程序设计语言原理

Principle of Programming Languages

裘宗燕

北京大学数学学院

2012.2～2012.6

8. 面向对象

- 为什么需要面向对象?
- OO 语言的发展
- 面向对象的基本概念
- 封装和继承
- 初始化和终结处理
- 动态方法约束
- 多重继承
- 总结

重用的问题

实践中人们认识到重用已有开发结果的重要性，提出了软件重用的概念

- 最早的重用单元是子程序，如 **Fortran** 的子程序库
- 子程序是纯粹的过程抽象，基于子程序的重用有很大局限性
- 模块是更合适的重用单元，因为模块可以包装任何功能，更灵活

- 重用中有一种常见情况：软件开发中遇到的新问题常与解决过的问题（可以重用的库提供的功能）类似，但又不完全相同
  - 已有模块的功能与需要有差异，无法以其“现有”形式直接使用
  - 如果模块功能的改变只能通过修改源代码的方式进行，程序员就只能拷贝这个模块的源代码，深入研究后再设法修改，以满足新需求
- 但问题是有没有可以使用的源代码？常常没有：
  - 模块可能是购入的，提供商不提供源代码
  - 模块可能是过去的遗产，源代码已经丢失或部分缺失

重用和软件开发

即使有源代码，基于修改代码的方式重用，也有很多问题：

- 修改代码的代价可能很大（需要理解开发者的想法、设计和大量细节）
- 修改代码很容易引进错误
- 经过修改的代码，其可读性、易理解性和易维护性都会恶化，多次修改导致这些性质不断恶化，可能给整个系统的质量带来严重危害
- 修改后的模块，继续重用的可能性更小

总之，基于修改代码的重用，重用的价值大大降低了

在软件开发过程，重用也是非常有价值的

- 在同一软件中重复使用某些部分，可使重要设计决策得到集中处理
- 提高重用比率可能减少重复开发工作量
- 对重要基础功能的深度优化可能非常耗工耗时，重用已有的经过精心调整的代码，可能大大提高系统的性能

模块和程序组织

- 常规的程序单元缺乏弹性，定义好的子程序/模块都是固定功能的实体，难以提供“定制”的方式部分地改变功能以满足实际需要的变化
- 通过模块定义的抽象数据类型是相互独立的，不同模块之间无任何关系
  - 而实际情况中，常常需要定义和使用一些相互有关的类型，可能需要把它们送给同一个函数/过程去处理，以同样方式存储
  - 变体和联合机制就是为了迎合这方面的需要，但它们没有类型安全性，且未能提供解决类似问题的统一框架，难用于应付更复杂的情况
  - 支持相关类型，可能给程序的结构组织带来新的可能性
- 如何在抽象数据类型的框架中提供这一类功能，也是需要解决的问题

面向对象的概念在这些方面都能发挥很大的作用

面向对象（**Object-Oriented**）的方法和程序技术，为基于模块（一个类也可以看作一个模块）的重用问题提供了一条解决途径。

面向对象和重用

- 面向对象技术的最重要能力，在于使程序员比较容易以一种外部附加的方式，在已有数据抽象的基础上定义新的数据抽象
  - **OO** 也支持定义有弹性的操作框架，使新的数据抽象可以使用这些框架，并把针对该类抽象的实例的具体操作插入框架中（重用和调整）
  - 新定义的抽象可以继承原有抽象的行为，也可以根据需要调整改变已有功能的行为，或者添加新抽象所需要的新行为
  - 这样大大提高了代码重用的可能性（目标是实现真正不加修改的重用。当然，实际的重用可能性还与具体数据抽象的设计有关）

面向对象还有另外的许多重要价值（有些可能同样重要或更重要），并由此发展出“面向对象的系统分析”，“面向对象的设计”等

面向对象思想对于软件领域的影响是全面的，是结构化思想（结构化程序设计，结构化分析，结构化设计等）之后软件开发领域中的又一次革命

注意：面向对象并没有取代结构化，应该看作是在另一层次上的抽象

OO 发展史

OO 技术和思想中的一个基本方面是数据和操作的封装

- 这方面的基本想法：一组数据与关联之上相关的操作形成一个对象。其内部数据构成对象的状态，操作确定对象与外界交互的方式
- OO 并不是从模块化程序设计发展出来的，它有自己的发展历程
- OO 的思想与模块化的思想是并行发展，一直相互影响、相互借鉴

Simula 67 是 OO 概念的鼻祖，其设计目标是扩充 Algol 60，以更好地支持计算机在模拟方面的应用。1960 年代在挪威计算中心设计和实现，主持其工作的 Ole-Johan Dahl 和 Kristen Nygaard 获得 2001 年图灵奖

- OO 的三个基本要素：封装、继承和动态方法约束都源于 Simula
- 类的概念源自 Simula，其设计中提出用类定义把一组操作与一组数据包装起来。Simula 的这些重要想法是模块概念和 OO 的起源
- Simula 只提供了基本封装，并没有对封装的保护，也没有信息隐藏

OO 发展史

软件实践也需要 OO 的思想，并逐渐开发了 相关的支撑技术，包括：

- 封装的思想在面向模块的语言里发展，提出了许多重要概念和想法，如
  - 作用域规则，开的或者闭的作用域
  - 界面与实现
  - 透明类型与隐晦类型，访问控制，等等
- 数据驱动的程序设计技术：
  - 将计算功能（子程序）约束于程序里处理的数据（结构），使我们在程序里可以从数据对象出发去启动相应的计算过程
  - 在一些非常规的语言（如函数式语言）里，可以通过引用的概念提供函数/过程与数据之间的约束
  - 常规语言（如 C）引进了指向函数的指针，在实现数据驱动程序设计的过程中起到了重要作用，也成为面向对象语言实现的技术基础

OO 发展史

继承和动态约束等被 Smalltalk 发展，形成目前 OO 的基本概念框架

- 程序里以类的方式定义各种数据抽象
- 类可以通过继承的方式扩充新功能，这样定义的新类（子类，派生类）自动继承已有类（基类，超类，父类）的功能
- 对象是类的实例，是程序运行时的基本数据单元
- 派生类的对象也看作是原有基类的对象，可以当作基类的对象使用（子类就是子类型，Liskov 代换原理，2008 年图灵奖）
- 类定义了对象的状态成分（数据成员）和一组相关操作（称为方法）
- 方法调用总是针对某个对象进行的，将方法调用看作是给相应对象送一个消息，对象通过执行相应操作的方式对消息做出响应
- 对一个消息执行什么方法，由接收消息的对象的类型确定（根据该对象所属的类确定，这就是动态约束）
- 计算，就是一组对象相互通讯的整体效果（对计算的另一种看法）

OO 发展史

Smalltalk 还有一些独特的东西：

- 变量采用引用模型，变量无类型，可以引用任何对象
- 语言里的一切都是对象：
  - 类也是对象，通过给类送 new 消息的方式要求创建类的实例
  - 各种控制结构也是通过消息概念建立的
    - 条件和逻辑循环是逻辑对象对特定消息的响应
    - 枚举循环是整数对象对特定消息的响应
- 采用单根的类层次结构，以类 Object 作为所有类的超类
- 提供了块（block）的概念，作为控制结构的抽象机制
- 提出了容器的概念，开发了一个功能丰富的类库
- 与程序开发环境的紧密结合，并开发了 GUI 的基本概念和相关技术

Smalltalk 经过 72、76 发展到 Smalltalk 80，其概念和结构已臻成熟

OO 发展史

随着 Smalltalk 的成功，人们看到了 OO 的潜在威力

- 许多人开始研究如何把 OO 概念有效集成到常规语言里，提出了一批已有语言的 OO 扩充和许多新 OO 语言，如 Object-Pascal、Object-C 等
- 其中前期最成功并得到广泛应用的是 C++。C++ 在 OO 概念的广泛接受和应用方面功不可没（具体理由见后面讨论）。原因：
  - 在面向对象和高效程序之间取得较好的平衡
  - OO 概念与常规语言的合理集成（在当时），支持数据抽象和面向对象的系统设计和程序设计，支持多泛型程序设计的结合，使与数据抽象和 OO 有关的许多新概念和新技术逐渐被实际软件工作者接受
- 随后是 OO 分析、OO 设计和基于 OO 的软件开发等等
- 后来的其他成功语言包括 Java，微软提出 C#，等等
- 出现了一些基于对象的脚本语言，如 Python，Ruby 等
- 现在，面向对象的开发已经成为一种主流的软件开发技术

面向对象的基本概念

面向对象的基本概念：

- 在面向对象语言里定义数据抽象的基本定义机制是类，在一个类里可以定义数据成员和子程序成员（称为方法）
- 封装是数据抽象和模块化的概念，与面向对象的概念并没有必然关系，但封装有助于更好发挥面向对象机制的作用
- （实在的）类被看作类型，可以用于生成（定义）实例，称为对象
- 已有的类可以作为定义新类的基础（基类、超类）
  - 可通过继承方式定义新类（子类，派生类），子类继承基类的行为
  - 子类可以修改基类已经定义的行为，或者增加所需的新行为
- 把子类看作是子类型（通常），如果 D 是 B 的子类，那么：
  - 若 o 是 D 类型的对象，那么 o 也看作是 B 类型的对象
  - 若变量 x 可以引用 B 类的对象，那么它也可以引用 D 类的对象

面向对象的基本概念

- 继承有两方面作用
  1. 建立类型之间的层次关系
  2. 重用基类的行为（代码和数据描述）

  对于面向对象的行为而言，前一方面的功能更为重要
- 类中的子程序成员称为方法，方法需要通过具体的对象调用
- 在运行中调用方法时，实际调用的方法由作为调用出发点的那个对象的类型确定的（动态约束）
  - 动态约束是实现面向对象行为的关键
  - 它为面向对象的机制提供了模块机制所不具有的弹性，使新的功能扩充可以比较自然地结合到已有的操作过程里
  - 理解动态约束是理解面向对象的关键，动态约束的高效实现也是面向对象语言的实现的关键

面向对象的语言

虽然基本框架类似，不同面向对象语言之间也存在很大差异：

基本问题：采用什么样的对象模型

- 采用单根的类层次结构，还是任意的类层次结构？
- 提供那些继承方式？
  - 例如 C++ 里提供了三种继承方式
- 允许多重继承？还是只允许单继承？
- 是否提供丰富完善的访问控制机制？
- 采用基于继承的模型，还是基于指派的模型
- 基于类的模型，还是基于对象或原型的模型（如 JavaScript）
- 对象本身的独立性（是否允许不属于任何一个类的对象）
- 类本身是不是对象？

面向对象的语言

其他情况：

- 是不是追求“纯粹”的面向对象语言？
  - **Smalltalk** 尽可能追求“面向对象”理想，完全是重新设计的新语言
  - **Java** 是接近理想的语言，但希望在形式上尽可能靠近常规语言
  - **C++** 设法在支持系统程序设计的过程性语言 **C** 上“扩充”支持面向对象的机制，是一种多范型语言，支持多种程序设计方式
  - 另外的一些语言（如**Ada**）采用可能很不同的方式支持面向对象的程序设计，这里不准备详细介绍
- 采用值模型还是引用模型。从本质上说，只有采用引用模型才能支持方法的动态约束，因此大多数面向对象语言采用引用模型
  - **C++** 采用值模型，可以创建静态对象或栈对象，但只有通过对象引用或指向对象的指针才能实现面向对象的动态约束行为
  - **Java** 只能把 **OO** 功能应用于用户定义类型，基本类型采用值模型

面向对象的语言

- 是否允许静态对象或者堆栈对象（自动对象）？多数面向对象语言只支持堆对象（通过动态存储分配创建的对象）
  - **C++** 支持静态对象和自动对象，这种设计是希望尽可能借助于作用域规则来管理对象，避免依赖自动存储管理系统（**GC**）
  - 为在这种环境下编程，人们开发了许多利用自动对象的对象管理技术，如句柄对象，对象的“创建即初始化”技术等
- 是否依赖自动废料收集（**GC**）。由于 **OO** 程序常（显式或隐式地）创建和丢弃对象，对象之间常存在复杂的相互引用关系，由人来完成对象的管理和回收很困难。大多数 **OO** 语言都依赖于自动存储回收系统
  - **GC** 的引入将带来显著的性能损失，还会造成程序行为更多的不可预见性（**GC** 发生的时刻无法预见，其持续时间长短也无法预计）
  - **Java** 等许多语言都需要内置的自动废料收集系统
  - **C++** 是例外，其设计目标之一是尽可能避免对自动存储回收的依赖，以支持系统程序设计，提高效率，减少运行时间上的不确定性

面向对象的语言

- 是否所有方法都采用动态约束？
  - 动态约束很重要，但调用时会带来一些额外的开销，如果需要调用的方法能够静态确定，采用静态约束有速度优势
  - 大部分语言里的所有方法都采用动态约束
  - **C++** 和 **Ada** 提供静态约束（默认）和动态约束两种方式
- 一些脚本语言也支持面向对象的概念。例如，
  - **Ruby** 是一个纯面向对象的脚本语言，其中的一切都是对象，全局环境看作一个匿名的大对象，全局环境里的函数看作这个对象的成员函数。它还有另外一些独特性质
  - **JavaScript** 支持一种基于对象和原型的面向对象模型。其中没有类的概念，只有对象。对象的行为继承通过原型获得

面向对象的语言

- 人们还提出了许多与面向对象机制有关的新想法和模型
- 许多新近的脚本语言提供了独特的面向对象机制：例如
  - 基于对象原型（而不是类）的 **OO** 模型
  - 在基于类的模型中允许基于对象的行为覆盖（可修改个别对象的行为）
  - 等等
- 总而言之，虽然今天面向对象的模型和语言已成为主流程序设计方法和主流程序语言，但是这类语言还远未成熟，还正在发展和研究中
  - 许多语言的 **OO** 机制非常复杂，实际还不断提出一些新要求，使一些 **OO** 语言在发展中变得越来越复杂
  - 如何提供一集足够强大，而且又简洁清晰的机制支持 **OO** 的概念和程序设计，还是这个领域中需要继续研究的问题
  - **OO** 语言有关的理论研究还处在起步阶段，也是本领域不成熟的标志

OO 语言需要提供的新机制

- 定义类的语言机制（语言提供特殊的描述结构）
- 描述或定义对象的机制
- 继承机制，描述类之间的继承关系。可能定义继承关系的性质（如 C++ 里的类继承有 **public**、**protected** 和 **private** 三种方式）
- 与对象交互的机制（方法调用，消息传递）
- 初始化新对象的机制（最好能自动进行，避免未初始化就使用的错误）
- 类类型对象的动态转换机制（转换对一个具体对象的观点）
- 控制类成员的访问权限的机制
- 对象销毁前的临终处理机制（最好能自动进行）
- 对象的存储管理机制

可能还有其他机制：

- 运行中判断对象的类属关系的机制、自反等等

OO 程序

```
class list_err {                              // exception
    public:
    char *description;
    list_err (char *s) {description = s;}
};

class list_node {
    list_node* prev;
    list_node* next;
    list_node* head_node;
public:
    int val;                                  // the actual data in a node
    list_node () {                            // constructor
        prev = next = head_node = this;       // point to self
        val = 0;                              // default value
    }
    list_node* predecessor () {
        if (prev == this || prev == head_node) return 0;
        return prev;
    }
    list_node* successor () {
        if (next == this || next == head_node) return 0;
        return next;
    }
```

定义 **list_node** 类，用于实现带头结点的双向循环链接表

每个结点里有一个域指向表头结点

OO 程序

```
    int singleton () {
        return (prev == this);
    }
    void insert_before (list_node* new_node) {
        if (!new_node->singleton ())
            throw new list_err ("attempt to insert node already on list");
        prev->next = new_node;
        new_node->prev = prev;
        new_node->next = this;
        prev = new_node;
        new_node->head_node = head_node;
    }
    void remove () {
        if (singleton ())
            throw new list_err ("attempt to remove node not currently on list");
        prev->next = next;
        next->prev = prev;
        prev = next = head_node = this;     // point to self
    }
    ~list_node () {                         // destructor
        if (!singleton ())
            throw new list_err ("attempt to delete node still on list");
    }
};
```

OO 程序

定义 list 类

```
class list {
    list_node header;
public:
    // no explicit constructor required;
    // implicit construction of 'header' suffices
    int empty () {
        return (header.singleton ());
    }
    list_node* head () {
        return header.successor ();
    }
    void append (list_node *new_node) {
        header.insert_before (new_node);
    }
    ~list () {                  // destructor
        if (!header.singleton ())
            throw new list_err ("attempt to delete non-empty list");
    }
};
```

注意：**header** 是个 **list_node**

定义的是有头结点的循环链表

OO 程序

```
class queue : public list {                    // derive from list
public:
    // no specialized constructor or destructor required
    void enqueue (list_node* new_node) {
        append (new_node);
    }
    list_node* dequeue () {
        if (empty ())
            throw new list_err ("attempt to dequeue from empty queue");
        list_node* p = head ();
        p->remove ();
        return p;
    }
};
```

通过继承定义 **queue** 类。（只是作为示例）

通用的表结点类

```
class gp_list_node {
    gp_list_node* prev;
    gp_list_node* next;
    gp_list_node* head_node;
public:
    gp_list_node ();        // assume method bodies given separately
    gp_list_node* predecessor ();
    gp_list_node* successor ();
    int singleton ();
    void insert_before (gp_list_node* new_node);
    void remove ();
    ~gp_list_node ();
};
```

派生的 **int** 表结点类

```
class int_list_node : public gp_list_node {
public:
    int val;                // the actual data in a node
    int_list_node () {
        val = 0;
    }
    int_list_node (int v) {
        val = v;
    }
};
```

使用这种 **int** 表的问题：

如果需要访问结点的数据内容，必须对取出的结点做强制

面向对象概念的实现

- 实现面向对象的语言，需要考虑它的几个标志性特征的实现
- 封装是一种静态机制，如 C++/Java 一类语言的各种访问控制机制也是静态的，都可以通过在符号表里记录信息，在编译中检查和处理
- 方法的实现与以模块为类型时局部子程序的实现一样。由于每个方法调用有一个调用对象，因此方法需要一个隐含指针，被调用时指向调用对象，所有对该对象的数据成员的访问都通过这个指针和静态确定的偏移量进行
- 许多语言以这一指针作为一个伪变量，称为 **this** 或者 **self**，通过这种指针访问调用对象，方式上与通过指针访问普通结构一样

实现面向对象语言的关键是两个问题：

- 继承的实现，使派生类型的对象能当作基类的对象使用
- 动态约束的实现，能够从（作为变量的值或者被变量引用的）对象出发，找到这个对象所属的类里定义的方法

下面讨论实现的一些具体问题

封装

- 封装是一种静态机制，仅仅在程序加工阶段起作用，有关情况与模块机制类似，在加工后的程序里（可执行程序里）完全没有关于封装的信息
- 不同语言里对类的访问控制可能不同：
  - 作为 “开模块”（允许以特定方式任意访问类成员）
  - 作为“闭模块”（凡是没有明确声明可访问的都不可访问）

对基本封装机制的扩充是引进进一步的控制

- C++ 引进成员的 **public**、**protected** 和 **private** 属性，提供细致的访问控制
- C++ 还允许定义派生类的不同继承方式，控制对基类成员的访问：
  - **public** 继承
  - **protected** 继承，使基类的 **public** 成员变成派生类的 **protected** 成员
  - **private**继承，使基类的所有成员变成派生类的 **private** 成员

一些新语言借鉴了 C++ 的这方面思想，可能结合另外一些想法

静态域和静态方法

许多面向对象语言的类里可以定义静态域和静态方法

- **C++/Java** 允许类里定义静态数据域
- **Smalltalk** 把普通的对象域称为实例变量，表示在这个类的每个实例里都有这些成分的一份拷贝；把静态数据域称为类变量
- 类的静态数据域并不出现在实例对象里，它们是类封装的静态数据成分，提出具有静态生存期，在类的作用域里可直接访问。类外能否访问由语言确定（提出有与其他成员一样的访问控制）

静态方法和静态域的一些情况：

- 类的静态数据成员可以在静态区实现，在程序运行之前静态分配，在程序的整个执行期间保持其存储
- 类的静态方法可访问静态数据成员，其他方法也可以访问静态数据成员
- 可以把静态数据成员看作本类的所有对象共享的信息
- 类对象可以通过静态数据成员交换或者共享信息

静态域和静态方法

- 静态成员是静态创建的，其初始化在程序开始执行前完成（或者在语言定义的适当时刻完成），只做一次
- 静态成员的初始化中不能调用类的普通方法（因为没有对象）

静态方法相当于普通子程序，只是具有类封装（类作用域）。特点：

- 没有调用对象
- 不能引用 **this/self**，不能引用类定义的普通数据成员（如 **Smalltalk** 里不能引用实例变量），只能引用本类的静态数据成员
- 通常采用某种基于定义类的语法形式调用

仅有静态数据成员和静态方法的类，相当于一个简单模块

- 提供模块的内部状态，可以通过所提供的方法修改状态
- 不能生成有用的实例（生成的是空实例，没有局部的实例状态）
- 静态数据成员的静态方法的封装，可能定义内部数据和操作

对象和继承：数据布局

继承关系的数据部分通过对象的适当存储布局实现

- 对象的实际表现就是数据成员的存储
- 假定 B 是一个类，有自己的数据成员
- D是B的派生类，增加了数据成员。D类对象的前部仍是B类的所有成员，扩充的成员排在后面
- 在D类对象里，所有B类成员相对于对象开始位置的偏移量与它们在一个B类对象里的偏移量相同
- 这样，D 类对象就可以作为 B 类对象使用，B 类里的方法能正确操作，它们只看属于 B 对象的那部分
- D 类里的方法既可以使用对象中的 B 类数据成员，也可以使用对象里的 D 类数据成员

用D类对象给B类对象“赋值”（值 copy，或者值语义时）会产生“切割”现象，D 类数据成员不能拷贝

初始化和终结处理

对象可能具有任意复杂的内部结构

- 要求创建对象的程序段做对象初始化，需反复描述，很麻烦，易弄错
- 对象可能要求特殊的初始化方式和顺序，对象的使用者难以贯彻始终
- 继承使对象的初始化更复杂化，因为需要正确初始化继承来的数据成员
- 为更容易处理对象初始化的问题，OO 语言通常都提供了专门的机制，在对象创建时自动调用
- 初始化操作保证新创建对象具有合法的状态。自动调用非常有意义，可以避免未正确初始化造成的程序错误
- 现在常把对象初始化看作调用一个称为构造函数（constructor）的初始化子程序，它（们）在对象的存储块里构造出所需要的对象
- 语言通常支持参数化的初始化操作，以满足不同对象的需要。对象创建可能有多种需要，为此 C++/Java 等都支持一个类有多个不同的构造函数

初始化和终结处理

- 如果变量采用引用语义，所有（值）对象都需要显式创建，有明确的创建动作。这样很容易保证在存储分配之后调用构造函数
- 如果变量是值，为保证初始化，语言需要对变量创建提供特殊语义，要求变量创建包含隐式的构造函数调用
- 对象初始化必须按一定的顺序进行
  - 对象内部的基类部分必须在派生类部分之前完成初始化，因为派生类新增的数据成员完全可能依赖于基类成员的值
  - 数据成员本身也可能是某个类的对象，在执行整体对象的构造函数的过程中，就需要执行这些对象成员的构造函数
  - 这种构造规则是递归的，语言必须严格定义对象的构造顺序
- 如果变量采用值语义（例如 C++），在进入一个作用域的过程中，就可能出现许多构造函数调用
  - 进入作用域可能是代价很大的动作

初始化和终结处理

- 在对象销毁之前，可能需要做一些最后动作（终结处理），例如释放对象所占用的各种资源，维护有关状态等
- 忘记终结处理，就可能导致资源流失，或者状态破坏
- 有些 OO 语言提供终结动作定义机制，销毁对象前自动执行所定义动作
- C++ 采用值语义，终结动作以类的析构函数的形式定义：
  - 类变量是堆栈上的对象，在其作用域退出时，自动调用它们的终结动作
  - 堆对象需要显式释放，释放之前恰好应该执行终结动作，易于处理
- 采用引用语义的语言（如 Java），通常并不提供销毁对象的显式操作（以防悬空引用），对象销毁由 GC 自动进行
  - 有了 GC，对终结动作的需求大大减少，终结动作由 GC 自动进行
  - 执行终结动作的时间不可预计，出现了（时间和顺序的）不确定性
  - 对象关联和 GC 顺序的不确定性使终结动作很难描述

静态和动态约束的方法

- OO 语言里的方法调用通常采用 x.m(...) 的形式，其中
  - x 是一个指向或者引用对象的变量
  - m 是 x 的定义类型（类，假定为 B）的一个方法

问题：x.m(...) 所调用的方法何时/根据什么确定？两种可能性：

- 根据变量 x 的类型（在程序里静态定义）确定（静态约束）
- 根据方法调用时（被 x 引用/指向）的当前对象的类型确定（动态约束）
  - 由于 x 可能引用（指向）B 类或其任何子类的对象，因此同为这个方法调用，不同的执行中实际调用的完全可能是不同的方法

所有 OO 语言都支持动态方法约束（否则就不是 OO 语言），多数以它作为默认方式。少数语言同时也支持静态约束的方法，如 C++、Ada 等

- C++ 把动态约束的方法称为虚方法（virtual 方法），而且以静态约束作为默认方式。这种设计与它的 C 基础有关

静态约束的实现

调用静态约束的方法，实现方式就像是调用普通子程序（过程/函数），唯一不同之处就是需要传入一个指向调用对象的指针

在符号表里，每个类的记录项都包含了一个基类索引，依靠这个索引形成的基类链就可以静态（编译时）完成静态约束的方法的查找：

1. 首先在变量所属的类（静态已知）里查找（查找符号表）。如果在这里找到了所需要的方法，就生成对它的调用；如果不存在就反复做下一步
2. 转到当前类的基类里去查找相应方法，如果找到就生成对它的调用；如果找不到就继续沿着基类链上溯查找
3. 如果已无上层基类，查找失败。报告“调用了无定义的方法”错误

所有对静态约束的方法的调用都可以静态（编译时，一次）处理

- 运行时的动作与一般子程序调用完全一样，没有任何额外运行开销
- 如果语言允许静态约束的方法，采用静态约束可以提高效率。静态约束的方法还可以做 inline 处理

方法的动态约束

```
class B {
  ... ...
  T0 tem(...) { ... sp(...) ... }
  virtual T1 sp(...) { ... ... }
}

void fun(B &x) {... x.tem(...) ...}

class D : public B {
  T1 sp(...) { ... ... }
}

B b;
D d;

fun(b);
fun(d);
```

- **B** 类里定义了一个一般性操作 **tem**，对所有 **B** 类对象都有价值
- **tem** 中调用了一个特殊操作 **sp**，该操作可能因子类不同而不同
- 子类 **D** 覆盖操作 **sp**后，仍能正常地调用操作 **tem**，而且其中对 **sp** 的调用能调用到 **D** 类里新的操作定义

这是 **OO** 程序设计里最重要的东西：

这一特征使新类给出的行为扩充（或修改）可以自然地融合到已有功能里，包括放入已有的操作框架里（这个例子就是）

动态约束的实现：一般模型

对最一般的对象模型，运行中调用动态约束的方法时要做一次与编译时处理静态约束方法一样的查找，这种查找可能非常耗时

为完成这种方法查找：

- 每个类需要有一个运行时表示（把类也作为程序对象），类表示中需要有一个成分是基类索引，还有一个成分是方法表
- 每个对象里必须保存所属类的信息（一个类指针，指向其类）
- 每个动态方法调用都启动一次方法查找。如果找到就调用，找不到就发出一个 **“message is not understood”**（**Smalltalk**）动态错误

这种方式普遍有效，可以处理具有任何动态性质的对象模型，如动态类层次结构构造和动态方法更新（修改、添加和删除）、动态类属关系等

- 查找的时间开销依赖于继承链的长度和继承链上各个类中方法的个数
- 这种方法的缺点是效率太低。如果所采用的对象模型在动态特性方面有所限制，就可能开发出效率更高的方法

动态约束的实现：受限模型

早期 OO 语言（包括 Smalltalk）都采用功能强大灵活的对象模型

- 在提供了极大灵活性的同时，也带来效率上的巨大开销
- 这也是早期 OO 语言及其概念难被实际软件工作者接受的最关键原因

提高算法效率的最基本途径是限制要解决的问题（对更特殊一些的问题，可能找到效率更高的算法），并设计优化的实现模型

对于 OO 语言，就是要找到一个受限的对象模型，它能比较高效地实现，同时又能满足绝大部分实际 OO 程序开发的需要

常规 OO 语言中的对象模型有如下特性（足以支持常见程序设计工作）：

- 类层次结构是静态确定的
- 每个类里的动态约束方法的个数和顺序都静态确定

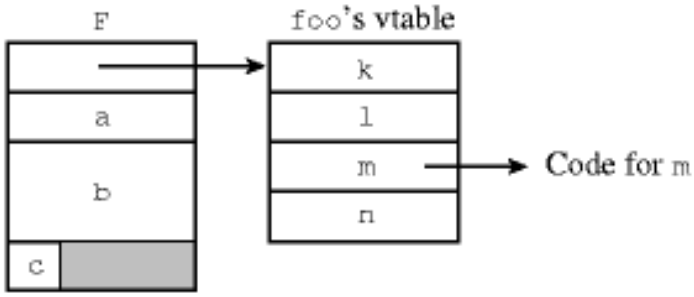
在这种模型里就可以避免动态方法查找，使方法调用的执行效率接近普通的子程序调用的执行效率（C++ 和 Stroustrup 的贡献）

动态约束的实现

优化实现模型，其中绝大部分工作都能静态完成：

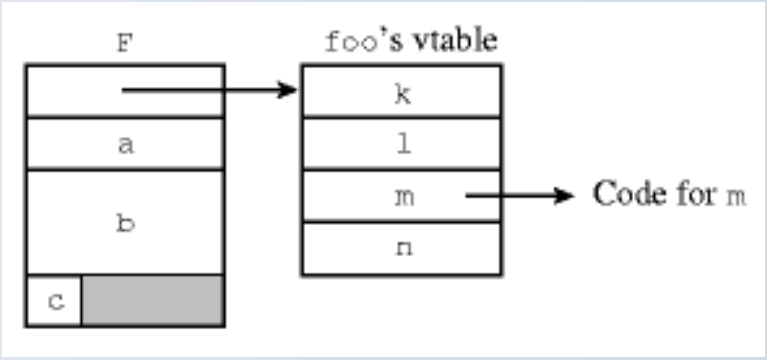
- 每个类的运行时体现是一个动态约束方法的表（称为虚表，vtable），这是一个指针表，指针指向本类的对象需要调用的方法的代码体
- 虚表的指针按方法在类里的顺序排列，每个方法对应于一个顺序下标

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k ( ...
    virtual int l ( ...
    virtual void m ();
    virtual double n( ...
    ...
} F;
```



动态约束的实现

- 在每个对象开头（数据域之前）增加一个指针 **vt**
- 创建对象时，设置其 **vt** 指向其所属的类的虚表（运行中始终不变）



f 是指向 **F** 的指针（或引用）

调用 **f->m(...)**的实现

```
to call f->m():

    r1 := f
    r2 := *r1                       -- vtable address
    r2 := *(r2 + (3–1) × 4)         -- assuming 4 = sizeof(address)
    call *r2
```

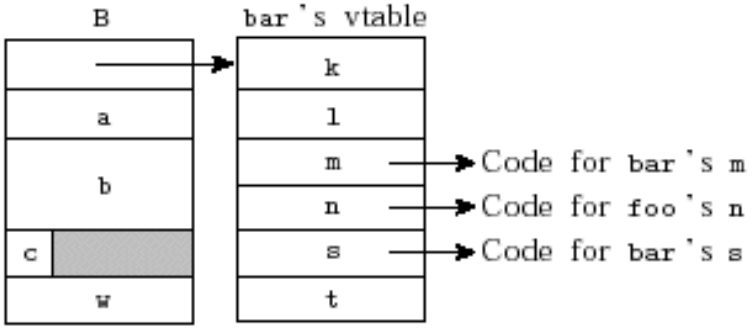
比调用静态约束的方法多了中间的两条指令，它们都需要访问内存

动态约束的实现

虚表的创建：

- 如果类 **B** 没有基类，就将其定义里的动态约束方法的代码体指针依次填入它的虚表（下标从 **0** 或者 **1** 开始算）
- 若类 **D** 的基类是 **B**，建立 **D** 的虚表时先复制 **B** 的虚表。如 **D** 覆盖了 **B** 的某个（某些）动态约束方法，就用新方法的指针覆盖虚表里对应的已有指针。若 **D** 定义了新的动态约束方法，就将它们顺次加入虚表，放在后面

```
class bar : public foo {
    int w;
public:
    void m ();  //override
    virtual double s ( ...
    virtual char *t ( ...
    ...
} B;
```



如果 **f** 指向的对象是 **B**，那么 **f->m(...)** 也会调用正确的方法

动态约束的实现

重温受限的对象模型（对一般程序设计已经足够强大）：

- 类层次结构是静态确定的
- 每个类里的动态约束方法的个数和顺序都静态确定

优化实现的效果：

- 构造方法表的工作在编译时完成
- 每个对象里需要增加一个指向其类的方法表的指针
- 每次方法调用需要多执行两条指令（典型情况），多访问内存两次

- 对这种受限对象模型，动态约束方法调用的额外开销不大，一般软件系统（包括系统软件的绝大部分情况）都可以接受

  Stroustrup 在设计和实现 **C++** 语言时特别希望能够得到高效的实现，最后选择了这种对象模型，并设计了这种高效的实现方法

动态约束的实现

- 对数据抽象和面向对象技术的支持，以及高效的实现，使实际软件工作者看到了 **C++**（和 **OO**）的潜力，最终导致了面向对象的革命
- 以后的主流面向对象语言也都采用了这种技术。当然，采用这种选择，也就对它们可能采用的对象模型提出了严格的限制
- **Pragmatics** 里本章最后的练习里还讨论了其他高效实现方法，《**C++** 语言的设计与演化》里也有讨论（通过几条指令构成的一段“蹦床代码”，将控制转到实际应该调用的方法，主要是要解决多重继承问题）

虚方法（动态约束方法）的一个重要缺点是不能做 **inline** 处理（在线展开要求静态确定被调用的方法），使编译器难以进行跨过程的代码优化

C++ 希望支持高效的系统程序设计，认为虚方法带来的效率损失有时也是不能容忍的，因此它同时支持静态方法约束

注意：如果一个类里只有静态约束的方法，该类编译之后就不会生成方法表，该类的对象也没有一个指针的额外存储开销

类层次结构和强制转换

```
class foo { ...
class bar : public foo { ...
...
foo F;
bar B;
foo* q;
bar* s;
...
q = &B;          // ok; references through q will use prefixes
                 // of B's data space and vtable
s = &F;          // static semantic error; F lacks the additional
                 // data and vtable entries of a bar
```

- C++ 代码：
- 基类指针可以安全地引用派生类的对象，这时的（非变换）自动类型转换称为“向上强制”，**upcasting**
- 但子类指针不能引用基类对象

- 向上强制总是安全的，不会出问题，总可以自动进行。因为派生类包含基类所有数据成分，因此可以支持基类所有操作
- 后一个赋值是编译时错误，派生类指针不能引用基类对象

类层次结构和强制转换

- 如果用 **foo** 类的指针 **q** 传递一个对象
- 可保证该对象一定是 **foo** 的或它的某个派生类的对象
- 如果由 **foo** 类指针 **q** 传递的实际上是一个 **bar** 对象，我们有时需要把它作为 **bar** 对象使用，例如想对它调用 **foo** 里没有的方法
  - `q->s(..)` 是静态类型错误（**q** 的指向类型是 **foo**，**foo** 无方法 **s**）
  - `s = q` 也是静态类型错（不能保证 **q** 指向的是 **bar**，赋值不安全）

能不能用 `s = (bar*)q` ？

- 如果 **q** 指向的确实是一个 **bar** 对象，当前情况下恰好可以，因为
  - **(bar*)** 对指针是“非变换转换”，导致把 **foo** 指针当做 **bar** 指针
  - 恰好 **bar** 对象的起始位置和各成分的偏移量与 **foo** 一样

这些条件有时不成立（下面会看到，在存在多重继承时）

这种转换不安全，它要求 **q** 指向的确实是 **bar**。动态怎么检查类型？

类层次结构和强制转换

- C++ 为安全的向下强制转换提供了专门运算符 `dynamic_cast`。上述转换的正确写法：

```
bar *x = dynamic_cast<bar*>(q);
```

- 如果 q 指向的确实是 bar 类的对象，转换将成功，x 指向该 bar 类对象
- 如果 q 指向的不是 bar 类的对象，转换失败，x 被赋空指针值 0
- 通过检查 x 的值，可以判断转换是否成功

实现 dynamic_cast，就要求在运行中能判断对象的类型和类型间关系

这就是运行时类型识辨（Run Time Type Identification，RTTI）

要像支持安全的向下转换，C++ 的实现需要在 虚表里增加一个类描述符

- 常放在虚表最前。一些 C++ 编译器要求用户指明需要用 RTTI，在这种情况下才按这种方式创建虚表（虚表的形式与没有类描述符时不同）
- dynamic_cast 检查类型关系，确定能否转换，在能转换就完成转换

类层次结构和动态强制

多数 OO 语言（如 Java 等）默认支持 RTTI，虚表里总保存类描述符

如何描述类型是编译器的具体实现问题，不必关心

RTTI 机制可保证类型安全的转换

虽然 Java 的类型转换采用 C 语言类型转换的描述形式，但功能不同

- 在牵涉到基本类型时，可能需要做值的转换
- 在牵涉到类类型时，需要做动态的类型转换合法性检查
  - 如果发现错误，就抛出异常 ClassCastException
  - 否则做“非变换类型转换”，把相应引用直接当作所需的类型的引用
- 从基本类型值到类类型的合法转换，还需要自动构造对象（boxing）；从类对象到基本类型值的转换需要提取对象内的值（unboxing）

运行时类型描述机制还被用于支持“自反”（reflection）功能

多重继承

有时需要从多个基类出发进行继承，例如需要管理所有同年级（一、二、三、四年级）的学生记录，要建立学生的表，可能希望从两个类继承：

class student : public person, public gp_list_node { ... ... }

这样 **student** 类的对象将同时具有：

- **person** 类对象的成分，因此可以记录个人信息并处理这些信息
- **gp_list_node** 类对象的成分，因此可作为表结点链入表中，进行管理
- 还可以有其自身的特殊的数据成员和操作

通过多重继承定义的类是它的各个基类的子类

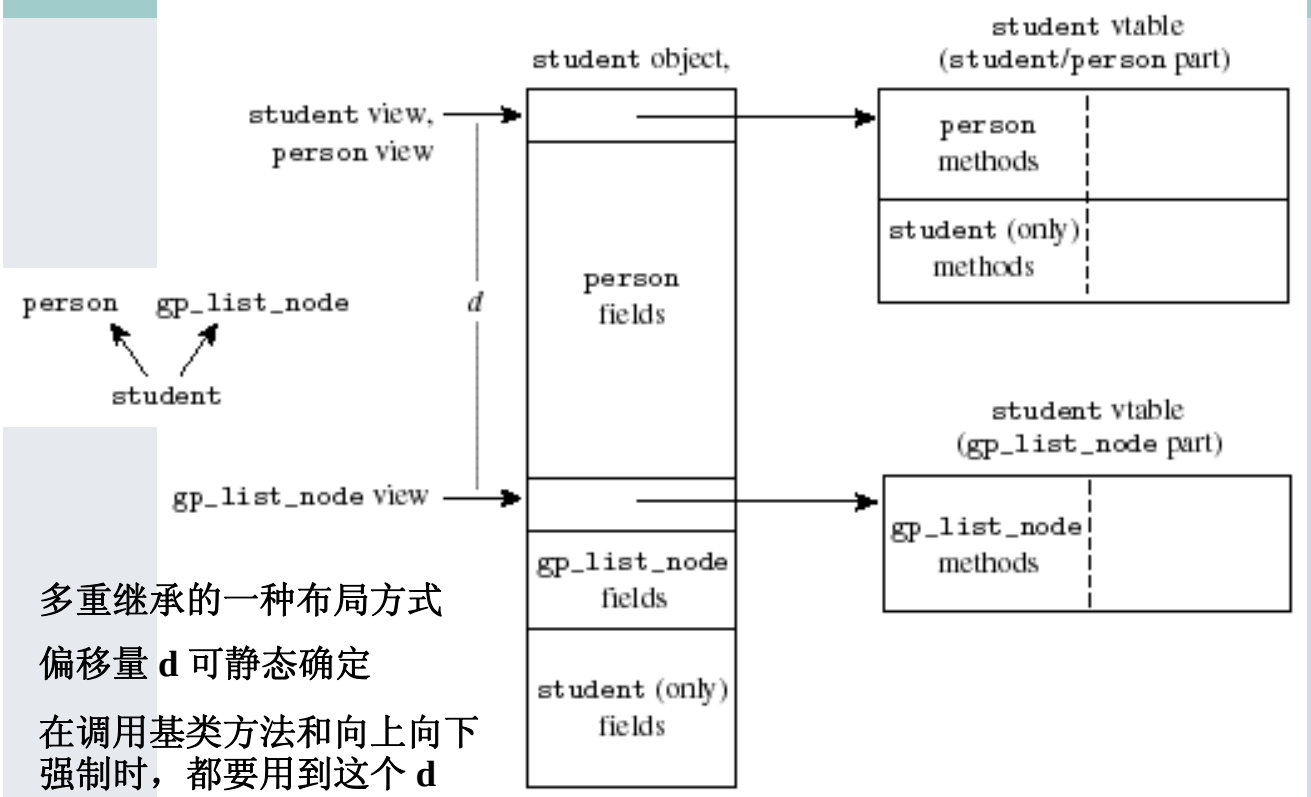
这种派生类的对象也是它的各个基类的对象，在需要这个类的任何一个基类的对象，都可以用这个类的对象

“student 类的对象是 person，也是 gp_list_node”

多重继承

- 如果只有一个基类，我们把基类对象的数据成员按照原来的位置存放在派生类对象数据区域的最前面，这就使
  - 可以对一个对象调用其任意基类的方法，因为数据成员的位置相同
  - 也可以对派生类对象调用其所属类里定义或覆盖的方法
- 如果语言允许多重继承，对象布局首先就出问题了：不可能把多个基类的数据成员都放在派生类对象的数据区域的最前面
- 要实现多重继承，首先必须
  - 设计一种数据布局，把多个基类的数据成员的位置安排好
  - 在需要调用基类的方法时，必须能够找到位于当前对象里的相应的基类对象所在的位置
  - 具体例子：需要能从任何一个 **student** 类的对象产生出它的**“person** 观察点”和**“gp_list_node** 观察点”。注意：最多只能把一个基类对象（的数据成分，例如 **person**）放在 **student** 对象的最前面

多重继承：实现



多重继承的一种布局方式

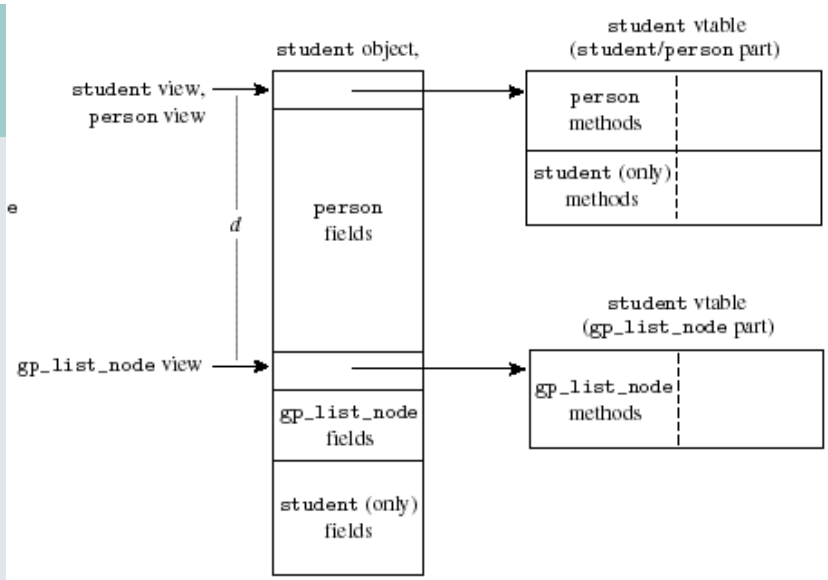
偏移量 **d** 可静态确定

在调用基类方法和向上向下强制时，都要用到这个 **d**

多重继承：实现

student 的虚表分两部分：

- 第一部分是 **person** 的虚方法（可能有覆盖）和 **student** 新增虚方法，虚表指针放在对象开始
- 第二部分是**gp_list_node** 的虚方法（可能有覆盖），虚表指针放在偏移量 **d** 处



student 对象的每个数据成分相对于对象开始的偏移量可静态确定

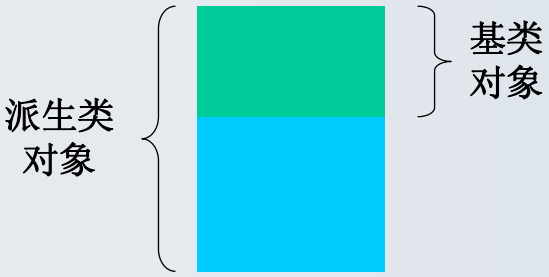
虚方法相对所在虚表的偏移量可静态确定。确定被调用的方法：

- 调用第一部分的方法，确定方法指针的方式与单继承一样
- 调用 **gp_list_node** 定义的方法，取对象地址值加 **d** 得到 **gp_list_node** 的虚表地址，而后按偏移量找到相应的方法指针

多重继承：实现

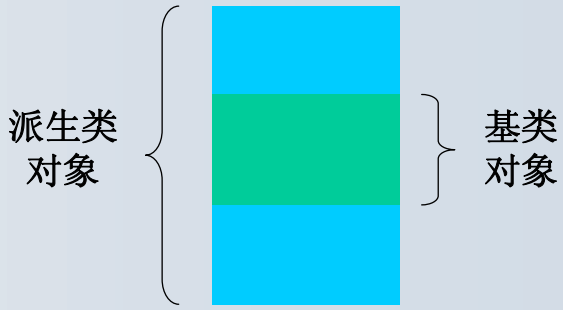
问题：如何给被调用的方法传送 this 指针？

如果只有单继承，基类对象总出现在派生类对象的最前面



基类的 this 指针也就是整个（派生类）对象的 this 指针

当允许多重继承时，基类对象可能出现在派生类对象里的任何位置



如果只有对象里基类对象的 this 指针，如何找到整个对象的 this？

```
func (gp_list_node *x) { ... x->md(...) ... }
```

假定 gp_list_node 有方法 md。实际调用 func 时送的可能是一个 student 对象。在调用 md 时，送去的 this 指针应该是什么？

多重继承：实现

```
func (gp_list_node *x)
{ ... x->md(...) ... }
```

假定调用 func 时的实参确实是个 student 对象，那么：

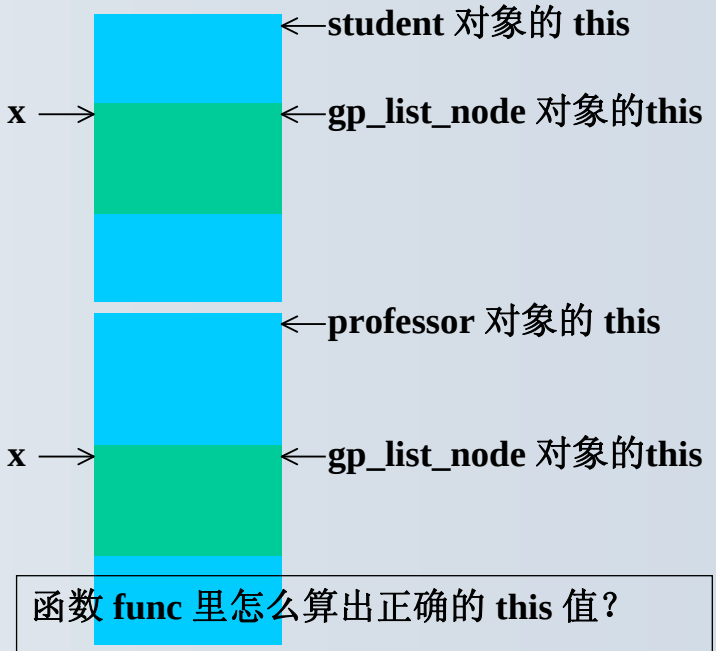
- 若 student 没有覆盖 md，送给 md 的 this 指针就是 x
- 若 student 覆盖 md，送给 md 的 this 应指向包含 x 所指对象的那个 student 对象

假定 professor 从 person 的子类 employee 和 gp_list_node 派生

gp_list_node 对象在 professor 里的位置可能与在 student 里不同

func 也可能用于 professor

调用 func 时，参数可能是个 student 对象。调用 md 时，送的 this 指针应是什么？



函数 func 里怎么算出正确的 this 值？

注意：这一计算对每个派生类可能不同！

多重继承：实现

```
func (gp_list_node *x)
{ ... x->md(...) ... }
```

- 定义派生类（例如 **student**、**professor**）之前，并不知道与之对应的 **d** 值是什么（而且，不同派生类可能有不同的 **d** 值）
- 同一派生类的不同方法，覆盖的就需要调整，未覆盖的不调整
- 由于动态约束，编译时（静态）不知道该不该调整 **this** 值
- 结论：如果没有其他信息，编译不知道怎么计算送给 **md** 的 **this** 值

细节情况：

- 调整或者不调整，是由具体的方法确定的
- 即使要调整，对于不同的类也可能需要不同的调整值

解决办法是在派生类（的虚表）里为计算 **this** 值提供信息。一种方法：

- 在虚表里为每个方法指针附一个 **this** 校正值
- 如果 **student** 覆盖方法 **m**，其校正值为 **-d**，没覆盖时调整值为 **0**
- 调用 **m** 时给当时的 **this** 加上这个校正值，而后转入方法体

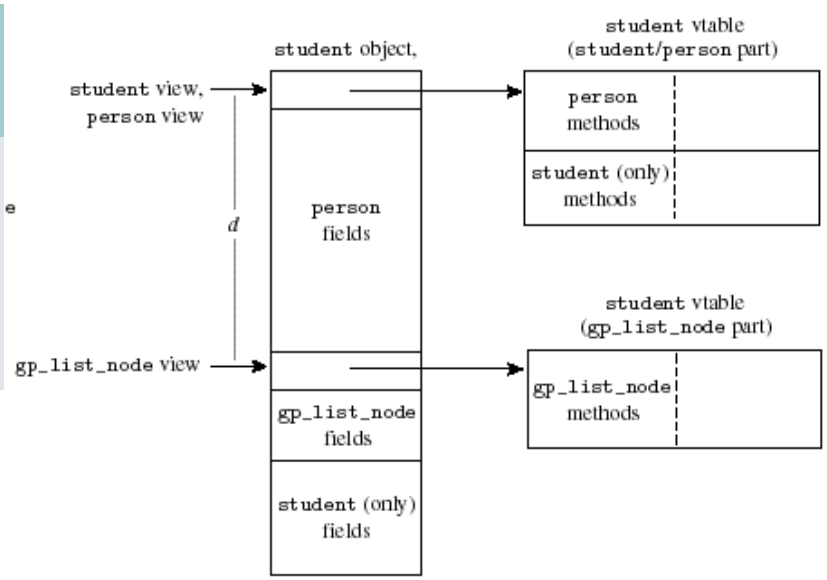
多重继承：实现

- 假定对象是 **my_student**
- 假定调用**gp_list_node** 的第三个方法

实现调用的指令：

```
r1 := my_student
r1 := r1 + d
r2 := *r1
r3 := *(r2 + (3–1) × 8)
r2 := *(r2 + (3–1) × 8 + 4)
r1 := r1 + r2
call *r3
```

r3 是被调方法指针，**r1** 是送给方法的 **this** 值

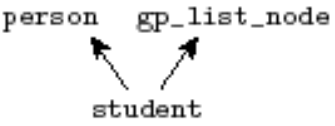


如果语言里存在多重继承，所有方法表都要增加 **this** 校正值，每次方法调用都必须考虑 **this** 校正（回想前面例子）

实现的复杂性，空间开销和方法调用的时间开销都增加了

无论一个类是否用到多重继承，都需要付出额外代价。代价是全局性的，即使不用（不用多重继承）也要付出代价

多重继承：歧义性



- 假定 **person** 和 **gp_list_node** 里都定义了 **debug_print** 方法
- 现在有 **student*** 类型的变量 **s**，问：

  `s->debug_print()` 调用哪个方法

这是多重继承引起的语义歧义性问题。不同语言采用了不同处理方法

C++ 要求明确重新定义存在歧义的方法（否则编译报错）：

```
void student::debug_print () {
    person::debug_print ();
    gp_list_node::debug_print ();
}
```

如果程序里总是要顺序地调用两个方法

```
void student::debug_print_person () {
    person::debug_print ();
}
void student::debug_print_list_node () {
    gp_list_node::debug_print ();
}
```

如有可能需要分别调用两个方法中的任何一个

多重继承：歧义性

如同名方法是虚方法，需要在多重继承的派生类里覆盖，问题更难办（覆盖哪一个）。**Stroustrup** 提出了定义转接类作为过渡的技术：

```
class person_interface : public person {
    virtual void debug_print_person () = 0;
    void debug_print () {debug_print_person ();}
        // overrides person::debug_print
};
class list_node_interface : public gp_list_node {
    virtual void debug_print_list_node () = 0;
    void debug_print () {debug_print_list_node ();}
        // overrides gp_list_node::debug_print
};
class student : public person_interface, public list_node_interface {
public:
    void debug_print_person () { ...
    void debug_print_list_node () { ...
    ...
};
```

这里就可以分别写覆盖定义

其他提供多重继承的语言可能支持不同方式

多重继承：歧义性

如果派生类 D 继承基类 B 和 C，而 B 和 C 都继承一个公共基类 A

- 在 D 类型的对象里到底有几个 A 对象？（一个还是两个？）

书上的例（在职研究生）：

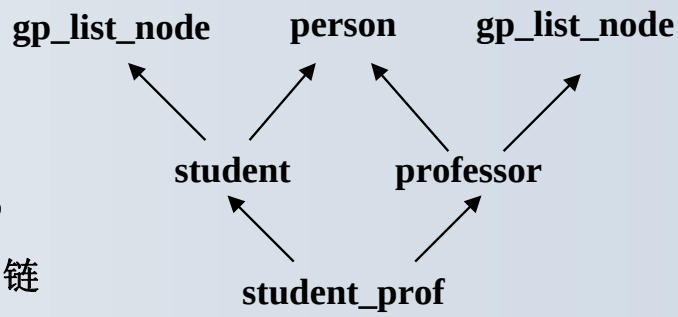
class professor : public person, public gp_list_node { ... ... }

class student_prof : public student, public professor { ... ... }

student_prof 继承 person 和 gp_list_node 各两次

可能希望 **student_prof** 对象里：

- 只有一个 **person**（是同一个人）
- 有两个 **gp_list_node**（以便可以链接到两个不同的表里）



希望的继承形式

多重继承：歧义性

两种不同需要和相应的继承概念

- 对 **gp_list_node** 的继承要求为继承树的不同分支分别提供不同的副本，这种方式称为复本式继承
- 对 **person** 的继承要求所有分支只有一个副本，称为共享式继承

C++ 里默认为复本式继承

可以为基类加 **virtual** 修饰，要求采用共享式继承

class professor : public virtual person, public gp_list_node { ... ... }

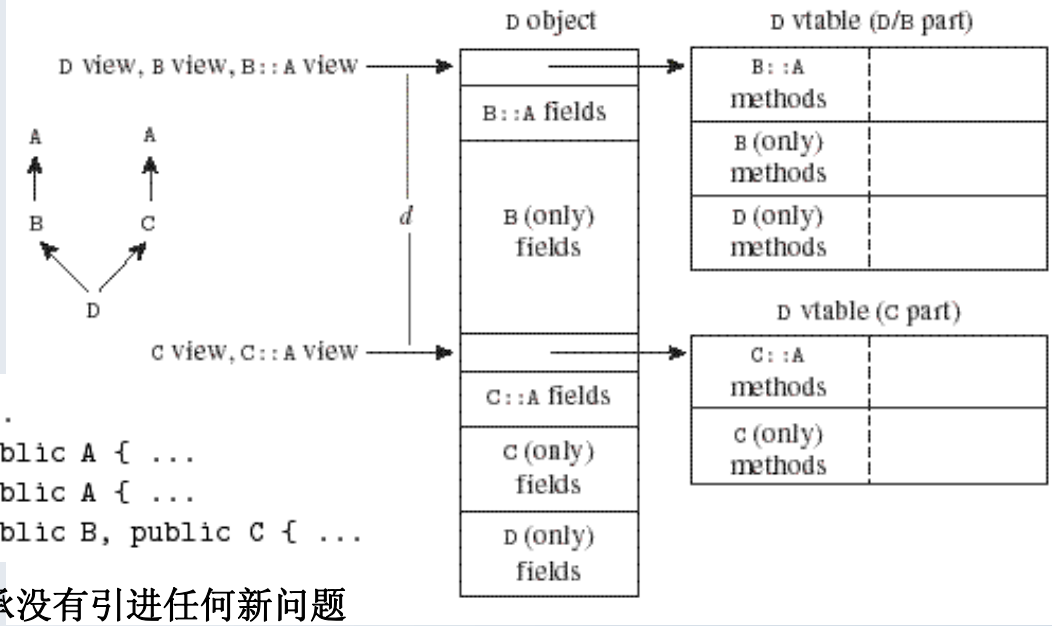
class student : public virtual person, public gp_list_node { ... ... }

class student_prof : public student, public professor { ... ... }

前一定义保证最终的 **student_prof** 对象符合我们的需要

其他支持多重继承的 **OO** 语言在这方面可能有不同的规定

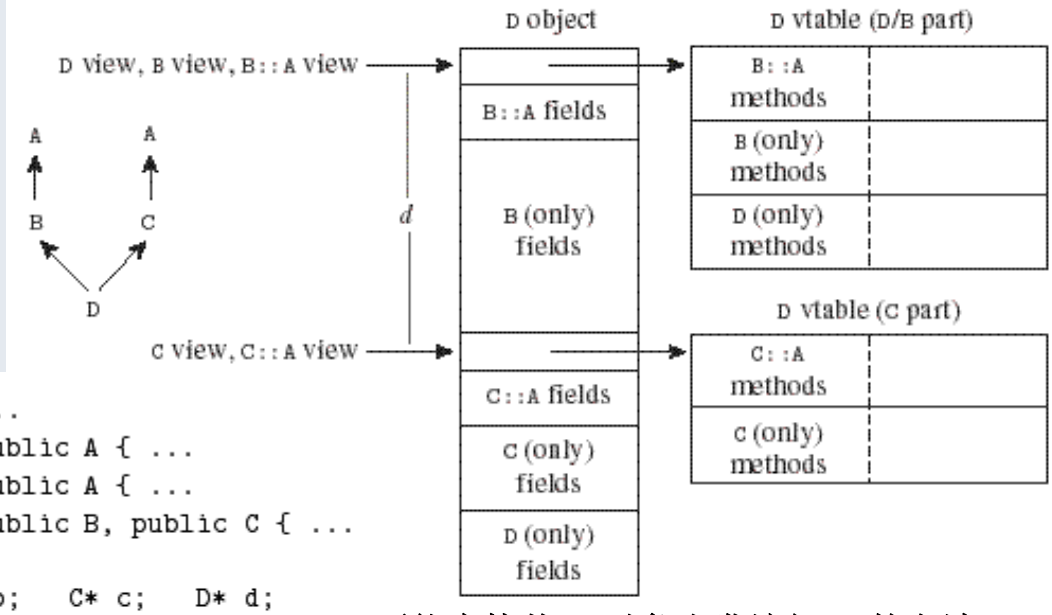
多重继承：复本式




```
class A { ...
class B : public A { ...
class C : public A { ...
class D : public B, public C { ...
```

- 复本式继承没有引进任何新问题
- 类 **D** 通过两条不同路径继承两个基类 **A**，其对象里有两份 **A** 数据成分
- 方法表里有两组 **A** 的虚方法项。究竟调用那里的方法，要根据取得 **D** 对象的哪个基类观察点（**B** 或 **C**）确定

多重继承：复本式



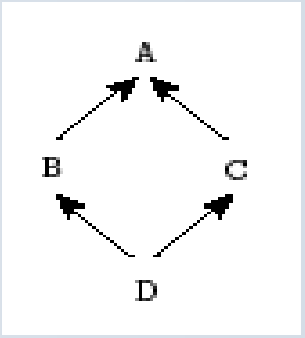

```
class A { ...
class B : public A { ...
class C : public A { ...
class D : public B, public C { ...
...
A* a;   B* b;   C* c;   D* d;
a = d;  // error; ambiguous
b = d;  // ok
c = d;  // ok
a = b;  // ok; a := d's B's A
a = c;  // ok; a := d's C's A
```

- 不能直接从 **D** 对象出发访问 **A** 的方法
- 必须先转到 **B** 或 **C** 观点（通过指针或引用方式），而后才能调用 **A** 的方法
- 虚表项里也需要校正值

多重继承：共享式

```
class A {
public:
    virtual void f ();
    ...
}

class B : public virtual A { ...
class C : public virtual A { ...
class D : public B, public C { ...
```



现在 **D** 对象里只有一个 **A** 对象，应该只有一组 **A** 的虚方法

如果没有覆盖，情况还比较简单

若 **B** 覆盖了 **A** 的虚方法 **f**，那么 **D** 应该通过 **C** 继承原来 **A** 的方法 **f**，还是由 **B** 继承新定义的 **f**？————出现歧义！（两个类都覆盖的情况类似）

这是共享式继承带来的新问题

多重继承：共享式

C++ 禁止这种歧义性，要求：

- 如果存在覆盖，必须有一个定义位于其他相关类的公共派生类里
- 上面情况不满足这一规定，因此不合法

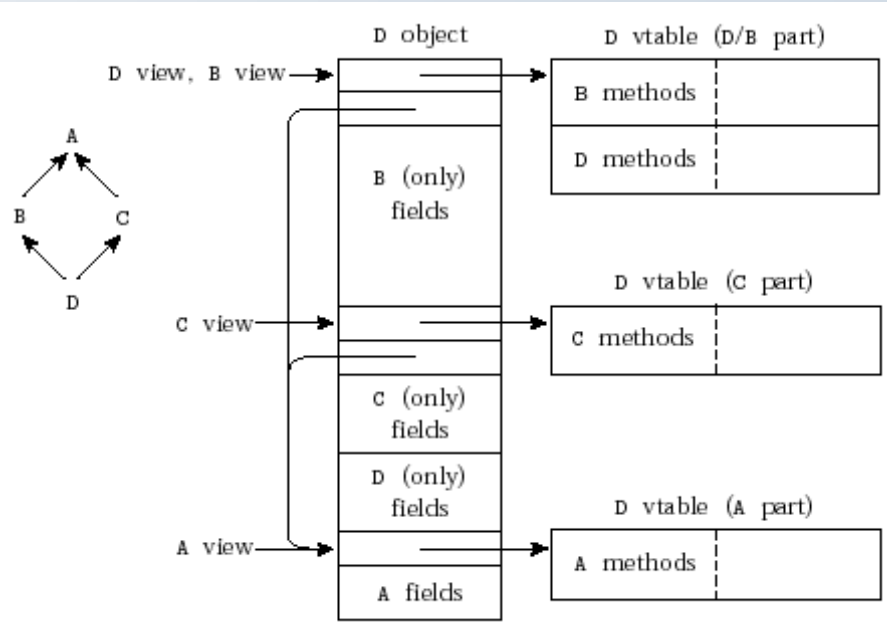
其他语言（如果有多重继承）可能采取不同的管理策略处理这个问题

- 在 **Eiffel** 语言里
  - 如果一个类可能从多个基类继承来同一方法的不同版本，可以在这个类里明确说明要继承哪一个
  - 也可以通过重命名的方式消解这种歧义性

多重继承：共享式

布局：

- 在 **D** 对象里只有一个 **A** 对象
- **D** 里的 **B** 对象和 **C** 对象不可能都连续
- 图中方式是让 **D** 里的 **B** 和 **C** 都不连续，**A** 部分单独放置
- 这时需要在各个派生类对象里保存 **A** 对象的地址
- 这个地址相对于各子对象的距离都是编译确定的常量



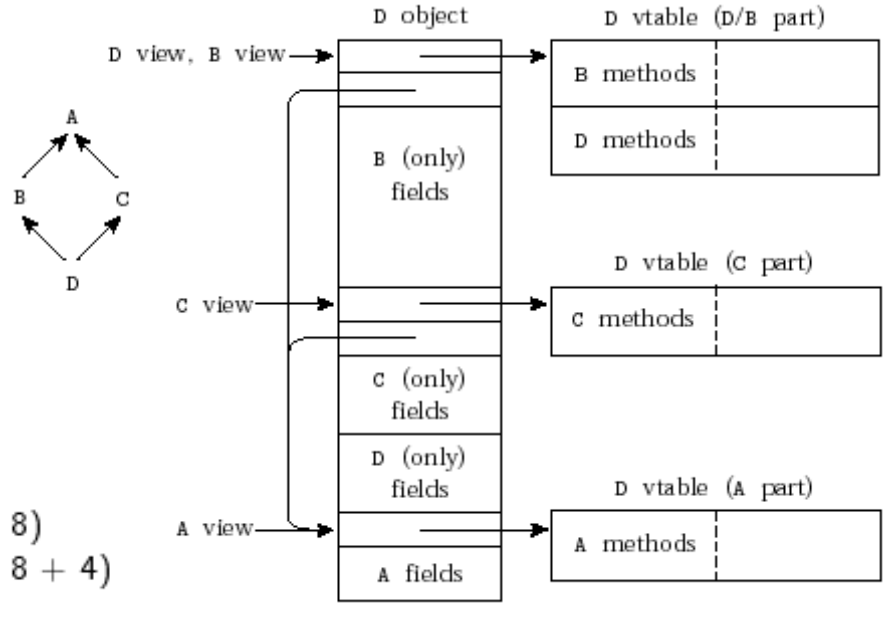
多重继承：共享式

数据成员访问：

要访问 **A** 的成员，根据偏移量计算后间接取成员

要调用 **A** 里的第 **n** 个虚方法：

```
r1 := my_D_view
r1 := *(r1 + 4)
r2 := *r1
r3 := *(r2 + (n − 1) × 8)
r2 := *(r2 + (n − 1) × 8 + 4)
r1 := r1 + r2
call *r3
```



这里还是需要在虚表里保存 **this** 校正值

存在解决问题的其他方式（蹦床代码）

混入式继承

```
public class widget { ...
}
interface sortable_object {
    String get_sort_name ();
    bool less_than (sortable_object o);
    // All methods of an interface are automatically public.
}
interface graphable_object {
    void display_at (Graphics g, int x, int y);
    // Graphics is a standard library class that provides a context
    // in which to render graphical objects.
}
interface storable_object {
    String get_stored_name ();
}
class named_widget extends widget implements sortable_object {
    public String name;
    public String get_sort_name () {return name;}
    public bool less_than (sortable_object o) {
        return (name.compareTo (o.get_sort_name ()) < 0);
        // compareTo is a method of the standard library class String.
    }
}
```

Java 代码示例：向一个（一组）类里逐步混入若干接口（interface）的虚方法

三个接口：

- 排序
- 显示
- 保存

继承 **widget**，

混入一个接口的方法实现

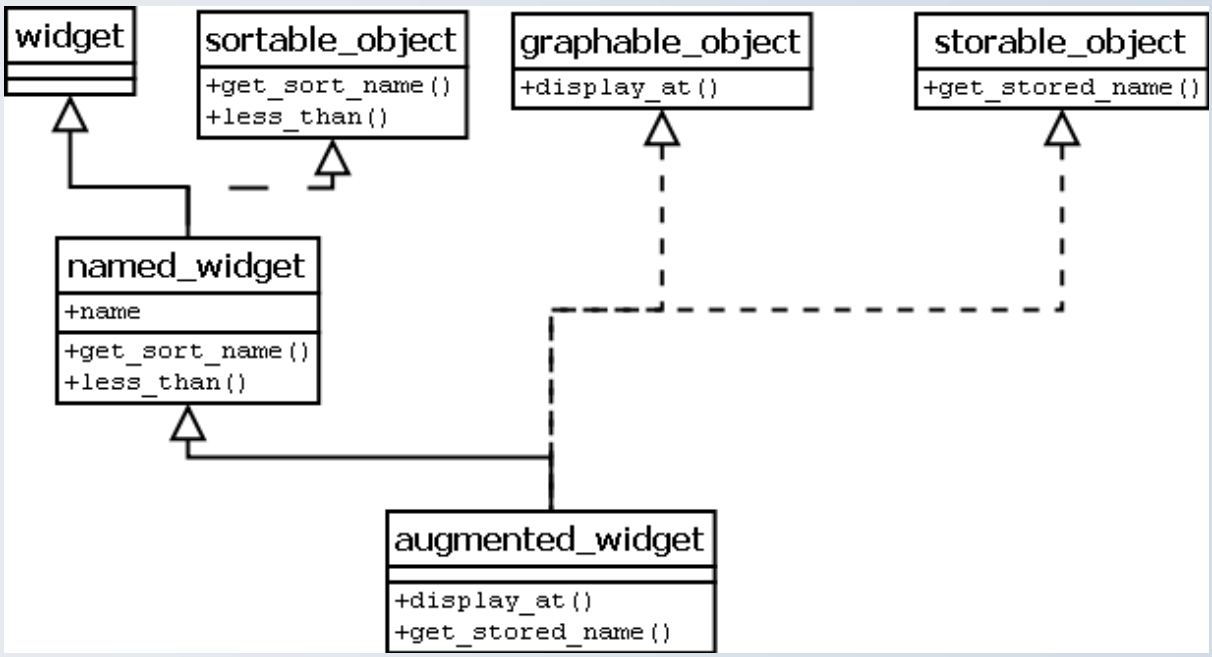
混入式继承

```
class augmented_widget extends named_widget
        implements graphable_object, storable_object {
    ...         // more data members
    public void display_at (Graphics g, int x, int y) {
        ...     // series of calls to methods of g
    }
    public String get_stored_name () {return name;}
}
...
class sorted_list {
    public void insert (sortable_object o) { ...
    public sortable_object first () { ...
    ...
}
class browser_window extends Frame {
    // Frame is the standard library class for windows.
    public void add_to_window (graphable_object o) { ...
    ...
}
class dictionary {
    public void insert (storable_object o) { ...
    public storable_object lookup (String name) { ...
    ...
}
```

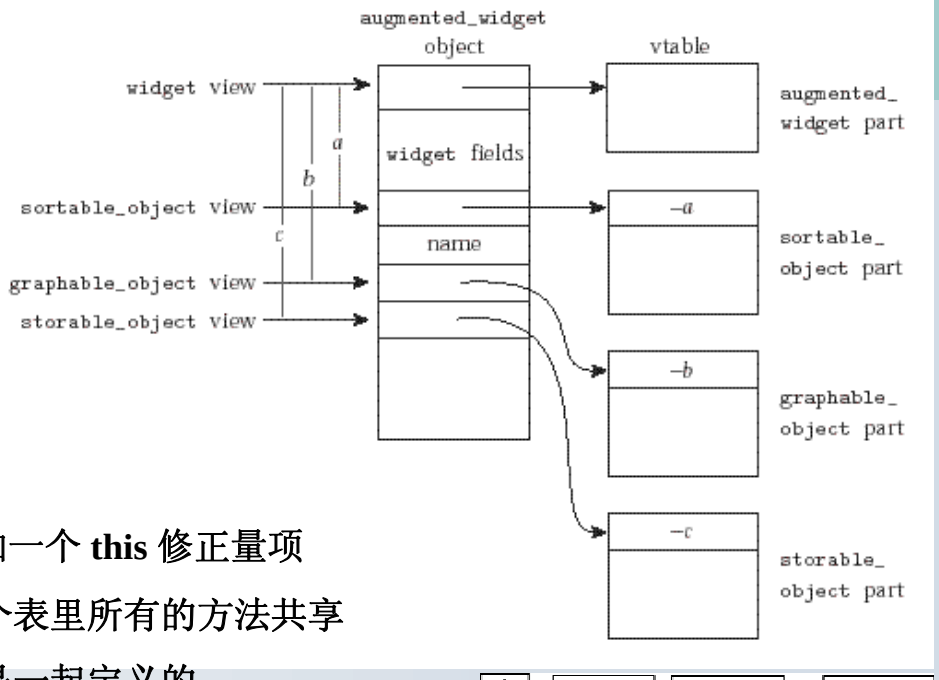
混入另外两个接口的方法实现

augmented_widget

- 可以放入排序表
- 可以在窗口里显示
- 可以保存到字典里



混入式继承



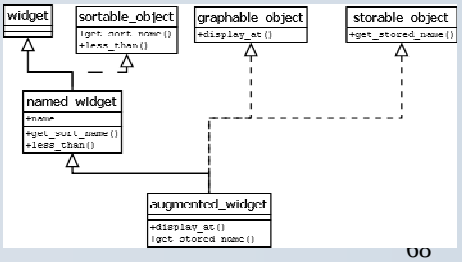
混入式继承的布局和方法调用：

成员布局可以简单地根据逐步混入的顺序排列

各个混入部分建立独立的方法表

在每个方法表前面加一个 **this** 修正量项

- 这一修正量由这个表里所有的方法共享
- 因为这些方法都是一起定义的



不能直接把“混入”的方法直接加在原来的方法表之后，因为每个观点都需要像一个独立的对象，可以独立操作和使用

面向对象的部分总结

- 面向对象的三个基本概念：
  - 封装（数据抽象）
  - 继承（类型扩充，代码重用）
  - 动态方法约束（调整抽象的行为方式）
- 实现面向对象的支持：
  - 需要引用或者指针概念的支持
  - 需要考虑自动的初始化和终结处理
  - 通常需要以废料收集为主要特征的存储管理机制的支持
- 运行中最重要的问题是支持动态约束
  - 在受限的对象模型中，可以通过虚表技术得到很好的性能
- 多重继承可以解决一些编程问题，但也带来概念和实现方面的困难和代价
- 混入方式可以在许多情况下替代多重继承