面向对象程序设计简介

周练

2017.9.25

主要内容

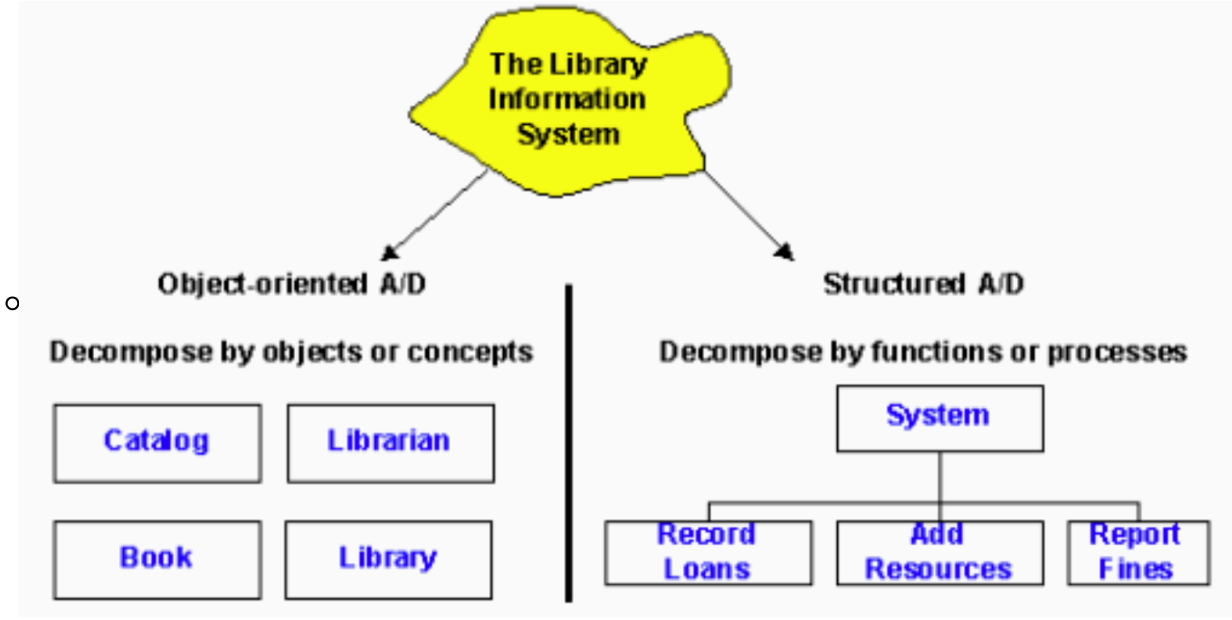
- 程序设计范型
- 类和对象
- 构造和析构函数
- 类的继承与派生
- 重载
- 覆盖，虚函数与多态
- 友元

程序设计范型

设计范型	思维方法
过程程序设计 **(Procedural Programming)**	基于过程抽象，以对数据进行分步骤地加工和处理作为思维的主干。
模块化程序设计 **(Modular Programming)**	基于模块抽象，以对程序的模块结构进行划分和组织作为思维的主干。
函数程序设计 **(Functional Programming)**	基于元数学的计算概念(如λ演算)，以对给定的问题表示进行分步骤地替换和化简为思维的主干。
逻辑程序设计 **(Logical Programming)**	基于一阶谓词演算理论，以从已知条件向结论进行分步骤的推理作为思维的主干。
面向对象程序设计 **(Object-Oriented Programming)**	基于**数据抽象**、**继承性**和**消息传递**，以对**实体**(包括结构、状态和行为)进行分类、组织和协同作为思维的主干。

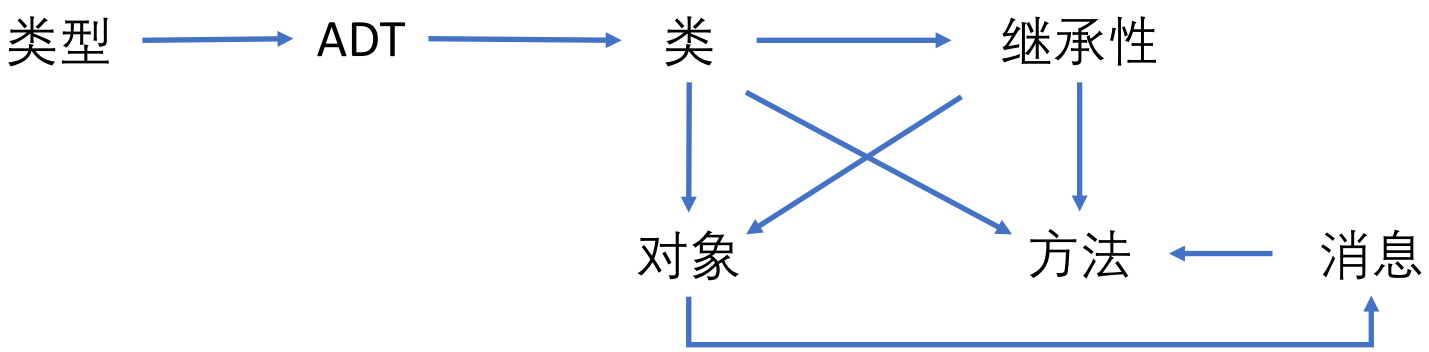
面向对象的基本概念体系

- 面向对象技术将**计算**看成是一个系统的演变过程，系统由**对象**组成，经历一连串的**状态变化**以完成计算任务。
- 对象具有状态保持能力和自主计算能力。
- 面向对象设计和实现的重点是多个对象的网状组织结构和协同计算，而不是过程调用的层次结构。



面向对象的基本概念体系

- 面向对象的基本概念:
  - 类(**Classes**)、对象(**Objects**)、 继承性(**Inheritance**)、方法(**Methods**)、消息 (**Messages**)



类-自定义类型

- 面向对象语言的要求：使用一个**自定义类型**应当与使用一个**基本类型**没有本质区别，看到的也应当只是**类型名**和**一组操作**的声明(包括操作名、参数、操作涵义、操作使用规则)，而看不到操作的**具体实现**，也看不到该类型所定义的**内部数据结构**。
- **类**：面向对象语言中**自定义的**抽象数据类型，支持继承性和多态，具有实例化能力。
- 类的定义、实现和使用通常是分离的，有确定的分离机制。

类-成员

- **数据成员**：定义在一个类中的**变量**，每个数据成员都有确定的类型，因而用它们的值集可以构造出这个类的**值集**。
- **成员函数**：定义在一个类中的**函数**，这些成员函数构成了这个类的**操作集**。

```
// Date.h
class Date {
private:
   int  d, m, y;
public:
   void init(int dd, int mm, int yy);
   void add_year(int n);
   void add_month(int n);
   void add_day(int n);
};
```

类-访问控制(Access Control)机制

- 访问控制：外界对于类的访问控制程度，也称为“能见度”。
- **C++**语言提供了三种访问控制程度:
  - **public**：允许在同一程序的任何地方引用;
  - **private**：只允许在本类的成员函数实现体中引用;
  - **protected**：允许在本类的成员函数实现体中引用，以及在满足一定条件的子类的成员函数实现体中引用。

类-访问控制机制的目的

- 使得这个类的**成员函数**确实构成了这个类的**操作集**。
- 易于进行出错定位(这是调试程序时首先要解决、也是最难解决的一个问题)。
- 用成员函数的声明组成这个类的**接口**，将它的数据成员和成员函数的实现**封装**起来，以减少程序的修改对外部的影响。
- 有利于掌握一个类型的使用方式(了解数据结构后才能使用类型，实际上是不得已而为之)。

类-构造函数

- 一个类中可以根据需要定义多个**Constructors**（函数名重载），编译程序根据调用时实参的数目、类型和顺序自动找到与之匹配者。

```
//Date.h
class Date{
    int d, m, y;

public:
    Date(int dd, int mm, int yy);
    Date(int dd, int mm); // today's year
    Date(int dd);              // today's month and year
    Date();                    // default Date: today
    Date(const char* p);  // date in string representation
    /* … */
};
```

```
#include "Date.h"

void f(){
    Date today(27);
    Date july4("July 4, 1983");
    Date guy("5 Nov");
    Date now();
    /* … */
}
```

类–构造函数

- 有时很难估计将来对**Constructor**形参的组合会有怎样的要求，一种有效的策略是对**Constructor**也声明有省缺值的形参（**Default Arguments**）。

```
//Date.h
class Date{
    int d, m, y;

public:
    Date(int dd=0, int mm=0, int yy=0);
    /* … */
};
```

```
#include “Date.h”

void f(){
    Date today(27);
    Date someday(27, 9);
    Date aDay(27, 9, 2017);
    Date now;
    /* … */
}
```

类-析构函数

- 任何一个系统的**资源**（信号量等。如果在包括**Constructor**（列的生存期结束时释放它们（**C++**编（在程序运行过程中耗尽。
- 需要在一个类中定义（善后处理）的成员函数，这就是**Des**
- **Destructor**的命名规
- **Destructor**不允许有

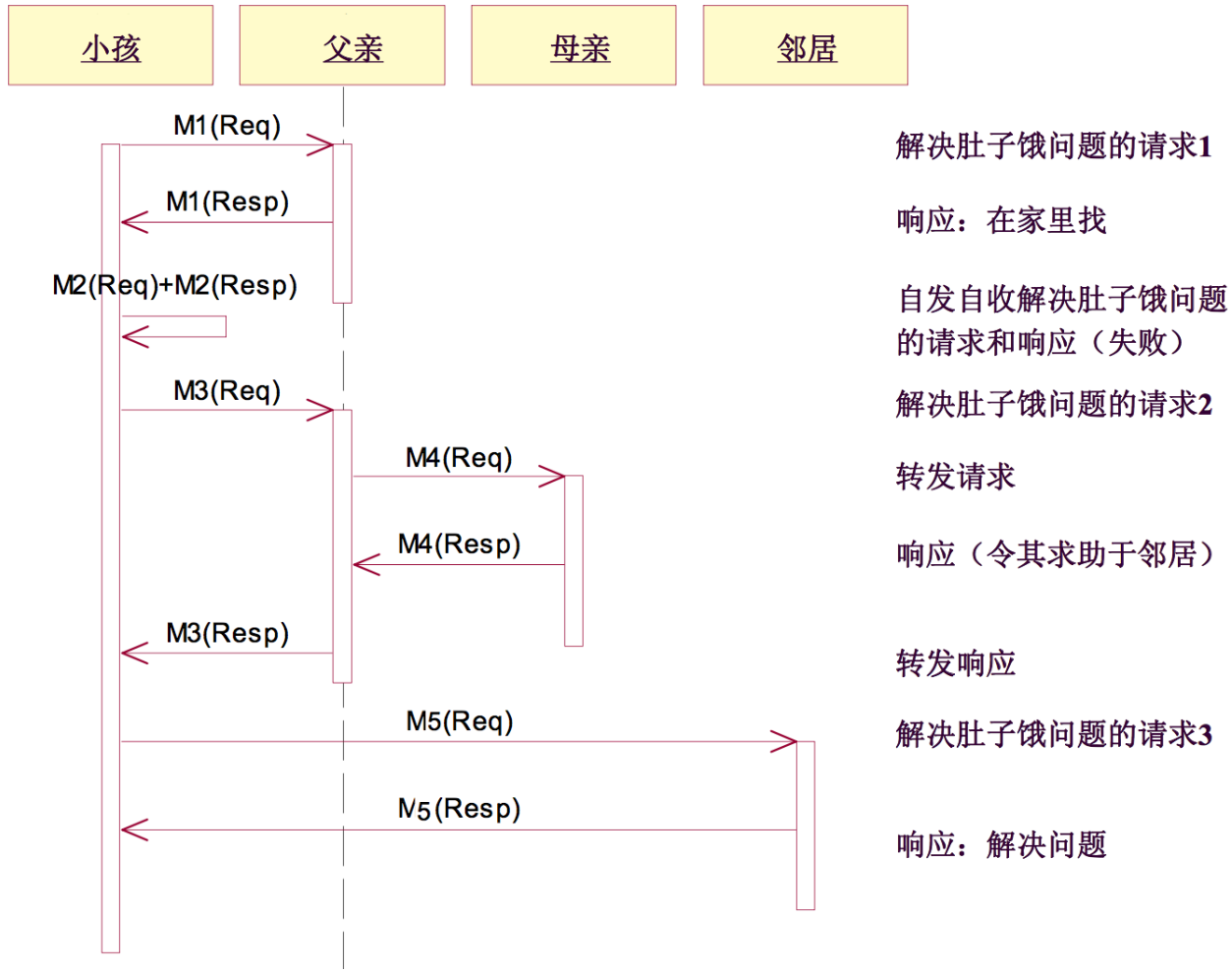
```
class Name{
   const char *s;
   /* ... */
};

class Table{
   Name *p; //元素个数为sz的数组，空间是动态申请的
   size_t sz;
public:
   Table(size_t s = 15) {  p = new Name[sz = s]; }
   ~Table() { delete [] p;}
   Name * lookup(const char*) ;
   bool insert(Name*);
};
```

对象

- 某个特定类的实例。

- 同一个类产生的对象具有：
  - 相同的数据结构、操作集合和能见度
  - 不同的标识
  - 相同或不同的初始状态
  - 拥有和保持不同的运行状态

- 对象之间通过消息传递方式进行交互。



- 对象与一般数据类型之实例的区别：对象是主动的数据，对象之间通过消息传递方式进行通信，而一般数据只能被动地由过程来加工。

对象

- 大部分面向对象语言的每个对象有：
  - **独立的数据空间**(以保持各自的状态)，
  - 同一个类的对象**共享操作代码**(以节省代码空间、保证操作一致性)。
- 大部分面向对象语言支持**类属性**(即同一个类的对象共享的数据)，如**C++**中的静态数据成员。
- 大部分面向对象语言采用为对象连续分配数据空间的策略(以支持继承性的实现)。

实例化及其机制

- **实例化**:依据对应的一个类(或一组有继承关系的类)，根据作用域控制规则，进行实例生成和实例消除的过程。
- **实例生成**:在用一个类定义一个变量时，或者用这个类的指针类型定义一个指针、且显式执行了**new**操作时，编译程序将该声明语句翻译成关于这个类的**Constructor**(如果定义了的话)的一次调用，按照该类所规定的空间大小，分配一块存贮空间(通常是一块连续的空间)，使之与该变量绑定，对这块空间执行在**Constructor**中定义的那些操作。
- **实例消除**:当到达了该变量的作用域结束位置，或者显式地对该变量执行了**delete**操作，则执行**Destructor**(如果定义了的话)中规定的操作，释放这个实例占用的存贮空间。

实例化及其机制

- 在类中定义的一个**数据成员**，是这个类的每个实例的一个组成部分，在每个实例中都有对应的、结构相同的存储空间，因而可以使每一个实例保持不同的值，具有不同的状态。
- 类的实例就是对象。
- 类是对象的模板：用一个类可以产生一组有相同的存储结构、相同的行为规律、相同的接口(通信协议)、不同状态的对象。

继承性与类层次结构

- 孤立的类只能描述实体集合的**特征同一性**，而客观世界中实体集合的划分通常还要考虑实体特征方面有关联的**相似性**。
- “相似”无非是既有共同点，又有差别。
  - 内涵的相似性:在客观世界中具有“一般–特殊”的关系。
    - 例如:雇员和经理。
  - 结构的相似性:具有相似的表示。
    - 例如在**Smalltalk**类库中，类**Bag**利用类**HashTable**作为基本存储结构

继承性与类层次结构

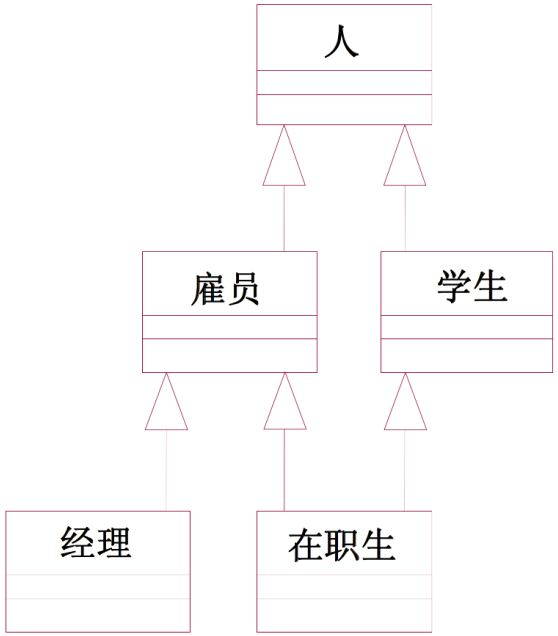
- **继承性**:在类之间既能体现其共性和差别，又能给出其间存在共性和差别关系的信息，还能将这样的关系按照需要进行传递的类型化机制。

- 继承性是面向对象技术实现基于差别的开发的一种主要机制。

- 继承的传递性将增加类之间的耦合度，故应防止滥用继承。

- 重视多重继承带来的问题，使用多重继承不可能代替对象组装的设计。

继承性与类层次结构




```
#include “Employee.h”
// Manager是Employee的子类
class Manager: public Employee{
   int Level;
public:
   void changeLevel(int l);
   Manager(char *n, int a, int l);
   ~Manager();
};
```

共性描述

差别定义

```
// Employee是Manager的父类
class Employee{
   char *Name;
   int Age;
public:
   void changeAge(int newAge);
   void retire();
   Employee(char *n, int a);
   ~Employee();
};
```

继承的方法

```
#include “Manager.h”
void userFunc(){
   Manager m1(“Wangwu”, 35, 2);
   *m2 = new Manager(“Zhaoliu”, 28, 3);
   m1.changeAge(36);
   m2->changeLevel(2);
}
```

继承性与类层次结构

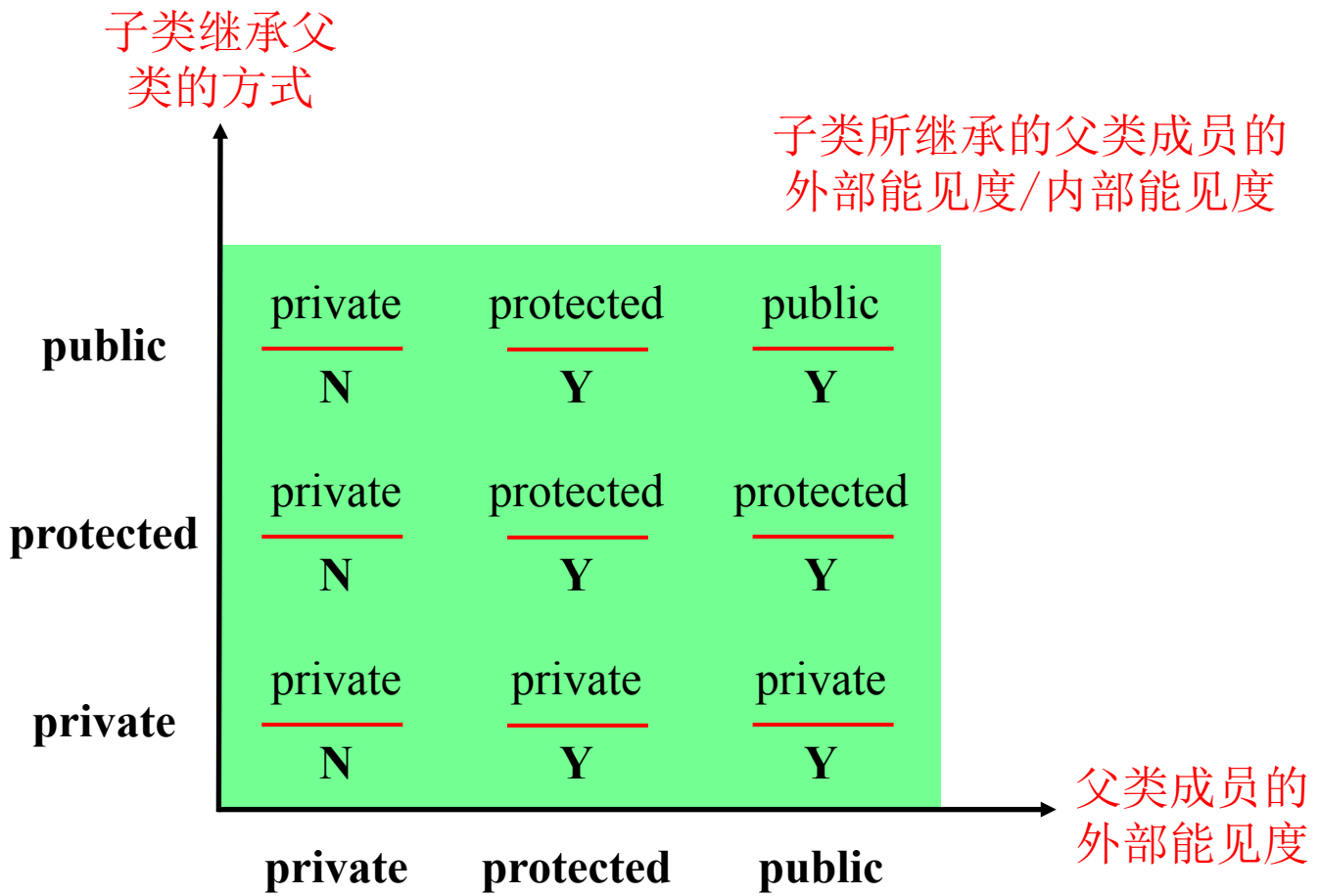
- 上例中，类**Manager**并不是其实例类型的全部定义，另一部分由类**Employee**定义，体现了类与类型之间的差别。
- 类之间的继承关系是传递的。
- 一个类的所有直接和间接父类(子类)统称为这个类的父类(子类)。
- 单重继承：只允许一个类最多有一个直接父类。
- 多重继承：允许一个类有多个直接父类。

继承性与类层次结构

- 父类的成员在子类中的**外部能见度**，是指被子类继承的父类成员在子类中的**外部**访问控制程度，也分为**public**、**private**、**protected**三种。

- 父类的成员在子类中的**内部能见度**，是指被子类继承的父类成员在子类中定义的方法中的**内部**访问控制程度，分为**Y**(可访问)和**N**(不可访问)两种。

继承性与类层次结构




```
class B {
  void pvB();
protected:
  void ptB();
public:
  void pbB();
};
class D : private B {
  void f()
  { pvB();✗ptB();✓
    pbB();✓}
  // ...
};
void g( D& d)
{
  d.pvB();✗d.ptB();✗
  d.pbB();✗
}
```

重载(overload)

- 重载：在同一范 但是这些同名函数的形式参数（ 顶不同。

- 如果需要对一个 语义，而**Constructor**的名 名，因此必须支持**Constructor**的重

```
class Employee{
   char *Name;
   int Age;
   char *Address;
   char *Telephone;
public:
   void changeAge(int newAge);
   void changeAddr(char *newAddr);
   virtual void retire();
   Employee(char *n, int a);
   Employee(char *n, int a, char *addr, char *tel);
   Employee();
   ~Employee();
};
```

重载(overload)

- 如果能支持**Constructor**的重载，自然应支持其他方法名的重载。

```
#include “Employee.h”
void userFunc(){
   Employee e1(“Zhangsan”, 24);
   int i;
   // ...
   ++e1;
   // ...
  for(i=0; i<10; i++)
   // ...
}
```

根据变量类型执行对应的操作

```
class Employee{
protected:
   char *Name;
   int Age;
   char *Address;
   char *Telephone;
public:
   // ...
   Employee & operator ++() {
      Age++; return *this;
   }
   // ...
};
```

重载(overload)

- C++**允许**在用户自定义类型中重载的操作符是：

```
+       -       *       /       %       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ->*     ,
->      []      ()      new     new[]   delete  delete[]
```

- C++**不允许**在用户自定义类型中重载的操作符是：

```
::      .       .*
? :     sizeof  typeid
```

- C++也**不允许**组合定义操作符，例如**。

覆盖(overriding)，虚函数与多态

- 问题:要求所继承的方法符合子类语义

```
#include “Employee.h”

class Manager: public Employee{
   int Level;
public:
   void changeLevel(int l);
   Manager(char *n, int a, int l);
   ~Manager();
};
```

```
class Employee{
   char *Name;
   int Age;
public:
   void changeAge(int newAge);
   void retire();
   Employee(char *n, int a);
   ~Employee();
};
```

```
#include “Employee.h”
void Employee::retire(){
   if(Age > 55)
      cout<<“employee retire”;
}
```

不符合子类的语义要求

覆盖(overriding)，虚函数与多态

- 覆盖机制：覆盖存在类中，子类重写从基类继承过来的函数。但是函数名、返回值、参数列表都必须和对应的基类函数相同。

- **C++**中的覆盖机制：虚拟函数(**virtual functions**)

覆盖(overriding)，虚函数与多态

```
#include “Employee.h”

class Manager: public Employee{
   int Level;
public:
   void changeLevel(int l);
   void retire();
   Manager(char *n, int a, int l);
   ~Manager();
};
```

覆盖方法

```
#include “Manager.h”
void Manager::retire(){
   if(Age > 60)
      cout<<“manager retire”;
}
```

```
class Employee{
protected:
   char *Name;
   int Age;
public:
   void changeAge(int newAge);
   virtual void retire();
   void retire();
   Employee(char *n, int a);
   ~Employee();
};
```

允许覆盖

```
#include “Employee.h”
void Employee::retire(){
   if(Age > 55)
      cout<<“employee retire”;
}
```

覆盖(overriding)，虚函数与多态

- 覆盖机制的特点:
  - 子类不改变父类中的已有接口定义(尽管它可能有不同语义的方法体)
- 虚函数：通过动态绑定机制，在运行时才确定接收消息的对象类型并调用相应的方法。

```
Employee *base = new Employee(“Zhangsan”, 56);
Manager *subclass = new Manager(“Lisi”, 61);
base = subclass; //基类指针指向子类对象
base->retire(); // “manager retire”
```

友元

- friend函数和friend类：关于局部破坏封装的描述。
- 这种机制的合理使用，可以限定一个类具体到某个(些)其他类或函数的外部能见度(但是语言是无法防止滥用这种机制的)

```
class a {
// 类 a 对于函数 f 和类 b 是不封装的。
friend void f(); friend class b;
private:
  int a1;
  void g();
/* 类 a 没有public成员，意味着只有函数 f
   和类 b 的实例能够访问 a 的实例 */
};
```