13 类继承

重用机制

- 提供可重用的代码是面向对象编程的主要目的之一
  - 开发新项目，尤其是当项目十分庞大时，重用经过测试的代码比重新编写代码要好得多
    - 使用已有的代码可以节省时间，由于已有的代码被使用和测试过，有助于避免引入错误
    - 必须考虑的细节越少，有利于专注在当前程序的整体策略
- C的重用机制：库函数
  - 有些专用的C库没有源代码
  - 源码的修改存在一定风险。如不经意地修改了函数的工作方式或改变了库函数之间的关系
- C++的一种重用机制：类继承
  - 可以没有源码的情况下，从已有的类派生出新的类， 而派生类继承了原有类的特征，包括方法
    - 添加新功能
    - 添加新数据
    - 修改类方法的行为

一个简单的基类

1 一个简单的基类

P13.1 tabtenn0.h P13.2 tabtenn0.cpp P13.3 usett0.cpp

- 从一个类派生出另一个类
  - 原始类称为基类
  - 继承类称为派生类
- 基类TableTennisPlayer
  - 用于乒乓球俱乐部跟踪会员信息
  - string替代char，存储姓名
  - 构造函数中，采用成员初始化列表

```
TableTennisPlayer::TableTennisPlayer (const string &
fn, const string & ln, bool ht) : firstname(fn),
        lastname(ln), hasTable(ht) {}
```

```
1.  class TableTennisPlayer
2.  {
3.  private:
4.      string firstname;
5.      string lastname;
6.      bool hasTable;
7.  public:
8.      TableTennisPlayer (const string & fn = "none",
9.           const string & ln = "none", bool ht = false);
10.     void Name() const;
11.     bool HasTable() const { return hasTable; };
12.     void ResetTable(bool v) { hasTable = v; };
13. };
```

1.1 派生一个类

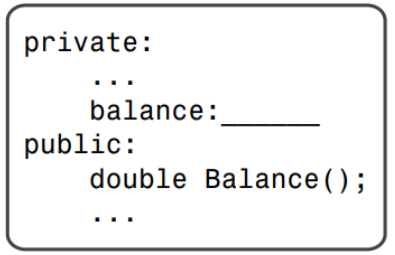
P13.4 tabtenn1.h P13.5 tabtenn1.cpp P13.6 usett1.cpp

- 新派生一个类RatedPlayer
  - 包含成员在比赛中的比分
- RatedPlayer从TableTennisPlayer继承
  - 继承基类的
    - 数据成员
    - 成员函数
  - 添加
    - 自己的构造函数【新类，新构造函数】
    - 额外的数据成员和成员函数

```
1.  // simple derived class
2.  class RatedPlayer : public TableTennisPlayer
3.  {
4.  private:
5.      unsigned int rating;
6.  public:
7.      RatedPlayer (unsigned int r = 0, const string & fn
    = "none", const string & ln = "none", bool ht = false);
8.      RatedPlayer(unsigned int r, const
    TableTennisPlayer & tp);
9.      unsigned int Rating() const { return rating; }
10.     void ResetRating (unsigned int r) {rating = r;}
11. };
```

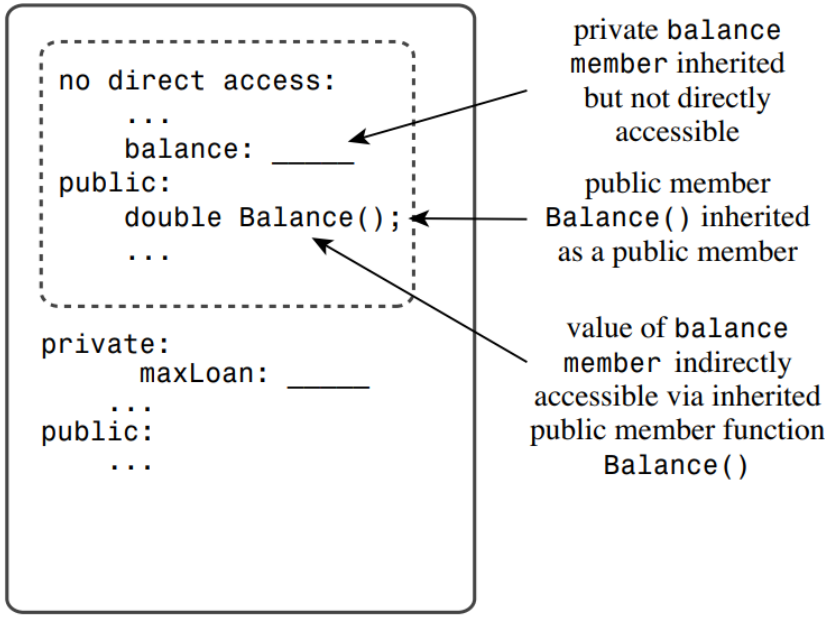
示例

- 继承
  - private:
    - 继承，但不能直接访问
  - public:
    - 继承，可以直接访问
- Balance进行了重载



BankAccount object

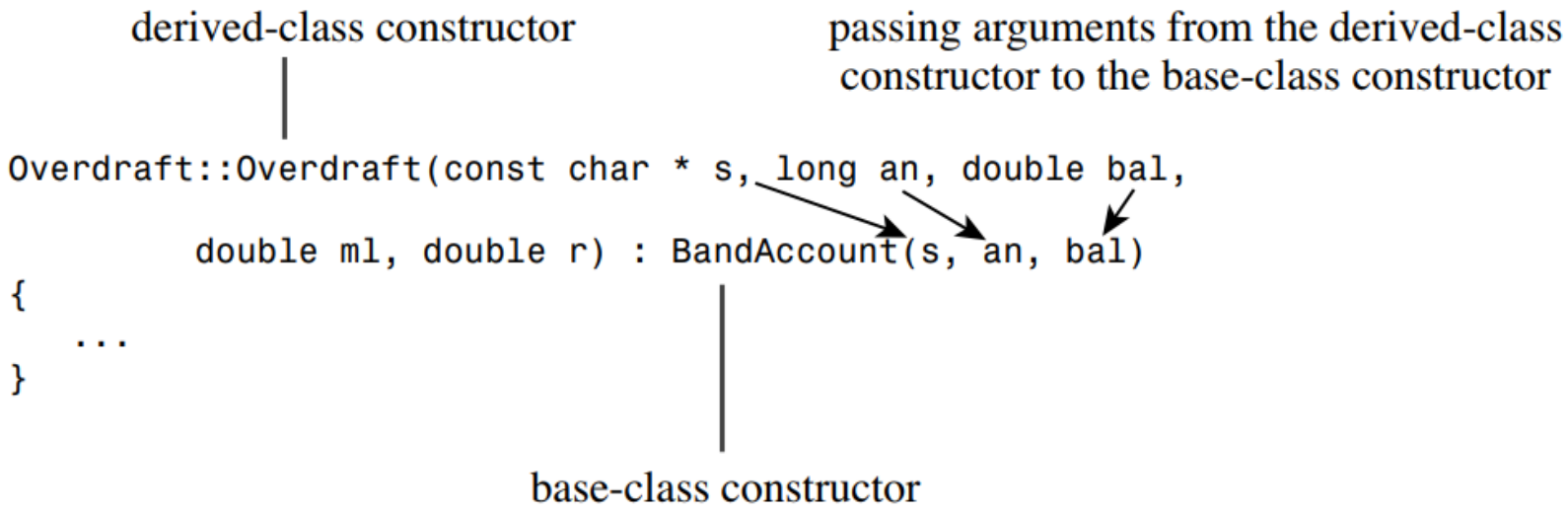
```
class Overdraft : public BankAccount {...};
```



Overdraft object

1.2 构造函数：访问权限的考虑

- 基类中的私有成员，派生类不能直接访问，只能通过基类公有方法访问
- 创建派生类对象时，程序首先创建出基类对象
  - 从概念上说，基类对象先被创建，然后进入派生类的构造函数
  - 构造函数可使用成员初始化列表：基类构造函数，数据成员初始化，逗号’,’隔开
    - 如省略成员初始化列表，则先创建基类对象，如果不显式调用基类构造函数，则使用默认的基类构造函数



派生类构造函数要点

- 派生类构造函数的要点
  - 首先，创建基类对象
  - 然后，通过成员初始化列表，将基类信息传递给基类构造函数
  - 最后，通过派生类构造函数，初始化派生类新增的数据成员
- 析构刚好相反
  - 先析构派生类对象
  - 再析构基类对象
- 构造和析构的顺序，后进先出
  - 先构建的后析构
  - 栈？

1.3 使用派生类

P13.4 tabtenn1.h P13.5 tabtenn1.cpp P13.6 usett1.cpp

- 一般情况下，每个类放在不同文件里
- 要使用派生类，程序必须要能够访问基类声明
  - 派生类和基类在同一个文件
  - 派生类和基类在不同文件，派生类声明前include基类声明头文件

1.4 派生类和基类之间的特殊关系

- 派生类对象可以使用基类的非私有方法
  - public和protected
- 基类对象一般对象，派生类对象是特殊对象，特殊包含一般；特殊对象可以当成一般对象用，但一般对象不能当成特殊对象看待
- 在不进行显式转换的情况
  - 基类指针，可指向派生类对象
  - 基类引用，可引用派生类对象

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
TableTennisPlayer &rt = rplayer;
TableTennisPlayer *pt = &rplayer;
rt.Name();  // invoke Name() with reference
pt->Name(); // invoke Name() with pointer
```

- 在不进行显式转换的情况
  - 基类指针或者引用，只能调用基类方法
    - 【基类看不到派生类的东西】
  - 不可将基类对象和地址，赋给派生类引用和指针
    - 【基类对象实体，没有派生类的部分，所以，基类对象不能转换成派生类对象用】
    - 一般的对象不能当成特别对象用

```
TableTennisPlayer player("Betsy", "Bloop", true);
RatedPlayer &rr = player;  // NOT ALLOWED
RatedPlayer *pr = &player; // NOT ALLOWED
```

引用和指针的兼容性属性

- 一些特别的基类引用和指针指向派生类对象情况
  - 函数调用，形参基类引用/指针
    - 实参基类对象/指针或派生类对象/指针
  - 基类对象初始化
    - 派生类对象作为参数
  - 基类对象赋值
    - 派生对象赋值给基类对象
- 为什么可以？
  - 派生对象就是特殊的基类对象

```
1.  void Show(const TableTennisPlayer & rt){
2.      cout << "Name: ";
3.      rt.Name();
4.      cout << "\nTable: ";
5.      if (rt.HasTable()) cout << "yes\n";
6.      else cout << "no\n";
7.  }

8.  TableTennisPlayer player1("Tara", "Boomdea", false);
9.  RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
10. Show(player1); // works with TableTennisPlayer
11. Show(rplayer1); // works with RatedPlayer
12. RatedPlayer olaf1(1840, "Olaf", "Loaf", true);
13. TableTennisPlayer olaf2(olaf1);

14. RatedPlayer olaf1(1840, "Olaf", "Loaf", true);
15. TableTennisPlayer winner;
16. winner = olaf1; // assign derived to base object
```

继承：is-a关系

2 继承：is-a关系

- has-a
  - 午餐可能包括水果，是组合/包含关系
  - 但午餐并不是水果
- is-a（is-a-kind-of）
  - 派生类对象也是一个基类对象，基类对象上的操作也可以在派生类对象上执行
    - 香蕉是一种特殊的水果，可以从Fruit 类派生出Banana类。
  - 不是is-like-a
    - 不采用明喻：律师是鲨鱼
  - 不是is-implemented-as-a（作为……来实现）
    - 可以使用数组来实现栈，但从Array类派生出Stack类不合适！
      - 栈不是数组。但通过让栈包含一个私有Array 对象成员来隐藏数组实现
  - 不是uses-a
    - 计算机可以使用激光打印机，但从Computer类派生出Printer类没有意义

多态公有继承

3 多态公有继承

- （在继承的结构中）多态
  - 同一个方法，在派生类和基类中的行为不同【因而可以通过新的派生类进行功能扩展】
  - 方法的行为取决于调用该方法的对象（的类别）
    - 对基类指针/引用的方法调用，程序根据指针/引用对象的实际类型来选择相应方法
  - 多态（具有多种形态），即同一个方法的行为随上下文而异
- 多态机制的好处
  - 提升可扩展性
  - 通过基类指针/引用实现应用程序的逻辑，应用程序的功能随着指针/引用对象类型的派生而扩展
- 实现多态的可能方法
  - 【广义的】在派生类中重新定义基类的方法（函数重载）
    - 派生类对象指针/引用，调用非虚方法/函数。实际调用派生类重载的方法
  - 【通常的】虚方法（virtual）
    - 基类对象指针/引用，调用虚方法/函数。实际调用指针/引用所属类型的方法

Pontoon银行

- Brass Account
  - Brass Account支票账户的信息：
    - 客户姓名；
    - 账号；
    - 当前结余。
  - 下面是可以执行的操作：
    - 创建账户
    - 存款
    - 取款
    - 显示账户信息
- Brass Plus（透支保护）
  - 包含Brass Account的所有信息及如下信息
    - 透支上限
    - 透支贷款利率
    - 当前的透支总额
  - 两种操作的实现不同
    - 取款操作有透支保护
    - 显示操作包含其他信息

3.1 开发brass类和brassplus类

P13.7 brass.h P13.8 brass.cpp P13.9 usebrass1.cpp

- 定义派生类
- 实现虚方法
  - 在基类中声明虚函数
- 演示虚方法
  - 通过对象（而不是指针或引用）调用的，没有使用虚方法特性
  - 用数组保存对象信息，因此数组的元素可采用基类指针
    - 通过基类指针调用来引发虚方法

- 为什么需要虚析构函数
  - 可以不加区分地delete 基类指针，引发正确的delete行为

静态联编和动态联编

4 静态联编和动态联编

- 函数调用时，使用哪个具体代码(调用哪个具体函数，选择其入口地址)?

```
class cls{};
class deri : public cls{};
cls obj;  obj.Fun();
deri obj2; obj2.Fun();
```

  - Fun()到底是哪个函数?
  - 编译器将源代码中的函数调用解释为执行特定的函数代码块，称为函数名联编(binding)
- 静态联编(static binding)
  - 在编译过程中进行（完成）的联编被称为静态联编 ，又称为早期联编(early binding)
- 动态联编(dynamic binding)
  - 编译器不知道用户将选择哪种类型的对象【如，调用函数的是基类指针，此时不能确认指针指向的具体类型】
    - 编译器生成能够在程序运行时选择正确虚方法的代码，称为动态联编，又称为晚期联编(late binding)

4.1 指针和引用类型的兼容性

- 由上往下，基类到派生类
- 指向基类的引用或指针可以引用派生类对象
  - 不必进行显式类型转换
  - 将派生类对象当基类对象用，称向上强制转换(upcasting)。该规则是is-a关系的一部分
- 将基类指针或引用转换为派生类指针或引用，称向下强制转换(downcasting)
  - 如不使用显式类型转换，向下强制转换不被允许
  - 派生类中操作新增数据成员的函数不能应用于基类
- 使用基类引用或指针作为参数的函数调用，进行（隐式 ）向上转换
  - 基类指针或引用，指向基类对象或派生类对象，调用虚成员函数，需要动态联编

```
class Employee
{
private:
    char name[40];
    ...
public:
    void show_name();
    ...
};
class Singer : public Employee
{
    ...
public:
    void range();
    ...
};
...
Employee veep;
Singer trala;
...
Employee * pe = &trala;
Singer * ps = (Singer *) &veep;
...
pe->show_name();
ps->range();
```

upcast—implicit type cast allowed

downcast—explicit type cast required

Upcasting leads to a safe operation because a `Singer` is an `Employee` (every `Singer` inherits `name`).

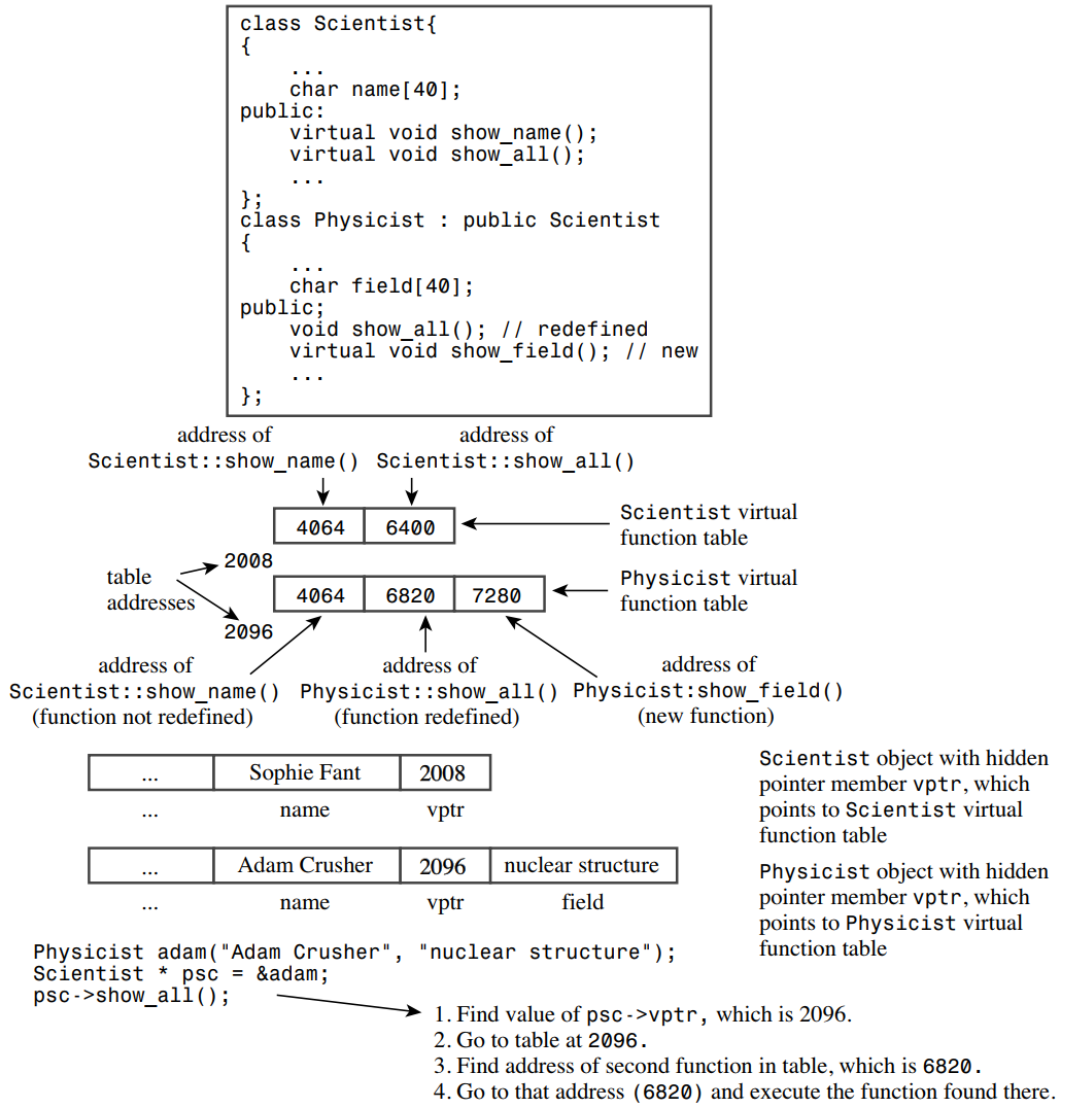
Downcasting can lead to an unsafe operation because an `Employee` isn't a `Singer` (an `Employee` need not have a `range()` method).

4.2 虚成员函数和动态联编

- 编译器
  - 对虚方法使用动态联编
  - 对非虚方法使用静态联编
- 为什么有两种类型的联编，以及为什么默认为静态联编
  - 效率原因
    - 静态联编的效率更高
    - Strousstrup：C++的指导原则之一，不要为不使用的特性付出代价（内存或者处理时间）
      - 仅当程序设计确实需要虚函数时，才使用它们
  - 概念模型
    - 仅将那些预期被重新定义的方法，声明为虚方法
    - 如一个方法在派生类中可能不会被重新定义，设置为非虚方法

虚函数的工作原理

- 虚函数机制(编译器实现)：对包含虚方法的类
  - 每个类包含一个虚函数表
    - 虚函数表：一个数组，用于存储虚函数的地址
  - 每个对象增加了一个隐藏成员【相当于具体类的标志】
    - 一个指针，指向虚函数表（virtual function table, vtbl）
- 不同对象的不同【指向虚函数表的】指针指向
  - 基类对象，指针指向基类中所有虚函数的地址表
  - 派生类对象，指针指向独立地址表
    - 如果派生类提供了虚函数的新定义，该虚函数表将保存新函数的地址
    - 重新定义虚函数，该vtbl将保存函数原始版本的地址
  - 【每个对象均有一个指针，而不是整个类唯一一个】



虚函数的工作原理

- 原理
  - 调用虚函数时，程序先查看该对象指针中的vtbl地址，然后转向相应的函数地址表
  - 如果使用类声明中定义的第n个虚函数，则程序使用该函数地址，并执行具有该地址的函数

- 使用虚函数时，在内存和执行速度方面有一定的成本，包括：
  - （内存）每个对象多了一个指向虚函数表的指针
  - （内存）每个类多了一个虚函数地址表（数组）
  - （执行速度）每个函数调用，需要执行一项额外的操作：到表中查找地址
- 虽然非虚函数的效率比虚函数稍高，但不具备动态联编功能

4.3 有关虚函数注意事项

- 基类方法的声明中使用关键字virtual ，该方法在基类以及所有的派生类（包括从派生类派生．出来的类）中是虚的

- 指向对象的引用或指针，调用虚方法
  - 程序将使用该对象所属的类型中定义的方法，而不使用引用或指针类型所定义的方法
    - 此现象称为动态联编或晚期联编
  - 通过基类指针或引用可以指向派生类对象，这样可以扩展到未知的派生类

- 如果一个类是基类，那么应将需要在派生类中重新定义的类方法，声明为虚的

其它注意事项

- 构造函数不能是虚函数
  - 派生类的构造函数将使用基类的一个构造函数，这种顺序不同于继承机制
- 析构函数应为虚函数（可通过delete基类指针删除派生类对象）
- 类静态成员函数不能是虚函数
- 友元不是虚函数
  - 友元函数不是类的成员【只有成员函数才能是虚函数】

- 派生链中使用的是最新的虚函数版本
  - 如果派生类没有重新定义函数，使用基类版本
  - 如果派生类位于派生链中，则将使用最新的虚函数版本
- 不要重载基类的虚函数
  - 第一，如果重新定义继承的方法，应确保与原来的原型完全相同
  - 第二，如果基类声明被重载了，则应在派生类中重新定义所有的基类版本

访问控制：protected

13.5 访问控制：protected

- 关键字 protected
  - 针对派生类设定的
- 派生类
  - 派生类中，可以访问基类的protected（保护）成员，而不能访问基类private成员
- 其他情况
  - 在类外，派生类对象只能访问public成员

抽象基类

6 抽象基类

- 抽象基类(abstract base class，ABC)
- 开发一个图形程序，该程序会显示圆和椭圆等
  - 圆是椭圆的一个特殊情况，因此，所有的圆都是椭圆，可以从Ellipse类派生出Circle类
    - 虽然圆是一种椭圆，但是这种派生是笨拙的 。简单继承不是很好的解决方法
  - 另一种解决方法，从Ellipse和Circle类中抽象出它们的共性，将这些特性放到一个ABC中。然后从该ABC派生出Circle和Ellipse类
    - 这样，便可以使用基类指针数组同时管理Circle和Ellipse对象，即可以使用多态方法
    - 不能在ABC中提供的函数，定义成纯虚函数=0(没有足够细节)
- ABC描述的是至少使用一个纯虚函数的接口，从 ABC 派生出的类将根据派生类的具体特征，使用常规虚函数来实现这种接口

6.1 应用ABC概念

P13.11 acctabc.h　P13.12 acctabc.cpp　P13.13 usebrass3.cpp

- Brass和BrassPlus

6.2 ABC理念

- 就类的设计方法来说，ABC方法更具系统性、更规范

- 设计ABC之前，首先应开发一个模型：指出所需的类以及它们之间相互关系
  - 一种学院派思想认为，如果要设计类继承层次，则只能将那些不会被用作基类的类设计为具体的类。这种方法的设计更清晰，复杂程度更低

- 可以将ABC看作是一种必须实施的接口。ABC 要求具体派生类覆盖其纯虚函数–迫使派生类遵循ABC设置的接口规则
  - 该模型在基于组件的编程模式中很常见
  - 组件设计人员制定“接口约定”，确保从ABC派生的所有组件都至少支持ABC指定的功能

继承和动态内存分配

7 继承和动态内存分配

- 继承是怎样与动态内存分配（使用new和delete）进行互动的呢？
  - 如果基类使用动态内存分配，并重新定义赋值和复制构造函数，这将怎样影响派生类的实现呢？

7.1 第一种情况：派生类不使用new

- 比较正常
  - 构造函数
  - 复制构造函数
  - 赋值运算

7.2 第二种情况：派生类使用new

- 显式析构
  - 只需要delete派生类中的内存
- 复制构造函数
  - 需要调用基类复制构造函数（初始化列表）
- 赋值运算
  - 需要调用基类赋值运算（显示调用）

7.3 使用动态内存分配和友元的继承示例

- 友元不是成员

类设计回顾

8 类设计回顾

- C++可以用于解决各种类型的编程问题，但不能将类设计简化成带编号的例程
- 有些常用的指导原则

8.1 编译器生成的成员函数

- 默认构造函数
- 复制构造函数
- 赋值运算符

8.2 其他的类方法

- 构造函数
  - 只能在创建时调用。构造函数在完成其工作之前，对象并不存在
- 析构函数
  - 一定要析构函数，用于释放使用了new分配的所有内存
    - 并完成类对象所需的任何特殊的清理工作。对于基类，即使它不需要析构函数，也应提供一个虚析构函数。
- 转换
  - 使用一个参数就可以调用的构造函数定义了从参数类型到类类型的转换

8.2 其他的类方法

- 传值还是引用
  - 使用对象作为参数的函数时，应按引用而不是按值来传递对象
  - 按值传递对象涉及到生成临时拷贝，即调用复制构造函数，然后调用析构函数
  - 如果函数不修改对象，应将参数声明为const引用
- 返回对象和返回引用
  - 直接返回对象与返回引用之间唯一的区别在于函数原型和函数头
  - 如函数返回在函数中创建的临时对象，则不要使用引用
  - 如函数返回的是通过引用或指针传递给它的对象，则应按引用返回对象
- const
  - 用它来确保方法不修改参数

8.3 公有继承的考虑因素

- is-a关系
  - 要遵循is-a 关系。如果派生类不是一种特殊的基类，则不要使用公有派生
  - 表示is-a 关系的方式之一，无需进行显式类型转换，基类指针就可以指向派生类对象，基类引用可以引用派生类对象。
  - 反过来，不能在不进行显式类型转换的情况下，将派生类指针或引用指向基类对象
- 什么不能被继承
  - 构造函数不能继承
  - 析构函数不能继承
  - 赋值运算符不能继承
- 赋值运算符
  - 如果类构造函数使用new 来初始化指针，则需要提供一个显式赋值运算符
  - 如果派生类使用了new，则必须提供显式赋值运算符。必须给类的每个成员提供赋值运算符，而不仅仅新成员

8.3 公有继承的考虑因素

- 私有成员与保护成员
  - 对派生类而言，保护成员类似于公有成员；但对于外部而言，保护成员与私有成员类似。派生类可以直接访问基类的保护成员，但只能通过基类的成员函数来访问私有成员。
  - 将基类成员设置为私有，可以提高安全性；设置为保护成员，则可简化代码编写工作，并提高访问速度
- 虚方法
  - 设计基类时，必须确定是否将类方法声明为虚的
  - 如果希望派生类能够重新定义方法，则应在基类中将方法定义为虚的，这样可以启用晚期联编（动态联编）；
  - 如果不希望重新定义方法，则不必将其声明为虚的
    - 虽然无法禁止他人重新定义方法，但表达了意思：不希望它被重新定义

8.3 公有继承的考虑因素

- 析构函数
  - 基类的析构函数应当是虚的
    - 当通过指向对象的基类指针或引用来删除派生对象时，程序将首先调用派生类的析构函数，然后调用基类的析构函数，而不仅仅是调用基类的析构函数
- 友元函数
  - 友元函数并非类成员，因此不能继承
    - 如希望派生类的友元函数能够使用基类的友元函数，可通过强制类型转换，将派生类引用或指针转换为基类引用或指针，然后使用转换后的指针或引用来词用基类的友元函数
  - 也可使用运算符dynamic_cast<>来进行强制类型转换

8.3 公有继承的考虑因素

- 有关使用基类方法的说明
  - 派生类对象，自动使用继承而来的基类方法，如果派生类没有重新定义该方法
  - 派生类的构造函数，自动调用基类的构造函数
  - 派生类的构造函数，自动调用基类的默认构造函数，如果没有在成员初始化列表中指定其他构造函数
  - 派生类构造函数，显式地调用成员初始化列表中指定的基类构造函数
  - 派生类方法，可以使用作用域解析运算符来调用公有的和受保护的基类方法
  - 派生类的友元函数，可以通过强制类型转换，将派生类引用或指针转换为基类引用或指针，然后使用该引用或指针来调用基类的友元函数

8.4 类函数小结

表 13.1 成员函数属性

函数	能否继承	成员还是友元	默认能否生成	能否为虚函数	是否可以有返回类型
构造函数	否	成员	能	否	否
析构函数	否	成员	能	能	否
=	否	成员	能	能	能
&	能	任意	能	能	能
转换函数	能	成员	否	能	否
()	能	成员	否	能	能
[]	能	成员	否	能	能
->	能	成员	否	能	能
op=	能	任意	否	能	能
new	能	静态成员	否	否	void*
delete	能	静态成员	否	否	void
其他运算符	能	任意	否	能	能
其他成员	能	成员	否	能	能
友元	否	友元	否	否	能

9 总结

- 继承通过使用基类定义派生类，使得能够根据需要修改编程代码
  - 公有继承建立is-a关系，这意味着派生类对象也应该是某种基类对象
- 如果要将类用作基类，则可以将成员声明为保护的
  - 派生类将可以直接访问这些成员
- 如果希望派生类可以重新定义基类的方法，则可以使用关键字virtual将它声明为虚的
  - 对于通过指针或引用访问的对象，能够根据对象类型来处理，而不是根据引用或指针的类型来处理
- 可以考虑定义一个ABC：只定义接口，而不涉及实现