程序设计实习（实验班-2023春）
面向对象编程：继承与多态

授课教师：姜少峰

助教：张宇博 楼家宁

Email: shaofeng.jiang@pku.edu.cn

继承: X is a Y

复合与继承

- 复合：has a
  - 是一种常见设计范式
  - 例如：A类含有其他类的对象（课程类的对象含有学生类的对象）
- 继承：is a
  - 今天要介绍的范式
  - 具体就是：Child对象也是一个Parent对象

带着问题：
两种范式的异同？是否可互相替代？

继承的若干动机

- 复用：可利用父类已有的成员（变量和函数）
- 扩展：可在父类基础上添加新的成员
- 改写：可将父类已经定义的成员改写、覆盖，以实现新的功能

语法

- 设Parent是父类（基类）的名字，Child是子类（派生类）的名字

  class Child : 继承访问权限 Parent {};

- 其中继承访问权限必须是

  private，public，protected之一

可以不指定；若不指定，则默认是private

（继承）访问权限

访问权限

- 区分两种：
  - 类成员访问权限：规定谁可以访问类成员 （上节课介绍的）
  - 类继承时声明的继承访问权限：规定子类中对父类成员的访问权限

（类继承时声明的）继承访问权限

为什么需要继承访问权限?

- 发生继承时，Child和Parent其实应该看作不同的类
- 但又很特殊：Child可以复用/调用Parent的成员
    - 问题：如何确定哪些成员Child中仍可以调用?

继承访问权限：指定Parent类的成员访问权限在Child类应该是怎样的

- 即，在Parent类中是public/private/protected的，到了Child中应该是什么?

继承访问权限的规则

总规则：

注：绝不是Parent的private在Child也是private，
而是更进一步，Child不可见这些Parent的private成员

- Parent类中的private成员都对Child完全不可见
- Parent类中除private成员都对Child可见，但具体访问权限依据下页规则确定

继承访问权限的规则

public继承：父类成员全部保持原访问权限

private继承：父类成员全部访问权限改为private

protected继承：父类成员全部访问权限改为protected

- 即：public继承保持原有权限不变，其余继承会重设权限

继承访问权限的规则总结

这里总结的是Child类进行public/private/protected继承后，Parent的public/private/protected成员在Child类中新的访问权限是什么

	public成员	protected成员	private成员
public继承	public成员	protected成员	不可见
private继承	private成员	private成员	不可见
protected继承	protected成员	protected成员	不可见

绝大多数情况应该用public继承（也是我们讨论的重点）
对于极少数private/protected继承合理例子最后会探讨

protected成员访问权限

回忆public和private?

我们在此补充上节课没有具体介绍的protected成员的访问权限规定

protected成员：仅在1) 当前类内部，2) 子类内部，以及3) 友元可以访问

- 介于private和public之间
- 与private的主要区别是子类内也可以访问，共同点是其他一切来源不能访问
- “内部”是指?

在类的作用域内，也可以理解成用于实现当前类的代码里

特殊规则：本类其他对象在本类内部的访问权限

该特殊规则主要为了方便实现本类的函数

- 在访问规则之上，C++有一个特殊规则：

特别地：x可见的父类成员

设当前类为X，在X内部使用X类型的其他对象x时可以访问x的所有可见成员

可以是传入的X对象，也可以是本类内部创建的其他X对象

```
class Point {
    private:
    int x, y;
    public:
    Point(int _x, int _y) : x(_x), y(_y) {}
    Point operator+(const Point& p) {
        return Point(x + p.x, y + p.y);
    }
};
```

此处p是Point的其他对象，p.x根据特殊规则允许使用

- 为什么说是“特殊规则”？private/protected不是说成员允许从类内部访问吗?
  - 原因：严格来说，“成员”应该只算this指向的，其他X的对象 (如p)不能算成员

例子：特殊规则与继承

```
struct Parent {
    protected:
    void h() {cout << "parent" << endl;}
};
struct Child : public Parent {
    void f() {cout << "child" << endl;}
    void g() {
        Child child;
        Parent parent;
        f(); // 输出child
        h(); // 访问父类函数，输出parent
        child.f(); // 根据特殊规则允许调用任何可见函数，输出child
        child.h(); // 根据特殊规则允许调用任何可见函数，输出parent
        parent.h(); //编译错误!
    }
};
```

虽然h()是protected，但可以访问 —
因为h()算做是（this的）可见成员

此处parent对象不是（this的）成员，且不是Child型，因此特殊规则不适用于parent
（特殊规则只对“本类”Child类型对象child适用）

子类的构造函数

当父类有“无参数构造”时会被默认调用

Child所有构造函数必须确保父类Parent被构造

很多时候父类变量是private的，在子类中甚至都不可见！

- 一般通过调用父类构造函数进行构造，而不要试图直接为父类变量初始化

必须在子类构造函数的初始化列表进行，而不允许在函数体内调用！

规则：
父类在子类构造之前构造

```
class Student {
    int id;
    public:
    int getId() const {return id;}
    Student(int _id) : id(_id) {}
};
class GradStudent : public Student {
    string advisorName;
    public:
    GradStudent(string an, int id) :
        Student(id), advisorName(an) {}
};
```

子类的析构函数

规则：
父类析构会在子类析构之后调用

```
struct Parent {
    int *arr;
    Parent() {arr = new int[10000];}
   ~Parent() {
        cout << "parent destructor" << endl;
        delete [] arr;
    }
};
struct Child : public Parent {
    int *arr_child;
    Child() {arr_child = new int[10000];}
    ~Child() {
        cout << "child destructor" << endl;
        delete[] arr_child;
    }
};
```

```
int main() {
    Child *p = new Child;
    delete p;
    return 0;
}
```

输出：
child destructor
parent destructor

多次继承

- 继承可以递归进行下去，例如可以有类A_1继承A_2，A_2继承A_3...
- 称A_2为A_1的直接父类，$A_3, \ldots$为间接父类
- 在声明A_1时，只列出直接父类A_2，不列出$A_3, \ldots$
  - 编译器会自动递归解析所有间接父类

```
struct A1 {};
struct A2 : public A1 {};
struct A3 : public A2 {};
```

继承与友元

友元关系不会因继承扩展到子类（但父类的友元关系依然成立）

- 即继承后，父类的友元依然可以访问父类的private成员，但对子类不成立

```
class Child;
struct X {
    Child *p;
    void f();
};
struct Parent {
    private:
    int x;
    friend class X;
};
struct Child : public Parent {
    private:
    int y;
};
```

```
void X::f() {
    p = new Child;
    cout << p->y << endl; // 编译错误：子类的友元关系不存在，y不可见
    cout << p->x << endl; // 可以调用：父类的友元关系依然存在
}
```

覆盖
Override

覆盖

思考：
与重载（overload）的区别和联系
- 重载只能是函数，变量无法重载
- 重载函数的形参表必须不同
- 重载与覆盖都是同函数名

这种行为叫做覆盖（override）

- 子类允许定义与父类中“相同”的成员（变量和函数）
- 对于变量来说：同变量名（类型可以相同或者不同）

例：父类有public的void f()，则子类的private的void f()也算是覆盖

注意：这些规则与访问权限无关！
权限并不是区分函数的标准！

- 对于函数来说：
  - 同函数名且同参数表（返回值类型可以相同也可以不同）
- 特点：若放在同一个类里会因重定义导致编译错误，但在子类重定义就是覆盖！
  - 深层原因：子类与父类处在不同的作用域

举例

子类Child中成员变量k和成员函数f都覆盖了父类Parent对应的成员

```
struct Parent {
    int k;
    void f() {cout << "parent" << endl;}
};

struct Child : public Parent {
    double k;
    void f() {cout << "child" << endl;}
};
```

如何调用父类被覆盖的成员?

- 在Child中，对覆盖的成员的调用默认会使用Child的版本
- 如何调用Parent中的版本?

Parent::成员

- 类比用this指针指定作用域

一般地，如果有多个父类，应该用
类名::成员

例子

```
struct Parent {
    protected:
    int k = 1;
    void f() {cout << "parent" << endl;}
};
struct Child : public Parent {
    double k = 0.9;
    int t = 1;
    void f() {cout << "child" << endl;}
    void g() {
        Child child;
        f(); // 输出child
        Parent::f(); // 输出parent
        child.f(); // 根据特殊规则允许调用，输出child
        child.Parent::f(); // 根据特殊规则允许调用，输出parent
        cout << k << " " << Parent::k << " " << t << endl;
    }
};
```

```
int main() {
    (Child()).g();
    return 0;
}
```

原则：
尽量不要定义同名**成员变量**
但是同名**成员函数**没问题

注意这种写法

此处输出0.9 1 10

t变量只在Parent定义过，所以此处可以不加Parent::

父类-子类的类型转换与赋值兼容

大前提：public继承，否则不可赋值

原因在后面会讨论

- 子类对象可直接赋值给父类对象，以下都是合法的

  Parent parent = child，Parent &parent = child，Paring *parent = &child

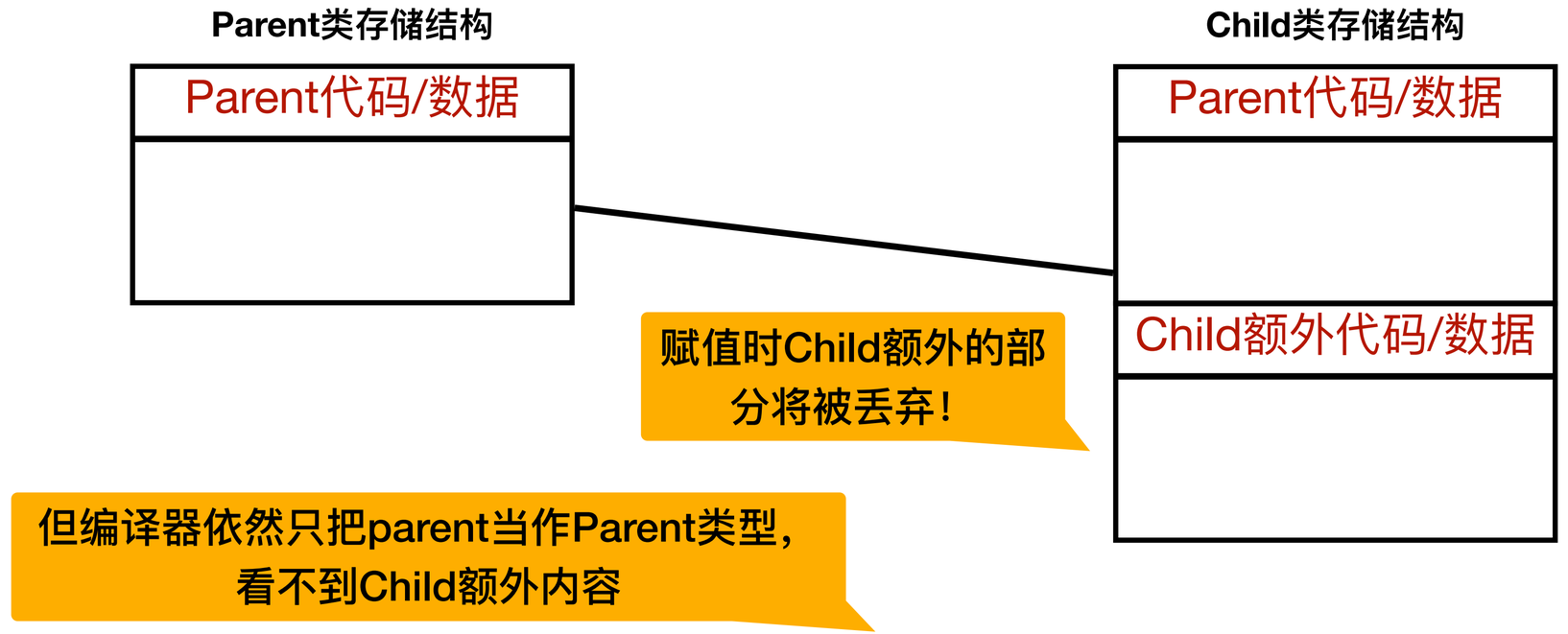
- 三种写法共同行为：丢失子类信息，编译器只会把左值父类变量当作父类类型

```
struct Parent {
    void f() {cout << "parent" << endl;}
};
struct Child : public Parent {
    void f() {cout << "child" << endl;}
    void g() {cout << "child" << endl;}
};
```

```
int main() {
    Parent *p = new Child;
    p->f(); // 输出parent，且无法调用Child版本的f()
    p->g(); // 编译错误：只能看到p是parent类型的，因此看不到g
    return 0;
}
```

赋值转换规则

- Parent parent = child这里发生了什么?



- 如果是Parent* parent = &child，那么Child所存储的内容仍存在

父类指针强制转换为子类指针

一种“找回”子类信息的方法

- 考虑parent指针实质指向的是child对象内存的情况，即

```
Parent *parent = &child;
```

- 此时仍可通过强制把指针转换回Child*来要调用所指向的Child成员

```
Child *p = (Child*) parent;
```

程序员需确保parent指向了Child类型的数据，否则可能发生为定义行为

- 然而：若采用Parent parent = Child()写法，则parent完全丢掉了Child信息！

会调用Parent的复制构造进行类型转换！

之后强行Child *p = &parent也不再安全

但是……

我们还是无法处理这种应用

类：Role

数据：
共有的数据，如攻击，血量，防御，暴击，暴伤等数值，人物3D模组等

函数：
1. 对各项属性的修改
2. 行动有关的函数，走，跑，冲刺，跳
3. 接口：普攻，元素战技，元素爆发

类：Role1

数据：继承自父类型，外加独特信息，例如技能有关数据（如CD）

函数：继承自父类型，按照接口实现三种攻击方式

```
struct Role {
    void attack() {
        // ...
    }
};
struct RoleX : public Role {
    void attack() {
        // ...
    }
};
```

现在这种写法还只能调用父类的attack

```
int main() {
    Role* roleList = {new Role1(), new Role2(), new Role3()};
    roleList[i]->attack();
    return 0;
}
```

想要把具体的角色子类放入一个父类Role*数组，然后通过roldeList[i]->attack()调用子类对应的普攻函数

如何“不丢失”子类信息：
用父类指针调用子类版本的函数?

virtual

在Parent的f函数声明中加virtual关键字，即可让parent指针调用子类版本的函数f

```
struct Parent {
    virtual void f() {cout << "parent" << endl;}
};
struct Child : public Parent {
    void f() {cout << "child" << endl;}
};
```

```
int main() {
    Parent *parent = new Child();
    parent->f();
    return 0;
}
```

输出child!

虚函数

- 有virtual关键字的成员函数被称为虚函数
- virtual只能用在函数声明中出现（加在返回值前/后均可，但要在函数名之前）
  - 若定义是之后给出的，则不要再加virtual修饰
- 静态成员函数不能是虚函数

```
struct Parent {
    void virtual f();
};
void Parent::f() {cout << "parent" << endl;}
```

- 注：只有函数可以加virtual，变量是不行的！

虚函数的行为：动态绑定

指针、引用

```
struct Parent {
    virtual void f() {cout << "parent" << endl;}
};
struct Child : public Parent {
    void f() {cout << "child" << endl;}
};
```

- 设Parent有虚函数f，且子类覆盖了该函数
- 指针：Parent *parent可赋值Child或Parent对象，调用的f会根据指向对象选择

这种行为叫做动态绑定
这种编程概念叫做“多态”

```
int main() {
    Parent p; Child c;
    Parent *parent = &p;
    parent->f(); // 输出parent
    parent = &c;
    parent->f(); // 输出child
    return 0;
}
```

```
int main() {
    Parent p; Child c;
    Parent& parent = c;
    parent.f(); // 输出child
    return 0;
}
```

采用引用类型效果也是类似的

动态绑定的触发条件

- 但是：不用指针、引用，而直接用平常变量进行赋值，将丧失动态绑定行为！

```
int main() {
    Parent p; Child c;
    p = c;
    p.f(); // 输出parent!
    return 0;
}
```

p = c这句调用的是赋值，仅仅是将c的Parent部分拷贝给了p，丢失了c所有其他信息

此时的p存储的内容完全是一个Parent型，因此只可能调用Parent的f，即使f是virtual的！

因此产生动态绑定行为的条件是：

- 在子类中被覆盖的函数在父类是virtual的
- 调用该函数时要用指针或者引用
- (子类是public继承，否则甚至无法把子类地址赋值给父类指针)

虚函数与访问权限

- 一般虚函数需要是public或者protected，如果是private会发生这种情况：

```
struct Parent {
    private:
    void virtual f() {cout << "parent" << endl;}
};
struct Child : public Parent {
    void f() {cout << "child" << endl;}
};
```

```
int main() {
    Parent p; Child c;
    Parent* parent = &c;
    parent->f(); // 编译错误
    return 0;
}
```

因此f函数被当作不可见的

- 这里编译错误是因为编译器始终会依据parent的类型信息来解析

虚函数与访问权限 (cont.)

- 反过来，Parent的虚函数是public的，但Child的覆盖是private的，会怎样?

```
struct Parent {
    public:
    void virtual f() {cout << "parent" << endl;}
};
struct Child : public Parent {
    private:
    void f() {cout << "child" << endl;}
};
```

```
int main() {
    Child* child = new Child;
    child->f();
    return 0;
}
```

但是直接调用child的f()
就会编译错误!

可以编译通过：因为编译器无法检查Child的f()的情况

```
int main() {
    Parent* parent = new Child;
    parent->f();
    return 0;
}
```

很难仅依赖编译时信息来禁止这种访问，
除非明确禁止子类以private重写f()

程序输出child：因为动态绑定生效了!
这绕过了子类的private权限限制!

更“离谱”的行为...

Good practice：

子类对虚函数的重写尽量保持与父类访问权限一致

父类声明虚函数时尽量不要private，可以考虑protected

```
struct Parent {
    private:
    void virtual f() {cout << "parent" << endl;}
    public:
    void g() {
        f();
    }
};
struct Child : public Parent {
    private:
    void f() {cout << "child" << endl;}
};
```

```
int main() {
    Parent* parent = new Child;
    parent->g();
    return 0;
}
```

程序依然编译通过并输出child

Parent的虚函数f()在public继承时对Child甚至是不可见的，但动态绑定依然发生了

（虽然不可见，但对象内存依然有Parent的部分）

根本原因依然是：

C++只能利用编译时、父类信息

如何选择不采用动态绑定

- 如果有时候想暂时不要动态绑定的行为，想调用父类的f，怎么办?
  - 可以通过::Parent.f()来实现

```
int main() {
    Parent p; Child c;
    Parent* parent = &c;
    parent->f(); // 输出child
    parent->Parent::f(); // 输出parent
    return 0;
}
```

测试题

- 输出parent还是child?

```
struct Parent {
    void virtual f() {cout << "parent" << endl;}
    void g() {
        f();
    }
};
struct Child : public Parent {
    void f() {cout << "child" << endl;}
};
```

这个f()等价于this->f()
调用的是哪个类的f()?

```
int main() {
    Parent p; Child c;
    Parent* parent = &c;
    parent->g(); // 输出???
    return 0;
}
```

多层继承下的virtual

- virtual会传递：
  - 即某个父类的函数f一旦加了virtual，那么所有子类f也是virtual的

即使子类没有显式加virtual

```
struct Parent {
    virtual void f() {cout << "parent" << endl;}
};
struct Child : public Parent {
    void f() {cout << "child" << endl;}
};
struct Child2 : public Child {
    void f() {cout << "child2" << endl;}
};
```

f在Child2也依然是virtual的

```
int main() {
    Parent p; Child c; Child2 c2;
    Parent* parent = &c2;
    parent->f(); // 输出child2
    return 0;
}
```

测试题

- 输出parent还是child?

```
struct Parent {
    void f() {cout << "parent" << endl;}
};
struct Child : public Parent {
    virtual void f() {cout << "child" << endl;}
};
struct Child2 : public Child {
    void f() {cout << "child2" << endl;}
};
```

virtual改到这里

```
int main() {
    Parent p; Child c; Child2 c2;
    Parent* parent = &c2;
    parent->f(); // 输出???
    return 0;
}
```

编译器怎么实现动态绑定?

- C++生成的代码完全依赖于编译时的信息，不会借助运行时
- 问题：编译时无法知道一个父类指针指向的子类类型

为什么编译时不能知道?

```
int main() {
    Parent* parent = new Child;
    parent->f();
    return 0;
}
```

“知道”指向类型的例子

```
int main() {
    Parent* parent = NULL;
    int t; cin >> t;
    if (t == 0) parent = new Parent;
    else parent = new Child;
    return 0;
}
```

无法编译时确切知道指向类型的例子

- 因此只能根据父类指针的类型进行编译 — 那如何实现到子类函数的动态绑定?

编译器怎么实现动态绑定?

vtable

这个表类似于一个static的成员，是独立于具体对象、与类一一对应的

- 每个类会有一个表 (vtable)，汇总了自己的类型和各个（重写后的）函数入口
- 父类的指针指向的不光是子类对象的地址，同时也指向对象类型的这个表

实现上：创建对象的时候，vtable也会复制一份拼接在对象内存区域的前面，因此通过父类指针的地址也可以得到子类vtable

- 运行时就找到指向的对象的内存，结合这个表，得到应该调用的函数的入口

这只能实现虚函数/动态绑定
仍不能看到超出父类编译时可以得到的信息

动态绑定确实是运行时确定的
但实现上只需要借助编译时

规定：构造、析构函数内无动态绑定行为

在构造、析构函数中，即使调用虚函数，也不会动态绑定

- 永远只会调用自己类（如果有的话）/父类的虚函数实现

```
struct Parent {
    void virtual f() {cout << "parent" << endl;}
    Parent() {f();}
    void g() {f();}
};
struct Child : public Parent {
    void f() {cout << "child" << endl;}
};
```

```
int main() {
    Parent *p = new Parent();
    Child *c = new Child();
    p = c;
    p->g();
    return 0;
}
```

父类构造内调用f()
输出parent

子类构造时调用父类构造内的f()
输出parent（无动态绑定！）

输出child：只有这里发生动态绑定！

输出：parent parent child

虚析构函数

动态绑定场景下，编译时只会看到父类类型

- 因此：delete父类对象时只会调用父类析构函数，指向的子类未调用自身析构！

```
struct Parent {
    int *arr;
    Parent() {arr = new int[10000];}
   ~Parent() {
        cout << "parent destructor" << endl;
        delete [] arr;
    }
};
struct Child : public Parent {
    int *arr_child;
    Child() {arr_child = new int[10000];}
    ~Child() {
        cout << "child destructor" << endl;
        delete[] arr_child;
    }
};
```

```
int main() {
    Parent *p = new Child;
    delete p;
    return 0;
}
```

只会输出
parent destructor

解决方案：析构函数也设置成virtual

例子

一般而言：
一旦有虚函数就应该考虑把析构也设置成virtual

```
struct Parent {
    int *arr;
    Parent() {arr = new int[10000];}
    virtual ~Parent() {
        cout << "parent destructor" << endl;
        delete [] arr;
    }
};
struct Child : public Parent {
    int *arr_child;
    Child() {arr_child = new int[10000];}
    ~Child() {
        cout << "child destructor" << endl;
        delete[] arr_child;
    }
};
```

```
int main() {
    Parent *p = new Child;
    delete p;
    return 0;
}
```

输出
child destructor
parent destructor

严格来说这里和之前介绍的动态绑定不太一样，
因为~Child并不是~Parent的覆盖

是否与“析构函数内无动态绑定行为”矛盾?

- 不矛盾：析构函数内无动态绑定，但析构函数是系统在外部销毁对象时调用的
- 相关问题：析构函数可以是private的吗?

```
struct X {
    private:
    ~X() {}
};

int main() {
    X x;
    return 0;
}
```

编译错误，因为编译器无法调用析构函数

纯虚函数与抽象类

- 纯虚函数：virtual函数声明的尾部加上 = 0且不提供定义（即virtual void f() = 0;)
- 包含纯虚函数的叫做抽象类

  子类若实现了所有纯虚函数，就不再是抽象类

  - 只能用来创建子类，不可创建本类对象
  - 但是指针和引用可以指向（非抽象）子类创建的对象

```
struct Parent {
    void virtual f() = 0;
};
struct Child : public Parent {
    void f() {cout << "child" << endl;}
};
```

f()是纯虚函数
Parent是抽象类

f()不再是纯虚函数
Child不是抽象类

```
int main() {
    Parent p; // 编译错误：抽象类不可创建对象
    Parent* parent = new Child;
    parent->f();
    return 0;
}
```

输出child

非抽象类
可以创建对象

关于纯虚函数

- 构造和析构中不能调用纯虚函数（但可以调用虚函数）
- 纯虚函数也是虚函数，因此可以当作平常（虚）函数来使用

```
struct Parent {
    // Parent() {f();} 编译错误：因为构造函数不能调用纯虚函数
    void virtual f() = 0;
    void g() {f();}
};
struct Child : public Parent {
    void f() {cout << "child" << endl;}
};
```

纯虚函数可在普通成员函数调用

```
int main() {
    Parent* parent = new Child;
    parent->g();
    return 0;
}
```

输出child

多继承

多继承

- C++支持一个子类继承自多个父类
- 语法：

```
class Child: 访问权限 Parent1, 访问权限 Parent2, … { };
```

二义性

- 当两个父类都有某个成员时，需要显式用 类名:: 来指定
- 二义性在检查访问权限之前就会进行，不能通过private等消除二义性！

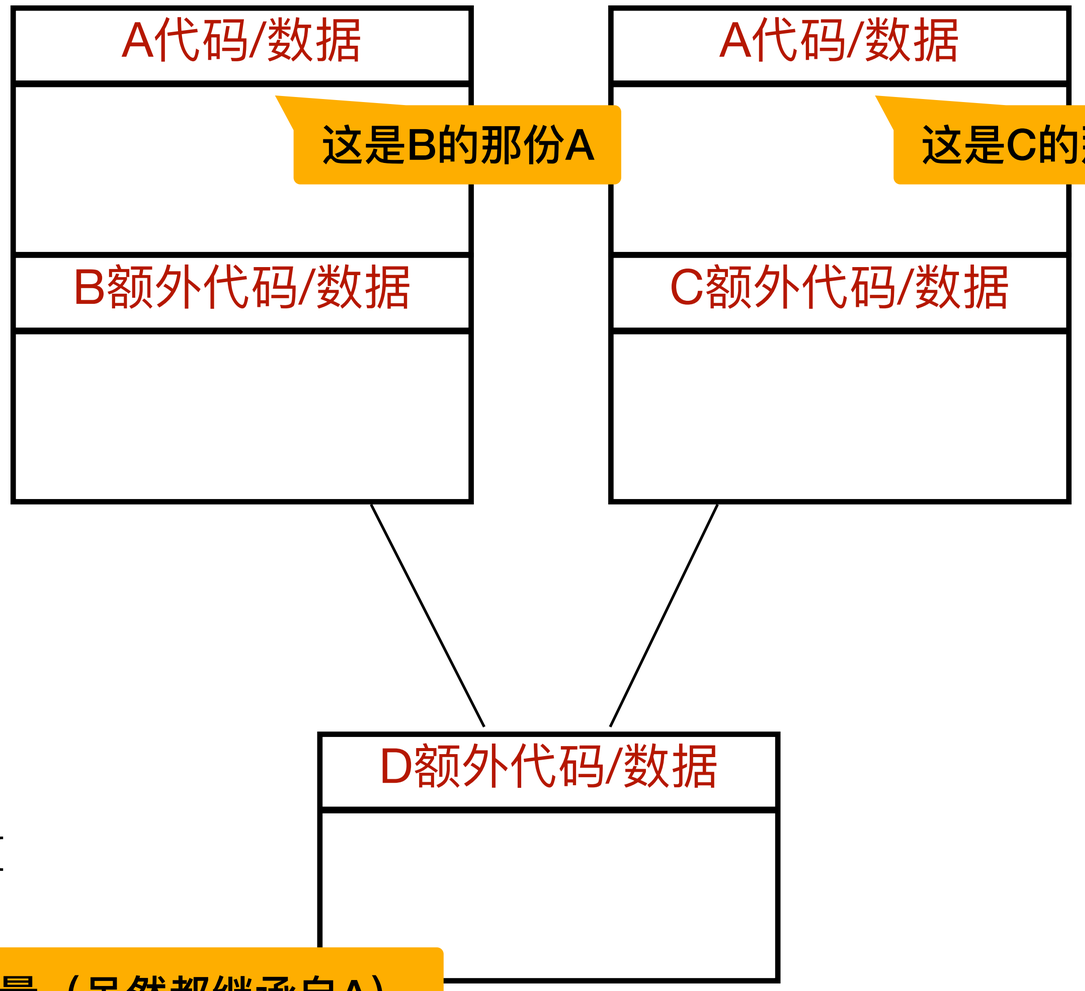
```
struct Parent {
    int k;
    void virtual f() {cout << "parent" << endl;}
};
struct Parent1 {
    private:
    int k;
};
struct Child : public Parent, public Parent1 {
    void f() {cout << k << endl;}
};
```

因歧义编译错误：
即使Parent1的k对Child不可见！

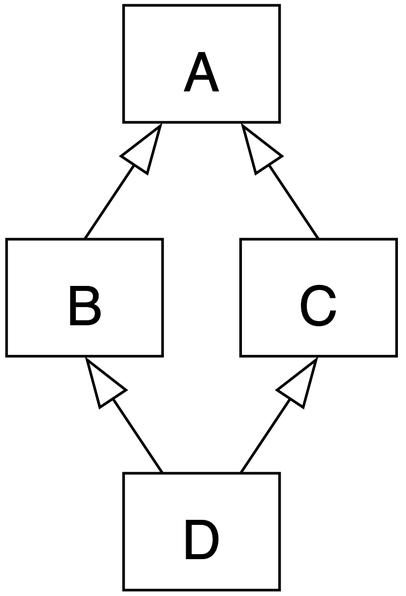
更严重的二义性

菱形继承

```
struct A {
    int x;
};

struct B : public A {
};

struct C : public A {
};

struct D : public B, public C {
};
```



因此调用A::x就有歧义：
是B的还是C的那份？



在D中，::B.x和::C.x是独立的变量（虽然都继承自A）
但是调用::A.x会编译错误，因为有两份::A.x，有歧义

这也是多继承的主要问题之一
应该尽可能避免带二义性的多继承

* 解决菱形继承问题：virtual继承

注：这里的virtual与虚函数的virtual字面上一样，但是功能上没有相似点，不要混淆！

- 改变A被复制两次这一行为：继承时采用virtual关键字

struct B : virtual public A {};

struct C : public virtual A {};

virtual需要加在父类名称A之前，可以在public之前也可以是之后

行为：只留一份A，B和C共用这份A

```
struct A {
    int x;
};

struct B : public virtual A {};

struct C : public virtual A {};

struct D : public B, public C {};
```

两个virtual都要加，只加一个依然有歧义

```
int main() {
    D d;
    d.A::x = 5;
    cout << d.B::x << " " << d.C::x << " " << d.x << endl;
    return 0;
}
```

这里输出5 5 5
也就是x确实是共用的同一份

多继承与纯虚函数的推荐用法：接口类

接口类：只包含少量纯虚函数的类，这几个纯虚函数通常共同规定了某种功能

- 一般有配套的类库：只要实现了接口类规定的纯虚函数，就可以完成某些操作
- 例如：

用于实现哈希表

```
struct Hashable {
    virtual int hashCode() = 0;
    virtual bool equals() = 0;
};
```

用于实现GUI/画板

```
struct Drawable {
    virtual int* getPixels() = 0;
};
```

```
struct BinaryOpt {
    virtual std::string toBin() = 0;
    virtual void restoreFromBin(std::string) = 0;
};
```

用于实现保存/恢复当前类的对象的功能

```
struct HammingEmbedable {
    virtual HammingPoint
      getHammingCoordinate(int i) = 0;
}
```

用于实现Hamming空间上的最近邻查找

例子

```
struct BinaryOpt {
    virtual string toBin() = 0;
    virtual void restoreFromBin(stringstream&) = 0;
};
```

- 可以写一个统一的写入/读出的函数
  - 接受BinaryOpt* p，只需要调用p->toBin()和p->restoreFromBin(ss)

```
struct C1 : public BinaryOpt {
    int x, y;
    string toBin() {
        stringstream ss;
        ss << x << " " << y << endl;
        return ss.str();
    }
    void restoreFromBin(stringstream& ss) {
       ss >> x >> y;
    }
};
```

```
struct C2 : public BinaryOpt {
    C1 *c1;
    double z;
    string toBin() {
        stringstream ss;
        ss << c1->toBin() << endl;
        ss << z << endl;
        return ss.str();
    }
    void restoreFromBin(stringstream& ss) {
        c1->restoreFromBin(ss);
        ss >> z;
    }
};
```

可以对变量递归的进行restoreFromBin

好处

- 模块化/解耦合：
  - 与使用接口的类本身的功能独立，且接口间一般也相互独立
  - 工具类库只需要接口，不需要知道任何类内的其他细节
- 避免多继承的二义性：没有成员变量，只是规定功能

接口也实现了声明与定义的分离、解耦合

关于非public继承

思考：非public继承为何禁止子类指针赋值给父类?

- 我们的讨论大多是针对Parent* parent = new Child这种用法的
- 回忆大前提：需要是public继承，否则编译器禁止Parent *parent = new Child
- 为什么要禁止?
  - 要使这种parent = new Child的用法有意义，需要让Parent是Child的子集
  - 如果非public继承，Parent的public成员在Child中访问权限降低了
    - 有些Parent的public成员外界不可访问!

至少是“外界”看来的子集

概念上不再是is-a关系了

思考：何时用protected/private继承?

一般认为是“功能”/“代码实现“上的继承，而不是父-子逻辑的继承

绝大多数时候不需要使用

public继承是is-a关系，protected/private继承是is-implemented-in-terms-of关系

- 对外界来说，protected继承和private继承都会导致父类的public成员不再可用
  - 很多时候这不太自然：子类移除了父类的某些功能?
- 因此，主要应用场景是要“隐藏”父类的一些功能
- 例如：一个支持append等修改的String类想继承出只读的ReadOnlyString类
  - 这个ReadOnlyString类要去掉/隐藏String的修改有关的函数
  - 然后再添加若干高效匹配/查询函数（基于只读假设才能用的算法）

但是……

- 刚刚的例子，是否也可以用复合来实现呢?
  - 比如说，ReadOnlyString里面直接创建一个String成员变量?
  - 若ReadOnlyString仅仅想隐藏一部分String的函数而不复用数据，则可行
    - 而且一般而言我们应先尝试复合而不是private继承，降耦合度/避免多继承
- 但：若需要复用String的数据，则需要private/protected继承

用protected继承可将这些数据结构继续暴露给未来的子类

当然，这需要String类把数据结构都声明成protected而不是private

```
struct String {
    protected:
    char* arr;
};
```

另一个非public继承的场景

- 考虑另一种“接口”类：
  - 本身实现较为独立、但复杂的算法/功能
  - 但这种实现需要的一些基本操作依赖于一个纯虚函数
  - 换句话说，子类可以继承这个接口，并实现纯虚函数，来利用该接口的功能
- 这种接口类把一切成员定义成protected，然后子类也protected继承

与普通接口不同：普通接口只是抽象出需要依赖的基本操作，如何利用这种操作在其他类库实现

一般用于封装一些独立的功能模块，与使用这些功能的完全没有is a关系
并且这种功能主要是面向过程的，不适合用复合

一个将一般LSH映射到Hamming Space的例子

你还能想到其他合理应用吗?

LSH纯虚函数：接受seed返回当前对象的hash value（需要满足是LSH）

```
struct LSHToHamming {
    protected:
    virtual int LSH(int seed) = 0;
    Point toHammingPoint() {
        //
    }
};
```

这个省略了细节的toHammingPoint函数返回当前对象在Hamming space的坐标

一个维护整数元素的集合类，采用LSHToHamming增加了映射到Hamming space的功能，并且最后把这个功能暴露给外界

```
struct Set : protected LSHToHamming{
    public:
    void addElememnt(int);
    Point toHammingPoint() {return LSHToHamming::toHammingPoint();}
    protected:
    int LSH(int seed) {
        // 采用MinHash来给出Jaccard Similarity的LSH值
    }
};
```

确定多态对象的运行时类型

需求

- 假设有Student类，UndergradStudent和DoctorStudent都继承了Student
- Student中有一个代表身高的成员变量
- 现有一堆UndergradStudent和DoctorStudent的对象，要求放一起按照身高排序
- 排序后输出学生类型等父类Student可能不含有、只在子类有的信息
- 实现上的问题：
  - 用Student* list数组存储所有子类对象后sort(list, list + n)，但然后呢？？

需要定义<操作符按照身高比较

```
struct Student {
    int height;
};

struct UndergradStudent : public Student {
    void f();
};

struct DoctorStudent : public Student {
    void g();
};
```

```
void func(Student list[], int n) {
    sort(list, list + n);
    for (int i = 0; i < n; i++) {
        // 如何得到list[i]的类型，并且根据类型调用f()或者g()?
    }
}
```

在C++中得到变量运行时/动态绑定类型

直接得到类型名称：typeid

- typeid(x).name()可以给出一个指示x在运行时类型的字符串 (C++ 11引入)
  - 然而： 输出是编译器决定、没有保证的

甚至没有文档说明输出应该是什么

```
int main() {
    Parent *p = new Parent();
    Child *c = new Child();
    cout << typeid(*p).name() << endl;
    p = c;
    cout << typeid(*p).name() << endl;
    cout << typeid(Parent()).name() << endl;
    print(p);
    cout << typeid(std::string).name() << endl;
    return 0;
}
```

这是在我的Mac上的输出：

```
6Parent
5Child
F6ParentvE
NSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE
```

？？？

不推荐使用：因为行为不明确，且输出可能比较难以parse

在C++中得到变量运行时/动态绑定类型

间接得到类型：dynamic_cast

可以使用
if (dynamic_cast<目标类型>(变量))
来判断变量是否是目标类型

- C++中有一种专门针对多态对象的继承关系转换操作符

dynamic_cast<目标类型>(变量)

目标类型必须为指针或者引用类型，因此变量的类型也应该是对应的指针/引用类型

- 行为（以指针型为例）：
  - 若变量指向的对象匹配目标类型，则返回目标类型指针（但地址不变）
  - 否则返回NULL

如果允许下面操作则为匹配：
Parent *parent = new Child

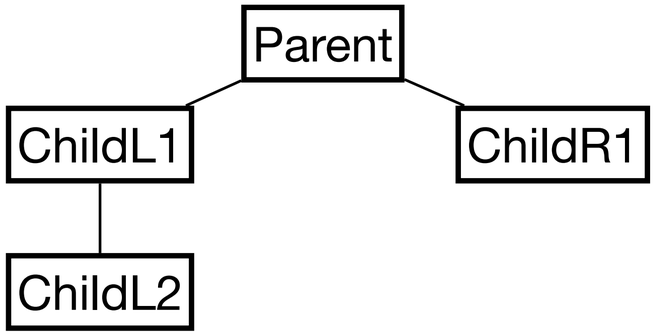
- 要求：必须在定义过至少一个虚函数的对象上使用（否则编译错误）

不构成限制：因为一个good practice就是总是为Parent定义virtual析构（即使该析构在Parent类什么也不需要做）

例子

good practice：总是定义虚析构函数

```
struct Parent {
    virtual ~Parent() {}
};
struct ChildL1 : public Parent {
};
struct ChildL2 : public ChildL1 {
};
struct ChildR1 : public Parent {
};
```




```
int main() {
    Parent *p = new Parent();
    cout << dynamic_cast<ChildL1*>(p) << endl; // 输出 NULL
    p = new ChildL2();
    cout << dynamic_cast<Parent*>(p) << endl; // 非 NULL
    cout << dynamic_cast<ChildL1*>(p) << endl; // 非 NULL
    p = new ChildR1();
    cout << dynamic_cast<ChildL1*>(p) << endl; // NULL
    cout << dynamic_cast<Parent*>(p) << endl; // 非NULL
    return 0;
}
```

如果不是菱形继承，则继承关系可以用树来表示
且这可以保证任何节点的类型可以转换成祖先类型

因此可以用这种方法判断继承关系树上的相对关系！

关于cast

- 除了dynamic_cast还有其他几种类型转换操作
  - static_cast, const_cast, reinterpret_cast
- 可以参考下面的材料自学：
  - https://cplusplus.com/doc/tutorial/typecasting/
  - https://stackoverflow.com/questions/332030/when-should-static-cast-dynamic-cast-const-cast-and-reinterpret-cast-be-used

reinterpret_cast较危险

另外一种可能性

定义一个接口，“人为”实现支持输出类名的功能

```
struct WithName {
    virtual string getName() const = 0;
};
```

```
struct Parent : public WithName {
    virtual ~Parent() {}
    string getName() const {return "Parent";}
};
struct Child : public Parent {
    string getName() const {return "Child";}
};

int main() {
    Parent* parent = new Child;
    cout << parent->getName() << endl;
    return 0;
}
```

动态绑定，输出Child

由此可以利用getName得知指针指向的对象类型

缺点：已有类库无法直接加上WithName功能

一定要支持并且不想重写的话，可以写一个继承该类X并且实现WithName接口的子类

```
struct X {};

struct XWithName : public X, public
WithName {
    string getName() const {return "X";}
}
```