软件构造
SOFTWARE CONSTRUCTION

主讲教师：李翔　冯桂焕

2013年秋季

Sub-Part II:

- 类的设计
  - 抽象数据类型
  - 类接口设计
  - 设计和实现中的问题
  - 为什么使用类

什么是类?

- 程序员经历了如下思考编程过程
  - 基于语句的思考->基于子程序的思考->以类为基础的思考
- 类是一组数据和子程序构成的集合，这些数据和子程序共同拥有一组内聚的、明确定义的职责
  - 也可以是一组子程序集合，提供一组内聚的服务
- 类使得在开发程序时，能安全的忽视尽可能多的其余部分

抽象数据类型（ADT）

- 类的基础：抽象数据类型
  - 类是一组数据和子程序构成的集合，这些数据和子程序共同拥有一组内聚的、明确定义的职责。
  - ADT是指一组数据以及对这些数据所进行的操作的集合

- 以上两段描述有何本质区别？
  - 从数据的角度来看类和ADT没有区别
  - 从OO的角度来看，类还涉及到面向对象中的概念

需要用到ADT的例子：文本编辑程序中对字体控制的部分

字体数据项：

CurrentFont
size: int
bold: bool
italic: bool

currentFont.size = 16
currentFont.size = PointsToPixels(12)

！调用方代码直接控制数据成员！

CurrentFont
size: int; bold: bool; italic: bool
SetSizeInPoints(sizeInPoints)
SetSizeInPixels(sizeInPixels)
SetBoldOn()
SetBoldOff()
SetItalicOn()
SetItalicOff()

SetSizeInPoints(12)
SetSizeInPixels(16)
SetBoldOn()

！对字体的操作都隔离到一组子程序里了！

体会ADT与类的区别

- 使用ADT的益处
  - 可以隐藏实现细节
  - 改动不会影响到整个程序
  - 让接口提供更多信息
  - 更容易提高性能
  - 让程序的正确性更显而易见
  - 程序更具自我说明性
  - 无需在程序内到处传递数据
  - 可以在像现实世界中那样操作实体

- 在类中对应的体现
  - 隐藏私有信息
  - 封装性
  - 真正意义的接口
  - 更易于修改
  - 易于识别错误
  - 可读性高
  - 封装数据和操作
  - 更接近人的思维

ADT和类

- ADT构成了类概念的基础
- 类可以看做是ADT再加上继承和多态两个概念

ADT：

一些数据以及对这些数据进行操作的集合。

＋ 面向对象 ＝ 类

继承、多态

类接口的设计

- 类设计的第一步，也是最重要的一步

```
class Employee {

public:

   // public constructors and destructors
   Employee();
   Employee(
      FullName name,
      String address,
      String workPhone,
      String homePhone,
      TaxId taxIdNumber,
      JobClassification jobClass
   );

   virtual ~Employee();
   // public routines
   FullName GetName() const;
   String GetAddress() const;
   String GetWorkPhone() const;
   String GetHomePhone() const;
   TaxId GetTaxIdNumber() const;
   JobClassification GetJobClassification() const
..};
```

```
class Program {

public:
   ...

   // public routines

   void InitializeUserInterface();

   void ShutDownUserInterface();

   void InitializeReports();

   void ShutDownReports();
   ...

private:

};
```

类接口设计建议

- 一：好的抽象
  - 类的接口应能提供一组明显相关的子程序
- 二：好的封装
  - 不要暴露自身数据和实现细节
- 二者相关，或者两者都有，或者两者皆失

好的抽象

- 类的接口应该尽量展现一致的抽象层次
  - 每一个类应该实现并仅实现一个ADT
  - 当实现了多个ADT时，可以考虑重新组织类

```
class EmployeeCensus: public ListContainer {
public:
    void AddEmployee( Employee employee );
    void RemoveEmployee( Employee employee );

    Employee NextItemInList();
    Employee FirstItem();
    Employee LastItem();
    ......
Private:
    ......
} ;
```

针对Employee抽象

针对List抽象

使用容器类实现内部逻辑，却没有把该事实隐藏起来

不符合“is a”关系

对EmployeeCensus的重新组织

- EmployeeCensus包含了两个ADT
  - Employee显然才是EmployeeCenus要处理的ADT
  - ListContainer是用于存放Employee的容器，相当于数据库的角色

```
class EmployeeCensus {
public:
    void AddEmployee( Employee employee );
    void RemoveEmployee( Employee employee );
    Employee NextEmployee();
    Employee FirstEmployee();
    Employee LastEmployee();
private:
    ListContainer m_EmployeeList;
```

好的抽象-续

- 理解类接口应该捕获的抽象到底是哪一个
  - 150个子程序的table控件vs.15个子程序的grid控件
- 提供成对的服务
  - 不要盲目创建相反操作
- 把不相关的信息转移到其他类
  - 思考子程序与数据之间的引用关系
- 让接口可编程，而不是表达语义
  - 可编程部分：接口数据类型+其他属性，编译器可检查
  - 语义部分：本接口将会被怎样使用，编译器不可检查
- 不要添加与接口抽象不一致的公用成员
  - “这个子程序与现有接口所提供的抽象一致吗？”
- 同时考虑抽象性和内聚性
  - 抽象性和内聚性之间的关系非常紧密

- 谨防在修改时破坏接口抽象

```
class Employee {
public:
   ...
   // public routines
   FullName GetName() const;
   Address GetAddress() const;
   PhoneNumber GetWorkPhone() const;
   ...
   bool IsJobClassificationValid( JobClassification jobClass );
   bool IsZipCodeValid( Address address );
   bool IsPhoneNumberValid( PhoneNumber phoneNumber );

   SqlQuery GetQueryToCreateNewEmployee() const;
   SqlQuery GetQueryToModifyEmployee() const;
   SqlQuery GetQueryToRetrieveEmployee() const; ……}
```

好的封装

- 封装比抽象更严格，二者相辅相成
  - 抽象：提供可以忽略实现细节的模型来管理复杂度
  - 封装：阻止你看到细节
- 尽可能限制类和成员的可访问性
  - 采用最严格且可行的访问级别
  - 如何最好地保护接口抽象的完整性
- 不要公开暴露成员数据

```
float x;
float y;
float z;
```

内部用float存储的吗？

```
float GetX();
float GetY();
float GetZ();
void SetX( float x );
void SetY( float y );
void SetZ( float z );
```

- 避免把私有的实现细节放到类的接口中

```
Class Employee {
public:
   ...
   Employee(
      Fullname name, String Address,
      String workphone,
      StringhomePhone
      TaxId taxIdNumber
      JobClassification jobClass
   );
   ...
   Fullname GetName() const;
   String GetAddress() const;    ...
Private:
   String m_name;
   String m_Address;
   int m_jobClass; ...
 };
```

Private段在类的头文件中，暴露了实现细节

```
class Employee {
public:
   ...
   Employee( ... );
   ...
   FullName GetName() const;
   String GetAddress() const;
   ...
private:

EmployeeImplementation
*m_implementation;
};
```

只对Employee类可见

- 不要对类的使用者做任何假设

```
// 请把x,y,z初始化为1.0，如果初始化为0，DerivedClass就
// 会崩溃
```

- 避免使用友元类
- 不要因为一个子程序只使用公用子程序，就把它归入公开接口
  - 接口抽象还一致吗？
- 让阅读代码比编写代码更方便
  - 不要为了编码方便而降低代码的可读性
- 警惕从语义上破坏封装性！

调用方从语义上破坏封装性的例子

- 不在调用employee.Retrieve(database)之前调用database.connect()，因知道未建立数据库连接时Retrieve会去连接
- 即便在ObjectA离开作用域之后，仍使用由其创建的指向ObjectB的指针或引用，因知其在静态存储空间，或还能用
- 不去调用A类的Terminate()子程序，因为知道A类的PerformFinalOperation()子程序已经调过它了
- 使用ClassB.MAXIMUM_ELEMENTS 而不用ClassA.MAXIMUM_ELEMENTS，因知二者相等

在透过接口通过内部实现编程

解决方案:针对接口编程

- 当从类的接口无法得知如何使用时
  - 联系类的作者

- 类的作者
  - 面对面地告诉答案 ×
  - 修改类的接口文档 √

留意过于紧密的耦合关系

- 尽可能限制类和成员的可访问性
- 避免友元类，因其之间是紧密耦合的
- 在基类中把数据声明为private而不是protected，以降低派生类和基类的耦合度
- 避免在类的公开接口中暴露成员数据
- 要对从语义上破坏封装性保持警惕
- 警觉“Demeter”法则

Design & Implementation问题：包含

- 面向对象编程中的主力技术
- 实现“有一个/has a”的关系
  - 雇员有一个姓名、有一个电话号码
- 在万不得已时使用private继承实现“有一个”的关系
  - 为了让外层的包含类能够访问内层被包含类的Protected成员函数与数据成员
  - 破坏了封装性，要慎重！
- 警惕超过7个数据成员的类
  - Magic Number 7+-2

D&I问题：继承

- 表示一个类是另一个类的特例
- 目的是写出更精简的代码，避免重复
  - 定义能为两个或多个派生类提供共有元素的基类

- 使用时，需谨慎决策
  - 成员函数应该对派生类可见吗？有默认实现吗？能被覆盖（override）吗？
  - 数据成员应该对派生类可见吗？

继承-续

- 用public继承来实现“是一个”的关系
  - 基类对派生类将做什么设定了预期
- 对不可继承的类明确禁止
  - Non-virtual(C++), final(Java)
- 遵循Liskov替换原则
  - “派生类能通过基类接口被使用，且使用者无需了解两者之间的差异”
  - 基类中定义的所有子程序，用在它的任何一个派生类中含义都应该是相同的
    - E.g. Account、CheckingAccount、SavingAccount、AutoLoanAccount

继承-续

	可覆盖的	不可覆盖的
提供默认实现	可覆盖的子程序	不可覆盖的子程序
未提供默认实现	抽象且可覆盖的子程序	不会用到(一个未经定义但又不让覆盖的子程序是没有意义的)

- 只继承需要继承的部分
  - 慎重对待继承来的子程序
- 不要“覆盖”一个不可覆盖的成员函数
  - 派生类中的成员函数不要与基类中不可覆盖的成员函数重名
- 把共用内容放到继承树中尽可能高的位置
  - 位置越高，派生类使用的时候就越容易
  - 根据抽象性决定高度
- 以下情况值得怀疑
  - 只有一个实例的类，和只有一个派生类的基类
  - 覆盖基类子程序却什么都不做
    - E.g. ScratchlessCat的Scratch()

继承-续

- 让所有数据都是private(而非protected)
  - 对需访问属性提供protected访问器函数
- 尽量使用多态，避免大量类型检查

C++示例：多半应该用多态来替代的case语

```
switch ( shape.type ) {
    case Shape_Circle:
        shape.DrawCircle();
        break;
    case Shape_Square:
        shape.DrawSquare();
        break;
    ...
}
```

C++示例：也许不该用多态来替代的case语句

```
switch ( ui.Command() ) {
    case Command_OpenFile:
        OpenFile();
        break;
    case Command_Print:
        Print();
        break;
    case Command_Save:
        Save();
        break;
    case Command_Exit:
        ShutDown();
        break;
    ...
}
```

小结：包含与继承

实际情况	选择
多个类共享数据而非行为	创建类可以包含的共用对象
多个类共享行为而非数据	继承共同的基类，在基类定义共用子程序
多个类既共享数据也共享行为	从一个基类继承，在基类定义共用数据和子程序
想由基类控制接口	使用继承
想自己控制接口	使用包含

D&I问题：成员函数和数据成员

- 让类中子程序的数量尽可能少
  - 子程序数量越多，出错率就越高
- 禁止隐式产生不需要的成员函数和运算符
  - 通过定义为private，从而禁止调用方代码访问
- 减少对其他类的子程序的间接调用
  - Demeter法则
- 减小类和类之间相互合作的范围
  - 实例化对象的种类、在被实例化的对象上直接调用不同子程序的数量等

Demeter法则（最少知识原则）

- 在面向对象的方法中，一个方法“M”和一个对象“O”只可以调用以下几种对象的方法：
  1. O自己
  2. M的参数
  3. 在M中创建的对象
  4. O的直接组件对象

如果你去店里买东西，你会把钱交给店员，还是会把钱包交给店员让他自己拿？

```
class Clerk {
Store store;
void SellGoodsTo(Client client)
{
money = client.GetWallet().GetMoney();//店员自己从钱包里拿钱了！
store.ReceiveMoney(money);
}
};
```

D&I问题：构造函数

- 尽可能在所有构造函数中初始化所有数据成员
- 优先采用深层副本
  - 深层副本在开发和维护方面比浅层副本简单
- 单件属性的实现：私有构造函数
  - 规定某个类只能有唯一一个对象实例

```
public class MaxId {
  // constructors and destructors
  private MaxId() {  ...  }     ...

  // public routines
  public static MaxId GetInstance() {
    return m_instance;
  }    ...

  // private members
  private static final MaxId m_instance = new MaxId();      ...
}
```

外部如何使用？

为什么创建类？

- 为现实世界中的对象建模
- 为抽象对象建模
  - 如Student
- 降低复杂度、隔离复杂度
- 隐藏实现细节
- 限制变动的影响范围
- 隐藏全局数据
  - 所谓“全局数据”可能只是对象的数据
- 让参数传递更顺畅
  - 一个参数在多个子程序之间传递？为什么不做到一个类里
- 让代码更易于重用
- ……

CHECKLIST: Class Quality
核对表：类的质量

抽象数据类型

- ❑ 你是否把程序中的类都看做是抽象数据类型了？是否从这个角度评估它们的接口了？

抽象

- ❑ 类是否有一个中心目的？
- ❑ 类的命名是否恰当？其名字是否表达了其中心目的？
- ❑ 类的接口是否展现了一致的抽象？
- ❑ 类的接口是否能让人清楚明白地知道该如何用它？
- ❑ 类的接口是否足够抽象，使你能不必顾虑它是如何实现其服务的？你能把类看做黑盒子吗？
- ❑ 类提供的服务是否足够完整，能让其他类无须动用其内部数据？
- ❑ 是否已从类中除去无关信息？
- ❑ 是否考虑过把类进一步分解为组件类？是否已尽可能将其分解？
- ❑ 在修改类时是否维持了其接口的完整性？

封装

- ❑ 是否把类的成员的可访问性降到最小？
- ❑ 是否避免暴露类中的数据成员？
- ❑ 在编程语言所许可的范围内，类是否已尽可能地对其他的类隐藏了自己的实现细节？
- ❑ 类是否避免对其使用者，包括其派生类会如何使用它做了假设？
- ❑ 类是否不依赖于其他类？它是松散耦合的吗？

继承

- ❑ 继承是否只用来建立“是一个/is a”的关系？也就是说，派生类是否遵循了 LSP（Liskov 替换原则）？
- ❑ 类的文档中是否记述了其继承策略？
- ❑ 派生类是否避免了“覆盖”不可覆盖的方法？
- ❑ 是否把公用的接口、数据和行为都放到尽可能高的继承层次中了？
- ❑ 继承层次是否很浅？
- ❑ 基类中所有的数据成员是否都被定义为 private 而非 protected 的了？

跟实现相关的其他问题

- ❑ 类中是否只有大约七个或更少的数据成员？
- ❑ 是否把类直接或间接调用其他类的子程序的数量减到最少了？
- ❑ 类是否只在绝对必要时才与其他的类相互协作？
- ❑ 是否在构造函数中初始化了所有的数据成员？
- ❑ 除非拥有经过测量的、创建浅层复本的理由，类是否都被设计为当作深层复本使用？

作业1

- 每人提交一段类接口代码
  - 不需要包含类的实现细节
  - 对该类进行简要文字描述
    - 在何系统中，承担什么职能或体现何种抽象
  - 具有中等复杂度
    - 除get()、set()外，包括不少于8个功能子程序
  - 应用本章内容，对接口进行改进，并简要分析改进依据

- 提交内容，.doc文档
- 截止日期：9月15日