CHƯƠNG 20 LỚP DẪN XUẤT

20.1 Lớp Dẫn Xuất

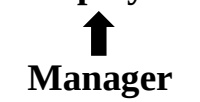
Một lớp được xây dựng thừa kế một lớp khác được gọi là lớp dẫn xuất; lớp dùng để xây dựng lớp dẫn xuất được gọi là lớp cơ sở.

Một lớp dẫn xuất ngoài các thành phần riêng của lớp đó, còn được thừa kế các thành phần của các lớp cơ sở có liên quan

Ví dụ:

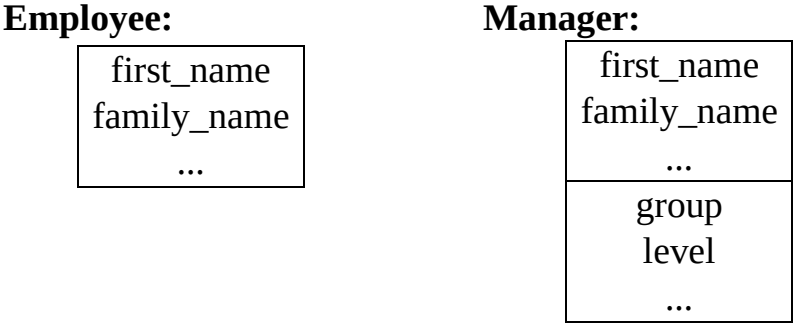
Manager có nguồn gốc từ Employee và ngược lại, Employee là một lớp cơ sở của Manager.Ngoài các thành viên của chính nó (nhóm, trình độ, v.v.), lớp Manager còn có các thành viên của lớp Employee (tên, phòng ban, v.v.).

Khởi tạo thường được biểu diễn bằng đồ thị bởi một con trỏ từ lớp dẫn xuất đến lớp cơ sở của nó cho biết rằng lớp dẫn xuất tham chiếu đến cơ sở của nó (thay vì ngược lại):



Một lớp dẫn xuất thường được cho là kế thừa các thuộc tính từ cơ sở của nó, vì vậy mối quan hệ còn được gọi là kế thừa. Một lớp cơ sở đôi khi được gọi là lớp cha và lớp dẫn xuất là lớp con. Một lớp dẫn xuất thường lớn hơn (và không bao giờ nhỏ hơn) so với lớp cơ sở của nó theo nghĩa là nó chứa nhiều dữ liệu hơn và cung cấp nhiều chức năng hơn.

Cách triển khai phổ biến và hiệu quả của khái niệm lớp dẫn xuất có một đối tượng của lớp dẫn xuất được biểu diễn như một đối tượng của lớp cơ sở, với thông tin thuộc về lớp dẫn xuất cụ thể được thêm vào cuối. Ví dụ:



Manager (cũng) là một Employee, vì vậy Manager có thể được sử dụng như một Nhân viên. Tuy nhiên, một Employee không nhất thiết phải là Manager, vì vậy một Employee không thể được sử dụng làm Manager. Nói chung, nếu một lớp dẫn xuất có một lớp cơ sở công khai, thì một dẫn xuất có thể được gán cho một biến kiểu cơ

sở mà không cần sử dụng chuyển đổi kiểu. Việc chuyển đổi ngược lại, từ cơ sở sang dẫn xuất, phải rõ ràng.Ví dụ:

```
void g(Manager mm, Employee ee){
        Employee* pe = &mm;
        pm->level = 2;
}
```

Nói cách khác, một đối tượng của một lớp dẫn xuất có thể được coi là một đối tượng của lớp cơ sở của nó khi được thao tác thông qua các con trỏ và tham chiếu.

20.1.1 Chức năngaa thành viên

Các cấu trúc dữ liệu đơn giản, chẳng hạn như Employee và Manager, thực sự không thú vị và thường không đặc biệt hữu ích. Chúng ta cần cung cấp một kiểu thích hợp với một tập hợp các thao tác phù hợp và chúng ta cần làm như vậy mà không bị ràng buộc vào các chi tiết của một biểu diễn cụ thể. Ví dụ:

```
Class NhanVien {
public:
void print () const;
string full_name () const {return first_name + '' + middle_initial + '' +
family_name;}
private:
string first_name, family_name;
char middle_initial;
// ...
};
Class Quanly: public NhanVien {
public:
void print () const;
};
```

Một thành viên của lớp dẫn xuất có thể sử dụng các thành viên công khai và được bảo vệ (xem) của một lớp cơ sở như thể chúng được khai báo trong chính lớp dẫn xuất.Ví dụ:

```
void Manager :: print () const
{
cout << "name is" << full_name () << '\ n';
}
```

Tuy nhiên, một lớp dẫn xuất không thể truy cập vào các thành viên riêng tư của một lớp cơ sở:

```
void Manager :: print () const
{
```

cout << "name is" << family_name << '\ n'; // lỗi!
}
Thông thường, giải pháp rõ ràng nhất là cho lớp dẫn xuất chỉ sử dụng các thành viên công khai của lớp cơ sở của nó. Ví dụ:
void Manager::print() const
{
Employee::print();
cout << level;
}

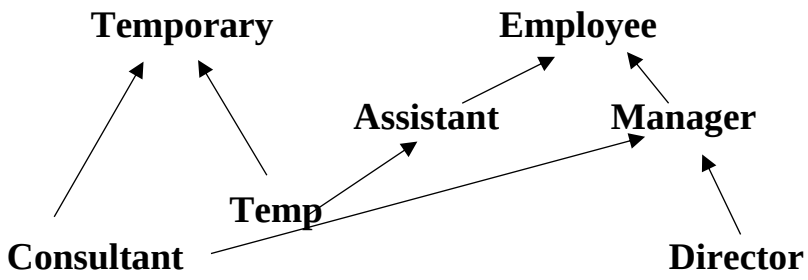
20.1.2 Hàm tạo và hàm huỷ

Như thường lệ, hàm tạo và hàm hủy là thiết yếu:
• Các đối tượng được tạo từ dưới lên (cơ sở trước thành viên và thành viên trước dẫn xuất) và hủy từ trên xuống (dẫn xuất trước thành viên và thành viên trước cơ sở);
• Mỗi lớp có thể khởi tạo các thành viên và cơ sở của nó (nhưng không trực tiếp là thành viên hoặc cơ sở của các cơ sở của nó)
• Thông thường, các hàm hủy trong một hệ thống phân cấp cần phải là ảo
• Các hàm tạo sao chép của các lớp trong một hệ thống phân cấp nên được sử dụng cẩn thận (nếu có) để tránh bị cắt;
• Độ phân giải của một lệnh gọi hàm ảo, một ép kiểu động, hoặc một kiểu trong một phương thức khởi tạo hoặc huỷ phản ánh giai đoạn khởi tạo và hủy (chứ không phải là kiểu của đối tượng chưa được hoàn thành); .
Trong văn bản nguồn, các định nghĩa của các lớp cơ sở phải xuất hiện trước các định nghĩa của các lớp dẫn xuất của chúng. Điều này ngụ ý rằng đối với các ví dụ nhỏ, các cơ sở xuất hiện phía trên các lớp dẫn xuất trên một màn hình. Ví dụ khi vẽ cây, ta nói về việc xây dựng các đối tượng từ dưới lên, ý là bắt đầu với những gì cơ bản nhất (ví dụ: các lớp cơ sở) và xây dựng những gì phụ thuộc vào nó (ví dụ: các lớp dẫn xuất). Sau đó xây dựng từ gốc (các lớp cơ sở) về phía lá (các lớp dẫn xuất).

20.2 Phân cấp lớp

Bản thân một lớp dẫn xuất có thể là một lớp cơ sở. Ví dụ:
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
Một tập hợp các lớp liên quan như vậy theo truyền thống được gọi là hệ thống phân cấp lớp. Hệ thống phân cấp như vậy thường là một dạng cây, nhưng nó cũng có thể là một cấu trúc đồ thị tổng quát hơn. Ví dụ:
class Temporary { /* ... */ };

class Assistant : public Employee { /* ... */ };
class Temp : public Temporary, public Assistant { /* ... */ };
class Consultant : public Temporary, public Manager { /* ... */ };
hoặc bằng đồ thị:



20.2.1 Type Fields
Để sử dụng các lớp dẫn xuất không chỉ là một cách viết tắt thuận tiện trong các khai báo, chúng ta phải giải quyết vấn đề sau: cho một con trỏ kiểu cơ sở, đối tượng được trỏ đến thực sự thuộc về kiểu dẫn xuất nào? Có bốn giải pháp cơ bản:
[1] Đảm bảo rằng chỉ các đối tượng của một kiểu duy nhất được trỏ tới.
[2] Đặt một trường kiểu trong lớp cơ sở để các chức năng kiểm tra.
[3] Sử dụng ép kiểu động.
[4] Sử dụng các hàm ảo.
Giải pháp [1] dựa vào kiến thức nhiều hơn là những gì có sẵn cho trình biên dịch. Nói chung, không phải là một ý kiến hay nếu cố gắng trở nên thông minh hơn hệ thống kiểu,(đặc biệt là khi kết hợp với việc sử dụng các khuôn mẫu), nó có thể được sử dụng để áp dụng- đề cập đến các vùng chứa đồng nhất (ví dụ: vectơ thư viện chuẩn và bản đồ) với hiệu suất vượt trội.
Các giải pháp [2], [3], và [4] có thể được sử dụng để xây dựng danh sách không đồng nhất, tức là danh sách (con trỏ tới) các đối tượng thuộc một số kiểu khác nhau. Giải pháp [3] là một biến thể được hỗ trợ ngôn ngữ của giải pháp [2].
Giải pháp [4] là một biến thể đặc biệt an toàn của giải pháp [2]. Sự kết hợp của các giải pháp [1] và [4] đặc biệt thú vị và mạnh mẽ; trong hầu hết mọi tình huống, chúng mang lại mã rõ ràng hơn so với các giải pháp [2] và [3].
Ví dụ về Manager /employee có thể được định nghĩa lại như sau:
struct Employee {
enum Empl_type { man, empl };
Empl_type type;
Employee() : type{empl} { }

```
string first_name , family_name;
char middle_initial;
Date hiring_date;
short depar tment;
};
struct Manager : public Employee {
Manager() { type = man; }
list<Employee*> group; // people managed
short level;
};
```

Với điều này, bây giờ chúng ta có thể viết một hàm in thông tin về mỗi Employee:

```
void print_employee(const Employee* e)
{
switch (e−>type) {
case Employee::empl:
cout << e−>family_name << '\t' << e−>department << '\n';
// ...
break;
case Employee::man:
{ cout << e−>family_name << '\t' << e−>department << '\n';
// ...
const Manager* p = static_cast<const Manager*>(e);
cout << " level " << p−>level << '\n';
// ...
break;
}
}
}
```

và sử dụng nó để in danh sách Employee như sau:

```
void print_list(const list<Employee*>& elist)
{
for (auto x : elist)
print_employee(x);
}
```

Điều này hoạt động tốt trong một chương trình nhỏ do một người duy trì. Tuy nhiên, nó có một điểm yếu cơ bản là nó phụ thuộc vào việc người lập trình thao tác các kiểu theo cách mà trình biên dịch không thể kiểm tra được. Vấn đề này thường

trở nên tồi tệ hơn vì các hàm như **print_employee ()** thường được tổ chức để tận dụng tính phổ biến của các lớp liên quan:

```
void print_employee(const Employee* e)
{
cout << e−>family_name << '\t' << e−>department << '\n';
if (e−>type == Employee::man) {
const Manager* p = static_cast<const Manager*>(e);
        cout << " level " << p−>level << '\n';
}
}
```

Có vẻ như bất kỳ dữ liệu phổ biến nào có thể truy cập được từ mọi lớp dẫn xuất, chẳng hạn như trường kiểu, sẽ thúc giục mọi người thêm nhiều dữ liệu như vậy. Do đó, cơ sở chung trở thành kho lưu trữ tất cả các loại ‘‘ thông tin hữu ích ’’. Điều này, đến lượt nó, giúp việc triển khai các lớp cơ sở và lớp dẫn xuất được đan xen vào nhau theo những cách không mong muốn nhất. Trong một hệ thống phân cấp lớp lớn, dữ liệu có thể truy cập (không phải riêng tư) trong một lớp cơ sở chung sẽ trở thành " biến toàn cục " của hệ thống phân cấp. Để thiết kế sạch sẽ và bảo trì đơn giản hơn, cần tách biệt các vấn đề riêng biệt và tránh phụ thuộc lẫn nhau.

20.2.2 Hàm ảo

Các hàm ảo khắc phục các vấn đề với giải pháp trường kiểu bằng cách cho phép lập trình khai báo các hàm trong một lớp cơ sở có thể được định nghĩa lại trong mỗi lớp dẫn xuất. Trình biên dịch và trình liên kết sẽ đảm bảo sự tương ứng chính xác giữa các đối tượng và các chức năng được áp dụng cho chúng.Ví dụ:

```
class Employee {
public:
Employee(const string& name, int dept);
virtual void print() const;
// ...
private:
string first_name , family_name;
short depar tment;
// ...
};
```

Từ khóa **virtual** chỉ ra rằng print () có thể hoạt động như một hàm print () được định nghĩa trong lớp này và các hàm print () được định nghĩa trong các lớp dẫn xuất từ nó. Các hàm print () như vậy được định nghĩa trong các lớp dẫn xuất, trình

biên dịch đảm bảo rằng print () phù hợp cho đối tượng Employee đã cho được gọi trong mỗi trường hợp.
Để cho phép một khai báo hàm ảo hoạt động như các hàm được định nghĩa trong các lớp dẫn xuất, các kiểu đối số được chỉ định cho một hàm trong lớp dẫn xuất không được khác với các kiểu đối số được khai báo trong cơ sở và chỉ cho phép những thay đổi rất nhỏ đối với kiểu trả về. Một hàm thành viên ảo đôi khi được gọi là một phương thức.
Một hàm ảo phải được định nghĩa cho lớp mà nó được khai báo lần đầu (trừ khi nó được khai báo là một hàm ảo thuần túy; xem §20.4).Ví dụ:

```
void Employee::print() const
{
cout << family_name << '\t' << department << '\n';
}
```

Một hàm ảo có thể được sử dụng ngay cả khi không có lớp nào được dẫn xuất từ lớp của nó. Khi dẫn xuất một lớp, chỉ cần cung cấp một chức năng thích hợp nếu nó cần thiết. Ví dụ:

```
class Manager : public Employee {
public:
Manager(const string& name, int dept, int lvl);
void print() const;
private:
list<Employee*> group;
short level;
};
void Manager::print() const
{
Employee::print();
cout << "\tlevel " << level << '\n';
}
```

Một hàm từ một lớp dẫn xuất có cùng tên và cùng một tập hợp các kiểu đối số như một hàm ảo trong cơ sở được cho là ghi đè phiên bản lớp cơ sở của hàm ảo. Hơn nữa, nó có thể ghi đè một hàm ảo từ một cơ sở có kiểu dẫn xuất trả về(§20.3.6).
Hàm toàn cục print_employee () (§20.3.1) bây giờ không cần thiết vì các hàm thành viên print () đã thay thế. Danh sách employee có thể được in như sau:

```
void print_list(const list<Employee*>& s)
{
for (auto x : s)
```

```
x−>print();
}
```
Mỗi Nhân viên sẽ được in ra như sau:
```
int main()
{
Employee e {"Brown",1234};
Manager m {"Smith",1234,2};
print_list({&e,&m});
}
produced:
Smith 1234
level 2
Brown 1234
```
Một kiểu có các chức năng ảo được gọi là kiểu đa hình hay. Để có được hành vi đa hình trong C ++, các hàm thành viên được gọi phải là ảo và các đối tượng phải được thao tác thông qua con trỏ hoặc tham chiếu. Khi thao tác trực tiếp một đối tượng (thay vì thông qua con trỏ hoặc tham chiếu), kiểu chính xác của nó được trình biên dịch biết nên không cần đến tính đa hình thời gian chạy.

Một hàm ảo được gọi từ một hàm tạo hoặc một hàm hủy phản ánh rằng đối tượng được cấu trúc một phần hoặc bị phá hủy một phần (§22.4). Do đó, thường là một ý tưởng tồi nếu gọi một hàm ảo từ một hàm tạo hoặc một hàm hủy.

20.2.3 Explicit Qualification

Gọi một hàm bằng toán tử phân giải phạm vi ::, như được thực hiện trong Manager :: print () đảm bảo rằng cơ chế ảo không được sử dụng:
```
void Manager::print() const
{
Employee::print();
cout << "\tlevel " << level << '\n';
}
```
20.2.4 Kiểm soát ghi đè

Nếu bạn khai báo một hàm trong một lớp dẫn xuất có cùng tên và kiểu với hàm ảo trong lớp cơ sở, thì hàm trong lớp dẫn xuất sẽ ghi đè hàm trong lớp cơ sở. Đó là một quy tắc đơn giản và hiệu quả. Tuy nhiên, đối với các cấu trúc phân cấp lớn hơn, có thể khó để chắc chắn rằng bạn thực sự ghi đè chức năng mà bạn định ghi đè. Xem xét:
```
struct B0 {
```

```
void f(int) const;
virtual void g(double);
};
struct B1 : B0 { /* ... */ };
struct B2 : B1 { /* ... */ };
struct B3 : B2 { /* ... */ };
struct B4 : B3 { /* ... */ };
struct B5 : B4 { /* ... */ };
struct D : B5 {
void f(int) const; // overr ide f() in base class
void g(int); // overr ide g() in base class
virtual int h(); // overr ide h() in base class
};
```

Điều này minh họa ba lỗi không rõ ràng khi chúng xuất hiện trong hệ thống phân cấp lớp thực, nơi các lớp B0 ... B5 mỗi lớp có nhiều thành viên và nằm rải rác trên nhiều tệp tiêu đề. Ở đây:

• B0 :: f () không phải là ảo, vì vậy bạn không thể ghi đè nó, chỉ ẩn nó (§20.3.5).

• D :: g () không có cùng kiểu đối số như B0 :: g (), vì vậy nếu nó ghi đè bất cứ thứ gì thì nó không phải là hàm ảo B0 :: g (). Rất có thể, D :: g () chỉ ẩn B0 :: g ().

• Không có hàm nào được gọi là h () trong B0, nếu D :: h () ghi đè lên bất cứ điều gì, nó không phải là một hàm từ B0.

Tôi đã không cho bạn thấy những gì trong B1 ... B5, vì vậy có thể điều gì đó hoàn toàn khác đang xảy ra do các khai báo trong các lớp đó. Cá nhân tôi không (dư thừa) sử dụng ảo cho một chức năng nghĩa là ghi đè. Đối với các chương trình nhỏ hơn (đặc biệt là với trình biên dịch có cảnh báo phù hợp về các lỗi thường gặp) việc ghi đè được thực hiện đúng cách không khó. Tuy nhiên, đối với các phân cấp lớn hơn, các điều khiển cụ thể hơn sẽ hữu ích:

• virtual: Chức năng có thể bị ghi đè (§20.3.2).

• = 0: Hàm phải ảo và phải được ghi đè (§20.4).

• override: Hàm được dùng để ghi đè một hàm ảo trong một lớp cơ sở (§20.3.4.1).

• final: Hàm không có nghĩa là bị ghi đè (§20.3.4.2).

Trong trường hợp không có bất kỳ điều khiển nào trong số này, một hàm thành viên không tĩnh sẽ là ảo khi và chỉ khi nó ghi đè một hàm ảo trong một lớp cơ sở. Một trình biên dịch có thể cảnh báo chống lại việc sử dụng không nhất quán các điều khiển ghi đè rõ ràng. Ví dụ, một khai báo lớp sử dụng ghi đè cho bảy trong số chín hàm lớp cơ sở ảo có thể gây nhầm lẫn cho người bảo trì.

20.2.4.1 Ghi đè

Chúng tôi có thể nói rõ về mong muốn ghi đè của chúng tôi:

```
struct D : B5 {
void f(int) const override; // error : B0::f() is not virtual
void g(int) override; // error : B0::f() takes a double argument
virtual int h() override; // error : no function h() to overr ide
};
```

Đưa ra định nghĩa này (và giả sử rằng các lớp cơ sở trung gian B1 ... B5 không cung cấp các hàm tương ứng), cả ba khai báo đều mắc lỗi.

Trong một hệ thống phân cấp lớp hoặc phức tạp với nhiều chức năng ảo, tốt nhất là chỉ sử dụng ảo để giới thiệu một chức năng ảo mới và sử dụng ghi đè trên tất cả các chức năng được dùng làm trình ghi đè.

Sử dụng ghi đè hơi dài dòng nhưng làm rõ ý định của lập trình viên.Bộ chỉ định ghi đè xuất hiện cuối cùng trong một khai báo, sau tất cả các phần khác. Ví dụ:

```
void f(int) const noexcept override;
override void f(int) const noexcept; // syntax error
```

Mã định nghĩa ghi đè không phải là một phần của kiểu hàm và không thể lặp được lặp lại trong định nghĩa ngoài lớp. Ví dụ:

class Derived : public Base {

void f() override; // OK if Base has a virtual f()

void g() override; // OK if Base has a virtual g()

};

void Derived::f() override // error : override out of class

{

}

void g() // OK

{

}

20.2.4.2 final

Khi chúng ta khai báo một hàm thành viên, chúng ta có một sự lựa chọn giữa ảo và không ảo (mặc định).

Chúng tôi sử dụng ảo cho các hàm mà chúng tôi muốn người viết các lớp dẫn xuất có thể định nghĩa hoặc xác định lại. Chúng tôi lựa chọn dựa trên ý nghĩa của lớp:

• Chúng ta có thể hình dung sự cần thiết của các lớp dẫn xuất tiếp theo không?

• Người thiết kế lớp dẫn xuất có cần xác định lại hàm để đạt được mục đích hợp lý không?

• Ghi đè một hàm có dễ bị lỗi không?

Nếu câu trả lời là ‘‘ không ’’ cho cả ba câu hỏi, chúng ta có thể để hàm không ảo để đạt được sự đơn giản của thiết kế và đôi khi là một số hiệu suất (chủ yếu là từ nội tuyến). Thư viện tiêu chuẩn đã đầy ví dụ về điều này.

Chúng tôi có thể ngăn người dùng ghi đè các hàm ảo bởi vì điều duy nhất mà những ghi đè như vậy có thể làm là thay đổi ngữ nghĩa của ngôn ngữ. Chúng tôi có thể đóng thiết kế của mình để sửa đổi từ người dùng. Ví dụ:

struct Node { // interface class

virtual Type type() = 0;

```
};
class If_statement : public Node {
public:
Type type() override final; // prevent further overriding
};
```

Việc thêm cuối cùng vào lớp không chỉ ngăn chặn việc ghi đè, nó còn ngăn chặn việc dẫn xuất thêm từ một lớp.Tuy nhiên, đừng sử dụng final một cách mù quáng như một chất tối ưu hóa sự giúp đỡ; nó ảnh hưởng đến thiết kế phân cấp lớp (thường là tiêu cực) và các cải tiến hiệu suất hiếm khi đáng kể. Sử dụng final khi nó phản ánh rõ ràng thiết kế phân cấp lớp mà bạn cho là phù hợp. Nghĩa là, sử dụng final để phản ánh nhu cầu ngữ nghĩa. Ví dụ:

```
class Derived : public Base {
void f() final; // OK if Base has a virtual f()
void g() final; // OK if Base has a virtual g()
// ...
};
void Derived::f() final // error : final out of class
{
}
void g() final // OK
{
}
```

Giống như ghi đè (§20.3.4.1), cuối cùng là một từ khóa theo ngữ cảnh. Nghĩa là, cuối cùng có một ý nghĩa đặc biệt trong một số ngữ cảnh nhưng có thể được sử dụng như một định danh thông thường ở những nơi khác. Ví dụ:

```
int final = 7;
struct Dx: Base {
```

```
int final;
int f () final
{
return final + :: final;
}
};
```

Đừng ham mê sự thông minh như vậy; nó làm phức tạp thêm việc bảo trì. Lý do duy nhất khiến cuối cùng là một từ khóa liên văn bản, chứ không phải là một từ khóa thông thường, là tồn tại một lượng đáng kể mã đã sử dụng cuối cùng làm mã định danh thông thường trong nhiều thập kỷ. Từ khóa theo ngữ cảnh khác được ghi đè (§20.3.4.1).

20.2.5 Sử dụng thành viên cơ sở

Các hàm không quá tải trên các phạm vi. Ví dụ:

```
struct Base {
void f(int);
};
struct Derived : Base {
void f(double);
};
void use(Derived d)
{
d.f(1); // call Derived::f(double)
Base& br = d
br.f(1); // call Base::f(int)
}
```

Một số khai báo sử dụng có thể mang tên từ nhiều lớp cơ sở. Ví dụ:

```
struct B1 {
```

```
void f(int);
};
struct B2 {
void f(double);
};
struct D : B1, B2 {
using B1::f;
using B2::f;
void f(char);
};
void use(D d)
{
d.f(1); // call D::f(int), that is, B1::f(int)
d.f('a'); // call D::f(char)
d.f(1.0); // call D::f(double), that is, B2::f(double)
}
```

Chúng ta có thể đưa các hàm tạo vào một phạm vi lớp dẫn xuất. Một tên được đưa vào phạm vi lớp dẫn xuất bởi một khai báo using có quyền truy cập xác định bởi vị trí của khai báo using. Chúng ta không thể sử dụng các chỉ thị using để đưa tất cả các thành viên của một lớp cơ sở vào một lớp dẫn xuất.

20.2.5.1 Khởi tạo kế thừa

Giả sử tôi muốn một vectơ giống như std::vector, tôi có thể thử điều này:

```
template<class T>
struct Vector : std::vector<T> {
        T& operator[](size_type i) { check(i); return this−>elem(i); }
        const T& operator[](size_type i) const { check(i); return this−>elem(i); }
```

```
        void check(siz e_type i) { if (this−>size()<i) throw rang
        e_error{"Vector::check() failed"}; }
};
```

Thật không may, chúng tôi sẽ sớm phát hiện ra rằng định nghĩa này không đầy đủ.

Ví dụ: Vectơ <int> v {1, 2, 3, 5, 8}; // error: không có phương thức khởi tạo

Kiểm tra sẽ cho thấy rằng Vector không kế thừa bất kỳ hàm tạo nào từ std :: vector.

Giải quyết vấn đề đó bằng cách đơn giản nói rằng các hàm tạo phải được kế thừa:

```
template<class T>
struct Vector : std::vector<T> {
        using vector<T>::vector; // kế thừa các hàm  tạo
        T& operator=[](size_type i) { check(i); return this−>elem(i); }
        const T& operator=(size_type i) const { check(i); return this−>elem(i); }
        void check(siz e_type i) { if (this−>size()<i) throw Bad_index(i); }
};
Vector<int> v { 1, 2, 3, 5, 8 }; // OK
```

Nếu bạn chọn như vậy, bạn có thể tự bắn vào chân mình bằng cách kế thừa các hàm tạo trong một lớp dẫn xuất, trong đó bạn xác định các biến thành viên mới cần khởi tạo rõ ràng:

```
struct B1 {
B1(int) { }
};
struct D1 : B1 {
using B1::B1; //Khai báo ngầm D1(int)
string s; // Khởi tạo chuỗi s mặc định
int x; //Quên khởi tạo x
};
```

Để loại bỏ điều đó cần thêm trình khởi tạo thành viên trong lớp:

```
struct D1 : B1 {
using B1::B1; // khai báo ngầm D1(int)
int x {0}; // khởi tạo x
};
void test()
{
D1 d {6}; // d.x is zero
}
```

Thông thường, tốt nhất là nên tránh khôn khéo và hạn chế việc sử dụng các hàm tạo kế thừa trong các trường hợp đơn giản mà không có thành viên dữ liệu nào được thêm vào.

20.2.6 Return Type Relaxationss

Có một quy tắc nới lỏng rằng kiểu của một hàm ghi đè phải giống với kiểu của hàm ảo mà nó ghi đè. Nghĩa là, nếu kiểu trả về ban đầu là B∗, thì kiểu trả về của hàm ghi đè có thể là D∗, với điều kiện B là cơ sở công khai của D. Tương tự, kiểu trả về B & có thể được nới lỏng thành D &.

Việc nới lỏng này chỉ áp dụng cho các kiểu trả về là con trỏ hoặc tham chiếu,không phải cho ‘‘ con trỏ thông minh ’’ chẳng hạn như unique_ptr (§5.2.1). Đặc biệt, không có sự nới lỏng tương tự các quy tắc cho các kiểu đối số vì điều đó sẽ dẫn đến vi phạm kiểu.

Ngoài các toán hạng để thao tác các biểu thức, lớp cơ sở Expr sẽ cung cấp các phương tiện để tạo các đối tượng biểu thức mới của các kiểu biểu thức khác nhau:

```
class Expr {
public:
Expr(); // default constructor
Expr(const Expr&); // copy constr uctor
virtual Expr∗ new_expr() =0;
```

```
virtual Expr* clone() =0;
};
```

Một lớp dẫn xuất có thể ghi đè new_expr () và /hoặc clone () để trả về một đối tượng thuộc kiểu riêng của nó:

```
class Cond : public Expr {
public:
Cond();
Cond(const Cond&);
Cond* new_expr() override { return new Cond(); }
Cond* clone() override { return new Cond(*this); }
// ...
};
```

Điều này có nghĩa là với một đối tượng của lớp Expr, người dùng có thể tạo một đối tượng mới " chỉ cùng loại. " Ví dụ:

```
void user(Expr* p)
{
Expr* p2 = p->new_expr();
}
```

Con trỏ được gán cho p2 được khai báo để trỏ đến một " Expr thuần túy ", nhưng nó sẽ trỏ đến một đối tượng có kiểu dẫn xuất từ Expr, chẳng hạn như Cond.
Kiểu trả về của Cond :: new_expr () và Cond :: clone () là Cond * chứ không phải Expr *. Điều này cho phép một Cond được sao chép mà không làm mất thông tin.
Bởi vì các hàm như new_expr () và clone () là ảo và chúng (gián tiếp) xây dựng các đối tượng, chúng thường được gọi là các hàm tạo ảo.
Để tạo một đối tượng, một phương thức khởi tạo cần có kiểu chính xác của đối tượng mà nó sẽ tạo. Do đó, một phương thức khởi tạo không thể là ảo. Hơn nữa, một hàm tạo không phải là một hàm hoàn toàn bình thường.

20.3 Các lớp trừu tượng

Nhiều lớp giống với lớp Employee ở chỗ chúng hữu ích như chính chúng và làm khuôn mẫu cho các lớp dẫn xuất và là một phần của việc triển khai các lớp dẫn xuất.Tuy nhiên, không phải tất cả các lớp đều tuân theo khuôn mẫu đó. Một số lớp, chẳng hạn như một lớp Shape, đại diện cho các khái niệm trừu tượng mà đối tượng không thể tồn tại. Hình dạng chỉ có ý nghĩa như là cơ sở của một số lớp bắt nguồn

từ nó. Điều này có thể được thấy từ thực tế là không thể cung cấp các định nghĩa hợp lý cho các chức năng ảo của nó:
Hình dạng lớp {
public:
class Shape {
public:
virtual void rotate(int) { throw runtime_error{"Shape::rotate"}; }
virtual void draw() const { throw runtime_error{"Shape::draw"}; }
};
Một giải pháp thay thế tốt hơn là khai báo các hàm ảo của lớp Shape là các hàm ảo thuần túy.
Một hàm ảo được ‘‘tạo thuần túy’’ bởi ‘‘bộ khởi tạo giả’’ = 0:
class Shape { // abstract class
public:
virtual void rotate(int) = 0; // pure virtual function
virtual void draw() const = 0; // pure virtual function
virtual bool is_closed() const = 0; // pure virtual function
virtual ˜Shape(); // virtual
};
Một lớp có một hoặc nhiều hàm ảo thuần túy là một lớp trừu tượng và không có đối tượng nào của lớp trừu tượng đó có thể được tạo.
Một lớp trừu tượng được sử dụng như một khuôn cho các đối tượng được truy cập thông qua con trỏ và tham chiếu. Do đó, điều quan trọng đối với một lớp trừu tượng là có một trình hủy ảo. Bởi vì khuôn mẫu được cung cấp bởi một lớp trừu tượng không thể được sử dụng để tạo các đối tượng bằng cách sử dụng một hàm tạo, các lớp trừu tượng thường không có các hàm tạo. Một lớp trừu tượng chỉ có thể được sử dụng làm khuôn mẫu cho các lớp khác.Ví dụ:
class Point { /* ... */ };
class Circle : public Shape {
public:
void rotate(int) override { }
void draw() const override;
bool is_closed() const override { return true; }
Circle(Point p, int r);
private:

```
Point center;
int radius;
};
```
Một hàm thuần ảo không được định nghĩa trong lớp dẫn xuất vẫn là một hàm thuần ảo, vì vậy lớp dẫn xuất cũng là một lớp trừu tượng. Điều này cho phép chúng tôi xây dựng các triển khai theo từng giai đoạn:
```
class Polygon : public Shape { // abstract class
public:
bool is_closed() const override { return true; }
};
```
Một lớp trừu tượng cung cấp khuôn mẫu mà không để lộ chi tiết triển khai. Ví dụ:
```
class Character_device {
public:
virtual int open(int opt) = 0;
virtual int close(int opt) = 0;
virtual int read(char* p, int n) = 0;
virtual int write(const char* p, int n) = 0;
virtual int ioctl(int ...) = 0;
virtual ~Character_device() { } // hàm huỷ ảo
};
```
Có thể chỉ định trình điều khiển là các lớp bắt nguồn từ Character_device và thao tác nhiều hàm khác nhau thông qua khuôn mẫu đó. Chúng ta có thể định nghĩa và sử dụng các lớp cơ sở với cả trạng thái và chức năng thuần ảo. Tuy nhiên, những cách tiếp cận hỗn hợp như vậy có thể gây nhầm lẫn và cần phải cẩn thận hơn.

20.4 Kiểm soát truy cập

Một thành viên của một lớp có thể là riêng tư, được bảo vệ hoặc công khai:

• Nếu nó là private, tên của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của lớp mà nó được khai báo.

• Nếu nó được bảo vệ, tên của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của lớp mà nó được khai báo và bởi các hàm thành viên và bạn bè của các lớp dẫn xuất từ lớp này.

• Nếu nó là công khai, tên của nó có thể được sử dụng bởi bất kỳ hàm nào.

Điều này phản ánh quan điểm rằng có ba loại hàm truy cập vào một lớp: các hàm thực thi lớp(bạn bè và thành viên của nó), các hàm triển khai một lớp dẫn xuất (bạn bè và thành viên của lớp dẫn xuất) và các hàm khác.

Kiểm soát truy cập được áp dụng thống nhất cho các tên.Chúng ta có thể có các hàm thành viên riêng, kiểu, hằng số, v.v, cũng như các thành viên dữ liệu riêng. Thông tin và cấu trúc dữ liệu được sử dụng để thực thi có thể là riêng tư:

```
template<class T>
class List {
public:
void insert(T);
T get();
private:
struct Link { T val; Link* next; };
struct Chunk {
enum { chunk_siz e = 15 };
Link v[chunk_siz e];
Chunk* next;
};
Chunk* allocated;
Link* free;
Link* get_free();
Link* head;
};
```

Các định nghĩa của các chức năng công khai khá đơn giản:

```
template<class T>
void List<T>::insert(T val)
{
Link* lnk = get_free();
lnk−>val = val;
lnk−>next = head;
head = lnk;
}
```

Như thường lệ, định nghĩa của các hàm hỗ trợ (ở đây, riêng tư) phức tạp hơn một chút:

```
template<class T>
typename List<T>::Link* List<T>::get_free()
{
```

```
if (free == 0) {
}
Link* p = free;
free = free->next;
return p;
}
```
Các chức năng nonmember (ngoại trừ bạn bè) không có quyền truy cập như vậy:
```
template<typename T>
void would_be_meddler(List<T>* p)
{
List<T>::Link* q = 0; // error : List<T>::Link is private
q=p->free; // error : List<T>::free is private
if (List<T>::Chunk::chunk_siz e > 31) { // error :
List<T>::Chunk::chunk_size is private
}
}
```
20.4.1 Thành viên được bảo vệ
Khi thiết kế một cấu trúc phân cấp lớp, đôi khi chúng tôi cung cấp các hàm được thiết kế để được sử dụng bởi những người triển khai các lớp dẫn xuất mà không phải bởi người dùng chung.Ví dụ:
```
class Buffer {
public:
char& operator[](int i); // quyền truy cập được kiểm tra
protected:
char& access(int i); // quyền truy cập không được kiểm tra
};
class Circular_buffer : public Buffer {
public:
void reallocate(char* p, int s); // change location and size
};
void Circular_buffer::reallocate(char* p, int s)// change location and size
{
for (int i=0; i!=old_sz; ++i)
p[i] = access(i); // không kiểm tra dư thừa
}
```

void f(Buffer& b)
{
b[3] = 'b'; // OK (checked)
b.access(3) = 'c'; // error : Buffer ::access() is protected
}
Điều này ngăn ngừa các lỗi nhỏ có thể xảy ra khi một lớp dẫn xuất làm hỏng dữ liệu thuộc các lớp dẫn xuất khác.

20.4.1.1 Sử dụng các thành viên được bảo vệ

Mô hình ẩn dữ liệu riêng tư / công khai đơn giản phục vụ tốt khái niệm về các kiểu cụ thể.Tuy nhiên, khi các lớp dẫn xuất được sử dụng, có hai loại người dùng của một lớp: các lớp dẫn xuất và"công khai" Các thành viên và bạn bè thực hiện các thao tác trên lớp sẽ hoạt động trên các đối tượng của lớp. Mô hình private / public cho phép lập trình viên phân biệt rõ ràng giữa thực hiện và công khai, nhưng nó không cung cấp cấp cụ thể cho các lớp dẫn xuất.

Các thành viên protected dễ bị lạm dụng hơn nhiều so với các thành viên private. Việc khai báo các thành viên dữ liệu được bảo vệ thường là một lỗi thiết kế. Đặt một lượng lớn dữ liệu trong một lớp để tất cả các lớp dẫn xuất sử dụng khiến dữ liệu có thể bị hỏng. Tệ hơn nữa, dữ liệu được bảo vệ, như dữ liệu công khai, không thể dễ dàng tái cấu trúc vì không có cách nào tốt để tìm kiếm mỗi khi sử dụng. May mắn thay, bạn không muốn sử dụng dữ liệu được bảo vệ; private là mặc định trong các lớp và thường là lựa chọn tốt hơn.

20.4.2 Quyền truy cập vào các lớp cơ sở

Một lớp cơ sở có thể được khai báo là riêng tư, được bảo vệ hoặc công khai. Ví dụ:
class X : public B { /* ... */ };
class Y : protected B { /* ... */ };
class Z : private B { /* ... */
Có thể bỏ qua thông số truy cập cho một lớp cơ sở. Trong trường hợp đó, cơ sở mặc định là cơ sở riêng cho một lớp và cơ sở công khai cho một cấu trúc. Ví dụ:
class XX : B { /* ... */ }; // B is a private base
struct YY : B { /* ... */ }; // B is a public base
Bộ định nghĩa truy cập cho một lớp cơ sở kiểm soát quyền truy cập vào các thành viên của lớp cơ sở và việc chuyển đổi con trỏ và tham chiếu từ kiểu lớp dẫn xuất sang kiểu lớp cơ sở. Ví dụ lớp dẫn xuất D bắt nguồn từ một lớp cơ sở B:

• Nếu B là cơ sở, các thành viên public và protected của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của D. Chỉ bạn bè và thành viên của D mới có thể chuyển đổi D thành B.
• Nếu B là một cơ sở protected, các thành viên public và protected chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của D và bởi các hàm thành viên và bạn bè của các lớp bắt nguồn từ D. Chỉ bạn bè và thành viên của D và bạn bè và thành viên của các lớp dẫn xuất từ D có thể chuyển D∗ thành B∗.
• Nếu B là một cơ sở public, các thành viên public có thể được sử dụng cho bất kỳ hàm nào. Ngoài ra, các thành viên protected có thể được sử dụng bởi các thành viên và bạn bè của D và các thành viên và bạn bè của các lớp có nguồn gốc từ D. Bất kỳ hàm nào cũng có thể chuyển đổi từ D ∗ sang B ∗.

20.4.2.1 Đa kế thừa và kiểm soát truy cập

Nếu tên của một lớp cơ sở có thể được truy cập thông qua nhiều đường dẫn trong đa kế thừa,nó có thể truy cập được nếu có thể truy cập thông qua bất kỳ đường dẫn nào. Ví dụ:

```
struct B {
int m;
static int sm;
// ...
};
class D1 : public virtual B { /* ... */ } ;
class D2 : public virtual B { /* ... */ } ;
class D12 : public D1, private D2 { /* ... */ };
D12∗ pd = new D12;
B∗ pb = pd; // OK: accessible through D1
int i1 = pd−>m; // OK: accessible through D1
```

Nếu một thực thể duy nhất có thể truy cập được thông qua một số đường dẫn, chúng tôi vẫn có thể tham chiếu đến nó.

20.4.3 Sử dụng-Khai báo và Kiểm soát truy cập

Không thể sử dụng một khai báo đang sử dụng để truy cập thông tin bổ sung. Nó chỉ đơn giản là một cơ chế để làm cho thông tin có thể truy cập được thuận tiện hơn khi sử dụng. Mặt khác, khi đã có quyền truy cập, nó có thể được cấp cho những người dùng khác. Ví dụ:

```
class B {
private:
```

```
int a;
protected:
int b;
public:
int c;
};
class D : public B {
public:
using B::b; // make B::b publicly available through D
};
```

20.5 Con trỏ đến thành viên

Con trỏ đến thành viên là một cấu trúc giống như bù đắp cho phép tham chiếu gián tiếp đến thành viên của một lớp. Các toán tử -> ∗ và .∗ được cho là các toán tử C ++ chuyên dụng nhất và ít được sử dụng nhất các toán tử. Sử dụng ->, chúng ta có thể truy cập một thành viên của lớp m, bằng cách: p−> m. Sử dụng -> ∗, chúng ta có thể truy cập một thành viên có tên được lưu trữ trong một con trỏ tới thành viên, ptom: p -> ∗ ptom. Điều này cho phép truy cập các thành viên với tên của họ được chuyển làm đối số.

Một con trỏ tới thành viên không thể được gán cho void ∗ hoặc bất kỳ con trỏ thông thường nào khác. Một con trỏ null (ví dụ: nullptr) có thể được gán cho một con trỏ tới thành viên và sau đó đại diện cho "" no member ".

20.5.1 Con trỏ đến các thành viên hàm

```
class Std_interface {
public:
virtual void start() = 0;
virtual void suspend() = 0;
virtual void resume() = 0;
virtual void quit() = 0;
virtual void full_size() = 0;
virtual void small() = 0;
virtual ~Std_interface() {}
};
```

Ý nghĩa chính xác của mỗi thao tác được xác định bởi đối tượng mà nó được gọi. Thông thường, có một lớp phần mềm nằm giữa người hoặc chương trình đưa ra yêu cầu và đối tượng nhận yêu cầu.Tốt nhất, các lớp phần mềm trung gian như vậy

không cần phải biết bất kỳ điều gì về các hoạt động riêng lẻ như resume () và full_size (). Nếu có, các lớp trung gian sẽ phải được cập nhật mỗi khi một hoạt động thay đổi. Do đó, các lớp trung gian như vậy chỉ đơn giản là truyền dữ liệu đại diện cho hoạt động được gọi từ nguồn của yêu cầu đến người nhận.
Một cách đơn giản để thực hiện đó là gửi một chuỗi đại diện cho hoạt động được gọi. Ví dụ, để gọi Susan (), chúng ta có thể gửi chuỗi "Susan". Tuy nhiên, ai đó phải tạo chuỗi đó và ai đó phải giải mã nó để xác định nó tương ứng với hoạt động nào - nếu có.
Tuy nhiên, chúng ta có thể sử dụng một con trỏ tới thành viên để gián tiếp tham chiếu đến thành viên của một lớp. Hãy xem xét Std_interface. Nếu tôi muốn gọi hàm Susan () cho một số đối tượng mà không đề cập trực tiếp đến Susan (), tôi cần một con trỏ đến thành viên tham chiếu đến Std_interface :: Susan (). Tôi cũng cần một con trỏ hoặc tham chiếu đến đối tượng mà tôi muốn tạm dừng ví dụ:

```
using Pstd_mem = void (Std_interface :: ∗) (); // kiểu con trỏ đến thành viên
void f (Std_interface ∗ p)
{
Pstd_mem s = & Std_interface:: suspend; // con trỏ để tạm dừng ()
p−> Susan (); // gọi trực tiếp
p -> ∗ s (); // gọi thông qua con trỏ đến thành viên
}
```

Con trỏ tới một thành viên ảo là một loại bù đắp, nó không phụ thuộc vào vị trí của đối tượng trong bộ nhớ. Do đó, một con trỏ đến một thành viên ảo có thể được chuyển giữa các không gian địa chỉ khác nhau miễn là sử dụng cùng một bố cục đối tượng trong cả hai. Giống như con trỏ tới các hàm thông thường, con trỏ tới các hàm thành viên không phải ảo không thể được trao đổi giữa các không gian địa chỉ. Lưu ý rằng hàm được gọi thông qua con trỏ tới hàm có thể là ảo.
Thành viên tĩnh không được liên kết với một đối tượng cụ thể, vì vậy con trỏ đến thành viên tĩnh chỉ đơn giản là một con trỏ thông thường.

20.5.2 Con trỏ đến các thành viên dữ liệu

Đương nhiên, khái niệm con trỏ tới thành viên áp dụng cho các thành viên dữ liệu và cho các hàm thành viên với các đối số và kiểu trả về. Ví dụ:

```
struct C {
const char ∗ val;
int i;
void print (int x) {cout << val << x << '\ n'; }
```

```
int f1 (int);
void f2 ();
C (const char * v) {val = v; }
};
using Pmfi = void (C :: *) (int); // con trỏ tới hàm thành viên của C lấy int
sử dụng Pm = const char * C :: *; // con trỏ tới thành viên dữ liệu char * của
C
void f (C & z1, C & z2)
{
C * p = & z2;
Pmfi pf = & C :: print;
pm pm = & C :: val;
z1.print (1);
(z1. * pf) (2);
z1. * pm = "nv1";
p -> * pm = "nv2";
z2.print (3);
(p -> * pf) (4);
}
```

Loại con trỏ tới hàm được kiểm tra giống như bất kỳ loại nào khác.

20.5.3 Thành viên cơ sở và dẫn xuất

Một lớp dẫn xuất có ít nhất các thành viên mà nó kế thừa từ các lớp cơ sở của nó. Chứng ta có thể gán một cách an toàn một con trỏ cho một thành viên của lớp cơ sở cho một con trỏ tới một thành viên của lớp dẫn xuất, nhưng không phải ngược lại. Tính chất này thường được gọi là độ tương phản.Ví dụ:

```
class Text : public Std_interface {
public:
void start();
void suspend();
// ...
virtual void print();
private:
vector s;
};
void (Text::*pmt)() = &Std_interface::start; // OK
```

Quy tắc tương phản này dường như ngược lại với quy tắc nói rằng có thể gán một con trỏ cho lớp dẫn xuất, một con trỏ đến lớp cơ sở của nó. Trên thực tế, cả hai quy

tắc đều tồn tại để duy trì sự đảm bảo rằng một con trỏ có thể không bao giờ trỏ đến một đối tượng ít nhất không có các thuộc tính mà con trỏ hứa hẹn. Trong trường hợp này, Std_interface :: ∗ có thể được áp dụng cho bất kỳ Std_interface nào, và hầu hết các đối tượng như vậy có lẽ không thuộc loại Text. Do đó, chúng không có thành viên Text :: print mà chúng tôi đã cố gắng khởi tạo pmi. Bằng cách từ chối khởi tạo, trình biên dịch giúp chúng ta tránh khỏi lỗi thời gian chạy.