Data Abstraction and Object Orientation

第10章 数据抽象和面向对象

第3章已经说明了数据抽象发展过程的几个重要阶段，其中特别强调了控制名字可见性的作用域机制。那里的讨论从全局变量开始，它们的生存期是程序的整个执行期间。而后加入了局部变量，其生存期限制在一个子程序的执行期间；又加入嵌套的作用域，使子程序本身还可以是局部的；再加入了静态变量，其生存期是整个程序执行，而它们的名字只在一个作用域的内部可见。在所有这些之后的是模块，它们使一组子程序可以共享一集静态变量；再就是模块**类型**，它使程序员可以从一个给定抽象出发，通过实例化产生出多个实例；再后面是**类**，它使程序员可以定义一族相关的抽象。

常规模块鼓励一种“管理器”风格的程序设计，让每个模块导出一个抽象类型。模块类型和类使模块本身**就是**抽象类型。这两种方式之间的差异在下面两方面表现得特别明显：首先，原先需要从管理器模块导出的显式`create`和`destroy`例程，现在被模块类型的实例创建和撤销动作取代了。其次，对模块特定实例的例程调用取代了对公用例程的调用，后一种调用总期望有一个具有模块导出类型的变量作为参数。类是在模块类型的基础上建立起来的，其中增加了**继承**的机制，使我们可以把新抽象定义为已有抽象的某种精化或扩充；还增加了**动态方法约束**，使一个抽象的新版本可以表现出某种新的精化后的行为，即使是被用在期望这个抽象的老版本的上下文中。类的实例称为**对象**，基于类的语言和程序设计技术也称为面向对象的[1]。

[1] 前几章一直非形式地用术语“对象”指任何可以有名字的东西。本章将特别用这个术语指类的实例。

第 3 章给出了数据抽象机制的一步步发展轨迹。虽然那是一种组织思想的很好方式，但却没有完全反映有关语言特征发展的历史轨迹。特别的，上述介绍好像认为面向对象的程序设计是从模块概念里成长起来的，真实情况并非如此。事实上，面向对象程序设计的三个基本概念——封装、继承和动态方法约束——都根源于 20 世纪 60 年代开发的 Simula 程序设计语言。与各种新的面向对象语言相比，Simula 在封装的数据隐蔽方面比较弱，70 年代的 Clu、Modula、Euclid 以及其他相关语言在这个领域做出了最重要贡献。在此同时，有关继承和动态方法约束的思想被 Smalltalk 语言采纳，并经过整个 70 年代的进一步精化。Smalltalk 采用了另一种不同的“基于消息的”程序设计模型，采用动态类型以及很不寻常的术语和语法形式。采用动态类型使 Smalltalk 实现的速度相对较慢，也使对错误的报告推迟了。这一语言还和一个图形形式的程序设计环境紧密集成在一起，使它很难移植到其他系统上去。由于这些原因，Smalltalk 不像人们预期的那样使用广泛，对进一步开发的影响也不如预期的那样大。更新一些的面向对象语言包括 Eiffel、C++、Modula-3、Ada95 和 Java，它们在很大程度上代表着 Smalltalk 的继承和动态方法约束，与“主流”命令式语法和语义的再次集成。面向对象在函数式语言领域也已经变得非常重要，其中引领潮流的记述形式是 CLOS，Common Lisp Object System [Kee89，Ste90 第 28 章] 等。

在第 10.1 节里，我们将概述面向对象程序设计及其三个基本概念；第 10.2 节考察封装和数据隐蔽的进一步细节；第 10.3 节研究对象的初始化和终结处理；第 10.4 节讨论动态方法约束。在第 10.5 节，我们要考虑多重继承的问题，其中一个类是基于多个已有的类定义的。正如将要在那里看到的，多重继承带来了一些特别难处理的语义和实现问题。最后在第 10.6 节里重新考虑面向对象的定义，考察一个语言能够或者应该用对象去模拟所有事物的可能程度。我们的大部分讨论将注意力集中在 Smalltalk、Eiffel、C++和 Java，虽然有时也偶尔提到 Simula、Modula-3、Ada 95、Oberon 和 CLOS。

10.1 面向对象的程序设计 Object-Oriented Programming

随着越来越复杂的计算机应用的开发，数据抽象已经变成实际工程中最关键的东西了。由模块和模块类型提供的这种抽象至少带来如下三方面的重要利益：

1. 它可以最大程度地减少程序员必须同时考虑的细节的量，从而减少了人的**概念负担**。
2. 它起到一种**故障遏制**作用，可以防止程序员以不适当方式使用程序中的部件，限制给定部件中能用的程序正文部分，限制在查找程序错误的诱因时必须考虑的程序部分。
3. 它为程序部件之间的**独立性**提供了一个重要层次，使人比较容易设计好程序的结构，使各个部件相互分离，修改部件的内部实现时可以避免改动使用它们的外部代码，或者将它们安装在一个库里，使之能被其他程序使用。

不幸的是，有关模块和模块类型的经验说明，上面第三点所蕴含的重用可能性在实践中却很难做到。人们经常发现，某个原先构造好的模块几乎具有当前新应用所需要的全部性质，但是却不完全适用。或许我们手头有一个队列抽象，而需要的却是能在两端插入删除，而不是受限的完全先进先出顺序（FIFO）。或许我们手头有一个现存的图形用户界面的会话盒抽象，但是却没有提供一种高亮的默认选择机制。或许我们手头有一个用于符号数学的包，但它却假定所有的数都是实数，而不是复数。在所有这类情况中，如果程序员能做的只是复制现存代码，弄清它们的内部如何工作，而后再手工去修改它，而不能以其“现有”形式使用，这些抽象所提供的大部分优势也就丧失殆尽了。因为如果在未来某个时候，程序员必须对抽象做一些修改（纠正其中的错误，或者实现某种升级），那么就必须记住去修改所有的拷贝副本，而这注定是一种令人生厌而且又很容易出错的工作。

面向对象的程序设计可以看作是在这个方向上的一种追求，它使人能更容易地以**扩展**或**精化**现有抽象的方式定义新抽象，从而提高代码重用的可能性。作为起点的实例，现在考虑一个记录的表。图 10.1 展示的 C++代码描述了这种表的各方面要素。这个例子采用一种“模块作为类型”的抽象风格：表中的元素是类 `list_node` 的对象。类 `list_node` 包含数据成员（`prev`、`next`、`head_node` 和 `val`）和子程序成员（`predecessor`、`successor`、`insert_before` 和 `remove`）。子程序成员在许多面向对象语言里称为**方法**。C++ 关键字 `this` 表示以当前被执行的方法为成员的那个对象。Smalltalk 里有一个等价的关键字 `self`，Eiffel 里用的是 `current`。

有了这个 `list_node` 类之后，我们就可以按如下方式定义表了：

```
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;
   }
   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");
   }
};
```

图 10.1 C++语言里实现表结点的简单类。这个例子里考虑的是一个整数表

```
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");
    }
};
```

要创建一个空表，可以写：

```
list* my_list_ptr = new list;
```

准备插入表里的记录也以类似方式创建：

```
list_node* elem_ptr = new list_node;
```

在 C++里还可以简单地声明给定类的对象：

```
list my_list;
list_node elem;
```

`list` 类包含了一个作为数据成员的对象（`header`）。用 `new` 创建的该类的对象在堆里分配；如果通过加工声明创建对象，则根据生存期，该对象或者是静态分配，或者在堆栈分配。在任何情况下，创建对象都导致调用程序员描述的初始化代码。C++里的这种代码以特殊方法的形式出现，称为**构造函数**，这种方法的名字与类本身相同。C++还允许程序员描述一个析构函数方法，在对象销毁时将自动调用它，无论是程序员通过显式动作要求销毁，还是由于对象定义所在的子程序退出。析构函数的名字也与类一样，只是要求前面加上符号“~”。

在 `list_node` 的成员表里出现了 `public`（公用）标号，它把抽象的实现所需要的部分成员与抽象的用户使用的成员分隔开。按 3.3.1 节的术语，出现在 `public` 标号之后的那些成员都是从这个类导出的，出现在它前面的成员不导出。这个语言还提供了一个 `private`（私用）标号，如果需要，我们也可以把类中公共可见的部分写在前面。请注意，按 3.3.1 节的定义，C++的类具有开的作用域，什么都不需要显式导入。

与 Ada 里的程序包或者 Modula-2 里的外部（分别编译）模块类似，有关一个类的某些信息也可以放到类声明的外面，通过这一抽象的用户所不能看到的独立文件提供。在我们的例子里，也可以只声明 `list_node` 的公用方法而不提供它们的体：

```
class list_node {
    list_node* prev;
    list_node* next;
    list_node* head_node;
public:
    int val;
    list_node ();
    list_node* predecessor ();
    list_node* successor ();
    int singleton ();
    void insert_before (list_node* new_node);
    void remove ();
    ~list_node ();
};
```

而后可以把这种经过简化的类声明放到 a.*h*“头文件”里，把函数体放入 a.*cc*“实现”文件（有关 C 语言分别编译的规则在 3.6.2 节讨论过。这里的文件后缀命名采用了 GNU g++编译器的规则）。在这个 *.cc* 文件里，方法定义的头部必须用 :: 解析运算符标明所属的类：

```
void list_node::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;
}
```

有关把什么东西放入类声明里而不分开定义，人们提出了两条指导性原则。第一，声明中必须包含足够的信息，保证需要这一抽象的程序员能正确使用。第二，声明中必须包含编译器生成有关代码所需的全部信息。第二条规则的要求相当广泛，趋向于要求把第一条规则所不需要的信息（类的私用部分）都放入界面里，特别是在采用变量的值模型而不是引用模型的语言里。如果编译器必须生成（在堆栈帧里）完成空间分配以保存某对象类型的值的代码，它就必须知道对象的大小，这就是需要把私用数据成员的定义包含在类声明中的根本原因。此外，如果编译器要在线展开某些方法，它就需要这些方法的代码。在线展开面向对象程序里很小的最常用方法，对于得到很好的性能是至关重要的。

与常规命令式程序相比，面向对象的程序倾向于使用更多的子程序调用，而子程序也相对更短一些。在冯-诺伊曼语言中通过直接访问记录域而完成的许多事

情，在面向对象的语言中可能都隐藏到对象方法里了。事实上，许多程序员都认为声明公用数据成员是一种很坏的编程风格，因为这种做法使抽象的使用者可以直接访问内部表示。根据这种说法，我们应该把 list_node 的 val 域改为私用的，并定义 get_val 和 set_val 成员函数去读写它。

假定我们已经有了一个表抽象，但是现在需要一个队列抽象。我们可以从头开始定义队列抽象，但其中的许多代码都与图 10.1 类似。在面向对象的语言里还有另一种更好的方式：可以从表中**派生**出一个队列，让它**继承**已有的数据和子程序成员：

```
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;
    }
};
```

这里的queue称为一个**派生类**（也称**子类**），list称为是它的一个基类（也称**父类**或者**超类**）。派生类自动具有其基类的所有数据和子程序成员[2]，程序员要做的全部事情就是显式声明queue需要的但list缺少的那些成员，在目前这个例子里也就是enqueue和dequeue方法。我们很快会看到一些例子，其中派生类还需要新的数据成员。

通过从已有的类中派生新类，程序员可以建立起任意深度的**类层次结构**，在树的各个层次上加入所需要的功能。Smalltalk 的标准库有多至 7 层派生（图 10.2），类 FileStream 派生自 ExternalStream，它转而又派生自（按顺序）ReadWriteStream，WriteStream，PositionalStream，Stream 和 Object。（与 C++ 不同，Smalltalk 有一个唯一的根超类 Object，所有其他类都由它派生出来。Java 也有一个类似的类；Eiffel 也有，称为 ANY。）

机敏的读者可能已经注意到，我们在前面的表抽象里做了一个不幸的假设，假定表里的每个数据项都是整数。这一假设确实不是必需的。有了继承机制之后，

[2] 实际上，在C++里，如果派生类的用户要想看到基类的成员，就需要在派生类声明的第一行里的基类名字之前写关键字public。有关C++可见性规则的细节在 10.2 节讨论。

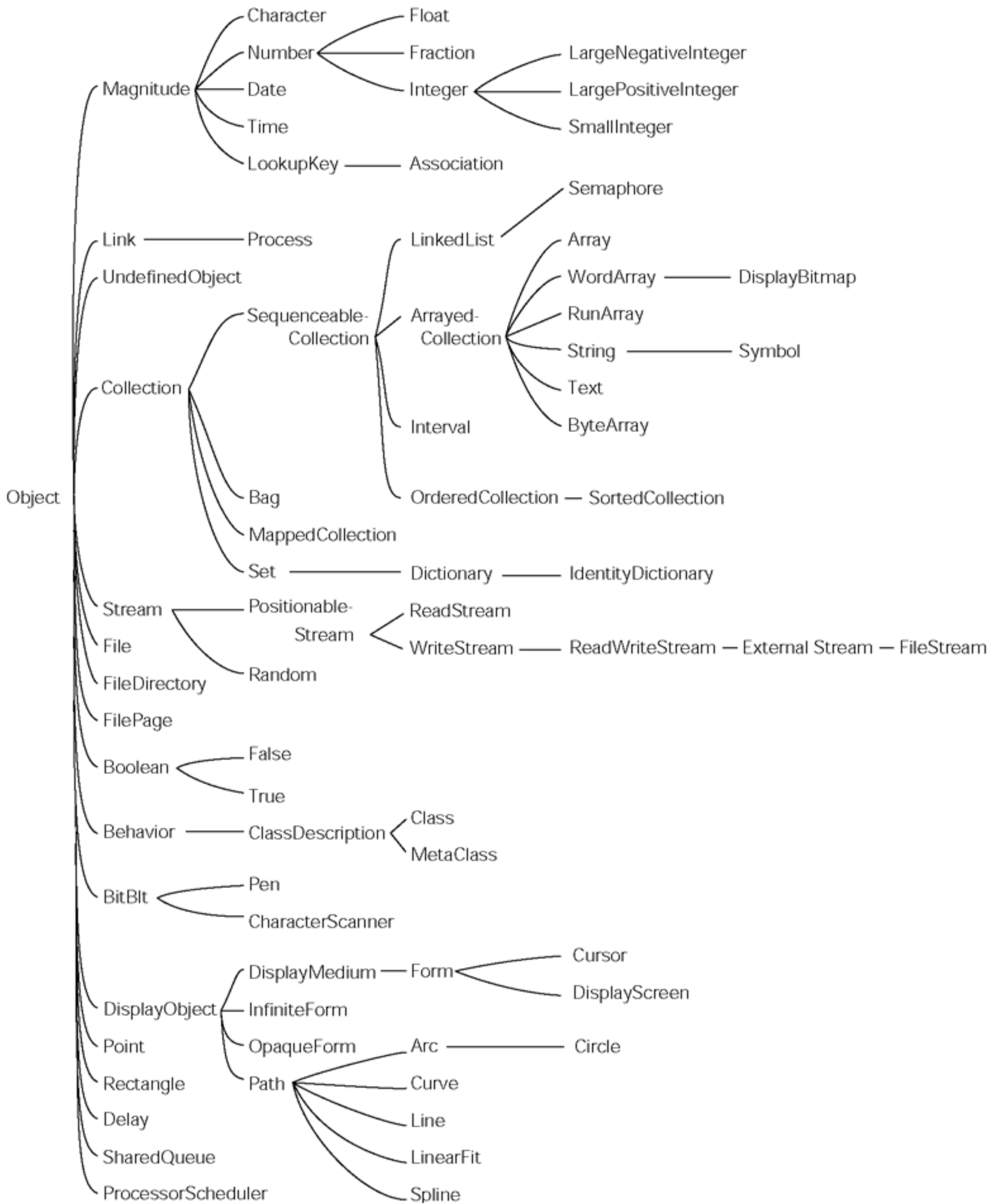

图 10.2 Smalltalk-80 的标准类层次结构

我们就可以创建一个通用元素的基类，其中只包含实现表操作所需的数据和子程序成员：

```
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 ();
};
```

现在我们就可以用这个通用基类派生出有特定类型的数据成员的表和队列了：

```
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;
    }
};
```

模板（通用型）机制为构造特定类型的类提供了更方便的方式，我们将在第 10.4.4 节讨论它。

我们在 int_list_node 里重载了构造函数，提供了两个不同实现，其中一个实现有一个参数，另一个没有参数。现在程序员可以通过给定或不给定初始值的方式创建 int_list_node 了：

```
int_list_node element1;                              // val = 0
int_list_node *e_ptr = new int_list_node (13);      // val = 13
```

在 C++里，编译器能保证在执行派生类的构造函数之前执行基类的构造函数。在我们的例子里，gp_list_node 的构造函数会首先执行，随后是 int_list_node 的构造函数。我们将在 10.3 节进一步讨论构造的问题。

除了定义新的数据和子程序成员之外，派生类还可以隐藏或者替换基类的成员，10.2 节将会讨论数据隐藏的问题。如果要替换基类里的一个方法，派生类里只需要简单地重新定义它。举个例子，假定我们正在创建一个 int_list_node 类，但希望其中 remove 方法的语义稍有不同。要是像图 10.1 那样写，当需要删除的结点不在表里时 gp_list_node::remove 就会抛出 list_err 异常。如果我们希望

int_list_node::remove 在这种情况下只简单返回而不做其他任何事情，那么就可以用下面方式显式地声明它：

```
class int_list_node : public gp_list_node {
public:
...
    void remove () {
        if (!singleton ()) {
            prev->next = next;
            next->prev = prev;
            prev = next = head_node = this;
        }
    }
};
```

这种重定义的方式也有缺点，因为它把 gp_list_node 的实现细节拉到 int_list_node 的一个方法里，有可能破坏了抽象（实际上 C++不接受上面这段代码。在第 10.2 节将会看到，如果想这样做，我们需要修改基类 gp_list_node 的定义，使成员 next 和 prev 变成派生类可见的）。另一种更好的方式是把有关实现细节留在基类里，只是在发生异常时捕捉它：

```
void int_list_node::remove () {
    try {
        gp_list_node::remove ();
    } catch (list_err) {
        ;   // do nothing
    }
}
```

这一版本的代码比前面那个稍微慢一点，具体情况依赖于 try 块的实现方式。但从维持抽象的角度看，现在的做法更好一些。请注意这里的作用域解析运算符（::），它使我们能显式访问基类的 remove 方法，虽然这个方法已经在 int_list_node 类里重新定义了。其他面向对象的语言也都以某种形式提供了这种访问方式。在 Smalltalk 和 Java 里，基类（超类）的成员可以通过关键字 super 访问：

```
gp_list_node::remove ();    // C++
super.remove ();            // Java
super remove.               // Smalltalk
```

在 Eiffel 里，程序员可以显式地**重命名**从基类继承的方法，使它们成为可访问的：

```
class int_list_node
inherit
    gp_list_node
        rename
            remove as old_remove
            ...     -- other renames
        end
```

这就使我们可以在 int_list_node 的方法里，用 old_remove 去访问 gp_list_node 里的删除方法了。我们将在 10.5 节中看到，C++和 Eiffel 不能采用关键字 supper，是因为在出现多重继承的情况下，这种描述是有歧义的。

在面向对象的程序设计里，保存着某给定类的一集对象的抽象通常称为**容器**。构造各种容器的方法有许多，本节展示的只是其中一种方法，采用从容器元素基类派生出对象的方式。这一途径存在一个基本问题：如果某对象不是从容器的元素基类派生出来的，那么我们就无法将它放入容器中。为了能把**任意**对象放入（例如）表里，我们可以换一种方式，把表结点定义为另一种对象，让这种对象里包含指向准备放入表里的对象的指针（或者引用），而不是在结点里直接保存这种对象的数据。第三种方式是把表结点作为要放入表里的对象的一个成员（子对象）。一般而言，设计出具有内在一致性的、直观而又有用的类层次结构是一项复杂而困难的艺术。容器问题不过只是冰山的一角。

10.2 封装和继承 Encapsulation and Inheritance

封装机制使程序员可以把数据和操作它们的子程序组织到一起，对有关抽象的用户隐藏起各种无关紧要的细节。在上面讨论里，我们把面向对象的程序设计说成是 Simula 和 Eiclid 的“模块作为类型”机制的一种扩充。也完全可以把面向对象程序设计纳入“模块作为管理器”的框架里。在下面第一小节，我们要考察一些非面向对象语言里的模块的数据隐藏机制，在第二小节里考察由于加入了继承，使模块变成了类而带来的新的数据隐藏问题。在第三小节里，我们将简单地考虑另一种替代方式，其中把继承加在记录上，同时仍通过（静态的）模块提供数据隐藏。

10.2.1 模块 Modules

有关数据隐藏的作用域规则，是 20 世纪 70 年代里 Clu、Modula、Euclid 和其他基于模块的程序设计语言里最重要的创新之一。在 Clu 和 Euclid 里，模块的声

明和定义（头部和体）总是一起出现，头部清晰说明了从模块里导出哪些名字。如果 Euclid 模块 *M* 导出了类型 *T*，按照默认规定，程序其他部分不能对类型 *T* 的对象做任何其他事情，只能把它们送给由模块 *M* 导出的子程序。*T* 被称为是**非透明**类型。如果需要，Euclid 程序员可以显式提供一些代码，使模块外面能执行按位赋值以及/或者对 *T* 类型的值的相等检测，或者访问 *T* 的域（如果是记录），通过下标访问（如果是数组），或者通过名字访问其值（如果是枚举）。举例说，有了下面声明，位于 `Database` 之外的代码就能做 `tuple` 之间的赋值，并可以访问其中的 `name` 域，但是不能检测它们是否相等，也不能访问其他域：

```
var Database : module
    exports (tuple with (:=, name))
    ...
    type tuple = record
        var name : packed array 1..80 of char
        ...
    end tuple
    ...
```

Clu 里的模块（在那里称为 `cluster`）实现了一个抽象类型，允许做这种类型的赋值和相等检测。但是，由于 Clu 采用的是变量的引用模型，因此这些赋值或者比较是对有关对象的引用做的，不是对于对象本身。

在 Modula-2 语言里，程序员可以把一个模块的头部和体分开。在第 3 章里我们只关注了所谓的“内部”模块，其中这两个部分被放在一起。对于“外部”模块（这也意味着分别编译）而言，头部应出现在一个源文件里，体将出现在另一个源文件里。不幸的是，在那里没有办法把头部分成公用部分和私用部分，放在头部的所有东西都是公用的（也就是导出的）。隐藏数据的唯一方式就是把类型做成非透明的，在头部只列出它的名字：

```
TYPE T;
```

在这种情况下，类型 T 的变量就只能赋值和比较相等，还可以送给这个模块的子程序。在 Modula-2 里没办法阻止赋值和比较。进一步说，由于编译器只看模块头部无法确定类型 T 的对象大小（用非形式说法），Modula-2 要求所有的非透明类型都是指针，因为指针的大小相同，可以在不了解其内部结构的情况下静态分配，或在堆栈上分配。（有些 Modula-2 实现允许其他非透明类型，条件是它们必须是用与指针一样多的二进制位实现的。）

Ada 也允许模块（称为 `package`，程序包）的头部和体分离，它消除了 Modula-2 的问题，采用的方法是允许把程序包的头部划分为公用部分和私用部分。要定义

一个可以非透明使用的类型，只要把它的具体定义放在头部的私用部分，公用部分只简单给出它的名字即可：

```
package foo is          -- header
    ...
    type T is private;
    ...
private        -- definitions below here are inaccessible to users
    ...
    type T is ...       -- full definition
    ...
end foo;
```

private 类型的变量可以赋值、比较相等，或者送给这个程序包的子程序。如果需要的话，Ada 程序员也可以禁止赋值和比较操作，为此应采用另一种更受限的形式：

```
type T is limited private;      -- replaces third line of example
```

静态的管理器风格的模块只影响名字的可见性，因此它们没有引出任何特殊的代码生成问题。在管理模块内部的变量和其他数据需要存储时，所用的方式正好和出现模块外的数据的存储完全一样。如果一个模块出现在全局作用域里，那么它的数据就可以静态分配。如果某模块出现在子程序里，那么它的数据就可以在堆栈上分配，放在已知偏移值的位置上，在子程序被调用时分配，在子程序返回时回收。

Euclid 里的模块类型更复杂一些，那里允许一个模块有任意数目的**实例**。明显的实现方式与记录类似。如果模块里的所有数据都具有静态已知的大小，那么每项数据都可以在模块的存储区中指定一个静态偏移值；如果某些数据的大小必须到运行时才能知道，那么就可以把模块的存储划分为固定大小的部分和变动大小的部分，把一个内情向量（描述字）放在固定大小部分的开始。模块的实例可以根据需要静态分配，或者在堆栈上分配，或者在堆中分配。位于模块内部的子程序也带来了一些麻烦，它们怎么知道应该用哪些变量？我们当然可以为每个模块实例复制这些子程序的一份代码，就像复制一份数据一样。但是这种重复复制太浪费了，因为不同拷贝之间的差异只在地址计算里的某些细节。更好的技术是只为每个模块子程序建立唯一实例，在运行中把适当的模块实例的存储地址传给它。这种地址的出现方式就像每个模块子程序有一个额外的隐藏的首参数。Eiclid 里的调用形式：

```
my_stack.push (x)
```

被翻译为就像是调用：

```
push (my_stack, x)
```

如果模块的头部和体出现在相互分离的不同文件里，只有体需要编译为可运行代码。在简单的实现里，编译器在编译模块 *M* 的体时读入 *M* 的头部，在编译任何使用 *M* 的模块时也读入 *M* 的头部。采用更复杂的实现技术可以设法避免这种重复扫描 *M* 头部，重复做语法分析和其他分析的额外开销。我们可以先把它编译到符号表里，在需要编译 *M* 的体或者编译使用它的模块时直接访问这个表。大部分 Ada 实现都把它们的模块头部编译好了。如第 3.6.2 节所述，所有的 C 和 C++实现都通过正文包含方式处理模块头部（*.h* 文件）。Modula-2 的实现方法多种多样，有些这样做，也有些那样做。

对模块体的修改绝不会要求重新编译模块的用户；对于模块头部中私用部分的修改可能要求我们重新编译模块的用户，但绝不会要求修改它们的代码。对模块头部中的公用部分的修改也就是对界面的修改，这时就要求修改用户代码了。

应该注意，虽然人们经常把模块与编译单位（进行分别编译的文件）联系在一起，实际上它们完全不必一一对应。举例来说，在 Java 语言里，每个分别编译的程序段里都要描述它所属的模块（在那里称为**程序包**），每个文件属于一个程序包，但这个程序包却可能跨越若干个文件。在一个程序包里，只有类可以在全局层次上声明；变量和子程序（方法）都只能在类里声明。除非把一个类显式地声明为 `public`，否则它就能且只能在这个程序包里看到。标准 C++包含一种 `namespace` 机制，提供了类似的功能。

10.2.2 类 Classes

随着继承的引入，面向对象的语言不但要支持基于模块的语言的作用域规则，还需要处理另外一些问题。举例说，基类里的私用成员对于派生类的方法应该是可见的吗？基类里的公用成员在派生类里也总是公用的吗（对于派生类的用户可见吗）？基类能对派生类的可见性做出哪些控制？

10.1 节的例子里忽略了这里的大部分问题。举例说，我们可能希望在 `queue` 里隐藏起 `append` 方法，因为它已经被 `enqueue` 取代了。为了在 C++里得到这种隐藏，可以在类 `queue` 的定义中把它的基类描述为 `private`：

```
class queue : private list {
public:
    using list::empty;
    using list::head;
    // but NOT using list::append
    void enqueue (gp_list_node* new_node);
    gp_list_node* dequeue ();
};
```

在这个声明的第一行出现了 private，说明 list 的公用成员只有再声明的随后部分描述为可见时，对 queue 的用户才是可见的。在上面声明里，我们通过 queue 公用部分的 using 声明使 empty 和 head 方法成为用户可见的方法。

除了 public 和 private 标号外，C++还允许把类成员指明为 protected（保护的）。保护成员只对本类的或者这个类的派生类里的方法可见。在我们的例子里，list 的保护成员 *M* 不仅可以由 list 本身的方法访问，也可以由 queue 的方法访问。与公用成员不同，*M* 对于 list 或者 queue 的任何用户都是不可见的。

关键字 protected 也可以用于描述基类：

```
class derived : protected base { ...
```

这样就使基类的公用成员都变成派生类的保护成员了。

在 C++的可见性规则背后的基本原则可以概述如下：

- 任何类都可以给它的成员的可见性增加限制。只要类声明在作用域里，其公用成员都是可见的。私用成员只在本类的方法里可见。保护成员在本类及其后裔的方法里可见。（作为正常规则的例外，一个类可以把某些特定的类或者子程序描述为自己的 friend，使它们能访问这个类的所有成员。）
- 派生类可以限制其基类成员的可见性，但却不能提升它们的可见性。基类的私用成员在派生类里根本不可见。共用基类的保护成员和公用成员，在派生类里仍然分别是保护的和公用的成员。保护基类里的保护和公用成员都将成为派生类里的保护成员。私用基类里的保护和公用成员都变成派生类的私用成员。
- 如果一个派生类通过把基类声明为 protected 或者 private 而限制了基类成员可见性，在这个派生类里还可以一个个地恢复基类成员的可见性，方法是在类声明的 protected 或者 public 部分加入适当的 using 声明。

其他面向对象语言对可见性采取了不同的方式。Eiffel 支持的可见性模式比 C++ 更灵活，但是它没有采纳 C++ 的上述第一条原则，那里的派生类既能限制也能提升基类成员的可见性。每个方法（在 Eiffel 里称为 feature，特征）可以描述其自身的**导出状态**。如果其状态是{NONE}，那么这个成员就是私用的（在 Eiffel 里称为**秘密的**）；如果状态是{ANY}那么该成员就是公用的（在 Eiffel 里称为**一般可用的**）。对于一般情况，状态可以是任意的类名字表，这时这一特征被说成是只

对这些类及其后裔**选择性可用的**。对于由基类继承来的任何特征，派生类都可以给它指定新状态。

Java 追随 C++有关 public、protected 和 private 声明的基本方式，但没有提供对基类的 protected 和 private 指定方式，这就使派生类既不能提升**也不能**限制基类成员的可见性。（当然，派生类总可以用一个方法重定义某个数据或子程序，让它在被使用时产生运行时错误。）Java 里 protected 关键字的意义与 C++ 里有少许不同。Java 类的 protected 成员不仅在派生类里是可见的，在本类所属的整个程序包里也都是可见的。

在 Smalltalk 里，可见性根本就不是一个问题。语言允许代码在运行中对任何对象试着调用任何方法名。如果该对象有给定名字的方法（而且带有数目正确的参数），那么这个调用就成功了；否则就将产生一个运行错误。在 Smalltalk 里，没有办法让一个方法只能为程序中某一部分使用，但却不能为另一部分使用。

10.2.3 类型扩展 Type Extensions

Smalltalk、Eiffel、C++和 Java 都是从一开始就作为面向对象的语言设计的，或者是从空白开始设计，或者是基于某种现存的没有强封装机制的语言出发。它们都支持以模块作为类型的抽象方式，其中都是用一种机制（类）提供了封装和继承。还有一些语言可以归类为已有语言的面向对象扩充，包括 Modula-3、Ada 95、Oberon 和 CLOS 等，其中的模块已经提供了封装。（Modula-3 和 Oberon 都不是 Modula-2 的严格扩充，但都大量吸取了其公共前驱的语法和语义。）这些语言并没有采取改动自己已有的模块机制的方式，而是通过一种扩充记录的机制提供继承和动态方法约束。举例来说，在 Ada 95 里，我们的表和队列抽象可以像图 10.3 那样定义。

为控制对类型的结构的访问，这里把它们隐藏在 Ada 程序包里。gp_list.queue 的过程 initialize、finalize、enqueue 和 dequeue 能把它们的参数从 self 转换到 list_ptr，因为 queue 是 list 的扩充。程序包 gp_list.queue 称为是包 gp_list 的子包，名字以父包的名字为前缀。Ada 95 里的子包类似于 Eiffel 或者 C++ 里的派生类，只是它仍然是管理器而不是类型。像 Eiffel 一样（与 C++ 不同），Ada 95 允许子包的体去看父包的私用部分。

图 10.3 里表和队列的所有子程序都用了一个显式的首参数，Ada 95、Oberon 和 CLOS 都没有采用“object.method()”的记法形式。Modula-3 确实用了这种形式，但只是作为一种语法包装：调用 A.B(C, D) 将被解释为调用 B(A, C, D)，其中的 B 被声明为三个参数的子程序。任意 Ada 代码都可以把 queue 类型的对象传给任何期望 list 的子程序。就像在 Java 里一样，这里的派生类型也无法隐藏基类型的公用成员。

```
package gp_list is
   list_err : exception;
   type gp_list_node is tagged private;
      -- 'tagged' means extendible; 'private' means opaque
   type gp_list_node_ptr is access all gp_list_node;
      -- 'all' means that this can point at 'aliased' nonheap data
   procedure initialize (self : access gp_list_node);
   procedure finalize (self : access gp_list_node);
   function predecessor (self : access gp_list_node) return gp_list_node_ptr;
   function successor (self : access gp_list_node) return gp_list_node_ptr;
   function singleton (self : access gp_list_node) return boolean;
   procedure insert_before (self : access gp_list_node; new_node : gp_list_node_ptr);
   procedure remove (self : access gp_list_node);

   type list is tagged private;
   type list_ptr is access all list;
   procedure initialize (self : access list);
   procedure finalize (self : access list);
   function empty (self : access list) return boolean;
   function head (self : access list) return gp_list_node_ptr;
   procedure append (self : access list; new_node : gp_list_node_ptr);
private
   type gp_list_node is tagged record
      prev, next, head_node : gp_list_node_ptr;
   end record;
   type list is tagged record
      header : aliased gp_list_node;
      -- 'aliased' means that an 'all' pointer can refer to this
   end record;
end gp_list;
...
package body gp_list is
   -- definitions of subroutines
   ...
end gp_list;
...
package gp_list.queue is     -- 'child' of gp_list
   type queue is new list with private
      -- 'new' means it's a subtype; 'with' means it's an extension
   procedure initialize (self : access queue);
   procedure finalize (self : access queue);
   procedure enqueue (self : access queue; new_node : gp_list_node_ptr);
   function dequeue (self : access queue) return gp_list_node_ptr;
private
   type queue is new list with null record;
      -- no new data members
end gp_list.queue;
```

图 10.3 Ada 95 里的表和队列。带标记类型的 **list** 和 **queue** 提供继承，程序包提供封装。**int_list_node** 可以按类似方式从 **gp_list_node** 派生出来。这里把 **self** 声明为具有类型 **access XXX**（而不是 **XXX_ptr**），使编译器能识别出这一子程序是该带标记类型里的一个方法（继续）。

```
package body gp_list.queue is
   procedure initialize (self : access queue) is
   begin
      initialize (list_ptr (self));
   end initialize;

   procedure finalize (self : access queue) is
   begin
      finalize (list_ptr (self));
   end finalize;

   procedure enqueue (self : access queue; new_node : gp_list_node_ptr) is
   begin
      append (list_ptr (self), new_node);
   end enqueue;

   function dequeue (self : access queue) return gp_list_node_ptr is
   rtn : gp_list_node_ptr;
   begin
      if empty (list_ptr (self)) then
         raise list_err;
      end if;
      rtn := head (list_ptr (self));
      remove (rtn);
      return rtn;
   end dequeue;
end gp_list.queue;
```

图 10.3（续）

10.3 初始化和终结处理 Initialization and Finalization

在 3.2 节，我们把对象的生存期定义为它占据空间并因此能保存数据的那段时间。大部分面向对象的语言都提供了某种特殊机制，在对象的生存期开始时能自动初始化它。采用 C++的术语，我们把这种机制称为**构造函数**。应强调指出，构造函数本身并不分配空间，它只是在已分配的空间里完成初始化工作。也有几个语言提供了类似的销毁函数（析构函数）机制，在对象生存期结束时自动去终结它。这样就出现了一些很重要的问题：

构造函数的选择：一个面向对象语言可能允许类有零个、一个或者多个构造函数。在最后一种情况下，不同构造函数可能有不同的名字，也可能只是通过参数个数或类型差异相互区分。

引用和值：如果变量是引用，那么所有对象都必须显式创建，这样就很容易保证能调用适当的构造函数。如果变量是值，那么对象的创建就必须隐式地作为声明加工的结果而发生。对后一种情况，语言或者必须允许对象以未初始化的形式开始其生存期，或者必须提供一种方式，使加工产生的每个对象也可以选择适当的构造函数。

执行顺序：在 C++里，要创建一个派生类的对象时，编译器保证它的每个基类的构造函数都在这个派生类的构造函数之前执行，最远的基类最先做。进一步说，如果一个类的某些成员本身也是某些类的对象，那么也会在执行包含它们的那个对象的构造函数之前，先执行这些成员的构造函数。这些规则带来了严重的语法和语义复杂性，因为在控制进入特定的作用域之前，许多构造函数、需要加工的对象和多重继承的组合，再加上还有重载解析问题，有时可能形成非常复杂的嵌套的构造函数调用序列。

废料收集：大部分面向对象语言都提供了某种构造函数机制，销毁函数方面的情况却有很大不同。在 C++一类语言里，析构函数的主要作用就是为了方便手工的存储释放。如果语言实现能自动地收集废料，那么对于销毁函数的需求也就大大降低了。

本节下面部分将考虑这些问题的更多细节。

10.3.1 构造函数的选择 Choosing a Constructor

在 Simula 语言里，类的初始化代码放在类定义的最后，也采用 `begin…end` 的形式。类的头部可以包含参数，每次调用 `new` 时都必须提供它们，初始化代码里可以访问这些参数。如果类 *B* 是从类 *A* 派生出来的，那么，创建类 *B* 的对象的调用就必须描述 *A* 和 *B* 的参数。按照默认方式，*A* 的初始化代码将被首先调用，而后是 *B* 的代码。如果需要的话，*A* 的代码也可以要求把 *B* 的代码执行**嵌入**在 *A* 的代码执行的中间：

```
CLASS A (I, J) INTEGER I, J;
...
BEGIN   COMMENT initialization code for A;
   initial;
   INNER;
   final;
END A;
...
```

```
A CLASS B (K) INTEGER K;
...
BEGIN   COMMENT initialization code for B;
   middle;
END B;
```

出现在这里的关键字 INNER 指明派生类的初始化代码的执行位置。要建立类 A 的对象，程序员需要写 myAobj :- NEW(E1, E2)，其中 E1 和 E2 分别是与 I 和 J 匹配的实参表达式。（Simula 的 :- 运算符表示做引用的赋值，而不是值的赋值。）NEW 将在执行完前面命名为 initial 和 final 的语句之后返回。创建类 B 的对象用 myBobj :- NEW(E1, E2, E3)，其中 E1、E2 和 E3 分别与 I、J 和 K 匹配。NEW 在按顺序执行完命名为 initial、middle 和 final 的语句之后返回。

采用 INNER 的理由与 Simula 支持协作程序（在 8.6 节讨论过）有很强的联系。如果 B 的代码里包含了一个 DETACH 语句，那么，位于 DETACH 之后的那个“初始化”部分就会在另一个协程里执行，这些执行可能与 B 的创建者的执行交错进行。换句话说，如果 B 是协程，那么上面标名 initial 的语句将在 B 开始其工作之前执行，而标名 final 的语句将在 B 结束其工作之后执行。

Smalltalk、Eiffel、C++和 Java 都允许程序员为一个类描述多个构造函数。在 C++和 Java 里，这些构造函数的行为方式就像是重载的子程序，它们必须通过参数个数或类型相互区分。在 Smalltalk 和 Eiffel 里，不同构造函数可以有不同的名字，创建对象的代码必须显式指明所用的构造函数名。在 Eiffel 里我们可以写：

```
class COMPLEX
creation
   new_cartesian, new_polar
feature {ANY}
   x, y : REAL;

   new_cartesian (x_val, y_val : REAL) is
   do
      x := x_val; y := y_val;
   end;

   new_polar (rho, theta : REAL) is
   do
      x := ro * cos (theta);
      y := ro * sin (theta)
   end;

   -- other public methods
```

```
feature {NONE}

   -- private methods

end -- class COMPLEX
...
a, b : COMPLEX;
...
!!b.new_cartesian (0, 1);
!!a.new_polar (pi/2, 1);
```

Eiffel 里的 `!!` 运算符相当于 `new`。由于类 `COMPLEX` 描述了几个构造函数（“创建函数”）方法，编译器要求对 `!!` 的每个使用都必须给定构造函数的名字和参数。这段代码无法在 C++里直截了当地实现，因为上面的两个构造函数都取两个 `real` 参数，这就意味着无法通过重载去区分它们。

在使用多个命名构造函数方面，Smalltalk 的情况与 Eiffel 类似，但在对对象的操作和对对象的类的操作方面，它们却截然不同。Smalltalk 采取一种拟物化的程序设计模型，其中的操作都被看作是由某个特定对象为响应来自其他对象的请求（“消息”）时执行的。这样，说对象 *O* 创建它自身就没什么意义，对象 *O* 必须由另外的某个代表着 *O* 的类的对象（例如称为 *C*）创建。当然，由于 *C* 本身也是对象，它也必须属于某个类。这一推理的结果就得到了一个系统，其中的每个类定义实际上引进了**一对**类，以及一对代表它们的对象。

以名字为 `Date` 的标准类为例。与 `Date` 相对应的有一个对象（现在称它为 *D*），它执行代表这个类的操作。特别的，正是由 *D* 创建类 `Date` 的对象，由于只有对象可以执行操作（而类则不行）。我们并不真正需要给 *D* 命名，完全可以直接用类名字代表它：

```
todaysDate <- Date today
```

这段代码导致 *D* 为 `Date` 执行构造函数 `today`，并把新创建的对象的引用赋给名为 `todayDate` 的变量。

那么 *D* 的类又是什么呢？显然它不会是 `Date`，因为 *D* **代表** `Date`。Smalltalk 说 *D* 是**元类** `Date class` 的对象（实际上是其唯一对象）。为了技术上的原因，现在还需要用某个对象代表 `Date class`。为了避免无穷回归，所有代表元类的对象都是一个名字为 `Metaclass` 的类的实例。

Modula-3 和 Oberon 都没有提供构造函数，因此程序员必须显式初始化程序里的每个东西。在 Ada 95 里，只有从标准库类型 `Controlled` 派生出的类的对象，才能使用构造函数和销毁函数（称为 `Initialize` 和 `Finalize` 例程）。

10.3.2 引用和值 References and Values

一些面向对象的语言采用让变量引用对象的程序设计模型，包括 Simula、Smalltalk 和 Java。其他语言允许变量有值，值是对象，这样的语言包括 C++、Modula-3、Ada 95 和 Oberon。Eiffel 的默认方式是引用模型，但也允许程序员将特定的类描述为 `expanded`（应该展开），此时这个类的变量就使用值模型。一般认为引用模型更优美一些，但这种模型要求在堆中分配所有的对象，并对每次访问强加了（如果没有编译器优化）一层额外的间接。一般说值模型的效率更高，但它也使初始化控制问题变得更加困难。

引用模型里的每个对象都是显式创建的，因此也就更容易保证调用适当的构造函数。如果采用值模型，创建变量对象就是声明加工的隐含结果。实际上，Modula-3、Ada 95 和 Oberon 都没有构造函数，加工产生的变量以未初始化的形式开始其生存期，因此就可能出现一个变量还没有值之前试图去使用它的情况。C++ 的编译器保证为每个被加工的变量调用适当的构造函数，但是它所用的标识构造函数及其参数的规则有时也使人感到困惑。

如果在声明 C++ 类 `foo` 的变量时没有给初始值，那么编译器就会去调用 `foo` 的无参构造函数（如果不存在这种构造函数但却有其他构造函数，那么这个声明就是一个静态语法错误——要求调用并不存在的子程序）：

```
foo b;                  // calls foo::foo ()
```

如果程序员希望调用其他构造函数，声明时就必须提供构造函数的参数，以驱动重载解析：

```
foo b (10, 'x');        // calls foo::foo (int, char)
```

最常见的参数表里只包含一个对象参数，它可能属于同一个类或者其他的类：

```
foo a;
bar b;
...
foo c (a);              // calls foo::foo (foo&)
foo d (b);              // calls foo::foo (bar&)
```

程序员的意图常常是想声明一个新对象，并要求其初始值与某个现存的对象“一样”。这时更自然的写法可能是：

```
foo a;                  // calls foo::foo ()
bar b;                  // calls bar::bar ()
...
foo c = a;              // calls foo::foo (foo&)
foo d = b;              // calls foo::foo (bar&)
```

由于认识到这种常见的意图，在 C++里把这种只有一个参数的构造函数称为**复制构造函数**。最重要的是应该认识到，在这个声明里的等于符号（=）表示的是初始化而不是赋值，其作用与下面的类似代码片段不同：

```
foo a, c, d;            // calls foo::foo () three times
bar b;                  // calls bar::bar ()
...
c = a;                  // calls foo::operator= (foo&)
d = b;                  // calls foo::operator= (bar&)
```

这里的是先用无参构造函数初始化 c 和 d，随之使用的等号是**赋值**而不是初始化。这一差异是在 C++里引起困惑的一个常见根源。由于变量的值模型和要求加工产生的对象必须由构造函数初始化，这两种要求的相互作用导致上述情况的出现。CLOS 要求把对象送给方法作为显式的首参数，对象创建和初始化靠名字为 make-instance 和 initialize-instance 子程序的重载版本完成。由于 CLOS 里统一使用引用模型，因此根本不会出现加工对象的初始化问题。

Eiffel 里的每个变量都用一个默认值初始化。内部类型（整数、浮点数、字符等等）都认为是 expanded，默认值都是 0。所有对象引用的默认值是 nil。对于 expanded 类类型的变量，默认值规则被递归地应用于各个成员。正如前面所说，创建新对象的方式是调用 Eiffel 的 !! 创建运算符：

```
!!var.creator (args);
```

这里的 var 是某个类类型 *T* 的变量，creator 是 *T* 的构造函数。一般情况下 var 是引用，创建运算符会为类 *T* 的对象分配空间，而后调用对象的构造函数。当 *T* 是一个 expanded 类类型时，也允许采用这种语法形式，只是此时的 var 是一个实际对象而不是引用。在这种情况下，!! 运算符就简单地把那个已经存在的对象送给构造函数。

10.3.3 执行顺序 Execution Order

正如我们已经看到的，C++强调每个对象都要在能用之前完成初始化。进一步说，如果该对象的类（现在称其为 *B*）是从另外某个类（称其为 *A*）派生的，C++强调要在调用 *B* 的构造函数之前调用 *A* 的构造函数，以保证派生类不会看到它所继承的数据成员处于不协调的状态。当程序员创建类 *B* 的对象时（无论是通过声明还是通过对 new 的调用），创建操作都会为 *B* 的构造函数描述所有的参数。这种参数使 C++编译器可以在出现多个构造函数的情况下完成解析工作。但是，

编译器从哪里得到类 *A* 的构造函数的参数呢？Simula 的方法是将它们加到创建的语法里，这显然违反了抽象的原则。C++采用的方式是允许在派生类构造函数的头部描述基类构造函数的参数：

```
foo::foo ( foo_params ) :  bar ( bar_args ) {
    ...
```

这里的 foo 由 bar 派生。表 foo_params 是 foo 的这个特定构造函数的形式参数。在参数表和子程序体的开括号之间，出现的是对基类 bar 构造函数的“调用”。提供给 bar 的构造函数的实际参数可以是任意复杂的涉及 foo 形参的表达式。编译器将会做出安排，在开始执行 foo 构造函数之前先执行 bar 的构造函数。

如果类的某些成员本身也是其他类的对象，C++程序员可以用类似的语法形式去描述对它们的构造函数的参数：

```
class foo :  bar {
    mem1_t member1;      // mem1_t and
    mem2_t member2;      // mem2_t are classes
    ...
}

foo::foo ( foo_params ) :  bar ( bar_args ), member1 ( mem1_args ),
        member2 ( mem2_args ) {
    ...
```

实际上我们在图 10.1 里也可以采用这种记法，在 list_node 的构造函数里初始化 prev、next、head_node 和 val：

```
list_node () : prev (this), next (this), head_node (this), val (0) {
    // empty body -- nothing else to do
```

这种替代形式确实很有价值，因为这是调用成员对象的复制构造函数，而不是先调用它们的默认（无参）构造函数，而后再用 operator=。对指针或者整数一类简单类型，两种做法生成的代码不会有显著差异。然而对那些有着复杂对象值的成员，显式构造函数在语义和效率上都可能与内部的赋值完全不同。

与 C++类似，Java 也强调基类的构造函数应该在派生类的构造函数之前调用，它采用的语法更简单，允许派生类构造函数代码起始行是对基类构造函数的“调用”：

```
super ( args );
```

如 10.1 节里所说的，Java 语言里的 super 用于引用当前类的基类。如果没有这种对 super 的调用，Java 编译器就会自动插入一个对基类的无参构造函数的调用（此

时就要求必须存在这种函数）。由于 Java 对于对象统一采用引用模型，任何本身是对象的类成员实际上也是**引用**，而不是“展开的”对象（按 Eiffel 的术语）。Java 把这种成员简单初始化为 `nil`。如果程序员不希望这样，那就必须在外围类的构造函数里显式调用 `new`。Smalltalk 和 Eiffel（在通常情况下）也采纳类似方式。如果一个 Eiffel 类里包含有 `expanded` 类类型的成员，那么就要求这个类类型提供一个无参构造函数。在创建这种外围类的对象时，Eiffel 编译器会安排好对这种构造函数的调用。

对于基类初始化问题，Smalltalk、Eiffel 和 CLOS 都比 C++更灵活一些。编译器或解释器会做出一些安排，在新建每个对象时自动去调用构造函数（创建函数，初始化函数），但是它们却不安排自动调用基类的构造函数，而只是将基类数据成员初始化为默认值（0 或者 `nil`）。如果派生类希望其他做法，其构造函数就必须显式调用基类的某个构造函数。

10.3.4 废料收集 Garbage Collection

当 C++对象被销毁时，首先会调用派生类的析构函数，而后按照与派生相反的顺序调用各基类的析构函数。在 C++里，析构函数的最常见用途就是完成手工的存储释放工作。举例说，假定我们要创建一个字符串名字的表或者队列：

```
class name_list_node : public gp_list_node {
   char *name;           // pointer to the data in a node
public:
   name_list_node () {
      name = 0;          // empty string
   }
   name_list_node (char *n) {
      name = new char[strlen(n)];
      strcpy (name, n);   // copy argument into member
   }
   ~name_list_node () {
      if (name != 0) {
         delete name;    // reclaim space
      }
   }
};
```

这个类的析构函数的作用就是释放构造函数在堆中分配的空间。Simula、Smalltalk、Eiffel、CLOS 和 Java 等语言里都提供了自动废料收集器，这些语言里对于析构函数的需要也就大大减弱了。事实上，析构的根本想法就是对带废料收集的语言的

质疑，因为在那些语言里，程序员几乎无法控制对象的销毁时间。Java 允许程序员声明一个 finalize 函数，废料收集器在回收对象前将自动被调用这个函数。但是这一特征并没有得到广泛使用。

10.4 动态方法约束 Dynamic Method Binding

继承/类型扩展的一个最主要结果就是，派生类 *D* 的对象有其基类 *C* 的所有（数据和子程序）成员。只要 *D* 不隐藏起 *C* 的任何公用可见成员（参见练习 10.7），把类 *D* 的对象用到任何期望类 *C* 对象的上下文里都有意义，这是因为所有希望对类 *C* 的成员做的事情，同样也可以去对类 *D* 的成员做。按照 Ada 的说法，凡是没有隐藏基类中任何公用可见成员的派生类，都是这个基类的**子类型**。

能在要求基类的上下文里使用其派生类，是多态性的一种形式。现在设想一个大学的计算机管理系统，我们可以从类 person 派生出类 student 和 professor：

```
class person { ...
class student : public person { ...
class professor : public person { ...
```

由于 student 对象和 professor 对象都具有 person 对象的所有性质，因此应该可以把它们用到要求 person 的上下文里：

```
student s;
professor p;
...
person *x = &s;
person *y = &p;
```

还有，下面这样的子程序：

```
void person::print_mailing_label () { ...
```

也将是多态的，因为它能接受多个不同的参数类型：

```
s.print_mailing_label ();   // i.e. print_mailing_label (s)
p.print_mailing_label ();   // i.e. print_mailing_label (p)
```

与其他多态性形式一样，我们在这里也依赖于一个事实：在自己的形式参数的所有特征中，print_mailing_label 只使用了所有可能的实际参数所共有的那一部分。

换一个问题，现在假定我们为两个派生类定义了各自的 print_mailing_label，希望能把某些特殊信息（例如学生在学校里的年级，或者教授所属的系）打印在邮件标签的角上。现在有了这个子程序的多个版本：student::print_mailing_label 和 professor:: print_mailing_label 等，而不是只有 person::print_mailing_label 一个版本。具体取得那个版本将依赖于对象：

```
s.print_mailing_label ();   // student::print_mailing_label (s)
```

```
p.print_mailing_label ();   // professor::print_mailing_label (p)
```

但下面这样写又会怎么样？

```
x->print_mailing_label ();   // ??
y->print_mailing_label ();   // ??
```

这时究竟是根据**变量** x 和 y 的类型选出需要调用的方法，还是根据被这些变量引用的**对象** s 和 p 的类去选择？

第一种方式（根据引用的类型选择）称为**静态方法约束**，而第二种方式（根据对象的类选择）称为**动态方法约束**。动态方法约束是面向对象语言的核心概念。举例说，设想我们的计算机管理程序已经创建了一个 person 的表，其中每个人都有逾期的图书。这个表里可能包括一些 student 和一些 professor。如果现在要遍历这个表打印每个人的邮件标签，动态方法约束保证对每个人都能调用正确的打印过程。在这种情况下，我们就说派生类里的定义**覆盖**了基类里的定义。

可惜的是，正如我们将在 10.4.3 节里看到的，动态方法约束会带来运行时的额外开销。虽然一般来说这种额外开销并不大，但对那些性能要求非常高的应用里的小子程序而言，这种情况可能还是很值得注意的。Smalltalk 和 Modula-3 对所有方法都采用动态方法约束；Java 和 Eiffel 以动态方法约束作为默认方式，但允许把方法标明为 final（Java）或者 frozen（Eiffel），此时这种方法就不会被派生类覆盖，可以采用优化实现方式。Simula、C++和 Ada 95 以静态方法约束作为默认方式，但允许程序员在需要时要求采用动态约束。后面这几种语言里通常需要区分两种不同术语：采用动态约束时是覆盖一个方法，而采用静态约束时（仅仅是）重新定义一个方法。

10.4.1 虚函数和非虚函数 Virtual and Nonvirtual Methods

在Simula和C++里，程序员可以把一个函数描述为virtual（虚的）。对虚函数的调用将在运行时根据对象的类（而不是引用的类型）分派到适当实现。在C++里，关键字virtual放在子程序声明的最前面[3]：

```
class person {
public:
    virtual void print_mailing_label ();
    ...
```

[3] 在C++里，在一定的环境里还能把virtual关键字放在基类名字之前，用在派生类声明的头部。但这样的使用是为了另一种完全不同的目的，10.5.3 节将讨论它。

在 Simula 里，虚函数列在类声明的最前面：

```
CLASS Person;
   VIRTUAL: PROCEDURE PrintMailingLabel;
BEGIN
   ...
   PROCEDURE PrintMailingLabel...
      COMMENT body of subroutine
   ...
END Person;
```

Ada 95 采用另一种方式。Ada 95 程序员不是把动态分派与特定方法关联，而是把动态分派与特定的**引用**相关联。在下面例子里，形式参数或者 access 变量（指针）可以声明为具有**泛类**类型 person'Class，在这种情况下，对这个参数或者变量的所有方法调用都将基于它们所引用的对象进行分派：

```
type person is tagged record ...
type student is new person with ...
type professor is new person with ...

procedure print_mailing_label (r : person) is ...
procedure print_mailing_label (s : student) is ...
procedure print_mailing_label (p : professor) is ...

procedure print_appropriate_label (r : person'Class) is
begin
   print_mailing_label (r);
   -- calls appropriate overloaded version, depending
   -- on type of r at run time
end print_appropriate_label;
```

反对采用静态方法约束，进而提出基于被引用对象的类型进行动态约束，最主要的理由就是静态方式会阻止派生类对自己的状态一致性的控制。举例说，假定我们要做一个 I/O 库，它包含了一个 text_file 类：

```
class text_file {
   char *name;
   long position;            // file pointer
public:
   void seek (long whence);    // virtual?
   ...
}
```

现在假定我们有派生类 read_ahead_text_file：

```
class read_ahead_text_file : public text_file {
   char *upcoming_characters;
public:
   void seek (long whence);    // redefinition
   ...
}
```

毫无疑问，对于 read_ahead_text_file::seek 的代码需要去修改缓存 upcoming_character 的值。如果这个方法不是 virtual 的，我们就没有办法保证这件事一定发生。因为，如果把一个 read_ahead_text_file 引用送给一个要求 text_file 引用的子程序作为参数，如果那个子程序里调用了 seek，那么得到的一定是基类的 seek 版本。

10.4.2 抽象类 Abstract Classes

在大部分面向对象的语言里，基类都可以不给出 virtual 方法的体。在 C++ 里，做这件事时需要采用下面这种“赋值”0 的子程序声明形式：

```
class person {
    ...
public:
    virtual void print_mailing_label () = 0;
    ...
```

没有体的虚函数称为**抽象**函数，C++ 称为**纯**虚函数。Simula 里的所有虚函数都是抽象的。

一个类被称为抽象类，如果它包含了至少一个抽象函数。我们不可能声明抽象类的对象，因为在这种类里至少有一个成员没有定义。抽象类的用途就是作为其他**具体**类的基类。具体类（或者它的某个中间的先驱类）必须为继承来的每个抽象方法提供实际定义。抽象类的每个抽象方法为动态方法约束提供了一个“挂钩”，使程序员写的代码可以通过这个基类的对象（引用）去调用方法，并知道在运行时一定能调用合适的方法。除了抽象方法之外什么也没有（没有任何数据成员和方法体）的类，在Java里称为接口（*interface*）。这种机制支持一种受限的，“混入式”的多重继承形式。我们将在 10.5.4 节考察它[4]。

10.4.3 成员查找 Member Lookup

对于静态方法约束（例如在 Simula、C++和 Ada 95 里），编译器总可以基于所用变量的类型确定应该调用的方法版本。对于动态方法约束，在被引用或指针

[4] 在Eiffel里的抽象虚方法称为缓期特征（注意，Eiffel里的特征都是虚的），抽象类称为缓期类，具体类则称为有效类。在Java中称为接口的东西，在Eiffel里称为完全缓期的类。

变量所引用的对象里就必须包含足够的信息，使编译器生成的代码能在运行时找到正确的方法版本。最常见的实现方式是用记录的形式表示对象，记录里的第一个域是一个指针，指向这个对象的类的**虚函数表**（**虚表**，*vtable*，见图 10.4）。虚表就是一个数组，其中的第 i 个项指明该对象的第 i 个虚函数的代码的地址。一个给定类的所有对象共享同一个虚表。

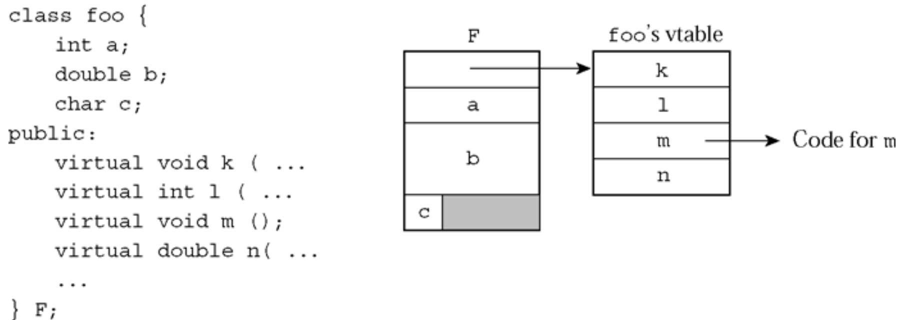

图 10.4　虚函数的实现。在对象 `F` 表示的开始处是类 `foo` 的虚表的地址（该类的所有对象都指向同一个虚表）。虚表本身是一个地址的数组，其中每项指向类里一个虚方法的代码。F 记录里的其余各项是它的数据成员。

假定送给方法的 `this`（或 `self`）指针在寄存器 `r1` 里传递，`m` 是类 `foo` 的第三个虚函数，而 `f` 是指向 `foo` 类的对象的指针，调用 `f->m()` 代码大致是下面的样子：

```
r1 := f
r2 := *r1                    -- 虚表地址
r2 := *(r2 + (3−1) × 4)      -- 假定 sizeof(地址) = 4
call *r2
```

在典型的 RISC 机器里，这一调用序列比静态识别的方法调用多用两条指令（这两条指令都需要访问内存）。如果编译器能在编译时推断出有关对象的类型，那么就可以避免这种额外开销。对于以对象为值的变量（不是引用或者指针），这种推断是非常简单的。

如果类 `bar` 由 `foo` 派生，它所加入的信息就会放在表示它的“记录”的后部。需要为 `bar` 建立一个虚表，采用的方法是首先复制 `foo` 的虚表，而后把类 `bar` 覆盖的所有虚方法项目代换进去，再把 `bar` 中声明的方法附在后面（图 10.5）。如果现在有一个类 `bar` 的对象，我们就可以很安全地把它的地址赋给类型为 `foo*` 的变量了：

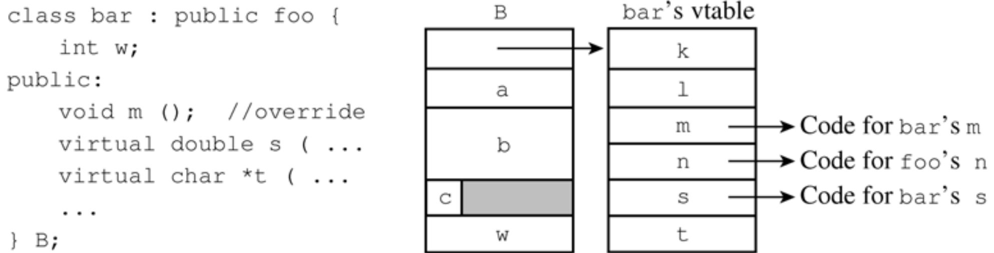

图 10.5　单继承的实现。与图 10.4 中一样，对象 **B** 的表示开始处也是它的类的虚表地址。这个表里的前四项是与 **foo** 的虚表里一样的成员，除其中的 **m** 已被覆盖，变成另一个不同子程序的代码地址。**bar** 新增的数据成员存放在 **B** 继承自 **foo** 的成员之后；在类 **bar** 的虚表里新增的虚方法也存放在由 **foo** 继承来的方法之后。

```
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
```

在 C++里（除 Smalltalk 和 CLOS 之外，我们所考虑的其他面向对象语言里也都是这样），编译器能静态检验这段代码的类型正确性。它当然无法知道 q 在运行时引用的是哪个类的对象，但却知道该对象必定或者属于 foo，或者属于 foo 的某个（直接或间接）派生类。这也就保证了该对象里有 foo 的特定代码可能访问的所有成员。

标准 C++允许通过 dynamic_cast 运算符“反向”赋值：

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

由于向后兼容性，C++也允许把 C 风格的强制用于对象的指针或引用：

```
s = (bar*) q;   // permitted, but risky
```

如果采用 C 风格强制，那就要求程序员保证所涉及的实际对象确实有合适的类型，执行时不做任何运行时检查。Java 允许采用传统的强制写法，但是却会执行动态检查。Eiffel 有一个**反向赋值**运算符 ?=，当且仅当运行时的类型可以接受时，才会把对象赋给变量：

```
class foo ...
class bar inherit foo ...
...
f : foo;
b : bar;
...
f := b;     -- always ok
b ?= f;     -- reverse assignment: b gets f if f refers to a bar object
          -- at run time; otherwise b gets nil
```

为了支持对反向赋值的动态检查，Eiffel、Java 和 C++实现都需要在虚表里包含一个运行时的**类型描述字**。在 C++里只能对多态类型的指针和引用使用 dynamic_cast，因为非多态类型的对象根本就没有虚表。

Smalltalk 变量是**无类型**的引用，可以把任何对象的引用赋给任何变量作为值。只有在运行中代码实际企图调用某个操作（送去一个“消息”）时，语言实现才去检查有关对象是否支持这一操作。实现是直截了当的：对象的数据成员都不是公用的，方法是与对象交互的唯一方式。对象表示的开始处放着类型描述字的地址，描述字里包含一个将方法名映射到代码段的字典。运行中 Smalltalk 解释器在字典里执行一次查找，看所需方法是否得到支持。如果没有就生成一个“message not understood”错误，这是与 Lisp 里类型崩溃错误类似的东西。CLOS 提供的语义与此类似，也采用了类似实现方式。有人说 Smalltalk/CLOS 的方式比更静态类型的语言灵活一些，但是也会引起显著的运行开销，而且推迟了报告错误的时间。

除了增加间接操作的开销之外，虚方法还使我们没办法在编译时做子程序的在线展开。对于那些很小而又频繁调用的子程序，无法在线展开可能引出严重的性能问题。与 C 一样，C++也希望尽可能避免额外的运行时开销，因此它采用静态方法约束作为默认方式，而且尽可能依靠对象值的变量。对于这种变量，即使是虚方法也可以在编译时完成分派工作。

但是在另一方面，在某些特定的环境中，我们也可能希望运行时再做方法查找，即使语言允许做编译时查找工作。举例说，Java 程序通常以一种可移植的“字节码”形式发布，这种形式可以解释执行，某些实现在执行前先进行编译。标准的字节码“虚拟机”解释器在运行时查找有关的方法，这样做可以避免众所周知的**脆弱基类**问题。Java 实现依赖于一个很大的标准库，这个库会随着时间推移而不断演化。虽然可以假定库的设计者在努力保证最大限度的向后兼容性——极少

删除一个类里的任何成员。但是老版本库的用户也可能偶尔需要去运行一些代码，而这些代码是按照库的新版本的要求写出来的。在这种情况下，依赖于库类表示方式的静态假设就会带来大灾难：某些代码试图使用新加入的库特征，它们的访问可能会做超出可用表示的结束位置。如果采用的是运行时的方法查找，此时就会产生一个有用的“在你所用的库版本里没有找到所需成员”的动态错误消息。

10.4.4 相关概念 Related Concepts

面向对象语言里的动态方法约束与本书中已经考察过的几个概念有关系，包括多态性、通用型和一级子程序。

我们早已注意到，动态方法约束把多态性引进了期望某个基类 foo 的引用的所有代码里。只要派生类的对象支持这个基类的所有操作，这种代码就能完全同等地对 foo 的任何派生类的对象工作。例如，在声明一个类 foo 的引用参数时，程序员实际上说的是，这个子程序里只使用参数的“foo 特征”，能对任何具有这种特征的对象工作。

有人可能会认为，有了继承后就再也不需要通用型了，实际情况并非如此。我们在 10.1 节看了 gp_list_node 及其后继的例子，通过替换基类抽象（这里是表）的一些结构侧面，很容易得到特定类型的表：int_list_node、float_list_node、student_list_node 等等。可惜的是，predecessor 和 successor 等方法返回的是基类型的引用，这种引用不能支持特定类型的操作。为访问由这些表操作子程序返回的对象的内容，就必须执行显式的类型强制：

```
int_list_node_ptr q, r;
...
r = q->successor ();        // error: type clash
gp_list_node_ptr p = q->successor ();
cout << p.val;              // error: gp_list_nodes have no val
r = (int_list_node_ptr) q->successor ();
cout << r.val;              // ok
```

代码里的第 5 行既不雅致也不安全。我们没办法在这里用 dynamic_cast，因为类 gp_list_node 里没有虚成员，因此也就没有虚表。限制这种不雅之处的方法是在 int_list_node 的定义里重新定义成员函数：

```
int_list_node* int_list_node::predecessor () {      // redefine
   return (int_list_node*) gp_list_node::predecessor ();
}
```

```
int_list_node* int_list_node::successor () {       // redefine
   return (int_list_node*) gp_list_node::successor ();
}
```

不幸的是，在每个派生类里重新定义所有具有这类参数和返回值类型的基类方法，同样是一项令人生厌的烦琐工作，而且得到的代码仍然是不安全的，因为编译器不可能检验其类型正确性。通用型技术可以避免所有这些问题。在 C++里我们可以写：

```
template<class V>
class list_node {
   list_node<V>* prev;
   list_node<V>* next;
   list_node<V>* head_node;
public:
   V val;
   list_node<V>* predecessor () { ...
   list_node<V>* successor () { ...
   void insert_before (list_node<V>* new_node) { ...
   ...
};

template<class V>
class list {
   list_node<V> header;
public:
   list_node<V>* head () { ...
   void append (list_node<V> *new_node) { ...
   ...
};

typedef list_node<int> int_list_node;
typedef list<int> int_list;
...
int_list numbers;
int_list_node* first_int;
...
first_int = numbers->head ();
```

简言之，通用型的存在就是为建立起某些跨类型的，继承性不能支持的抽象。（注意：在 ML 以及与之相关的语言里有类型推理系统，它**已经**足够处理跨类型抽象问题。因此 ML 不需要通用型。在另一方面，ML 提供了类似 Euclid 的模块类型，但却没有提供继承，因此不能认为它是一个面向对象的语言。）

Eiffel 提供了与 C++类似的通用型功能。作为方便使用的简写形式，它允许程序员把方法的参数和返回值声明为“锚固”到与类的某些成员同样的类型上。这样，如果在一个派生类里重新定义了某个锚，相应的参数和返回值类型也就自动重新定义了，完全不需要显式地重新描述：

```
class gp_list_node ...
...
class gp_list
feature {NONE}            -- private
   header : gp_list_node   -- to be redefined by derived classes
feature {ALL}             -- public
   head : like header is ...                -- methods
   append (new_node : like header) is ...
   ...
end
...
class student_list_node inherit gp_list_node ...
...
class student_list
   inherit gp_list
   redefine header end
feature {NONE}
   header : student_list_node
   -- don't need to redefine head and append
end
```

虽然这种 `like` 机制也没有消除对通用型的需要，但却使我们更容易定义它们，或者在一些简单情况里不去使用它。

由于虚函数的分派是推迟到运行时才进行的，动态方法约束提供了一种类似于一级子程序（参见 3.4 和 8.3.1 节）的机制。在 C++里，类似下面形式的代码：

```
typedef void (*F_INT) (int);
   // F_INT is a type: pointer to function from int to void
void p (int a) {
   ...
}
void q (double b, F_INT f) {
   ...
   f (3);
   ...
}
q (3.14, &p);
```

或多或少可以用下面形式的代码取代：

```
class foo {
public:
   virtual void f (int a) = 0;
};
void q (double b, foo& obj) {
   ...
   obj.f (3);
   ...
}
```

```
class bar : public foo {
public:
    virtual void f (int a) {
        ...
    }
} my_obj;
q (3.14, my_obj);
```

与 C++里指向子程序的指针版本相比，后面这种面向对象的代码版本具有一个重要的优势：通过在类 bar（和对象 my_obj）里增加数据成员，我们可以给方法 f 提供操作的数据。从效果上看，带有虚方法的对象的数据成员，在行为上很像是在允许嵌套作用域的语言里的闭包所表示的引用环境（请注意，C 和 C++里都不需要闭包，因为它们没有嵌套的子程序作用域）。在 Ada 95 里同时有嵌套的子程序作用域和类（带标志类型），因此虚表项本身就必须是闭包，不能只是简单的子程序地址。

在一些情况下，虚方法和一级子程序也可以互为补充。举个例子，假定我们正在写一个离散事件模拟系统，如 8.6.4 节所描述的那样。其中可能需要有一种一般性机制，使我们能调度任意的（带着任意参数集合的）子程序调用，使之在未来某个时刻执行。如果希望调用的子程序在参数个数和类型方面都可能变化，那么就没办法把它们直接传给一个一般性的 schedule_at 例行程序。虚方法可以解决这种问题，如图 10.6 所示。如我们将在 12.2.3 节里看到的，这一技术也被用在 Modula-3 里，用于封装新创建的控制线程的启动参数。

10.5* 多重继承 Multiple Inheritance

有些时候，让一个派生类继承多个基类的特征也是很有用的。举例说，假定我们希望写出的计算机管理系统能在某个表里保存所有同一年级（一年级、二年级、三年级、四年级、未注册生）的学生记录，那么就可能希望从 person 和 gp_list_node 两个类派生出一个类。在 C++里可以写成：

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

现在，类 student 的对象将具有 person 和 gp_list_node 两个类的所有数据和方法成员。Eiffel 里的声明形式与此类似：

```
class student
inherit
    person;
    gp_list_node
feature
    ...
```

```
class fn_call {
public:
   virtual void trigger () = 0;
};
void schedule_at (fn_call& fc, time t) {
   ...
}
...
void foo (int a, double b, char c) {
   ...
}
class call_foo : public fn_call {
   int arg1;
   double arg2;
   char arg3;
   void (*ptr) (int, double, char);
public:
   call_foo (int a, double b, char c) :    // constructor
      arg1 (a), arg2 (b), arg3 (c) {
      // member initialization is all that is required
   }
   void trigger () {
      foo (arg1, arg2, arg3);
   }
};
...
call_foo cf (3, 3.14, 'x');         // declaration/constructor call
schedule_at (cf, now () + delay);
   // at some point in the future, the discrete event system
   // will call cf.trigger (), which will cause a call to
   // foo (3, 3.14, 'x')
```

图 10.6　子程序指针和虚方法。类 `call_foo` 封装了一个子程序指针和传递给它的值，导出一个无参过程。这一过程可用于触发被封装的子程序调用。

CLOS 里也有多重继承，而 Simula、Smalltalk、Modula-3、Ada 95 和 Oberon 里都只有单继承。Java 提供了一种受限的、“混入式”的多重继承方式，10.5.4 节将进一步讨论它。

为了实现多重继承，我们必须能根据要求，对 `student` 对象产生出一个“`person` 的观察点”和一个“`gp_list_node` 的观察点”，例如在把 `student` 对象引用赋给 `person` 或者 `gp_list_node` 变量时就需要这样做。对其中的某个基类（譬如说 `person`），我们可以采用与单继承一样的做法：把该基类的数据成员都放在派生类表示的开始位置，把该类的虚方法都放在相应虚表的开始部分。这样，当我们需要把一个 `student` 引用赋给一个 `person` 变量时，操作这一 `person` 变量的代码使用的正好就是数据成员和虚表的前面部分。

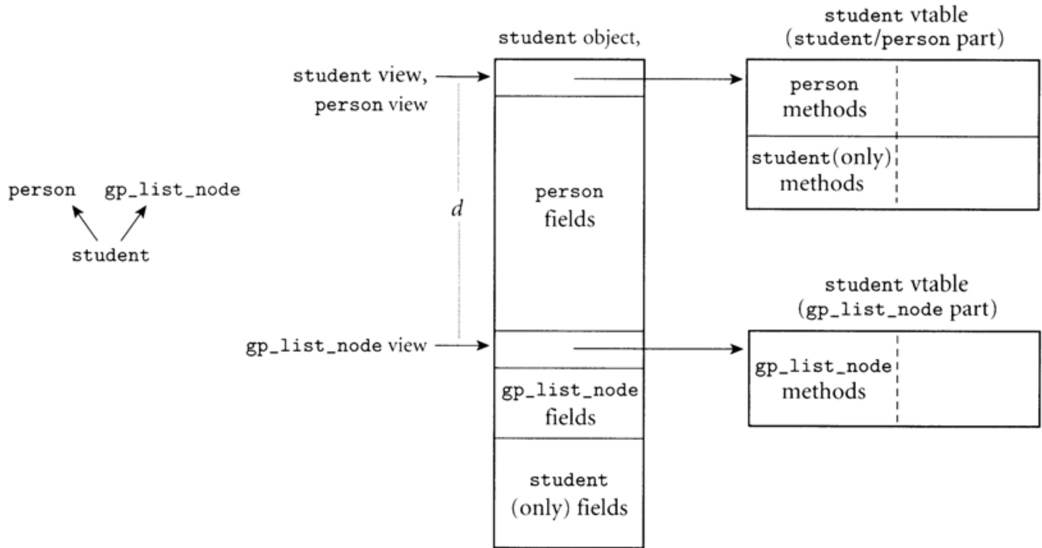

图 10.7　多重继承的（非重复性）实现。对象中 **person** 部分的大小 *d* 是编译时确定的常数。在访问虚表的 **gp_list_node** 部分时，需要在间接之前给 **student** 对象的地址加上 *d*。类似的，如果要建立 **student** 对象的 **gp_list_node** 观点，那么就要为对象的地址加上 *d*。如正文中所述，每个虚表项都包含一个方法地址和一个“**this** 校正值”，该值等于虚表访问的观察点和方法定义所在的类的观察点之间的距离（带正负号的整数）。

对于另一个基类（这里是 gp_list_node），事情就更麻烦一些了：我们不可能把两个基类都放到派生类的开始位置。一种可能解决方案如图 10.7 所示，这差不多就是在 Ellis 和 Stroustrup 在 [ES90，第 10 章] 里描述的实现方式。由于 gp_list_node 的域位于 person 的域之后，当我们需要把一个 student 引用赋给类型为 gp_list_node* 的变量时，就必须调整自己的“观察点”，加上一个编译时确定的常数偏移值 *d*。

student 的虚表现在分成了两部分。第一个部分里列出派生类和第一个基类（person）的虚方法，第二个部分里是第二个基类的虚方法。（我们已经在类 person 里引进了一个方法 print_mailing_lable。可以设想在 gp_list_node 里定义虚方法 debug_print，假定它把结点的内容以可打印形式送到标准输出。）把这种技术推广到三个或者更多基类也是直截了当的，参见练习 10.13。

student 对象的每个数据成员有一个相对于对象开始位置的偏移值，是编译时常量。与此类似，每个虚方法有相对于虚表的两部分之一开始位置的偏移值，也是编译时常量。虚表中 person/student 部分的地址保存在对象开始位置，虚表 gp_list_node 部分的地址保存在偏移值 *d*。注意，虚表的这两个部分都是 student

特定的东西，其中的 gp_list_node 部分也不与类 gp_list_node 的对象共享，因为 student 有可能覆盖 gp_list_node 的一些虚方法，那时这个表的内容就与原虚表不同了。

如果要调用 person 原来定义的 print_mailing_lable 虚方法，我们可以用 10.4.3 节所示的用于单继承的类似代码序列。在调用原来在 gp_list_node 里定义的虚方法时，就必须首先在对象地址上加偏移值 *d*，得到虚表 gp_list_node 部分的地址。而后再用下标在这个 gp_list_node 的虚表里找出要调用的方法的地址。但是现在还有一个问题没有解决：送给这一方法的 this 指针值应该是什么呢？

作为一个具体例子，现在假定 student 并没有覆盖 debug_print（虽然它可能应该做这件事）。如果当前对象属于类 student，我们就需要把它的 gp_list_node 观察点送给 debug_print，也就是为该对象的地址加上 *d*。而如果当前对象属于某个确实覆盖了 debug_print 的类（例如 transfer_student）的对象，那就应该把该对象的 transfer_student 观察点送给 debug_print。如果现在通过变量（引用或指针）访问对象，其方法是动态约束的，那么编译时就无法确定究竟应该采用哪种方式。实际情况更糟糕：我们甚至无法知晓如何产生出 transfer_student 的观察点（如果必需这样做），因为在编译这部分代码时，transfer_student 类可能还没有做出来，因此不可能知道它里面的 gp_list_node 域出现在什么地方！

一种普遍有效的解决方案是让虚表的每项由**一对**域组成，一个域是方法代码的地址，另一个是需要加在查找虚表的观察点上的“this 校正值”。现在回到图 10.7，如果 debug_print 被 student 所覆盖，与 debug_print 对应的虚表项里的“this 校正值”就应该是 –*d*；如果没有覆盖，这个校正值就是 0。在（现在还没有写出的）类 transfer_student 的虚表的 gp_list_node 部分里，这一“this 校正值”域可以包含另一个适当的值 –*e*。一般而言，“this 校正值”是一个带正负号的距离，是方法**声明**所在类的观察点（通过它访问虚表）和方法**定义**所在类的观察点（这是子程序的实现所期望的）之差。

如果运行时变量 my_student 包含对某对象（的 student 观察点）的引用，假定 debug_print 是类 gp_list_node 里的第三个虚方法，调用 my_student.debug_print 的代码大致具有下面形式：

```
r1 := my_student                 — 对象的 student 观察点
r1 := r1 + d                     — 对象的 gp_list_node 观察点
r2 := *r1                        — 相应虚表地址
r3 := *(r2 + (3−1) × 8)          — 方法地址
r2 := *(r2 + (3−1) × 8 + 4)      — this 校正
r1 := r1 + r2                    — shis
call *r3
```

在这段代码里，我们假定方法地址和 this 校正值的长度都为 4 个字节。在典型的计算机里，这段代码比实现单继承的代码多 3 条指令（包括 1 次内存访问），比静态识别的方法调用多 5 条指令（包括 3 次内存访问）。不幸的是，存在多重继承的语言还会产生更多的额外开销：即使程序里实际只用了单继承，我们也必须按多重继承的方式去做。练习 10.18 研究了另一些实现技术，其中采用可执行的代码表示调用虚方法的某些信息，而不是用数据结构表示。

10.5.1 语义歧义性 Semantic Ambiguities

多重继承除了给实现带来一些复杂性外（我们已经讨论的只是其中一部分），还会带来潜在的语义问题。现在假定 gp_list_node 和 person 里都定义了 debug_print 方法，如果我们有一个 student* 类型的变量 s，并想调用 s->debug_print()，得到的究竟是哪个方法呢？在 CLOS 里，这样得到的是出现在派生类头部的第一个基类里的版本。如果试图在 Eiffel 里定义有这种歧义性的派生类，我们将得到一个静态语义错误。在 C++里可以定义这种派生类，但试图使用名字有歧义性的成员时会得到一个静态语义错误。在 Eiffel 里定义派生类时，可以通过重命名机制摆脱名字冲突的困扰。在 C++里就必须显式重新定义有歧义性的成员：

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

这里写的是希望调用两个基类的 debug_print,其中用作用域解析运算符 :: 区分它们。我们也完全可以只调用其中之一，或者从空白开始另写自己的代码，还可以通过给基类子程序新名字的方式安排好对两者的访问：

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

如果两个基类里名字相同的方法之一是虚的，或者两个都是虚的，而我们又希望在派生类里覆盖它们，事情就更麻烦一些。按照 Stroustrup 的提议 [Str97，25.6 节]，我们可以通过在每个基类与派生类之间插入一个“接口类”的方式解决这种问题：

```
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 () { ...
    ...
};
```

如果把一个 student 对象赋给类型为 person* 的变量 p，而后调用 p->debug_print()，此时会发生什么事情？我们把对这一问题留作练习（练习 10.14）。

还有另一种更严重的歧义性情况，它出现在类 *D* 继承两个基类 *B* 和 *C*，而这两个类又都继承了某个公共基类 *A* 的时候。在这种情况下，类 *D* 的对象里应该包含类 *A* 的一个实例呢，还是应该包含两个？回答看起来要依赖于具体程序。例如，假定在前面讨论的计算机管理系统里，我们希望把同一个系的所有教授记录保存在一个链接表里。就像前面的 student 那样，我们也可能希望类 professor 从 person 和 gp_list_node 继承：

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

进一步假定教授偶尔也会作为未注册的学生去听课。在这种情况下，我们就可能希望有一个新类，它能同时支持教授和学生的两组操作：

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

这就使类 student_prof 继承了 person 和 gp_list_node 各两次，分别通过 student 和 professor。经过认真考虑，我们可能希望 student_prof 里只有类 person 数据成员的**一个**实例——一个名字，一个大学 ID 号，一个邮件地址；而应该有类 gp_list_node 的数据成员的**两个**实例——两个前驱结点和两个后继结点，一对用于链接到未注册学生的链表里，另一对用于链接到某个系的教工链表里：

类 gp_list_node 的情况中需要为继承树的不同分支分别提供副本，被称为**复本式继承**；类 person 的情况中树的不同分支只有一个副本，称为**共享式**继承；两者都是**重复继承**的具体形式。复本式继承是 C++里的默认方式，而共享式继承是 Eiffel 里的默认方式。在 C++里，可以通过将基类描述为 virtual 而得到共享式继承：

```
class student : public virtual person, public gp_list_node { ...
class professor : public virtual person, public gp_list_node { ...
```

如果在这种情况下再出现多条继承路径，类 person 的成员将被共享，而类 gp_list_node 的成员将建立复本。在 Eiffel 语言里，可以通过 10.2.2 节介绍过的重命名机制一个个地取得某些特征的复本式继承：

```
class student inherit person; gp_list_node ...
class professor inherit person; gp_list_node ...

class student_prof
inherit
    student
        rename
            prev as prev_student,
            next as next_student
        end;
    professor
        rename
            prev as prev_prof,
            next as next_prof
        end
feature
    ...
end -- class student_prof
```

最终通过不同名字继承的特征将建立不同复本，最终由同一名字继承的特征则共享。在 CLOS 里，多重继承总是共享的，除非显式插入如前所述的接口类，不存在其他重命名机制。

10.5.2 复本式继承 Replicated Inheritance

除了 10.5 节开头描述的实现问题外，复本式继承并没有引进其他问题。如图 10.8 所示，在继承树里，类 D 通过两条不同的路径继承基类 A，在一个类 D 对象的表示里有 A 的数据成员的两个副本，同时，在它的虚表的两个部分里各有一组针对类 A 的虚方法项。创建 D 对象的 B 观察点时（例如要把指向一个 D 对象的指针赋给一

个 B* 变量时），不需要执行任何代码；而在创建 C 观察点时（例如，赋给一个 C* 变量）就要求加上偏移值 d。

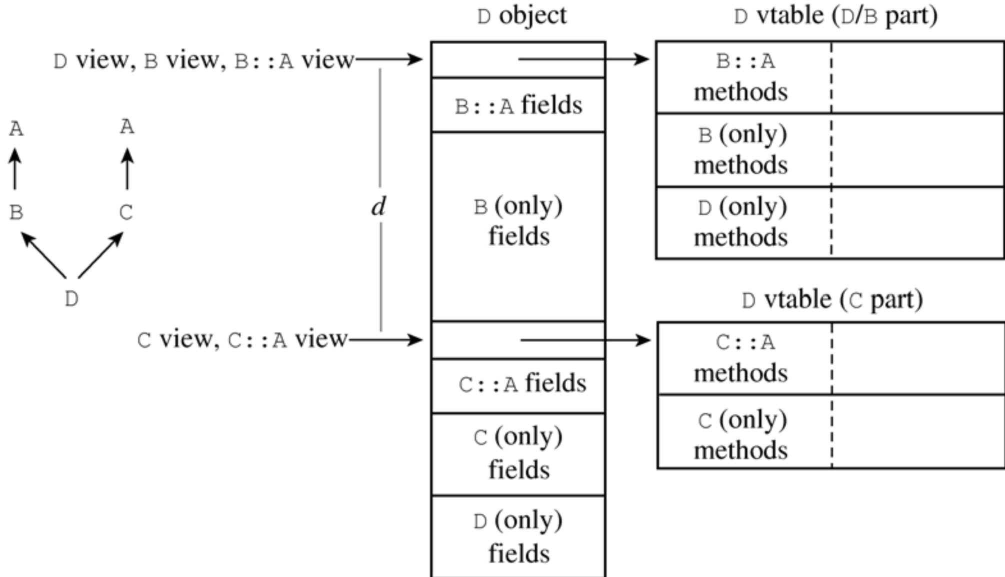

图 10.8　复本式多重继承的实现。每个基类都包含类 A 的一个完整副本。像在图 10.7 里一样，类 D 的虚表也分为两个部分，每个部分对应一个基类。每个表项是一个〈方法地址，this 校正值〉对。

由于存在歧义，我们现在不能从 D 对象出发直接通过名字访问其中的 A 成员。但还是可以访问它们，为此只要先把指向 D 对象的指针赋值给 B* 或是 C* 变量。与此类似，我们也不能直接把指向 D 对象的指针赋给 A* 指针，因为这样做没说清应该选择哪个 A 去创建观察点。但是可以通过 B* 或 C* 作为中介完成这种赋值：

```
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
```

正如在 10.5 节开头所述，虚表项必须是〈方法地址，this 校正值〉对。

10.5.3 共享继承 Shared Inheritance

共享继承引进了新的歧义可能和新的实现复杂性。与前节里一样，现在假定 D 继承自 B 和 C，这两者又都继承自 A。但这次假定 A 是共享的：

```
class A {
public:
    virtual void f ();
    ...
}
class B : public virtual A { ...
class C : public virtual A { ...
class D : public B, public C { ...
```

如果 B 或者 C 覆盖了 A 里声明的方法 f，就会出现新的歧义性问题： D 究竟应该继承哪个版本呢（如果它确实应该继承某一个）。C++定义说，如果 f 的所有定义中有一个定义制约了其他定义，那么对 f 的引用就是无歧义的（因而也是合法的）。所谓制约就是要求这个定义所在的类是其他有相关定义的类的后裔。在目前这个例子里，D 可以从 B 或者 C 继承 f 的覆盖版本。但如果这两个类都覆盖了 f，在 D 的任何代码里企图使用 f 都看作是静态错误。Eiffel 提供了一种更加精致的控制这种歧义性的机制。在那里，如果一个类从多条路径继承了一个覆盖方法，它可以描述自己希望继承哪一个。换种方式，通过重命名也使这个类可以访问所有不同版本。

为了实现共享继承，我们必须认识到如下事实：由于 A 的同一实例是 B 和 C 两者中的一部分，因此我们根本不可能使 B 和 C 的表示在内存中都连续。事实上，图 10.9 里的选择是让 B 和 C 的表示都不连续。当然，如果这样做，就要求在每个 B、C、D 对象的表示里（在下面派生类的对象的每个 B、C、D 观察点）里都必须包含对象中 A 部分的地址，用相对于该观察点开始的一个编译时常偏移值的方式表示。为访问 A 的数据成员，我们首先通过这个地址做间接，而后使用该成员在 A 里的偏移值。为调用 A 里声明的第 n 个虚方法，我们应执行如下的代码：

```
r1 := my_D_view                  —对象的原始观察点
r1 := *(r1 + 4)                  — A 观察点
r2 := *r1                        — 虚表 A 部分的地址
r3 := *(r2 + (n − 1) × 8)        — 方法地址
r2 := *(r2 + (n − 1) × 8 + 4)    — this 校正值
r1 := r1 + r2                    — this
call *r3
```

这段代码段的指令数与前面非虚基类的序列（第 10.5 节）一样，但是需要多做一次内存访问（通过 A 的地址做间接）。这段代码对任意对象的 D 观察点都可以工作，

包括在由 D 派生的类的对象里，虽然在那样的对象里，D 和 A 的观察点可能距离更远。第 2 行的常数 4 是假定地址长度为 4 个字节，因为 D 的 A 部分的地址就存放在 D 开始处的虚表地址之后。在带有多个基类虚表的对象里，通过相对于这个对象开始位置的不同偏移值，可以找到对应于各基类的那些部分的地址。

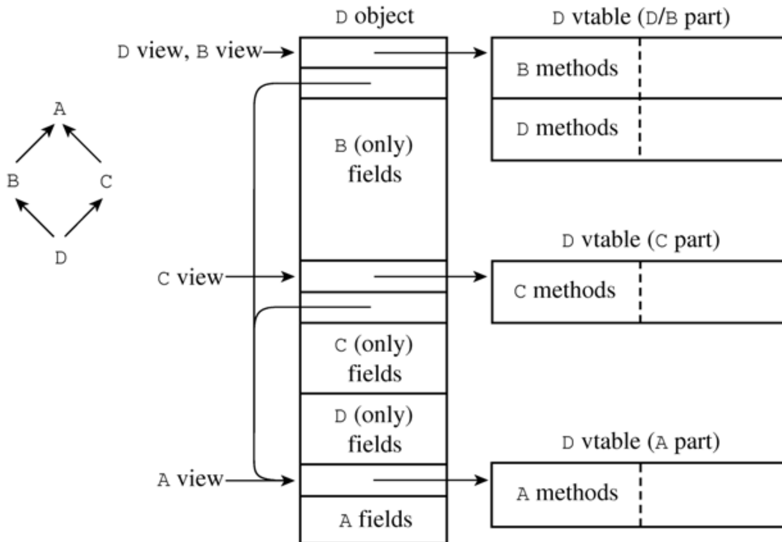

图 10.9　共享多重继承的实现。类 B、C、D 的对象里都包含了它们的 A 部分的地址，该地址放在编译时确定的常偏移值处（在图中放在虚表地址之后）。与图 10.7 和图 10.8 里的情况一样，虚表项里虚方法的 this 校正值也是相对于方法声明所在的类（即，访问虚表是通过这个类进行的）的观察点给出。

图 10.9 的实现策略在 C++里可行，是因为 C++ 总知道什么时候基类为 virtual（共享的）。对于那些非虚基类的数据成员和虚方法，我们仍然可以采用前两节里介绍（更廉价）的查找算法。另一方面，在 Eiffel 语言里，即使类层次结构中在某个层次上通过复本方式继承了一个特征，后来也可能采用共享继承。由于这种情况的存在，Eiffel 就需要采用某种更精细的实现策略（见练习 10.19）。

如果我们愿意为类的重复建立虚表里的一些部分，那么在 C++里访问虚基类的虚方法时就可以避免多做的一次间接。练习 10.20 探索了这种可能方式。

10.5.4 混入式继承 Mix-In Inheritance

在结束有关多重继承的讨论之前，让我们简单地回顾一下完全由抽象方法组

成的基类的概念，10.4.2 节说过，这种类在Java里称为**接口**。这种类既没有数据成员也没有方法实现[5]，因此能避免多重继承的大部分语义歧义性和实现复杂性。

从一个“实在的”基类和任意数目的接口继承，通常称为**混入式**继承——各个接口的虚方法被“混入”派生类的方法中。说“继承”一个接口似乎有些勉强，因为派生类必须为每个接口方法提供实现。然而，接口确实能通过多态性促进代码重用。如果某个子程序的形式参数被声明为具有某个接口类型，那么实现了（继承了）该接口的任何类都可以作为对应的实际参数传给它。可以合法传递的对象的类不必有公共的前辈类。

作为一个例子，假定我们现在有了一段 Java 代码，它按对象的某个正文域对对象进行排序，然后在万维网浏览器的窗口里显示对象的图形表示形式（需要完成适当的遮挡和刷新），并根据名字将这些对象的引用存入一个字典数据结构里。这些功能里的每一种都由一个接口表示。如果我们已经开发了一个复杂对象 `widget` 的类，那么通过把适当的界面混入由 `widget` 派生的类，就可以使用上述通用代码。有关情况如图 10.10 所示。

正如 10.4.3 节所说，Java 实现通常是在运行时用名字去查找方法。在这种情况下，接口的方法就很容易加入任何实现这一接口的类的字典里。如果没有运行时的方法查找，而又希望实现混入式继承，也有一种简单方法：扩大有关类的对象表示，加入被实现的那些接口的虚表地址，如图 10.11 所示。这些新加入的虚表指针就像新加入的数据成员，都放在基类对象的表示的后面，这样就做出了派生类的表示。如果在类层次结构中一些层次上加入接口和数据成员，那么虚表指针和数据成员就会散布在对象里的任何偏移位置上。

10.6 重温面向对象的程序设计 Object-Oriented Programming Revisited

在本章开始，我们提出了刻画面向对象的程序设计的三个基本概念：封装、继承和动态方法约束。封装使我们可以把抽象的实现细节隐藏在一个简单界面的背后；继承使我们可以把新的抽象定义为某些现存抽象的扩充或者精化，自动取得这些现存抽象的某些或者全部特性；动态方法约束使新定义的抽象能展现它自己的新行为方式，即使是被用在那些期望原有抽象的上下文中。

[5] Java实际上允许界面有数据成员，但这种成员总是常数，它们的值都必须在接口声明里描述。

```
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.
    }
}
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) { ...
    ...
}
```

图 10.10 Java 的接口类。通过在 **named_widget** 里实现 **sortable_object** 接口，在 **augmented_widget** 里实现 **graphable_object** 和 **storable_object** 接口，我们就能获得把这些类的对象传入传出像 **sorted_list.insert**、**browser_window.add_to_window** 和 **dictionary.lookup** 一类子程序的能力。

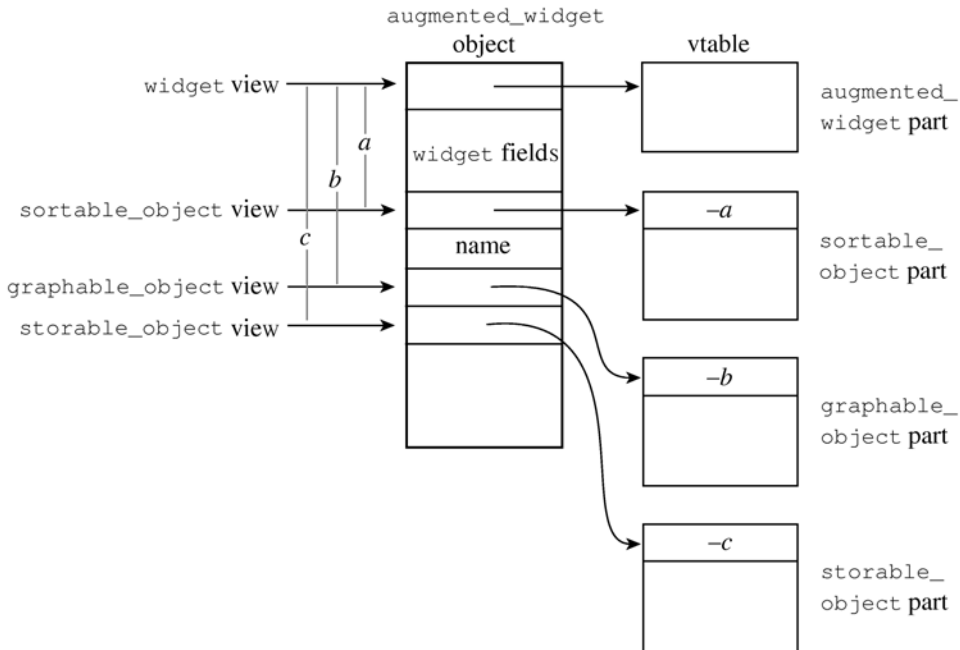

图 10.11　混入式继承的实现。类 **augmented_widget** 的对象里包含 4 个虚表地址，一个是该类自己的（如图 10.4 那样），另外 3 个虚表实现各个接口。送给接口子程序的对象观察点直接指向相应的虚表。这些虚表开始处是一个 this 校正值，它将被用于所有方法，以重新产生指向这一对象的指针。

不同的程序设计语言在不同程度上支持这些基本概念，特别是，不同语言对程序员以面向对象的风格写程序方面的**要求**也不同。有些作者争辩说，**纯**面向对象语言可能使人在写非面向对象的程序时遇到困难，甚至会束手无策。而按照纯粹论者的观点，面向对象语言就应该为计算提供一种**统一的对象模型**，其中每个数据类型都是类，每个变量都引用对象，每个子程序都是对象方法。进一步说，对象应该被看作一种**拟人化术语**：它们应该作为代表所有计算的主动性实体。

Smalltalk 非常接近这一理想状况。事实上，就像下一小节描述的，在 Smalltalk 里，甚至选择和迭代这样的流程控制机制也是通过方法调用模拟的。而在另一方面，Modula-3 和 Ada 95 则更应该说是冯•诺伊曼语言，但又**允许**程序员在愿意时采用面向对象风格写程序。

那么 C++又如何呢？这个语言确实有很丰富的特征，包括一些在面向对象程序里很有用的，甚至有些在 Smalltalk 里也找不到的特征（多重继承、精细的访问控制、严格的初始化顺序、析构函数、通用型特征等）。与此同时，它也包含了许

多很有争议的技术。其中的简单类型不是类，允许类之外的子程序。它采用静态方法约束和复本式多重继承作为默认方式，而没有采用代价更高的 `virtual` 方式。它的不加检查的 C 风格强制在类型检查和访问控制方面留下重大漏洞。它缺乏废料收集，给创建正确而自足的抽象设置了一个重要障碍。所有问题中最严重的，是它保留了 C 语言的所有低级机制，允许程序员完全摆脱或者颠覆程序设计的面向对象模型。有人建议说，最好的 C++程序员是那些**没有**先学 C 的人，因此他们不会倾向于在新语言里写“C 风格”的程序。公平地说，把 C++ 说成是面向对象的语言，就像把 Common Lisp 说成是函数式语言一样，可能不是大问题。除了废料收集问题之外，C++提供了所需要的全部工具，但它要求程序员经过很好的训练，以便能正确使用这些工具。

10.6.1* Smalltalk 的对象模型 The Object Model of Smalltalk

Smalltalk 在很大程度上是最正宗的面向对象语言。Smalltalk 的原始版本是由 Alan Kay 于 20 世纪 60 年代在犹他大学作为其博士工作的一部分设计的。这一工作被施乐公司的 Palo Alto 研究中心（PARC）的软件概念研究组采纳，经过 70 年代的五次重要修改，最终造就出作为其颠峰的 Smalltalk-80 语言。前面章节已经提到过 Smalltalk 的一些特征。现在我们集中讨论 Smalltalk 的拟人化程序设计模型。当然，对这一语言的完整介绍已超出了本书的范围。

Smalltalk 与其程序设计环境紧密集成在一起。事实上，与本书中提到的其他语言都不同，Smalltalk 程序并不是由一个字符序列构成，实际上，Smalltalk 程序的本意就是在一个 Smalltalk 实现的**浏览器**里被人观看，人们可以在其中通过字体改变和屏幕定位区分给定程序单位里的不同部分。Smalltalk 小组与当时 PARC 的 Interlisp 和 Pilot/Masa 项目一起，共同开发出今天被普遍采用的许多概念：点阵显式屏幕、窗口、菜单和鼠标等等。

Smalltalk 对所有变量采用一种无类型的引用模型，每个变量引用一个对象，静态时完全不必知道被引用对象所属的类。就像 10.3.1 节所说的那样（图 10.2 里展示），每个 Smalltalk 对象都是某个类的一个实例，这些类都是基类 `Object` 的后裔。所有数据都包含在对象里，其中最简单的是那些不变对象，例如 `true`（属于类 `Boolean`）和 3（属于类 `Integer`）。

从概念上看，所有操作都是送给对象的**消息**。举例说，表达式 3 + 4 表示送一个 + 消息给（不变）对象 3，并以对象 4 的引用作为参数。在响应这一消息时，

对象 3 将建立起一个对象 7 的引用。与此类似，表达式 a + b（其中 a 和 b 都是变量）表示将消息 + 送给 a 引用的那个对象，并以 b 所引用的对象作为参数。如果当时 a 恰好引用着 3 而 b 引用着 4，这个表达式的作用就像上面常数的例子一样。如前面在 6.1 节里介绍的，多参数消息采用的是多重的名字，在每个单词最后有一个冒号，每个参数跟在一个单词之后。表达式：

```
myBox displayOn: myScreen at: location
```

把 displayOn:at: 消息送给变量 myBox 引用的对象，以 myScreen 和 location 引用的对象作为参数。

在概念上，甚至 Smalltalk 的控制流也是消息传递。考虑下面选择结构：

```
n < 0
    ifTrue: [abs <- n negated]
    ifFalse: [abs <- n]
```

这段代码开始于送一个 < 0 消息（以 0 为参数的 < 消息）给 n 引用的对象。作为对这一消息的响应，n 引用的对象应该返回对两个不变对象 true 或者 false 之一的引用，这一引用也就是表达式 n < 0 的值。

Smalltalk 完全按照从左到右的顺序求值，不存在优先级和结合顺序。这样，n < 0 的值就成为 ifTrue:ifFalse: 消息的接受者。这个消息有两个参数，每个参数是一个**块**。Smalltalk 的块是由一对方括号括起的一段代码，也是一种不变对象，其语义大致相当于 Lisp 的 lambda 表达式。可以通过给块送 value 消息的方式要求它执行。

当不变对象 true 接到 ifTrue:ifFalse: 消息时，它就会给该消息的第一个参数（这个参数必须是一个块）送 value 消息，返回由此得到的值。在另一方面，对象 false 在响应同一个消息时，会把 value 消息送给它的第二个参数（ifFalse: 之后的那个块）。上面各块里的左箭头（<-）是赋值运算符。赋值不是消息，而是对其右边表达式求值的一种副作用。与 Algol 68 一类基于表达式的语言类似，赋值表达式的值也就是其右部表达式的值。这样，上述整个表达式的值就是其中的某个块的值，也就是说，或者是 n 的引用，或者是 n 的负数的引用，无论如何都是非负的。为了方便使用，Smalltalk 里的 Boolean 对象还实现了 ifTrue:、ifFalse: 和 ifFalse:ifTrue: 方法。

迭代也采用类似的方式模拟。为了完成通过枚举控制的循环，类 Integer 里实现了 timesRepeat: 和 to:by:do: 方法：

```
pow <- 1.
10 timesRepeat:
    [pow <- pow * n]
```

```
sum <- 0.
1 to: 100 by: 2 do:
   [:i | sum <- sum + (a at: i)]
```

第一段代码计算n^{10}。在响应timesRepeat: 消息时，整数k会把消息value送给参数（一个块）k次。第二个代码段加起由a引用的数组里下标为奇数的所有元素。在响应to:by:do: 消息时，整数k的行为如期：它把value: 消息送给第三个参数（一个块）$\lfloor (t-k+b)/b \rfloor$ 次，其中t是第一个参数，b是第二个参数。请注意value: 最后的冒号。简单的value是无参的，而value: 消息有一个参数，带有一个形参的块能接受它。在上面例子里，整数 1 把消息value: 1、value: 3、value: 5 等等送给块 [:i | sum <- sum + (a at: i)]。块开始处的 :i | 是这个块的形式参数。数组能接受at: 消息。为描述步长为 1 的迭代，整数还提供了一个to:do: 方法。

由于块也是对象，因此我们可以用变量引用它：

```
b <- [n <- n + 1].          "b is now a closure"
c <- [:i | n <- n + i].     "so is c"
...
b value.                    "increment n by 1"
c value: 3.                 "increment n by 3"
```

带有两个参数的块期望得到 value:value: 消息，带有 j 个参数的块期望得到名字由 j 个重复的 value: 组成的消息。双引号括起的是 Smalltalk 的注释（字符串用单引号括起）。

Smalltalk 用 whileTrue: 消息实现逻辑控制的循环，块能懂得这种消息：

```
tail <- myList.
[tail next ~~ nil]
   whileTrue: [tail <- tail next]
```

这段代码将把 tail 设置为 myList 的最后一个元素。双 ~ 运算符（~~）表示“没有引用同一个对象”。这里假定方法 next 返回位于其接受者之后的元素。在响应 whileTrue: 消息时，块给自己送一个 value 消息。如果这个消息的结果是一个 true 引用，那么块就把一个 value 消息送给原来那个消息的参数，并重复这样做。块还实现了 whileFalse: 方法。

Smalltalk 里的块使程序员可以构造自己需要的任何控制流结构。由于语法形式简单，Smalltalk 块比 Lisp 的 lambda 表达式更容易操作。从效果上看，to:by:do:

消息把迭代“翻开”，把循环体从 to:by:do: 方法的体里面翻出来，使之变成了一个可执行的（通过送给它 value 消息）简单的消息参数。Smalltalk 程序员也可以为其他容器类型定义方法，得到迭代器（6.5.3 节）的所有功能和 call_with_current_continuation（8.5.4 节）的部分功能：

```
myTree inorderDo: [:node | whatever]
```

应该注意，Smalltalk 里计算的统一对象模型并不意味着统一的实现方式。正如虽然 Clu 内部的不变对象采用引用语义，Clu 实现仍然把它们实现为值（6.1.2 节），Smalltalk 实现通常也会利用计算机算术的机器指令做整数计算，而不是实际地给整数送消息。与此类似，最常用的控制流结构（ifTrue:ifFalse:、to:by:do:、whileTure: 等等）一般也会由 Smalltalk 解释器重新组织，通过特定的快速代码实现。

在结束这一节时，我们看看 Smalltalk 里的递归，它工作得至少与其他语言里一样地好。下面是欧几里得算法的递归实现：

```
gcd: other                                 "other is a formal parameter"
    [self = other]
        ifTrue:   [↑self].                 "end condition"
    [self < other]
        ifTrue:   [↑self gcd: (other - self)]        "recurse"
        ifFalse:  [↑other gcd: (self - other)]       "recurse"
```

这段代码里的上箭头（↑）相当于 C 或者 Algol 68 里的 return，关键字 self 对应于 C++里的 this。这里我们用了混合字体显示代码，就像在一个 Smalltalk 浏览器里一样。方法的头部用黑体字标明。

10.7 总结和注记 Summary and Concluding Remarks

本章是我们有关语言设计的 5 个核心章中的最后一章。这些章包括名字（第 3 章）、控制流（第 4 章）、类型（第 6 章）、子程序（第 7 章）和对象（第 10 章）。

在 10.1 节开始，我们指出了面向对象程序设计的三个基本概念：封装、继承和动态方法约束，还介绍了类、对象和方法等术语。封装问题已经在第 3 章的模块里讨论过，它使数据抽象的细节能隐藏到一个相对简单的界面之后。继承扩展

了封装的能力，使程序员容易把新抽象定义为已有抽象的精化或者扩充。继承为子程序多态提供了一种自然的基础：如果一个子程序要求以某个给定类的实例作为实际参数，那么任何由这个类派生出的类的对象也可以用于这个子程序（假定该派生类维持了整个的现存界面）。动态方法约束扩展了这种形式的多态性，它在调用某个参数的方法时能做出适当安排，使用运行时实际对象的类所关联的方法。我们注意到，如 Modula-3 和 Ada 95 一类的语言通过某种类型扩充机制支持面向对象，其中的封装与模块相关联，而继承和动态方法约束则关联于一种特殊形式的记录。

随后几节里讨论对象初始化和终结操作、动态方法约束，以及多重继承的一些细节。在许多情况下，我们都看到了各种不同因素之间的此消彼长，以功能作为一个方面，简单性和执行速度作为另一方面。将变量作为引用而不是值，通常会导致更简单的语义，但要求多做一次间接。废料收集（如前面第 7.7.3 节所述）能极大地简化软件的创建和维护工作，但却会增加运行时的代价。动态方法约束（在一般情况下）要求使用虚表或者其他查找机制做方法分派。多重继承（无论使用与否）要求在虚表里存放 `this` 校正值。共享性多重继承要求更多做一次间接，以找到虚基类的数据成员。混入式继承简化了 Java 里的查找，但脆弱基类的问题限制了依赖于预先算出的表格的实现方式的适用范围。

在一些情况下，我们也看到了时间/空间之间的相互交换。正如 8.2.3 节所述，在线展开的子程序可能极大地改进包含许多小子程序的代码的性能，因为这种做法不仅可以消除子程序调用本身的开销，还使跨调用的寄存器分配、共用子表达式分析和其他“全局性”代码优化成为可能。但在另一方面，在线展开通常也会增大目标代码的规模。对于虚方法分派问题，练习 10.18 探索了用可执行代码代替虚表项的可能性，也是在节省时间的同时付出空间代价（类似的时空交换也出现在任何命令式语言 `case` 语句的实现中）。对于采用共享式虚继承的类，练习 10.20 探索了另一种重复虚表中某些部分，以减少一层间接的方式。

除了没有多重继承外，Smalltalk 被广泛认为是最纯最灵活的面向对象语言。然而由于不做编译时的类型检查，“基于消息的”计算模型，以及需要做动态方法查找，这些都使它的实现速度比较慢。C++有对象值变量、默认的静态约束、最少的动态检查以及高质量的编译器，在推动面向对象程序设计的大众化方面起了最重要的作用。虽然 Smalltalk 在可靠性、可维护性和代码重用方面有一些收获，

但这些或许不能作为其性能上高开销的理据，同样问题也适用于 Eiffel 里小一些的开销。几乎可以肯定的说，正是上面提出的因素决定了 C++相对较小的开销。随着软件规模越来越大，随着在 Internet 上的分布式计算的爆炸性增长，随着高度可移植的面向对象语言（Java）和二进制目标标准（ActiveX/OLE [Bro96]，CORBA [Sie96]/JavaBeans [Sun97]）的不断发展，我们可以肯定，面向对象的程序设计必然在 21 世纪的计算中扮演主要的角色。

10.8 复习 Review Questions

10.1 一般认为的面向对象语言的三个标志性特征是什么？

10.2 从 20 世纪 60 年代的哪个程序设计语言可以找到面向对象概念的根源？

10.3 对象界面的“私用”部分有什么用，为什么需要它？

10.4 什么是**容器**类？

10.5 请说明 C++语言里 `private`、`protected` 和 `public` 类成员之间的差异。

10.6 请说明 C++语言里 `private`、`protected` 和 `public` 基类之间的差异。

10.7 从模块里**不透明**导出是什么意思？

10.8 请说明在线展开子程序在面向对象语言里的特殊重要性。

10.9 请说明 Eiffel 里**选择性可用**的意思。

10.10 请以 Smalltalk、Eiffel 和 C++为一边，以 Oberon、Modula-3 和 Ada 95 为另一边，说明它们设计上的关键性差异。

10.11 什么是**构造函数**和**析构函数**？

10.12 为什么 C++比 Eiffel 更需要析构函数？

10.13 为什么在采用变量的引用模型（而不是变量的值模型）的语言里，对象初始化的问题更简单一些？

10.14 请说明 C++里初始化和赋值之间的不同。

10.15 请说明静态与动态方法约束（即**虚方法**和**非虚方法**）之间的不同。为什么 C++用静态方法约束作为默认方式？为什么 Smalltalk、Eiffel 和 Java 把所有方法都作为虚的？

10.16 请说明动态方法约束与多态性之间的联系。

10.17 什么是**纯虚函数**（即 Eiffel 里的**延迟特征**）。

10.18　请说明如何可以用虚函数得到子程序闭包的效果。

10.19　什么是**抽象类**（**延迟类**）？

10.20　请说明，虽然 C++和 Eiffel 已经有了对多态性的实质性支持，但是通用型机制（模板）在这些语言里仍然很有用。

10.21　请解释 Eiffel 里 `like` 的使用。

10.22　什么是**虚表**？它用于做什么？

10.23　什么是**脆弱的基类**问题？

10.24*　请解释复本式和共享式多重继承之间的差异。什么时候需要这种或那种多重继承？

10.25*　什么是**混入式**继承？

10.26*　请解释下面问题：即使是不重复出现的多重继承，在对象（的实现里）也需要多重观察点，需要在虚表里加入“`this` 校正值”域。

10.27*　请说明，为什么在共享式多重继承里访问特定父类的域时，需要多做一层间接。

10.28*　简单说明 Smalltalk 程序设计语言的一些新颖之处，也说明它的一些弱点。

10.9 练习 Exercises

10.1　有些语言设计者说面向对象消除了对嵌套子程序的需要。你同意这种观点吗？为什么？

10.2　参阅练习 6.23（如果你还没有解决那个问题，请现在做一做）。有了 C++一类的语言里那种创建“迭代器对象”的能力，你认为像 Clu 或者 Icon 风格的迭代器还有用吗？为什么？

10.3　请定义一个类层次结构，表示图 4.5 里的 CFG 的语法树。在每个类里提供一个返回结点的值的方法。请提供有关的构造函数，用它们扮演 `make_leaf`、`make_up_op` 和 `make_bin_op` 子程序的角色。

10.4　重做前面练习，但使用变体记录（联合）类型表示树结点。再用类型扩充的方式做这件事。从清晰、抽象、类型安全和可扩充性等方面比较这三个解。

10.5　采用另一种方式重写 10.1 节的表和队列类，使对象不是从容器基类派生，但仍然可以插入表与队列里。你可能需要在表/队列的结点里包含一个到数据的指针，而不是数据本身。

10.6　采用模板（通用型机制）对你在前一练习中的表/队列里的数据类型做抽象。

10.7 在几个面向对象语言里（包括 C++和 Eiffel），派生类可以隐藏基类的成员。例如，在 C++里我们可以把基类声明为 public、protected 或者 private：

```
class B : public A { ...
    // public members of A are public members of B
    // protected members of A are protected members of B
...
class C : protected A { ...
    // public and protected members of A are protected members of C
...
class D : private A { ...
    // public and protected members of A are private members of D
```

在所有的这些情况下，类 B、C、D 的方法都不能访问类 A 的 private 成员。请考虑 protected 和 private 基类对于动态方法约束的影响。在什么环境下对类 B、C、D 的对象的引用可以赋给类型为 A* 的变量？

10.8 如果我们在派生类里重新定义了一个数据成员，类的实现会发生什么情况？举例说，假定我们有：

```
class foo {
public:
    int a;
    char *b;
};
...
class bar : public foo {
public:
    float c;
    int b;
};
```

bar 对象的表示里包含一个还是两个 b 域？如果有两个，是两个都能访问，还是只能访问其中一个？在什么样的环境里有这些回答？

10.9 请讨论类和类型扩充方式的相对优势。你喜欢哪种方式？为什么？

10.10 你怎么看待 C++和 Ada 95 选择采用静态方法约束作为默认方式，而不是动态方法约束？这样做确实得到实现的速度优势而又没有损失抽象和可重用性吗？假定我们有时需要动态约束，你是喜欢 C++的按照方法确定的方式，还是 Ada 95 的按照变量确定的方式呢？为什么？

10.11 假设 foo 是 C++程序里的一个抽象类，为什么声明类型为 foo* 的变量可以被接受，而声明 foo 类型的变量就不行？

10.12* 请考察 10.6.1 节给出的欧几里得算法的 Smalltalk 实现，追踪在求值 4 gcd: 6 的过程中所涉及的消息。

10.13 假定类 D 继承类 A、B、C，而后面这几个类没有共享任何前辈。请画出类 D 的虚表在内存里的可能布局，再说明如何将一个对 D 对象的引用转换到对类 A、B 或 C 对象的引用。

10.14 考虑 10.5.1 节里描述的 person_interface 和 list_node_interface 类。假定 student 是由 person_interface 和 list_node_interface 派生的，在下面方法调用中会发生什么事情：

```
student s;
person *p = &s;
...
p.debug_print ();
```

你可能会想到用图说明 student 对象的表示，阐释所出现的方法查找和观察点计算的情况。你可以假定用的是 10.5 节开始所描述的实现方式，没有共享继承。

10.15* 给了 569 页里的继承树，请说明类 student_prof 对象的表示形式。你可能需要参考图 10.7、图 10.8 和图 10.9。

10.16* 有了图 10.7 的存储布局和下面声明：

```
student& sr;
gp_list_node& nr;
```

请给出下面赋值必须生成的代码：

```
nr = sr;
```

（缺陷：应该考虑 nil 指针）

10.17* 标准 C++为类提供了一种“到成员的指针”机制：

```
class C {
public:
    int a;
    int b;
} c;
int C::*pm = &C::a;
    // pm points to member a of an (arbitrary) C object
...
c->*pm = 3;    // assign 3 into c.a
```

指向成员的指针也允许指向子程序成员（方法），包括虚方法。在存在 C++风格的多重继承的情况下，你如何实现指向虚方法的指针？

10.18* 在处理多重继承的语言时，虚表项采用了〈方法地址, this 校正值〉对。另一种方式是只用一个简单指针，但让它们指向一段代码，让这段代码在线更新 `this` 的值之后跳到适当方法的开始位置。请展示这种模式下执行的指令序列。与 10.5 节给出的指令序列相比，什么因素会使这段新序列运行得更快些或者慢些？哪种模式使用的空间更少？（请注意计算代码的空间和数据结构的空间，并考虑哪些指令是必须在每个调用点重复设置的。）

还可以把采用可执行代码取代数据结构的事情再向前推进一步。请考虑另一种实现，其中的虚表本身就是可执行代码，说明这种代码大致具有什么样子，并（再次）讨论这种实现的时间和空间开销。

10.19* 在 Eiffel 里共享继承是默认的方式而不是例外，只有重命名的特征可以重复。作为结果，我们在看一个类时，就不可能说清它的成员被派生类以复本方式继承，还是以共享方式。请描述一种统一机制，用于查找继承自基类的各种成员，无论它们是以复本还是共享方式继承的。（提示：请考虑采用包含动态确定的数组的记录的那种描述向量，如 7.4.3 节所示。进一步的细节请参考 Wilhelm 和 Mauree 的编译程序教材 [WM95，5.3 节]。）

10.20* 图 10.9 里考虑了通过类 `B`、`C`、`D` 的观察点调用在类 `A` 里声明的虚方法的问题。通过将虚表的 `A` 部分的拷贝附加在虚表的 `D/B` 部分和 `C` 部分之后（带着适当调整的 `this` 校正值），我们就可能避免其中的一层间接。请给出采用这种实现方式时的调用序列。对于有 n 个前辈的类，在最坏情况下虚表可能大多少？

10.10 有关参考文献 Bibliographic Notes

附录 A 包含了本章中讨论各种语言（包括 Simula、Smalltalk、C++、Java、Modula-3、Ada 95、Oberon 和 CLOS）时引用的参考文献。Lisp 的其他面向对象版本包括 Loop [BS83a] 和 Flavors [Moo86]。Black 等讨论了 Emerald [BHL+87] 的类型系统，这是一个带有多态类型系统的并发语言，其类型系统基于一种**相似**概念，与基于类的分类之间有着有趣的不同。

Ellis 和 Stroustrup [ES90] 提供了有关 C++语义和语用的广泛讨论。Stroustrup 的 C++教程 [Str97] 的第 16 章到第 19 章包含了有关容器类的设计和实现的很好介绍。Deutsch 和 Schiffman [DS84] 描述了有效实现 Smalltalk 的一些技术。Borning 和 Ingalls [BI82] 讨论了多重继承，作为 Smalltalk-80 的一种扩充。Dolby 描述了

优化编译器如何（在 Eiffel 的意义下）识别出嵌套对象可以在其中展开的环境，而同时又保持其引用语义 [Dol97]。Driesen 给出了虚表的一种替代实现方式，它要求做整个程序的分析，但能提供特别高效的方法分派，即使语言里有动态类型和多重继承 [Dri93]。

组件系统是想为描述对象界面提供了标准，使得从任意语言的任何编译器生成的代码都能参与到一个工作程序里，这些代码常常还散布在许多机器中。CORBA [Sie96] 是由对象管理工作组（这是一个超过 700 个公司参加的共同体）颁布的一种组件标准。ActiveX（DCOM）[Bro96] 是微软公司与之竞争的一种标准。OLE [Bro96] 是 ActiveX 的二进制接口标准集合。JavaBeans [Sun97] 是用 Java 写的面向 CORBA 的组件的二进制标准。

面向对象程序设计的许多开创性文献出现在 ACM OOPSLA（面向对象的程序设计系统、语言和应用）系列会议文集中。这个系列会议从 1986 年开始，一直作为 ACM SIGPLAN Notices 的专集出版。Wegner [Weg90] 列举了面向对象的各种定义性特征，Meyer [Mey92，21.10 节] 解释了采用动态方法约束的理由。