4 继承

4.1 继承的概念和语法

- 子类继承基类，基类派生子类，子类对象“is a”基类对象
- 继承语法：

```
class 子类 : 继承方式1 基类1, 继承方式2 基类2, ... {
    ...
};
```

- 继承方式：公有继承（public）、保护继承（protected）、私有继承（private）

```
// human.cpp

// 继承的概念和语法

#include <iostream>

using namespace std;
/*
class Student {
public:
    Student(const char* name, int age, int num)
        : name(name), age(age), num(num) {}

    void who(void) const {
        cout << "我是" << name << "，今年" << age << "岁。" << endl;
    }

    void learn(const char* course) const {
        cout << "我在学" << course << "，我的学号是" << num << "。" << endl;
    }

private:
    string name;
    int age;
    int num;
};

class Teacher {
public:
    Teacher(const char* name, int age, const char* job)
        : name(name), age(age), job(job) {}

    void who(void) const {
        cout << "我是" << name << "，今年" << age << "岁。" << endl;
    }

    void teach(const char* course) const {
        cout << "我在教" << course << "，我的职位是" << job << "。" << endl;
    }

```

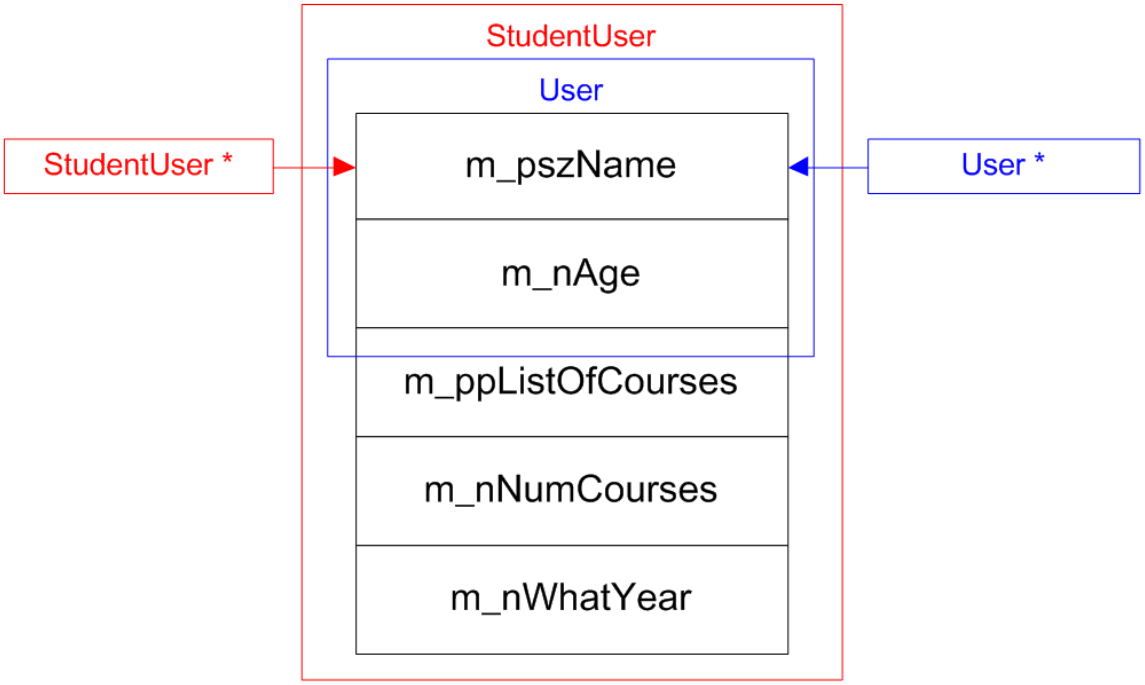
```cpp
private:
    string name;
    int age;
    string job;
};
*/
class Human {
public:
    Human(const char* name, int age)
        : name(name), age(age) {}

    void who(void) const {
        cout << "我是" << name << "，今年" << age << "岁。" << endl;
    }

private:
    string name;
    int age;
};

class Student : public Human {
public:
    Student(const char* name, int age, int num)
        : Human(name, age), num(num) {}

    void learn(const char* course) const {
        cout << "我在学" << course << "，我的学号是" << num << "。" << endl;
    }

private:
    int num;
};

class Teacher : public Human {
public:
    Teacher(const char* name, int age, const char* job)
        : Human(name, age), job(job) {}

    void teach(const char* course) const {
        cout << "我在教" << course << "，我的职位是" << job << "。" << endl;
    }

private:
    string job;
};

int main(void) {
    Student student("李明", 25, 1001);
    student.who();
    student.learn("英语");

    Teacher teacher("陈飞", 48, "教授");
    teacher.who();
    teacher.teach("数学");

    return 0;
```

```
97 }
```

4.2 公有继承

- 一个子类类型的对象在任何时候都可以作为一个基类类型的对象，而不必使用显示的类型转换，前提是两者（子类及其基类）都是通过指针或引用操作的

```
class User {
public:
    User(...) {
        ...
    }

private:
    char* name;
    int age;
};

class StudentUser : public User {
public:
    StudentUser(...) {
        ...
    }

private:
    char** listOfCourses;
    int numbOfCourses;
    int whatYear;
};

StudentUser* studentUser = new StudentUser(...);
User* user = studentUser; // 安全的向上造型
```

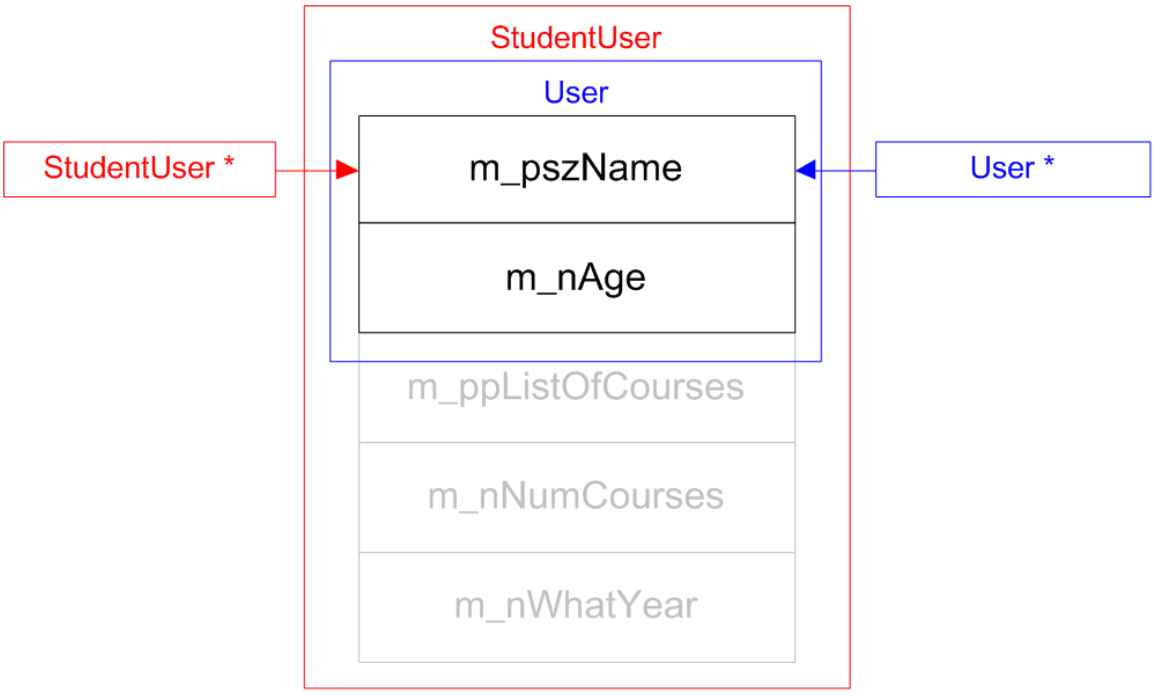


- 反之，任何时候都无法将一个基类类型的指针或引用，隐式转换为子类类型的指针或引用

```
User* user = new User(...);
StudentUser* studentUser = user; // 编译错误
```

使用静态造型（强制类型转换）固然可以规避语法上限制，但并不安全

```
StudentUser* studentUser = static_cast<StudentUser*>(user); // 不安全的向下造型
```



- 向下造型的安全性，取决于代码编写者能否确知，该基类指针或引用的目标对象真的是一个子类对象

```
User* user = new StudentUser(...);
StudentUser* studentUser = static_cast<StudentUser*>(user); // 安全的向下造型
```

- 在子类中可以直接使用基类的所有公有和保护成员，就象它们是在子类中声明的那样，但基类的私有成员在子类对象中虽然存在却不可见，故无法直接使用
- 基类的公有和保护成员在子类中不仅可见，而且可被重新定义
  - 由此产生的名字冲突可以通过“子类中的名字定义隐藏所有基类中的同名定义”规则而避免
  - 借助作用域分解操作符（::），可通过子类或在其中，访问那些在基类中定义却被子类所隐藏的名字

4.3 继承方式对成员访问控制的影响

4.3.1 类成员的访问控制

访控限定符	访控属性	自己	子类	外部	友元
public	公有成员	Yes	Yes	Yes	Yes
protected	保护成员	Yes	Yes	No	Yes

访控限定符	访控属性	自己	子类	外部	友元
private	私有成员	Yes	No	No	Yes

```cpp
// accctrl.cpp

// 类成员的访问控制

class A {
public:
    void pub_fun(void) {}
    char pub_var;

protected:
    void pro_fun(void) {}
    char pro_var;

private:
    void pri_fun(void) {}
    char pri_var;

    void foo(void) {
        pub_fun();
        pub_var = 10;

        pro_fun();
        pro_var = 10;

        pri_fun();
        pri_var = 10;
    }
};

class B : public A {
    void foo(void) {
        pub_fun();
        pub_var = 10;

        pro_fun();
        pro_var = 10;

        // pri_fun(); // 编译错误
        // pri_var = 10; // 编译错误
    }
};

int main(void) {
    A a;

    a.pub_fun();
    a.pub_var = 10;

    // a.pro_fun();
    // a.pro_var = 10;

```

```
    // a.pri_fun(); // 编译错误
    // a.pri_var = 10; // 编译错误

    return 0;
}
```

4.3.2 基类成员的访控属性在子类中的变化规则

基类中的	在公有子类中变成	在保护子类中变成	在私有子类中变成
公有成员	公有成员	保护成员	私有成员
保护成员	保护成员	保护成员	私有成员
私有成员	私有成员	私有成员	私有成员

```cpp
// public.cpp

// 基类成员的访控属性在公有子类中的变化规则：公保私->公保私

class A {
public:
    void pub_fun(void) {}
    char pub_var;

protected:
    void pro_fun(void) {}
    char pro_var;

private:
    void pri_fun(void) {}
    char pri_var;
};

class B : public A {};

class C : public B {
    void foo(void) {
        pub_fun();
        pub_var = 10;

        pro_fun();
        pro_var = 10;

        // pri_fun(); // 编译错误
        // pri_var = 10; // 编译错误
    }
};

int main(void) {
    B b;

    b.pub_fun();
    b.pub_var = 10;

```

```
    // b.pro_fun(); // 编译错误
    // b.pro_var = 10; // 编译错误

    // b.pri_fun(); // 编译错误
    // b.pri_var = 10; // 编译错误

    A* pa = &b;
    A& ra = b;

    return 0;
}
```

```
// protected.cpp

// 基类成员的访控属性在保护子类中的变化规则：公保私->保保私

class A {
public:
    void pub_fun(void) {}
    char pub_var;

protected:
    void pro_fun(void) {}
    char pro_var;

private:
    void pri_fun(void) {}
    char pri_var;
};

class B : protected A {};

class C : public B {
    void foo(void) {
        pub_fun();
        pub_var = 10;

        pro_fun();
        pro_var = 10;

        // pri_fun(); // 编译错误
        // pri_var = 10; // 编译错误
    }
};

int main(void) {
    B b;

    // b.pub_fun(); // 编译错误
    // b.pub_var = 10; // 编译错误

    // b.pro_fun(); // 编译错误
    // b.pro_var = 10; // 编译错误

    // b.pri_fun(); // 编译错误
```

```
    // b.pri_var = 10; // 编译错误

    // A* p1 = &b; // 编译错误
    // A* p2 = static_cast<A*>(&b); // 编译错误
    // A* p3 = dynamic_cast<A*>(&b); // 编译错误
    A* p4 = reinterpret_cast<A*>(&b);
    A* p5 = (A*)&b;

    // A& r1 = b; // 编译错误
    // A& r2 = static_cast<A&>(b); // 编译错误
    // A& r3 = dynamic_cast<A&>(b); // 编译错误
    A& r4 = reinterpret_cast<A&>(b);
    A& r5 = (A&)b;

    return 0;
}
```

```
// private.cpp

// 基类成员的访控属性在私有子类中的变化规则：公保私->私私私

class A {
public:
    void pub_fun(void) {}
    char pub_var;

protected:
    void pro_fun(void) {}
    char pro_var;

private:
    void pri_fun(void) {}
    char pri_var;
};

class B : private A {};

class C : public B {
    void foo(void) {
        // pub_fun(); // 编译错误
        // pub_var = 10; // 编译错误

        // pro_fun(); // 编译错误
        // pro_var = 10; // 编译错误

        // pri_fun(); // 编译错误
        // pri_var = 10; // 编译错误
    }
};

int main(void) {
    B b;

    // b.pub_fun(); // 编译错误
    // b.pub_var = 10; // 编译错误
```

```
39
40      // b.pro_fun(); // 编译错误
41      // b.pro_var = 10; // 编译错误
42
43      // b.pri_fun(); // 编译错误
44      // b.pri_var = 10; // 编译错误
45
46      // A* p1 = &b; // 编译错误
47      // A* p2 = static_cast<A*>(&b); // 编译错误
48      // A* p3 = dynamic_cast<A*>(&b); // 编译错误
49      A* p4 = reinterpret_cast<A*>(&b);
50      A* p5 = (A*)&b;
51
52      // A& r1 = b; // 编译错误
53      // A& r2 = static_cast<A&>(b); // 编译错误
54      // A& r3 = dynamic_cast<A&>(b); // 编译错误
55      A& r4 = reinterpret_cast<A&>(b);
56      A& r5 = (A&)b;
57
58      return 0;
59  }
```

class A
{
public:
int m_pub;
protected:
int m_pro;
private:
int m_pri;
};				

class B :
{
void foo (void)
{
m_pub = 10;
m_pro = 10;
m_pri = 10;
}
};
	public A	protected A	private A	

	Ok	Ok	Ok	
	Ok	Ok	Ok	
	No	No	No	

class C :
{
void foo (void)
{
m_pub = 10;
m_pro = 10;
m_pri = 10;
}
};
	public/protected/private B			

	Ok	Ok	No	
	Ok	Ok	No	
	No	No	No	

int main (void) {
B b;
b.m_pub = 10;
b.m_pro = 10;
b.m_pri = 10;
return 0;
}				

	Ok	No	No	
	No	No	No	
	No	No	No	

4.4 子类的构造与析构函数

- 子类隐式调用基类构造函数

```
1 // impbase.cpp
2
3 // 子类隐式调用基类构造函数
4
```

```cpp
#include <iostream>

using namespace std;

class User {
public:
    User(void) : name("Anonymous"), age(-1) {}

    User(const char* name, int age) : name(name), age(age) {}

private:
    string name;
    int age;

    friend ostream& operator<<(ostream& os, const User& user) {
        return os << user.name << ", " << user.age;
    }
};

class StudentUser : public User {
public:
    // 如果子类的构造函数没有显式调用基类的构造函数，那么系统
    // 将会调用基类的缺省构造函数。但是请注意，只有在为基类显
    // 式提供一个缺省构造函数，或者不提供任何构造函数（系统会
    // 提供一个缺省构造函数）的情况下，基类才拥有缺省构造函数

    StudentUser(const char* name, int age, const char* school)
        : school(school) {}

private:
    string school;

    friend ostream& operator<<(ostream& os, const StudentUser& su) {
        return os << static_cast<const User&>(su) << ", " << su.school;
    }
};

int main(void) {
    StudentUser su("Maura", 20, "Tarena");

    cout << su << endl;

    return 0;
}
```

- 子类显式调用基类构造函数

```cpp
// expbase.cpp

// 子类显式调用基类构造函数

#include <iostream>

using namespace std;

class User {
```

```
public:
    User(void) : name("Anonymous"), age(-1) {}

    User(const char* name, int age) : name(name), age(age) {}

private:
    string name;
    int age;

    friend ostream& operator<<(ostream& os, const User& user) {
        return os << user.name << ", " << user.age;
    }
};

class StudentUser : public User {
public:
    // 每个子类对象中都包含一个基类类型的子对象，即子类
    // 对象的基类部分。该子对象是由基类的构造函数初始化

    StudentUser(const char* name, int age, const char* school)
        : User(name, age), school(school) {}

private:
    string school;

    friend ostream& operator<<(ostream& os, const StudentUser& su) {
        return os << static_cast<const User&>(su) << ", " << su.school;
    }
};

int main(void) {
    StudentUser su("Maura", 20, "Tarena");

    cout << su << endl;

    return 0;
}
```

- 继承链的构造和析构顺序

```
// consorder.cpp

// 继承链的构造和析构顺序

#include <iostream>

using namespace std;

class A {
public:
    A(void) {
        cout << "A::A() invoked" << endl;
    }

    ~A(void) {
        cout << "A::~A() invoked" << endl;
```

```
    }
};

class B : public A {
public:
    B(void) {
        cout << "B::B() invoked" << endl;
    }

    ~B(void) {
        cout << "B::~B() invoked" << endl;
    }
};

class C : public B {
public:
    C(void) {
        cout << "C::C() invoked" << endl;
    }

    ~C(void) {
        cout << "C::~C() invoked" << endl;
    }
};

int main(void) {
    C c;

    return 0;
}
```

- 基类分配资源，基类提供析构，子类未分配资源，子类无需提供析构

```
// defdes.cpp

// 基类分配资源，基类提供析构，子类未分配资源，子类无需提供析构

#include <iostream>

using namespace std;

class X {
public:
    X(const int* data, size_t size)
        : data(new int[size]), size(size) {
        cout << "X::X() invoked" << endl;

        for (int i = 0; i < this->size; ++i)
            this->data[i] = data[i];
    }

    ~X(void) {
        cout << "X::~X() invoked" << endl;

        delete[] data;
    }
```

```
24
25  private:
26      int *data;
27      size_t size;
28  };
29
30  class Y : public X {
31  public:
32      Y(const int* data, size_t size) : X(data, size) {
33          cout << "Y::Y() invoked" << endl;
34      }
35
36      // 子类既没有分配资源也没有提供析构函数。对于子类，系
37      // 统所提供的缺省析构函数完全可以满足要求，它会自动调
38      // 用基类的析构函数，而后者负责释放在基类中分配的资源
39  };
40
41  int main(void) {
42      int data[1024] = {};
43
44      Y y(data, sizeof(data) / sizeof(data[0]));
45
46      return 0;
47  }
```

- 基类分配资源，基类提供析构，子类亦分配资源，子类必须提供析构

```
1   // mustdes.cpp
2
3   // 基类分配资源，基类提供析构，子类亦分配资源，子类必须提供析构
4
5   #include <string.h>
6
7   #include <iostream>
8
9   using namespace std;
10
11  class X {
12  public:
13      X(const int* data, size_t size)
14          : data(new int[size]), size(size) {
15          cout << "X::X() invoked" << endl;
16
17          for (int i = 0; i < this->size; ++i)
18              this->data[i] = data[i];
19      }
20
21      ~X(void) {
22          cout << "X::~X() invoked" << endl;
23
24          delete[] data;
25      }
26
27  private:
28      int *data;
29      size_t size;
```

```
};

class Y : public X {
public:
    Y(const int* data, size_t size, const char* text)
        : X(data, size), text(strcpy(new char[strlen(text)+1], text)) {
        cout << "Y::Y() invoked" << endl;
    }

    // 如果没有为子类提供自定义的析构函数，那么系统为该子类提供的
    // 缺省析构函数将在适当的时候被调用。该析构函数固然会调用基类
    // 的析构函数，但它并不会释放子类动态分配的资源，导致内存泄漏

    ~Y(void) {
        cout << "Y::~Y() invoked" << endl;

        delete[] text;
    }

private:
    char* text;
};

int main(void) {
    int data[1024] = {};

    Y y(data, sizeof(data) / sizeof(data[0]), "hello world");

    return 0;
}
```

- 通过基类指针析构子类对象的问题

```
// delbd.cpp

// 通过基类指针析构子类对象的问题

#include <string.h>

#include <iostream>

using namespace std;

class X {
public:
    X(const int* data, size_t size)
        : data(new int[size]), size(size) {
        cout << "X::X() invoked" << endl;

        for (int i = 0; i < this->size; ++i)
            this->data[i] = data[i];
    }

    ~X(void) {
        cout << "X::~X() invoked" << endl;

```

```
        delete[] data;
    }

private:
    int *data;
    size_t size;
};

class Y : public X {
public:
    Y(const int* data, size_t size, const char* text)
        : X(data, size), text(strcpy(new char[strlen(text)+1], text)) {
        cout << "Y::Y() invoked" << endl;
    }

    ~Y(void) {
        cout << "Y::~Y() invoked" << endl;

        delete[] text;
    }

private:
    char* text;
};

int main(void) {
    int data[1024] = {};

    X* x = new Y(data, sizeof(data) / sizeof(data[0]), "hello world");

    // delete一个指向子类对象的基类指针，只有基类的析构函数会被执行，
    // 因为没有调用子类的析构函数，子类动态分配的资源将形成内存泄漏

    delete x;

    return 0;
}
```

4.5 子类的拷贝构造函数和拷贝赋值操作符函数

```
// copy.cpp

// 子类的拷贝构造函数和拷贝赋值操作符函数

#include <iostream>

using namespace std;

class User {
public:
    User(void) : name("Anonymous"), age(-1) {}

    User(const char* name, int age) : name(name), age(age) {}

    // 拷贝构造函数
```

```
    User(const User& user) : name(user.name), age(user.age) {}

    // 拷贝赋值操作符函数
    User& operator=(const User& user) {
        if (&user != this) {
            name = user.name;
            age = user.age;
        }

        return *this;
    }

private:
    string name;
    int age;

    friend ostream& operator<<(ostream& os, const User& user) {
        return os << user.name << ", " << user.age;
    }
};

class StudentUser : public User {
public:
    StudentUser(const char* name, int age, const char* school)
        : User(name, age), school(school) {}

    // 子类的拷贝构造函数必须显式调用基类的拷贝构造函数。如果基类没
    // 有显式定义拷贝构造函数，那么系统会调用基类的缺省拷贝构造函数。
    // 如果子类的拷贝构造函数没有显式调用基类的任何构造函数，那么系
    // 统将试图调用基类的缺省构造函数，初始化子类对象中的基类子对象

    // 拷贝构造函数
    StudentUser(const StudentUser& su)
        : User(su), school(su.school) {}

    // 子类的拷贝赋值操作符函数必须显式调用基类的拷贝赋值操作符函数。
    // 如果基类没有显式定义拷贝赋值操作符函数，那么系统会以默认方式
    // 复制子类对象中的基类子对象。如果子类的拷贝赋值操作符函数没有
    // 显式调用基类的拷贝赋值操作符函数，那么子类对象中的基类子对象
    // 将不会被复制

    // 拷贝赋值操作符函数
    StudentUser& operator=(const StudentUser& su) {
        if (&su != this) {
            // User::operator=(su);
            static_cast<User&>(*this) = su;
            school = su.school;
        }

        return *this;
    }

private:
    string school;

    friend ostream& operator<<(ostream& os, const StudentUser& su) {
```

```
        return os << static_cast<const User&>(su) << ", " << su.school;
    }
};

int main(void) {
    StudentUser su1 ("Maura", 20, "Tarena");
    cout << su1 << endl;

    StudentUser su2 = su1; // 拷贝构造
    cout << su2 << endl;

    StudentUser su3("Betty", 22, "MIT");
    cout << su3 << endl;

    su3 = su1; // 拷贝赋值
    cout << su3 << endl;

    return 0;
}
```

4.6 子类的其它操作符函数

```
// operator.cpp

// 子类的其它操作符函数

#include <iostream>

using namespace std;

class Person {
public:
    Person(const char* name, int age) : name(name), age(age) {}

private:
    string name;
    int age;

    friend ostream& operator<<(ostream& os, const Person& person) {
        return os << person.name << ", " << person.age;
    }
};

class Employee : public Person {
public:
    Employee(const char* name, int age, const char* dept, double salary)
        : Person(name, age), dept(dept), salary(salary) {}

private:
    string dept;
    double salary;

    friend ostream& operator<<(ostream& os, const Employee& emp) {
        // 通过将子类引用向上造型为基类引用，迫使针对基类的操作符重载函
        // 数在针对子类的操作符重载函数中被调用，输出子类中的基类子对象
```

```
        return os << static_cast<const Person&>(emp) << ", " <<
            emp.dept << ", " << emp.salary;
    }
};

class Manager : public Employee {
public:
    Manager(const char* name, int age, const char* dept, double salary,
const char* title)
        : Employee(name, age, dept, salary), title(title) {}

private:
    string title;

    friend ostream& operator<<(ostream& os, const Manager& man) {
        // 通过将子类引用向上造型为基类引用，迫使针对基类的操作符重载函
        // 数在针对子类的操作符重载函数中被调用，输出子类中的基类子对象
        return os << static_cast<const Employee&>(man) << ", " << man.title;
    }
};

int main(void) {
    Manager man("Zaphod", 56, "Assembly", 5000.36, "Director");

    cout << man << endl;

    return 0;
}
```

4.7 名字隐藏

4.7.1 普通成员的名字隐藏

```
// genhide.cpp

// 普通成员的名字隐藏

#include <iostream>

using namespace std;

class A {
public:
    A(void) : n(100) {}

    void foo(void) const {
        cout << "A::foo() invoked" << endl;
    }

    int n;
};

class B : public A {
public:
```

```cpp
    B(void) : n(200) {}

    void foo(int) const {
        cout << "B::foo() invoked" << endl;
    }

    void bar(void) {
        // foo(); // 编译错误，自己的foo隐藏了基类的foo，参数不匹配
        foo(0); // 调用自己的foo
        A::foo(); // 调用基类的foo

        n = 300; // 访问自己的n
        A::n = 400; // 访问基类的n
    }

    int n;
};

class C : public B {
public:
    using A::foo;
    using B::foo;

    void foo(int, int) const {
        cout << "C::foo() invoked" << endl;
    }
};

int main(void) {
    B b;

    b.bar();
    cout << b.n << ", " << b.A::n << endl;

    // b.foo(); // 编译错误，B的foo隐藏了A的foo，参数不匹配
    b.foo(0); // 调用B的foo
    b.A::foo(); // 调用A的foo

    b.n++; // 访问B的n
    b.A::n++; // 访问A的n
    cout << b.n << ", " << b.A::n << endl;

    C c;
    c.foo(); // 重载匹配到A的foo
    c.foo(0); // 重载匹配到B的foo
    c.foo(0, 0); // 重载匹配到C的foo

    return 0;
}
```

4.7.2 静态成员的名字隐藏

```cpp
// statichide.cpp

// 静态成员的名字隐藏

#include <iostream>

using namespace std;

class A {
public:
    static void foo(void) {
        cout << "A::foo() invoked" << endl;
    }

    static int n;
};

int A::n = 100;

class B : public A {
public:
    static void foo(int) {
        cout << "B::foo() invoked" << endl;
    }

    static void bar (void) {
        // foo(); // 编译错误，自己的foo隐藏了基类的foo，参数不匹配
        foo(0); // 调用自己的foo
        A::foo(); // 调用基类的foo

        n = 300; // 访问自己的n
        A::n = 400; // 访问基类的n
    }

    static int n;
};

int B::n = 200;

class C : public B {
public:
    using A::foo;
    using B::foo;

    static void foo(int, int) {
        cout << "C::foo() invoked" << endl;
    }
};

int main(void) {
    B::bar();
    cout << B::n << ", " << B::A::n << endl;

```

```
    // B::foo (); // 编译错误，B的foo隐藏了A的foo，参数不匹配
    B::foo(0); // 调用B的foo
    B::A::foo(); // 调用A的foo

    B::n++; // 访问B的n
    B::A::n++; // 访问A的n
    cout << B::n << ", " <<  B::A::n << endl;

    C::foo(); // 重载匹配到A的foo
    C::foo(0); // 重载匹配到B的foo
    C::foo(0, 0); // 重载匹配到C的foo

    return 0;
}
```

4.8 私有继承（实现继承）和保护继承

- 不良的设计

```
// private1.cpp

// 不良的设计

#include <iostream>

using namespace std;

class SalaryDB {
public:
    int querySalary(const string& name) const {
        if (name == "关羽")
            return 10000;

        if (name == "刘备")
            return 60000;

        return 0;
    }
};

// 公有继承导致SalaryDB类中的公有成员也成为Employee类的公有成员
class Employee : public SalaryDB {
public:
    Employee(const string& name) : name(name) {}

    bool fire(void) const {
        return querySalary(name) > 20000;
    }

    string name;
};

int main(void) {
    Employee emp("关羽");

```

```
    cout << (emp.fire() ? "裁掉" : "留用") << emp.name << endl;

    // 可以从外部获取本应仅限于被Employee类访问的内部信息
    cout << emp.querySalary(emp.name) << endl;

    return 0;
}
```

- 改进的设计

```
// private2.cpp

// 改进的设计

#include <iostream>

using namespace std;

class SalaryDB {
public:
    int querySalary(const string& name) const {
        if (name == "关羽")
            return 10000;

        if (name == "刘备")
            return 60000;

        return 0;
    }
};

// 私有继承保证了SalaryDB类中的公有成员在Employee类中变成私有
class Employee : private SalaryDB {
public:
    Employee(const string& name) : name(name) {}

    bool fire(void) const {
        return querySalary(name) > 20000;
    }

    string name;
};

int main(void) {
    Employee emp("关羽");

    cout << (emp.fire() ? "裁掉" : "留用") << emp.name << endl;

    // 无法从外部获取只能被Employee类访问的内部信息
    // cout << emp.querySalary(emp.name) << endl; // 编译错误

    return 0;
}
```

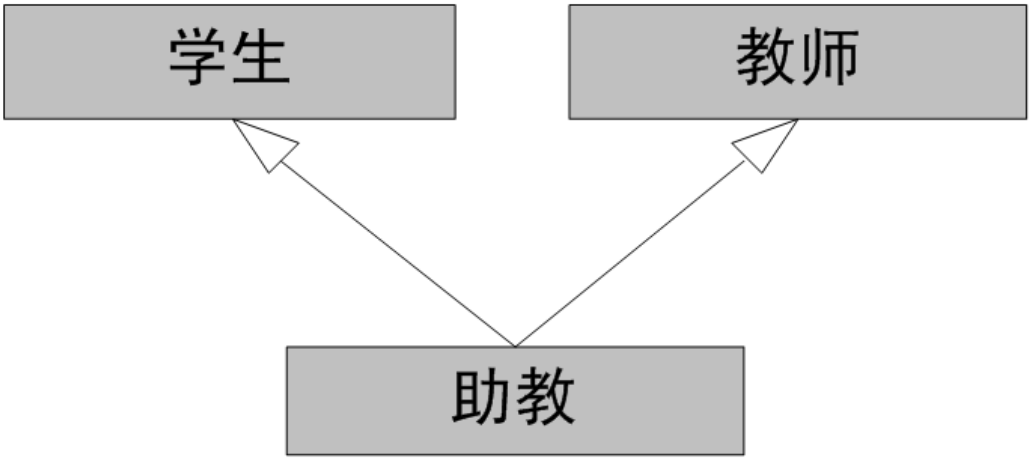
- 可扩展的设计

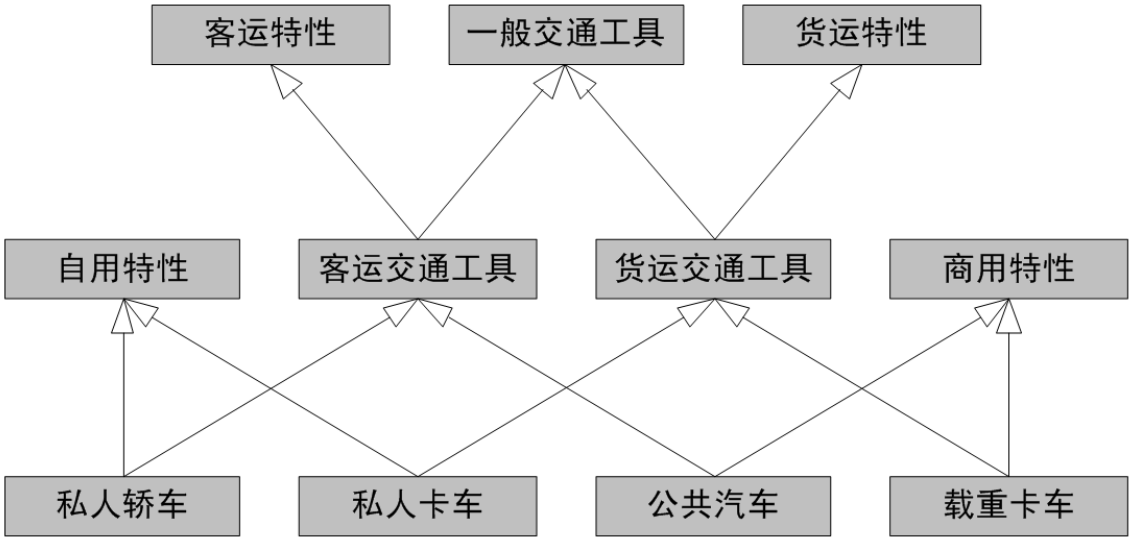
```cpp
// private3.cpp

// 可扩展的设计

#include <iostream>

using namespace std;

class SalaryDB {
public:
    int querySalary(const string& name) const {
        if (name == "关羽")
            return 10000;

        if (name == "刘备")
            return 60000;

        return 0;
    }
};

// 保护继承在确保SalaryDB的公有成员无法从Employee类外
// 部访问的同时，为其子类留有方便之门，以利于后续扩展
class Employee : protected SalaryDB {
public:
    Employee(const string& name) : name(name) {}

    bool fire(void) const {
        return querySalary(name) > 20000;
    }

    string name;
};

class Manager : public Employee {
public:
    Manager(const string& name) : Employee(name) {}

    bool fire(void) const {
        // querySalary成员函数在Employee类中是保护的，
        // 作为Employee类的子类，Manager类可以访问它
        return querySalary(name) > 50000;
    }
};

int main(void) {
    Employee emp("关羽");

    cout << (emp.fire() ? "裁掉" : "留用") << emp.name << endl;

    // 无法从外部获取只能被Employee类及其子类访问的内部信息
    // cout << emp.querySalary(emp.name) << endl; // 编译错误

    Manager man("刘备");

```

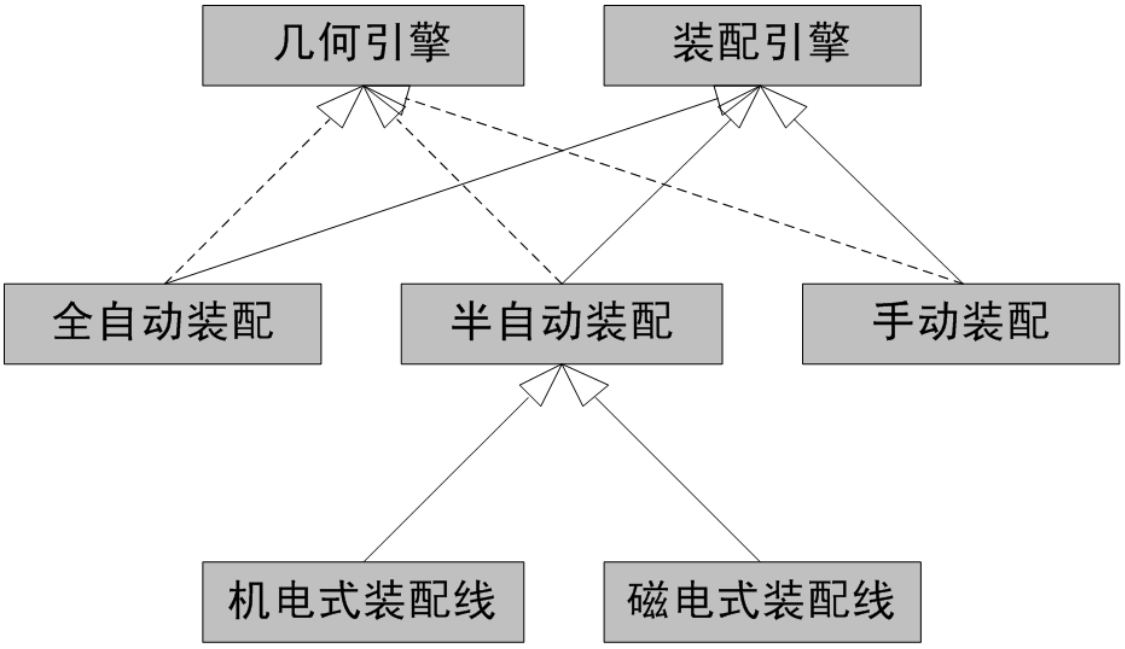
```
56      cout << (man.fire() ? "裁掉" : "留用") << man.name << endl;
57
58      // 无法从外部获取只能被Employee类及其子类访问的内部信息
59      // cout << man.querySalary(man.name) << endl; // 编译错误
60
61      return 0;
62  }
```

4.9 多重继承

4.9.1 应用场景





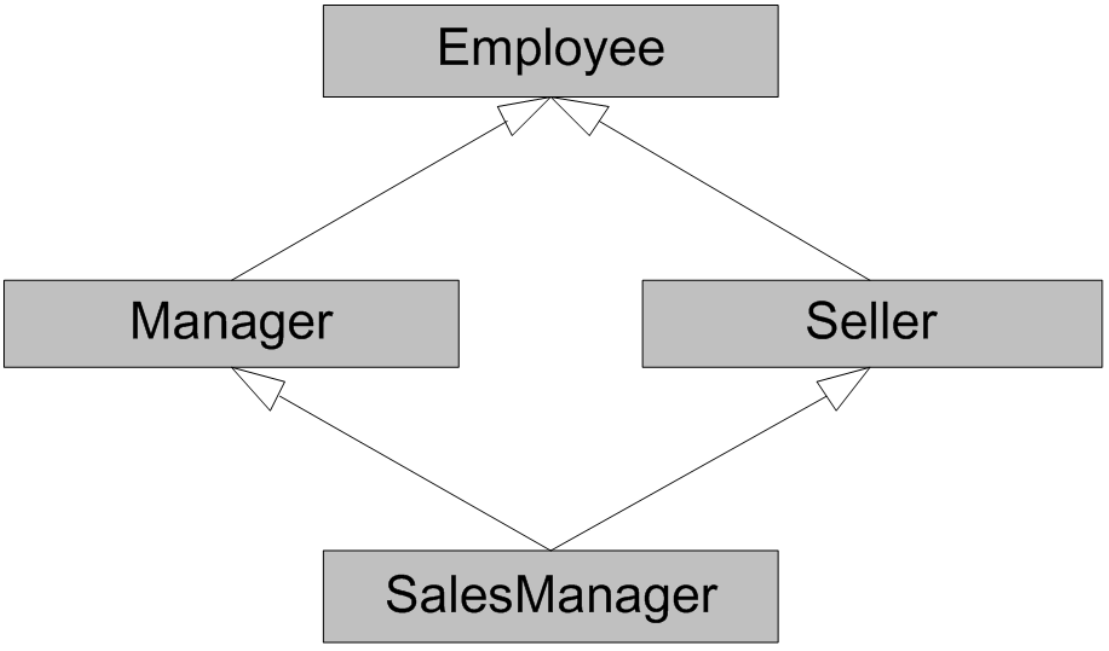


4.9.2 构造与析构顺序

```
// miorder.cpp

// 多重继承的构造与析构

#include <iostream>

using namespace std;

class X {
public:
    X(void) {
        cout << "X::X() invoked, this=" << this << endl;
    }

    ~X(void) {
        cout << "X::~X() invoked, this=" << this << endl;
    }

    int x;
};

class Y {
public:
    Y(void) {
        cout << "Y::Y() invoked, this=" << this << endl;
    }

    ~Y(void) {
        cout << "Y::~Y() invoked, this=" << this << endl;
    }

    int y;
};
```

```
34
35  class Z : public X, public Y {
36  public:
37      Z(void) {
38          cout << "Z::Z() invoked, this=" << this << endl;
39      }
40
41      ~Z(void) {
42          cout << "Z::~Z() invoked, this=" << this << endl;
43      }
44
45      int z;
46  };
47
48  int main(void) {
49      Z z;
50      cout << sizeof(z) << endl;
51
52      X* px = &z;
53      Y* py = &z;
54      Z* pz = &z;
55      cout << px << ", " << py << ", " << pz << endl;
56
57      pz = (Z*)py;
58      cout << pz << endl;
59
60      return 0;
61  }
```

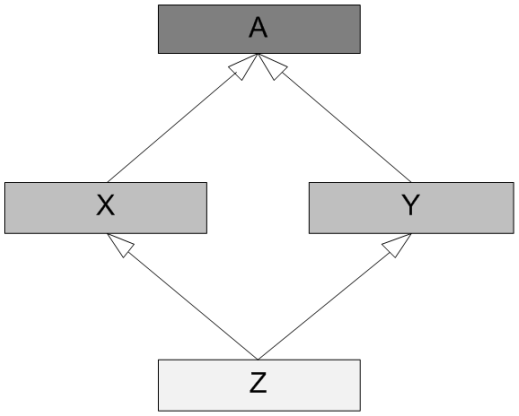
4.9.3 重复创建问题——钻石继承与虚继承

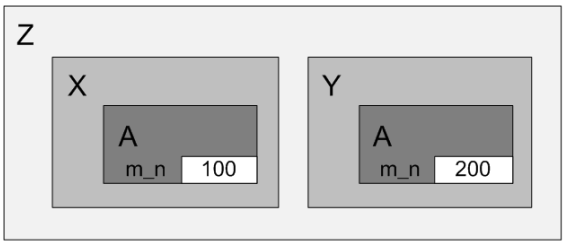



```
1  // diamond.cpp
2
3  // 多重继承的重复创建问题--钻石继承
```

```cpp

// 当一个子类沿多条不同的路径继承同一个基类时，这个
// 子类的实例化对象中将包含多个该基类的实例化子对象

#include <iostream>

using namespace std;

class A {
public:
    A(int n) : n(n) {
        cout << "A::A() invoked, this=" << this << endl;
    }

    int n;
};

class X : public A {
public:
    X(int n) : A(n) {
        cout << "X::X() invoked, this=" << this << endl;
    }

    int get(void) const {
        cout << &n << "->" << n << endl;
        return n;
    }
};

class Y : public A {
public:
    Y(int n) : A(n) {
        cout << "Y::Y() invoked, this=" << this << endl;
    }

    void set(int n) {
        this->n = n;
        cout << n << "->" << &this->n << endl;
    }
};

class Z : public X, public Y {
public:
    Z(int n) : X(n), Y(n) {
        cout << "Z::Z() invoked, this=" << this << endl;
    }
};

int main(void) {
    Z z(100);
    cout << sizeof(z) << endl;

    z.set(200);
    cout << z.get() << endl;

    return 0;
```

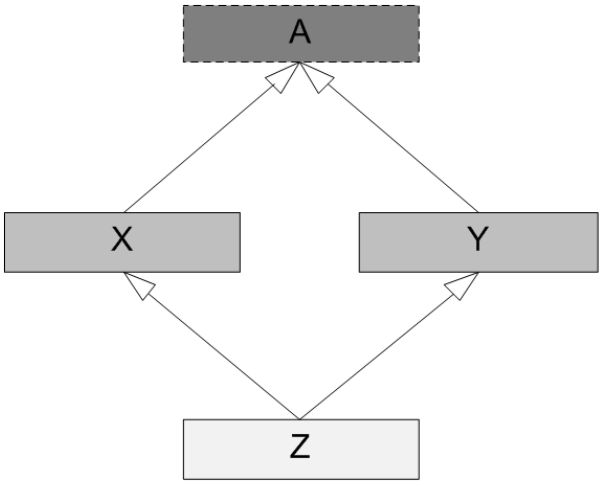
```
}
```

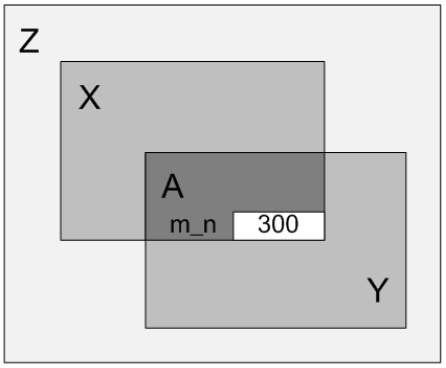





```cpp
// diamond.cpp

// 虚继承

// 将公共基类声明为虚基类，可以解决公共基
// 类子对象在间接子类对象中的重复创建问题

#include <iostream>

using namespace std;

class A {
public:
    A(int n) : n(n) {
        cout << "A::A() invoked, this=" << this << endl;
    }

    int n;
};

// 通过虚继承使A成为X的虚基类
class X : virtual public A {
public:
    X(int n) : A(n) {
        cout << "X::X() invoked, this=" << this << endl;
    }

    int get(void) const {
        cout << &n << "->" << n << endl;
        return n;
    }
};

// 通过虚继承使A成为Y的虚基类
class Y : virtual public A {
public:
    Y(int n) : A(n) {
        cout << "Y::Y() invoked, this=" << this << endl;
    }

    void set(int n) {
```

```
        this->n = n;
        cout << n << "->" << &this->n << endl;
    }
};

class Z : public X, public Y {
public:
    // 一般情况下，子类的构造函数不能调用其间接基类的构造函数。
    // 但是，一旦这个间接基类被声明为虚基类，它的所有子类（无
    // 论直接还是间接的）都必须显式调用该虚基类的构造函数。否
    // 则，系统将试图为每个子类对象调用该虚基类的缺省构造函数
    Z(int n) : X(n), Y(n), A(n) {
        cout << "Z::Z() invoked, this=" << this << endl;
    }
};

int main(void) {
    Z z(100);
    cout << sizeof(z) << endl;

    z.set(200);
    cout << z.get() << endl;

    return 0;
}
```





4.9.4 函数冲突与汇聚替代

```
// funcconf.cpp

// 多重继承的函数冲突问题

// 如果在共同派生同一个子类的两个基类中，含有名字完全相同的两个成员函
// 数，而且这个子类又没有对该成员函数进行隐藏或者覆盖，那么任何试图在
// 这个子类中，或者通过这个子类的实例化对象调用该成员函数的操作，都将
// 在编译期间引发一个链接歧义错误

#include <iostream>

using namespace std;
```

```

class X {
public:
    void foo(void) {
        cout << "X::foo() invoked" << endl;
    }
};

class Y {
public:
    void foo(void) {
        cout << "Y::foo() invoked" << endl;
    }
};

class Z : public X, public Y {
public:
    void bar(void) {
        // foo(); // 编译错误
    }
};

int main(void) {
    Z z;

    // z.foo(); // 编译错误

    return 0;
}
```

```
// converge.cpp

// 汇聚替代

// 在存在名字冲突的汇聚子类中，为该函数提供定
// 义，以隐藏或覆盖其从不同基类继承的同名函数

#include <iostream>

using namespace std;

class X {
public:
    void foo(void) {
        cout << "X::foo() invoked" << endl;
    }
};

class Y {
public:
    void foo(void) {
        cout << "Y::foo() invoked" << endl;
    }
};

```

```
class Z : public X, public Y {
public:
    // 对于多个基类中的同名成员函数，可以在汇聚子类中为其提供定义，
    // 以隐藏或覆盖该成员函数在所有基类中的版本。而该成员函数在汇
    // 聚子类中的定义很可能仅仅是根据某种条件，选择适当的基类，再
    // 辅以“::”操作符，以调用该成员函数在特定基类中的版本。这样做
    // 既避免了因名字冲突而导致的编译错误，同时又保护了基类的成果

    void foo(void) {
        cout << "Z::foo() invoked" << endl;

        X::foo();
        Y::foo();
    }

    void bar(void){
        foo();
    }
};

int main(void) {
    Z z;

    z.foo();

    return 0;
}
```

4.9.5 变量冲突与类名限定

```
// varconf.cpp

// 多重继承的变量冲突问题

// 如果在共同派生同一个子类的两个基类中，含有变量名完全相同的两个数据
// 成员，而且这两个数据成员在子类中都可见，那么任何试图在这个子类中，
// 或者通过这个子类的实例化对象访问该数据成员的操作，都将在编译期间引
// 发一个名字冲突错误

#include <iostream>

using namespace std;

class X {
public:
    X(int n) : n(n) {}

    int n;
};

class Y {
public:
    Y(int n) : n(n) {}

    int n;
```

```
};

class Z : public X, public Y {
public:
    Z(int n) : X(n), Y(n) {}

    void print(void) const {
        // cout << n << endl; // 编译错误
    }
};

int main(void) {
    Z z(100);
    z.print();

    // z.n = 200; // 编译错误
    z.print();

    return 0;
}
```

```
// scope.cpp

// 类名限定

// 一个位于多条继承路径汇聚点的类，为其不同基类中具有相同变量名的数
// 据成员提供自己的定义，固然可以解决由此而产生的名字冲突问题，但是
// 这些数据成员在基类中所体现的价值丝毫不会被继承到子类中。这里不妨
// 任由这种冲突的存在，而在对数据成员的访问上采取更加谨慎的态度

#include <iostream>

using namespace std;

class X {
public:
    X(int n) : n(n) {}

    int n;
};

class Y {
public:
    Y(int n) : n(n) {}

    int n;
};

class Z : public X, public Y {
public:
    // 在子类的构造函数中，为不同的基类构造函数传递不同的参数
    Z(int n1, int n2) : X(n1), Y(n2) {}

    void print(void) const {
        // 对不同基类中的同名数据成员，通过类名加“::”操作符加以区分
```

```
        cout << X::n << ", " << Y::n << endl;
    }
};

int main (int argc, char* argv[])
{
    Z z(100, 200);
    z.print();

    // 对不同基类中的同名数据成员，通过类名加“::”操作符加以区分
    z.X::n = 300;
    z.Y::n = 400;
    z.print();

    return 0;
}
```

4.9.5 类型转换的差别

```
// convcomp.cpp

// 通用类型转换、静态类型转换和重解释类型转换在多继承问题上的差异

#include <iostream>

using namespace std;

class A {
    int n[2];
};

class B {
    int n;
};

class C : public A, public B {};

int main(void) {
    A* pa;
    B* pb;
    C* pc = new C;

    // 通用类型转换可能（因编译器而异）做指针计算，
    // pb比pa和pc向下偏移两个整型（n[2]）
    pa = (A*)pc;
    pb = (B*)pc;
    cout << pa << ", " << pb << ", " << pc << endl;

    // 静态类型转换做指针计算，
    // pb比pa和pc向下偏移两个整型（n[2]）
    pa = static_cast<A*>(pc);
    pb = static_cast<B*>(pc);
    cout << pa << ", " << pb << ", " << pc << endl;

    // 重解释类型转换不做指针计算，
```

```
37      // pb、pa和pc的值完全一样
38      pa = reinterpret_cast<A*>(pc);
39      pb = reinterpret_cast<B*>(pc);
40      cout << pa << ", " << pb << ", " << pc << endl;
41
42      return 0;
43  }
```