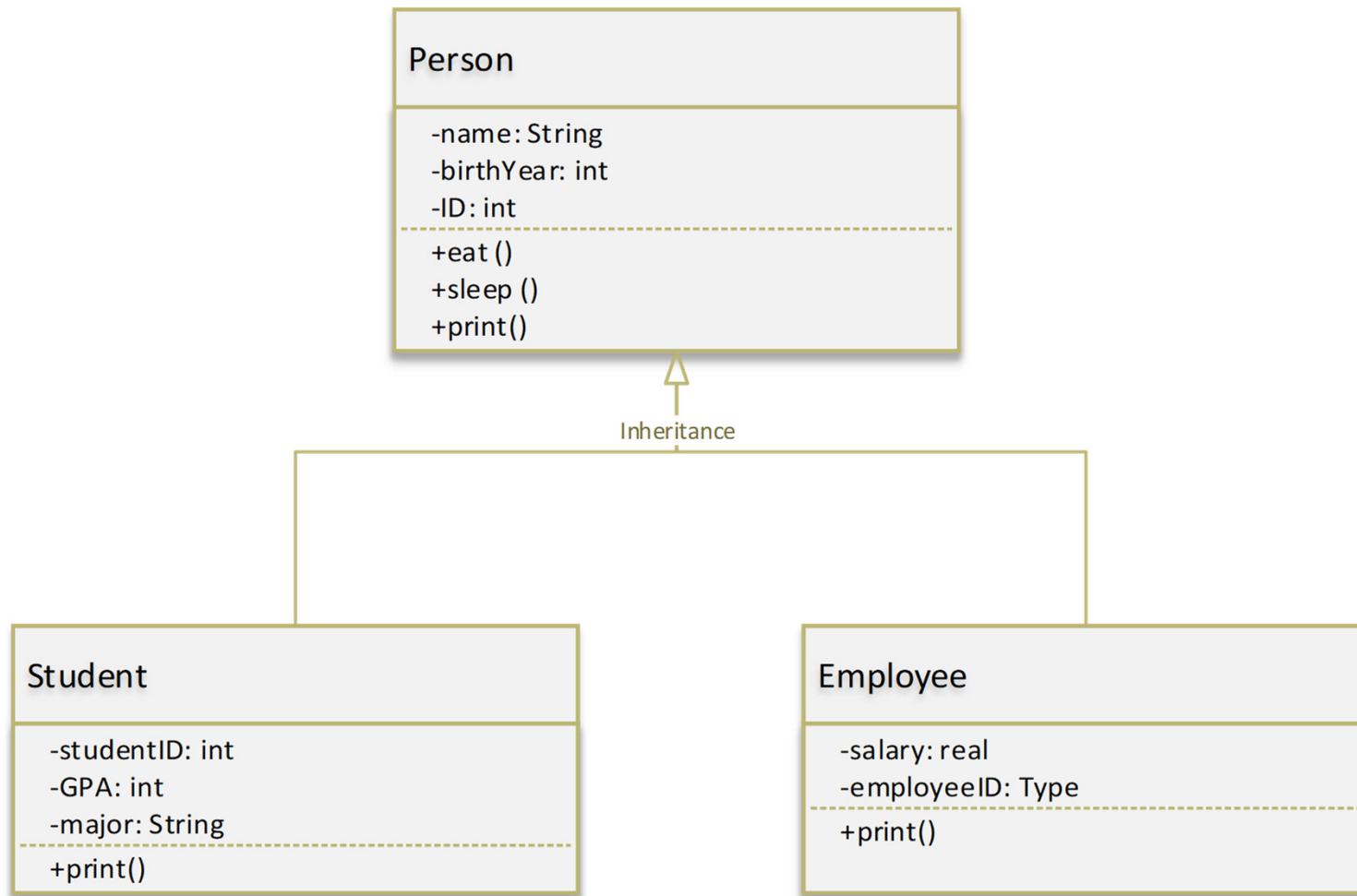
Giới thiệu

- Để phản ánh sự chia sẻ mã chương trình giữa các lớp không có quan hệ về mặt ngữ nghĩa nhưng có thể có tổ chức dữ liệu và mã chương trình tương tự nhau. Trong C++, cơ chế chia sẻ mã này thường được cài đặt dùng:
 - Kế thừa private.
 - Hàm thành phần không là phương thức ảo.

Kế thừa đơn

- Kế thừa đơn có được dùng để thiết lập mối quan hệ là một:
 - Một sinh viên *là một* người
 - Một người lao động *là một* người
 - Một con cọp *là một* động vật
 - Một hình tròn *là một* hình ellipse
 - Một xe lửa *là một* phương tiện di chuyển.

Kế thừa đơn



Kế thừa đơn – Ví dụ mở đầu

- Xét hai khái niệm người và sinh viên với mối quan hệ tự nhiên: một *sinh viên là một người*. Trong một ngôn ngữ lập trình hướng đối tượng như C++, Java hay Objective C, ta có thể biểu diễn khái niệm trên: một sinh viên là một người có thêm một số thông tin và một số thao tác (riêng biệt của sinh viên).
- Ta tổ chức lớp sinh viên kế thừa từ lớp người. Lớp người được gọi là lớp cha (superclass) hay lớp cơ sở (base class). Lớp sinh viên được gọi là lớp con (subclass) hay lớp dẫn xuất (derived class).

Ví dụ mở đầu (C++)

```
class Person
{
    friend class Student;
    char *name;
    int birthYear;
    int ID;
public:
    Person(const char *n, int y, int id) :
        birthYear(y), ID(id) { name = __strdup(n); }
    ~Person() { delete[] name; }
    void eat() const { cout << name << " eats 3 bowls
of rice"; }
    void sleep() const { cout << name << " sleeps 8
hours of rice"; }
    void print() const;
    friend ostream& operator << (ostream &os, Person&
p);
};
```

Ví dụ mở đầu (C++)

```
class Student : public Person
{
    char *major;
    int studentID;
    double GPA;
public:
    Student(const char *n, int y, int id, const char *m,
            int stId, double g) : Person(n, y, id),
        studentID(stId), GPA(g) { major = __strdup(m); }
    ~Student() { delete[] major; }
    void print() const;
};

ostream& operator << (ostream &os, Person& p)
{
    return os << "Person, name: " << p.name << ", birth
year: " << p.birthYear << ", ID: " << p.ID;
}
```

cont...

```
ostream& operator << (ostream &os, Person& p)
{
    return os << "Person, name: " << p.name << ", birth
    year: " << p.birthYear << ", ID: " << p.ID;
}

void Person::print() const
{
    cout << "Person, name: " << name << ", Identity: " <<
    ID << ", birth year: " << birthYear;
}

void Student::print() const
{
    // Person::print();
    // cout << ", student Identity: " << studentID << ",
    major: " << major << "GPA: " << GPA;
    cout << "Student, name: " << name << ", id: " <<
    studentID << ", major: " << major;
}
```

Ví dụ mở đầu (C++)

```
int main()
{
    Person p1("Le Van Nhan", 1980, 20978666);
    Student s1("Vo Vien Sinh", 2000, 20978662, "Toan Ung
    Dung", 180194, 9.2);

    cout << "1.\n";
    p1.eat(); cout << "\n";
    s1.eat(); cout << "\n";

    cout << "2.\n";
    p1.print(); cout << "\n";
    s1.print(); cout << "\n";
    s1.Person::print(); cout << "\n";

    cout << "3.\n";
    cout << p1 << "\n";
    cout << s1 << "\n";
}
```

Tự động kế thừa các đặc tính của lớp cha

Khai báo

```
class Student : public Person
{
    // ...
};
```

cho biết lớp sinh viên kế thừa từ lớp người. Khi đó sinh viên được thừa hưởng các đặc tính của lớp người.

Tự động kế thừa các đặc tính của lớp cha

- Về mặt dữ liệu: Mỗi đối tượng sinh viên tự động có thành phần dữ liệu họ tên và năm sinh của người.
- Về mặt thao tác: Lớp sinh viên được tự động kế thừa các thao tác của lớp cha. Đây chính là khả năng sử dụng lại mã chương trình.
- Riêng phương thức thiết lập không được kế thừa.

Ví dụ mở đầu (C++)

```
int main()
{
    Person p1("Le Van Nhan", 1980, 20978666);
    Student s1("Vo Vien Sinh", 2000, 20978662,
    "Toan Ung Dung", 180194, 9.2);

    p1.eat(); cout << "\n";
    s1.eat(); cout << "\n";    // From class Person

    p1.print(); cout << "\n";
    s1.print(); cout << "\n";
    s1.Person::print(); cout << "\n";
    // ...
}
```

Tự động kế thừa các đặc tính của lớp cha

- Kế thừa public như trên hàm ý rằng một đối tượng sinh viên là một đối tượng người. Bất kỳ nơi nào chờ đợi một đối tượng thuộc lớp cha (vd: người) có thể đưa vào đó một đối tượng thuộc lớp con (vd:sinh viên). Đó chính là cơ chế chuyển kiểu tự động.
- Khả năng thừa hưởng các thao tác của lớp cơ sở có thể được truyền qua vô hạn mức.

Định nghĩa lại đặc tính ở lớp con

- Lớp con có thể định nghĩa lại (override) các đặc tính đã có ở lớp cha, việc định nghĩa chủ yếu là phương thức, bằng cách khai báo giống hệt như ở lớp cha.
- Việc định nghĩa lại thao tác ở lớp con được thực hiện khi thao tác ở lớp con khác thao tác ở lớp cha. Thông thường là các thao tác xuất, nhập.

```

class Student : public Person
{
    char *major;
    int studentID;
    double GPA;
public:
    Student(const char *n, int y, int id, const char *m,
            int stId, double g) : Person(n, y, id),
        studentID(stId), GPA(g) { major = __strdup(m); }
    ~Student() { delete[] major; }
    void print() const;
                // Override Person::print
};

void Student::print() const
{
    cout << "Student, name: " << name << ", id: " <<
    studentID << ", major: " << major;
}

```

Định nghĩa lại đặc tính ở lớp con

- Ta cũng có thể định nghĩa lại thao tác ở lớp con trong trường hợp giải thuật ở lớp con đơn giản hơn (tô màu đa giác, tính modun của số ảo...).

```
class Polygon {  
    // ...  
public:  
    void draw() const;  
    void fill() const;  
};  
class Rectangle : public Polygon {  
public:  
    void fill() const;  
}
```

Định nghĩa lại đặc tính ở lớp con

- Hoặc thao tác ở lớp con không có tác dụng:

```
class Ellipse
{
public:
    void rotate(double rotangle) { /* rotate */}
};
```

```
class Circle:public Ellipse
{
public:
    void rotate(double rotangle){/*do nothing*/}
};
```

Ràng buộc ngữ nghĩa ở lớp con

- Kế thừa có thể được áp dụng cho quan hệ mang ý nghĩa ràng buộc, đối tượng ở lớp con là đối tượng ở lớp cha nhưng có dữ liệu bị ràng buộc.
 - Hình tròn là Ellipse ràng buộc bán kính ngang dọc bằng nhau.
 - Số ảo là số phức ràng buộc phần thực bằng 0.
 - Hình vuông là hình chữ nhật ràng buộc hai cạnh ngang và dọc bằng nhau...
- Trong trường hợp này, các hàm thành phần phải bảo đảm sự ràng buộc dữ liệu được tôn trọng.

Ràng buộc ngữ nghĩa ở lớp con

```
class CEllipse: public CShape
{
    Point T;
    double rx, ry;
public:
    CEllipse(double x, double y, double a, double b)
        :T(x,y), rx(a), ry(b) {}
    void Move(double dx, double dy) { T.Move(dx, dy); }
    void Scale(double s) { rx *= s; ry *= s; }
    void xScale(double s) { rx *= s; }
    void yScale(double s) { ry *= s; }
};
```

Ràng buộc ngữ nghĩa ở lớp con

```
class CCircle : public CEllipse
{
public:
    CCircle(double x, double y, double r):
        CEllipse(x, y, r, r) {}
    void xScale(double s) { Scale(sqrt(s)); }
    void yScale(double s) { Scale(sqrt(s)); }
};
// ...
CCircle c(100, 200, 50);
c.xScale(1.5)
```

Phạm vi truy xuất

- Khi thiết lập quan hệ kế thừa, ta vẫn phải quan tâm đến tính đóng gói và che dấu thông tin. Điều này dẫn đến vấn đề xác định ảnh hưởng của kế thừa đến phạm vi truy xuất các thành phần của lớp. Hai vấn đề được đặt ra là:
 - Truy xuất theo chiều dọc
 - Truy xuất theo chiều ngang

Phạm vi truy xuất

- *Truy xuất theo chiều dọc*: Hàm phương thức của lớp con có quyền truy xuất các thành phần riêng tư của lớp cha hay không? Vì chiều truy xuất là từ lớp con, cháu lên lớp cha nên ta gọi là truy xuất theo chiều dọc.
- *Truy xuất theo chiều ngang*: Các đặc tính (thuộc tính và phương thức) của lớp cha, kế thừa xuống lớp con, thì từ bên ngoài có quyền truy xuất thông qua đối tượng của lớp con hay không? Trong trường hợp này, ta gọi là truy xuất theo chiều ngang.

Truy xuất theo chiều dọc

- Lớp con có quyền truy xuất các thành phần của lớp cha hay không, hay tổng quát hơn, *phần chương trình* nào có quyền truy xuất các thành phần của lớp cha, hoàn toàn do lớp cha quyết định. Điều đó được xác định bằng *thuộc tính truy xuất*.
- Trong trường hợp lớp sinh viên kế thừa từ lớp người, truy xuất theo chiều dọc có nghĩa liệu lớp sinh viên có quyền truy xuất các thành phần họ tên, năm sinh của lớp người hay không. Chính xác hơn một đối tượng sinh viên có quyền truy xuất họ tên của chính mình nhưng được khai báo ở lớp người hay không?

Truy xuất theo chiều dọc

- *Thuộc tính truy xuất* của một là đặc tính của một thành phần của lớp cho biết những nơi nào có quyền truy xuất thành phần đó.
- Thuộc tính *public*: Thành phần nào có thuộc tính *public* thì có thể được truy xuất từ bất cứ nơi nào trong chương trình (từ sau khai báo lớp).
- Thuộc tính *private*: Thành phần nào có thuộc tính *private* thì nó là riêng tư của lớp đó. Chỉ có các hàm thành phần của lớp và ngoại lệ là các hàm bạn được phép truy xuất, ngay cả các *lớp con* cũng *không* có quyền truy xuất.

```

class Person
{
    char *name;
    int birthYear;
    int ID;
public:
    Person(const char *n, int y, int id) :
        birthYear(y), ID(id) { name = __strdup(n); }
    ~Person() { delete[] name; }
    void print() const;
};

void Person::print() const
{
    cout << "Person, name: " << name << ", Identity:
    " << ID << ", birth year: " << birthYear;
}

```

```

class Student : public Person
{
    char *major;
    int studentID;
    double GPA;
public:
    Student(const char *n, int y, int id, const char *m,
            int stId, double g) : Person(n, y, id),
        studentID(stId), GPA(g) { major = __strdup(m); }
    ~Student() { delete[] major; }
    void print() const; // Override Person::print
};

void Student::print() const
{
    cout << "Student, name: " << name << ", id: " <<
    studentID << ", major: " << major;
}

```

Error: Cannot
access name

Truy xuất theo chiều dọc

- Trong ví dụ trên, không có hàm thành phần nào của lớp sinh viên (Student) có thể truy xuất các thành phần riêng tư như họ tên, năm sinh của lớp Người (Person). Nói cách khác, lớp con không có quyền vi phạm tính đóng gói của lớp cha. Đoạn chương trình trên gây ra lỗi lúc biên dịch.
- Ta có thể khắc phục được lỗi trên nhờ khai báo lớp Student là bạn của lớp Person.

```
class Person
```

```
{
```

```
    friend class Student;
```

```
    char *name;
```

```
    int birthYear;
```

```
    int ID;
```

```
public:
```

```
    Person(const char *n, int y, int id) : birthYear(y),  
    ID(id) { name = __strdup(n); }
```

```
    ~Person() { delete[] name; }
```

```
    void print() const;
```

```
};
```

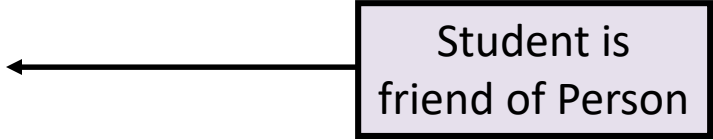
```
void Person::print() const
```

```
{
```

```
    cout << "Person, name: " << name << ", Identity: " <<  
    ID << ", birth year: " << birthYear;
```

```
}
```

Student is
friend of Person



```

class Student : public Person
{
    char *major;
    int studentID;
    double GPA;
public:
    Student(const char *n, int y, int id, const char *m,
            int stId, double g) : Person(n, y, id),
        studentID(stId), GPA(g) { major = __strdup(m); }
    ~Student() { delete[] major; }
    void print() const; // Override Person::print
};

void Student::print() const
{
    cout << "Student, name: " << name << ", id: " <<
    studentID << ", major: " << major;
}

```

Ok, name can
be accessed

Truy xuất theo chiều dọc

- Cách làm trên giải quyết được nhu cầu của người sử dụng khi muốn tạo lớp con có quyền truy xuất các thành phần dữ liệu private của lớp cha. Tuy nhiên nó đòi hỏi phải sửa đổi lại lớp cha và tất cả các lớp ở cấp cao hơn mỗi khi một lớp con mới ra đời.
- Ví dụ thêm các lớp nghệ sĩ, ca sĩ, sinh viên tại chức...

Truy xuất theo chiều dọc

- Cách làm trên giải quyết được nhu cầu của người sử dụng khi muốn tạo lớp con có quyền truy xuất các thành phần dữ liệu private của lớp cha. Tuy nhiên nó đòi hỏi phải sửa đổi lại lớp cha và tất cả các lớp ở cấp cao hơn mỗi khi một lớp con mới ra đời.
- Ví dụ thêm các lớp nghệ sĩ, ca sĩ, sinh viên tại chức...

Thuộc tính protected

- Thuộc tính protected là phương tiện để tránh phải sửa đổi lớp cơ sở khi có lớp con mới ra đời. Nhờ đó nó bảo được tính đóng của một lớp. Khai báo một thành phần nào có thuộc tính protected tương đương với qui định trước tất cả các lớp con, cháu sau này đều là bạn của thành phần đó.
- Thông thường ta dùng thuộc tính protected cho các thành phần dữ liệu và thuộc tính public cho hàm thành phần.

```

class Person
{
protected:
    char *name;
    int birthYear;
    int ID;
public:
    Person(const char *n, int y, int id) : birthYear(y),
    ID(id) { name = __strdup(n); }
    ~Person() { delete[] name; }
    void print() const;
};

void Person::print() const
{
    cout << "Person, name: " << name << ", Identity: " <<
    ID << ", birth year: " << birthYear;
}

```

```

class Student : public Person
{
protected:
    char *major;
    int studentID;
    double GPA;
public:
    Student(const char *n, int y, int id, const char *m,
            int stId, double g) : Person(n, y, id),
        studentID(stId), GPA(g) { major = __strdup(m); }
    ~Student() { delete[] major; }
    void print() const; // Override Person::print
};

void Student::print() const
{
    cout << "Student, name: " << name << ", id: " <<
    studentID << ", major: " << major;
}

```

Ok, name can
be accessed

Thuộc tính protected

- Các thuộc tính public, private, protected và khai báo friend cho những nơi nào có quyền truy xuất đến các thành phần của lớp. Cho hay không cho nơi nào trong chương trình được truy xuất đến (thành phần của) lớp hoàn toàn do lớp quyết định.

Truy xuất theo chiều ngang

- Thành phần (protected và public) của lớp khi đã kế thừa xuống lớp con thì bên ngoài có quyền truy xuất thông qua đối tượng thuộc lớp con hay không? Điều này hoàn toàn do lớp con quyết định bằng thuộc tính kế thừa.
- Có hai thuộc tính kế thừa là kế thừa public và kế thừa private.
- *Kế thừa public*: Lớp con kế thừa public từ lớp cha thì các thành phần protected của lớp cha trở thành protected của lớp con, các thành phần public của lớp cha trở thành public của lớp con.

Kế thừa public

- Trong kế thừa public: Mọi thao tác public của lớp cha được kế thừa xuống lớp con. Vì vậy ta có thể sử dụng thao tác của lớp cha cho đối tượng thuộc lớp con.
- Ta qui định kế thừa public bằng từ khoá public theo sau dấu hai chấm khi thiết lập quan hệ kế thừa.

```

class Student : public Person
{
protected:
    char *major;
    int studentID;
    double GPA;
public:
    Student(const char *n, int y, int id,
const char *m, int stId, double g) :
    Person(n, y, id), studentID(stId), GPA(g)
    { major = __strdup(m); }
    ~Student() { delete[] major; }
    void print() const; // Override
    Person::print
};

```

```
int main()
{
    Person p1("Le Van Nhan", 1980, 20978666);
    Student s1("Vo Vien Sinh", 2000, 20978662,
               "Toan Ung Dung", 180194, 9.2);

    p1.eat(); cout << "\n";
    s1.eat(); cout << "\n";

    p1.print(); cout << "\n";
    s1.print(); cout << "\n";
}
```

Kế thừa public

- Do được thừa hưởng các đặc tính của lớp cha nên ta dùng kế thừa public khi và chỉ khi có quan hệ là một từ lớp con đến lớp cha.
- Hầu hết các trường hợp kế thừa là kế thừa public, nó cho phép tận dụng lại mã chương trình, đồng thời tạo khả năng thu gom các đặc điểm chung của các lớp vào một lớp cơ sở (khái quát hoá), nhờ đó dễ dàng nâng cấp và sửa chữa (bảo trì).

Kế thừa private

- Có những trường hợp các lớp không có quan hệ với nhau về mặt ngữ nghĩa nhưng chia sẻ chung chi tiết cài đặt, nếu dùng kế thừa public thì sai khái niệm vì lớp con sẽ thừa hưởng các thao tác nó không có từ lớp cha.
- *Kế thừa private*: Lớp con kế thừa private từ lớp cha thì các thành phần protected và public của lớp cha trở thành private của lớp con. Nói cách khác mọi thao tác của lớp cha đều bị lớp con che dấu. Vì vậy, nhì từ bên ngoài thì lớp con không có các thao tác mà lớp cha có.

Kế thừa private

- Sử dụng kế thừa private, ta có thể chia sẻ chi tiết cài đặt (cấu trúc dữ liệu và mã chương trình) giữa các tương tự nhau nhưng vẫn giữ được tinh thần của từng lớp.
- *Ví dụ:* Lớp *List* biểu diễn khái niệm danh sách liên kết, lớp *Stack* biểu diễn khái niệm danh sách hoạt động theo nguyên tắc LIFO, lớp *Set* biểu diễn khái niệm tập hợp. Nếu tổ chức *Stack*, *Set* như danh sách liên kết, có thể dùng kế thừa private để tận dụng chi tiết cài đặt chung.


```
typedef int Item;
```

```
class Link  
{  
    friend class List;  
    friend class ListIterator;  
    Link *next;  
    Item e;  
    Link(Item a, Link *p) :e(a) { next = p; }  
};
```

```

class List
{
    friend class ListIterator;
    Link *last;
public:
    void Insert(Item a); // add at head of list
    void Append(Item a); // add at tail of list
    Item *GetFirst(); // return and remove head of list
    void CleanUp();
    List() { last = NULL; }
    List(Item a) { last = new Link(a, NULL); last->next = last; }
    ~List() { CleanUp(); }
    bool Empty() const { return last == NULL; }
    bool IsMember(Item x) const;
    int Count() const;
    void View() const;
};

```

```
class Stack : private List
{
public:
    Stack() :List() {}
    bool Push(Item x) { Insert(x); return
        true; }
    bool Pop(Item *px);
    bool Empty() const { return
        List::Empty(); }
};
```

Kế thừa private

- Các lớp Stack và Set tận dụng được cấu trúc dữ liệu và chi tiết cài đặt của lớp List, nhưng không bị trở thành List vì sử dụng kế thừa private. Các thao tác của List không bị kế thừa xuống lớp con Stack và Set.
- Một số hàm thành phần của lớp cơ sở List có thể cần thiết ở lớp con như hàm Empty trong lớp Stack, hàm Empty, IsMember, Count trong lớp Set... Ta định nghĩa lại những hàm này bằng cách gọi lại phiên bản trong lớp List.
- Một cách thay thế việc viết lại hàm như trên là khai báo lại các danh hiệu này trong phần public của lớp con bằng cách sử dụng phép toán phân giải phạm vi.

```

class Stack :private List {
public:
    Stack() :List() {}
    bool Push(Item x) { Insert(x); return true; }
    bool Pop(Item *px);
    List::Empty;
    bool Empty() const { return List::Empty(); }

};

class Set :private List {
public:
    Set() :List() {}
    void Add(Item x) { if (!List::IsMember(x)) Insert(x); }
    List::Empty;
    List::IsMember;
    List::Count;
    List::View;
};

```

Tận dụng mã chung không dùng kế thừa

- Ta dùng kế thừa private trong các trường hợp muốn tận dụng mã chương trình chung và có thể muốn kế thừa một phần nhưng không phải tất cả các thao tác của lớp cơ sở.
- Ta luôn luôn có thể tận dụng mã chương trình chung bằng kế thừa private như trên, nhưng cũng có thể không cần dùng kế thừa.

```

class Stack {
    List l;
public:
    Stack() :l() {}
    bool Push(Item x) { l.Insert(x); return true; }
    bool Pop(Item *px);
    bool Empty() const { return l.Empty(); }
};

class Set :private List {
    List l;
public:
    Set() :List() {}
    void Add(Item x) { if (!List::IsMember(x)) Insert(x); }
    bool Empty() const { return l.Empty(); }
    bool IsMember(Item x) const { return l.IsMember(x); }
    int Count() const { return l.Count(); }
    void View() const { l.View(); }
};

```

```

class Point
{
    double x, y;
public:
    Point(double xx, double yy);
    void move(double dx, double dy) { x += dx; y +=
dy; }
};

```

```

class Circle : private Point
{
    double radius;
public:
    Circle(double tx, double ty, double rr) :
    Point(tx, ty), radius(rr) {}
    Point::move;
};

```



```

class Point
{
    double x, y;
public:
    Point(double xx, double yy);
    void move(double dx, double dy) { x += dx; y += dy; }
};

class Circle
{
    double radius;
    Point center;
public:
    Circle(double tx, double ty, double rr) :center(tx, ty),
    radius(rr) {}
    void move(double dx, double dy) { center.move(dx, dy); }
};

```

Kế thừa trong Objective-C

- Trong Objective C, mọi lớp đều phải có lớp cơ sở. Lớp cơ sở có thể do người sử dụng định nghĩa, hoặc là lớp NSObject.
- Mọi lớp đều kế thừa từ lớp NSObject, trực tiếp hoặc gián tiếp.
- Lớp con mặc nhiên thừa hưởng các đặc tính của lớp cha, bao gồm cấu trúc dữ liệu và các thao tác.

Person class

```
@interface Person : NSObject
    @property (copy, nonatomic) NSString *name;
    @property int birthYear, ID;
    -(instancetype) initWithName: (NSString *)n andYear:
    (int)y andId: (int)i;
    -(void) setName: (NSString *)n andYear: (int)y andId:
    (int)i;
    -(void) eat;
    -(void) sleep;
    -(void) print;
@end
```

```
#import "Person.h"

@interface Student : Person
    @property int studentID;
    @property double GPA;
    @property (copy, nonatomic) NSString* major;
    -(instancetype)initWithName:(NSString *)n
    andYear:(int)y andId:(int)i andStId: (int)stId andGPA:
    (double) g andMajor: (NSString *) s;
    -(void) print;    // Override
@end
```

```

#import "Person.h"
@implementation Person
@synthesize name, birthYear, ID;
-(instancetype) init {
    self = [super init];
    if (self) {    name = @"To Van Ve";
        birthYear = 2000;
        ID = 209766213;
    }
    return self;
}
-(instancetype) initWithName: (NSString *) n andYear: (int)y
andId: (int)i {
    self = [super init];
    if (self) {
        name = n;
        birthYear = y;
        ID = i;
    }
    return self;
}
}

```

```
-(void) setName: (NSString *)n andYear: (int)y andId:
(int)i {
    name = n;
    birthYear = y;
    ID = i;
}

-(void) print {
    NSLog(@"%@ %d %d", name, birthYear, ID);
}

-(void) eat {
    NSLog(@"%@ eats three bowls of rice", name);
}

-(void) sleep {
    NSLog(@"%@ sleeps eight hours a day", name);
}
@end
```

```

#import "Student.h"
@implementation Student @synthesize studentID, GPA,
major;
-(instancetype)initWithName:(NSString *)n andYear:(int)y
andId:(int)i andStdId: (int)stdId andGPA: (double) g andMajor:
(NSString *) s {
self = [super initWithName: n andYear: y andId: i];
if (self) {
studentID = stdId;
GPA = g;
major = s;
}
return self;}
-(void)print { [super print]; NSLog(@"student id: %d,
GPA: %.2f, major: %@", studentID, GPA, major);} @end

```

```

#import "Student.h"
@implementation Student@synthesize studentID, GPA, major;
-(instancetype)initWithName:(NSString *)n andYear:(int)y
andId:(int)i andStId: (int)stId andGPA: (double) g
andMajor: (NSString *) s {
    self = [super initWithName: n andYear: y andId: i];
    if (self) {
        studentID = stId;
        GPA = g;
        major = s;
    }
    return self;
}
-(void)print {
    [super print];
    NSLog(@"student id: %d, GPA: %.2f, major: %@",
        studentID, GPA, major);
}
@end

```



```

#import <Foundation/Foundation.h>
#import "Student.h"

int main(int argc, const char * argv[]) {
@autoreleasepool {
    Person *n = [[Person alloc] initWithName:
@"Le Van Nhan" andYear:1991 andId:20976541];
    Student *p = [[Student alloc] initWithName:
@"Vo Vien Sinh" andYear:2000 andId:20976542
andStId: 182001 andGPA: 9.9 andMajor:
@"Computer Science"];
    [n print];
    [p print];
}
return 0;
}

```

Khởi động và dọn dẹp

- Mọi đối tượng thuộc lớp con đều chứa các thành phần dữ liệu của các lớp cơ sở. Có thể xem lớp con có các thành phần ngầm định là thành phần của lớp cơ sở.
- Khi một đối tượng thuộc lớp con được tạo ra, các thành phần thuộc lớp cơ sở cũng được tạo ra, nghĩa là phương thức thiết lập của các lớp cơ sở phải được gọi.
- Trình biên dịch tự động gọi phương thức thiết lập của các lớp cơ sở cho các đối tượng (cơ sở) nhúng vào đối tượng đang được tạo ra.

Khởi động và dọn dẹp

- Đối với phương thức thiết lập của một lớp con, công việc đầu tiên là gọi phương thức thiết lập của các lớp cơ sở.
- Nếu mọi phương thức thiết lập của lớp cơ sở đều đòi hỏi phải cung cấp tham số thì lớp con bắt buộc phải có phương thức thiết lập để cung cấp các tham số đó.
- Trong C++, cú pháp dấu hai chấm (:) được dùng để khởi động thành phần lớp cơ sở.

```
class Point
{
    double x, y;
public:
    Point(double xx, double yy);
    void move(double dx, double dy) { x += dx; y +=
dy; }
};
```

```
class Circle : private Point
{
    double radius;
public:
    Circle(double tx, double ty, double rr) :
    Point(tx, ty), radius(rr) {}
    Point::move;
};
```

```

class Student : public Person
{
protected:
    char *major;
    int studentID;
    double GPA;
public:
    Student(const char *n, int y, int id,
const char *m, int stId, double g) :
    Person(n, y, id), studentID(stId), GPA(g)
    { major = __strdup(m); }
    ~Student() { delete[] major; }
    void print() const; // Override
    Person::print
};

```

Khởi động và dọn dẹp

- Phương thức thiết lập bản sao là cần thiết trong trường hợp đối tượng có cấp phát tài nguyên (có dữ liệu luận lý).

```

class Person
{
protected:
    char *name;
    int birthYear;
    int ID;
public:
    Person(const char *n, int y, int id) :
        birthYear(y), ID(id) { name = __strdup(n); }
    Person(const Person &p):
        birthYear(p.birthYear), ID(p.ID) { name =
            __strdup(p.name); }
    ~Person() { delete[] name; }
    void print() const;
};

```

```

class Student : public Person
{
protected:
    char *major;
    int studentID;
    double GPA;
public:
    Student(const char *n, int y, int id, const
char *m, int stId, double g) : Person(n, y,
id), studentID(stId), GPA(g) { major =
__strdup(m); }
    Student(
    ~Student() { delete[] major; }
    void print() const; // Override Person::print
};

```



```

class Student : public Person
{
protected:
    char *major;
    int studentID;
    double GPA;
public:
    Student(const char *n, int y, int id, const char
    *m, int stId, double g) : Person(n, y, id),
    studentID(stId), GPA(g) { major = __strdup(m); }
    Student(const Student &s) : Person(s),
    studentID(s.studentID), GPA(s.GPA) { major =
    __strdup(s.major); }
    ~Student() { delete[] major; }
    void print() const; // Override Person::print
};

```

Khởi động và dọn dẹp

- Khi một đối tượng bị huỷ đi, phương thức huỷ bỏ của nó sẽ được gọi, sau đó phương thức huỷ bỏ của các lớp cơ sở sẽ được gọi một cách tự động. Vì vậy lớp con không cần và cũng không được thực hiện các thao tác dọn dẹp cho các thành phần thuộc lớp cha.
(Mỗi lớp cần được tôn trọng quyền tự chủ).

```

class Student : public Person
{
protected:
    char *major;
    int studentID;
    double GPA;
public:
    Student(const char *n, int y, int id, const char
    *m, int stId, double g) : Person(n, y, id),
    studentID(stId), GPA(g) { major = __strdup(m); }
    Student(const Student &s) : Person(s),
    studentID(s.studentID), GPA(s.GPA) { major =
    __strdup(s.major); }
    ~Student() { delete[] major; }
    void print() const; // Override Person::print
};

```

Khởi động trong Objective C

- Ta có thể khởi động đối tượng trong Objective C bằng cách định nghĩa lại (override) phương thức `init` của lớp cơ sở `NSObject`.

```
-(instancetype)init {  
    self = [super init];  
    if (self) {  
        numerator = 0;  
        denominator = 1; // or [self set: 0 over:  
        1];  
    }  
    return self;  
}
```

Tự khởi động 0/1
nếu quên khởi động

```
int main(int argc, const char * argv[]) {  
@autoreleasepool {  
    Fraction *a = [[Fraction alloc] init];  
    [a set : 1 over : 3];  
    Fraction *b = [[Fraction alloc] init];  
    [b set : 5 over : 6];  
    Fraction *c = [[Fraction alloc] init];  
    [a print];  
    [b print];  
    [c print];  
}  
return 0;  
}
```

Con trỏ và kế thừa

Con trỏ trong kế thừa hoạt động theo nguyên tắc sau:

- Con trỏ đến đối tượng thuộc lớp cơ sở thì có thể trỏ đến các đối tượng thuộc lớp con.
- Điều ngược lại không đúng, con trỏ đến đối tượng thuộc lớp con thì không thể trỏ đến các đối tượng thuộc lớp cơ sở.
- Ta có thể ép kiểu để con trỏ đến đối tượng thuộc lớp con có thể trỏ đến đối tượng thuộc lớp cơ sở. Tuy nhiên thao tác này có thể nguy hiểm.
- Sử dụng ép kiểu đúng cách có thể giải quyết bài toán quản lý một danh sách các đối tượng khác kiểu.

```

int main()
{
    Person n("Le Van Nhan", 1980, 20978666);
    Student s("Vo Vien Sinh", 1984, 20978662,
    "Toan Ung Dung", 180194, 9.2);
    Person *pn;
    Student *ps;
    pn = &n;
    ps = &s;
    pn = &s;
    ps = &n; // Sai
    ps = pn; // Sai
    ps = (Student *)&n; // Sai logic
    ps = (Student *)pn;
}

```

Chương 4 – Kế thừa

Q&A