10 对象和类

前言

- 面向对象编程(OOP) 是一种特殊的、设计程序的概念性方法。最重要的OOP 特性
  - 抽象
  - 封装和数据隐藏
  - 多态
  - 继承
  - 代码的可重用性
- 为了实现这些特性并将它们组合在一起， C++所做的最重要的改进是提供了类

1 过程性编程和面向对象编程

任务/问题

- 记录Genre Giants垒球队的统计数据
  - 输入每名选手的姓名、击球次数、击中次数、命中率以及其他重要的基本统计数据
  - 显示这些结果

过程性编程

- 过程性编程（面向过程）方法
  - 首先，考虑要遵循的步骤：将程序分解为一系列按顺序执行的过程或函数
    - 过程/函数：具有特定功能的代码块（逻辑单元），相对独立、功能完整
    - 通过函数名调用：接收输入参数，执行特定操作，并返回结果
  - 然后，考虑如何表示数据：确定程序所需的必要数据
- 过程性编程的特点
  - 步骤与数据的操作顺序不严格，可能需要多次迭代
  - 强调操作过程和流程控制，而非数据状态和对象交互
    - 不适合需要更高级抽象的场景

记录垒球队的统计数据

- 任务/问题
  - 输入每名选手的姓名、击球次数、击中次数、命中率等重要统计数据
  - 程序能够显示这些结果
- 首先，考虑要遵循的步骤
  - 【主要过程】main() 调用三个函数：获取输入 → 计算 → 显示结果
  - 【细化和扩展】添加更新统计数据的函数，在 main() 中提供菜单（输入、计算、更新、显示）
- 其次，表示数据
  - 用多个数组分别存储姓名、击球数、击中数等
  - 【细化和扩展】设计结构体存储每位选手信息，用该结构体数组表示整个球队

面向对象编程

- 面向对象编程(Object Oriented Programming, OOP)
  - 首先考虑数据【数据的描述，设计抽象数据类型（和实现语言无关）】
    - 如何表示数据【数据属性】
    - 如何使用数据【接口】
  - 完成接口和数据表示的细节
  - 使用接口设计方案创建程序

什么是接口

- 接口
  - 是一个共享框架，供多个系统(用户、设备或者模块)交互（通讯）时使用
  - 只强调交互的形式，而隐藏系统的内部处理细节
  - 接口一般定义为函数方式，其核心：
    - 完成的功能，接口名(函数名)，接口形式(参数，返回值)

面向对象编程

- 不仅要考虑如何表示数据，还要考虑如何使用数据
- 选手
  - 有一个对象表示整个选手的各个方面
    - 基本数据单元，一个表示选手的姓名和统计数据的对象
  - 一些处理该对象属性的方法
    - 首先，需要一种将基本信息加入到该单元中的方法
    - 其次，应计算一些东西，如命中率，因此需要添加一些执行计算的方法
    - 另外，还需要一些更新和显示信息的方法
    - 总之，用户与数据交互的方式有

      初始化、更新和报告，即为用户接口

面向对象编程

- 总之，OOP
  - 首先，从用户角度考虑问题【描述接口】
    - 描述对象所需要的数据
    - 描述用户与数据交互所需要的操作【定义接口】
  - 然后，确定如何实现接口和数据存储
    - 接口实现【实现具体的函数】
    - 数据存储【完善具体的数据表达】
  - 最后，使用设计方案创建程序【应用】
    - 用写好的类接口完成应用任务
- 定义类（类接口，类实现），类使用

抽象和类

2 抽象和类

- 抽象是处理复杂问题的核心原则
  - 抽取本质特征，忽略无关细节
    - 否则，难以真正理解和描述问题
  - 并设计出描述该问题的数据与操作
- 此原则在程序设计中同样适用
  - 首先，定义**信息与用户的接口**（即用户可对数据执行的操作集合）
    - 例：遥控器是用户与电视机的接口，用户只需按键操作，无需了解内部电路
    - 如垒球示例：接口规定了用户可对数据执行初始化、更新和显示操作
  - 然后，将接口用代码加以实现
  - 在C++中，**类**是程序员自定义的数据类型，用于描述现实事物及其操作

2.1 类型是什么

- 给数据对象指定类型，意味着
  - 决定数据对象所需的内存大小
  - 决定如何解释内存中的位（二进制表示规则）
    - 如，long 与 float 占用相同位数，但二进制解释规则不同，数值范围也不同
  - 决定可对数据对象执行的操作
- 【注】 "类型"的完整含义 = 内存大小 + 解释规则 + 可用操作
  - 内置类型（如 int、float）的以上信息均由编译器内置实现
  - C++ 的类机制允许程序员以同样的方式定义自己的类型

2.2 C++中的类

- 将"定义接口而不涉及实现"这一思想形式化，便得到抽象数据类型
- 抽象数据类型 (ADT)：规定数据类型的行为，而不关心内部实现细节
  - 数据对象集：该类型所有合法值的集合
  - 操作集：可对这些数据执行的所有操作
  - 行为规范：各操作的功能说明及其约束
- C++ 中，**类**是 ADT 的实现——将数据与操作封装为一个整体
  - 数据成员：描述对象属性（对应 ADT 数据对象集）
  - public 成员函数（方法）：描述对象操作，构成对外接口（对应 ADT 操作集）
- 【注】ADT 描述"做什么"，类的实现决定"怎么做"

什么是接口【回顾】

- 接口：供多个系统（用户、设备或模块）交互时共同遵循的规范
  - 只规定交互的形式，隐藏内部实现细节
  - 在程序设计中，接口通常以函数形式定义，包含
    - 函数名：说明该接口完成的功能
    - 参数列表与返回值：规定调用方式

C++类的接口

- 在类中，接口由**public成员函数**构成，称为公共接口
  - 交互的系统由类对象组成
  - 【类的开发者】 实现接口：编写类的 public 成员函数
  - 【类的使用者】 使用接口：通过调用 public 函数与类对象交互
    - 如，std::cin 的开发者实现了 >> 接口，使用者直接写 std::cin >> n; 即可完成输入
  - 【注】 同一程序员可以既是类的开发者，也是类的使用者

设计一个表示股票的类

- 设计 Stock 类【数据表示 + 操作方法】
  - 首先，确定数据表示：Stock 对象需要包含哪些信息
  - 其次，定义操作方法：确定Stock 类的公有接口
    - 接口设计完成后，再补充所需的数据成员
  - 最后，完成接口的代码实现
- C++ 中，类定义分两步
  - 类声明（.h 文件）：数据成员 + 成员函数原型
  - 类方法定义（.cpp 文件）：成员函数的具体实现
- 使用类：创建对象并调用成员函数
- P10.1 stock00.h

```
1.  #include <string>
2.  class Stock // class declaration
3.  {
4.  private:
5.    std::string company;
6.    long shares;
7.    double share_val;
8.    double total_val;
9.    void set_tot() { total_val = shares * share_val; }
10. public:
11.   void acquire(const std::string &c, long n, double p);
12.   void buy(long num, double price);
13.   void sell(long num, double price);
14.   void update(double price);
15.   void show();
16. }; // note semicolon at the end
```

类的定义(声明、类方法实现)、类的使用

```
// 类声明，.h文件中
class Stock {
private:
    std::string company;
    //…
    void set_tot() {total_val = shares * share_val;}
public:
    void acquire(const std::string & c, long n, double p);
};    // note semicolon at the end

//类方法实现，.cpp中
void Stock::acquire(const std::string &c, long n, double p)
{
    company = c;
    share_val = p;
    //…
    set_tot();
}
```

```
//类的应用，另一个.cpp中
#include "stock00.h"
int main()
{
    Stock fluffy_the_cat;
    fluffy_the_cat.acquire("NanoSmart", 20, 12.50);
    fluffy_the_cat.show();
    return 0;
}
```

类定义

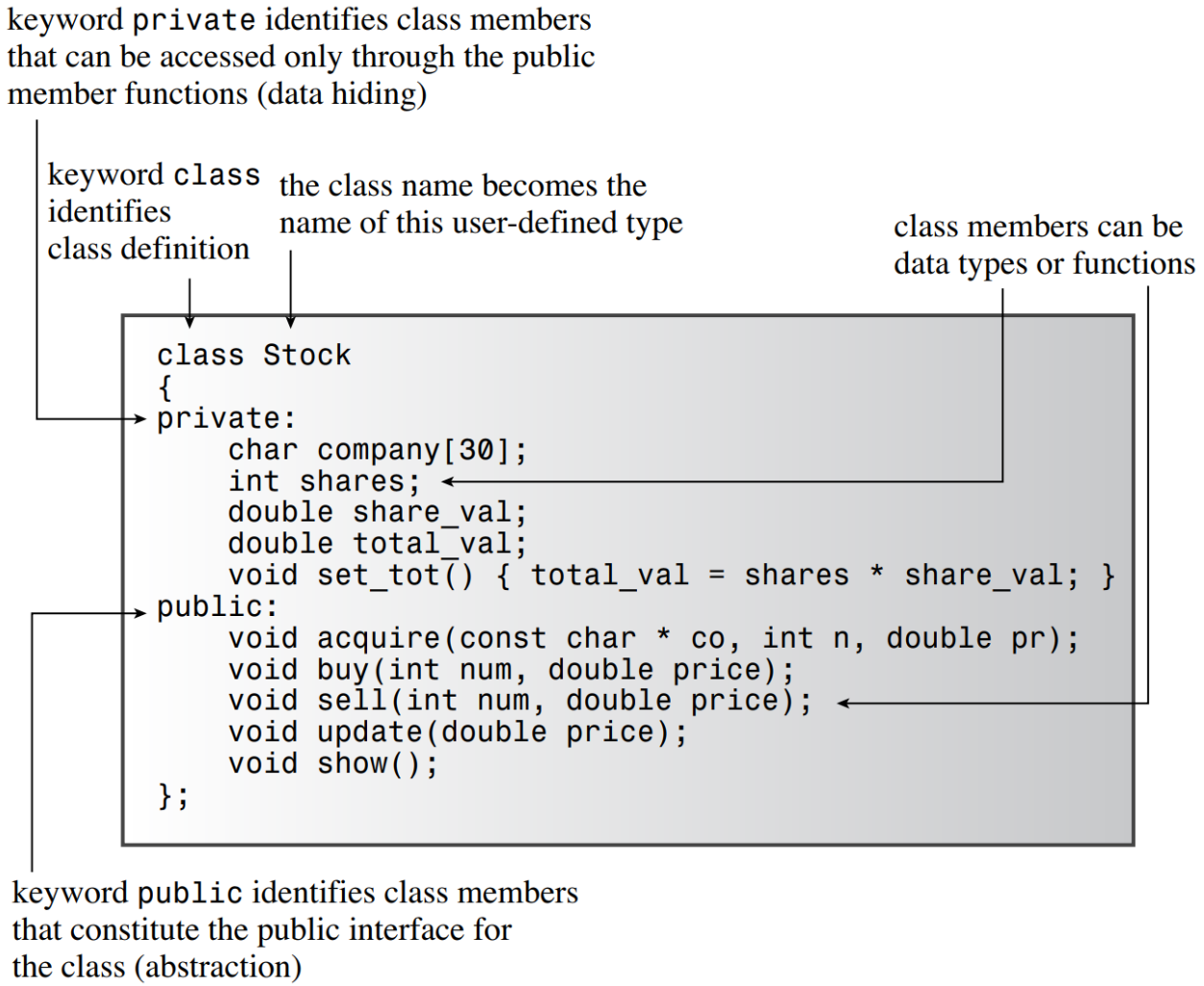
- Stock00.h(P10.1 tock00.h)
- 类定义两步骤
  - 提供类声明（.h 文件）
  - 实现类成员函数（.cpp 文件）
- 类声明包括
  - 数据成员【完整表示】
  - 成员函数/方法【可仅为原型】

```
1.  //  01-stock00.h
2.  class Stock // class declaration
3.  {
4.  private:
5.      std::string company;
6.      void set_tot() { total_val = shares * share_val; }
7.  public:
8.      void acquire(const std::string &c, long n, double p);
9.  }; // note semicolon at the end

10. // 01-stock00.cpp
11. #include "01-stock00.h“
12. // 类成员函数（类方法）实现
13. void Stock::acquire(const std::string & co, long n,
    double pr)
14. {
15. }
```

类声明

- 类声明包括
  - 数据成员【完整表示】
  - 成员函数/方法【可仅为原型】
- 访问控制：规定类成员的可访问范围
  - 关键字：private / public / protected
  - 类内（成员函数内）：不受限制
  - 类外（通过对象）：只能访问 public 成员



类声明 - 访问控制

- 访问控制：规定类成员的可访问范围
  - 类内（成员函数内）：不受限制
  - 类外（通过对象）：只能访问 public 成员
    - `Stock sto; sto.show();  // show() 是 public，可以`
- private
  - 类外不可访问，用于数据隐藏（封装核心）
  - `sto.set_tot();  // 不可以！set_tot() 是 private`
- public
  - 类外可访问，用于对外接口

```
1.  class Stock {
2.  private:
3.      std::string company;
4.      void set_tot() { total_val; } //类内，可
5.  public:
6.      void acquire(const std::string & c, long n, double p);
7.  };

8.  void Stock::acquire(const std::string & co, long n, double pr){//
    类内，都可以访问，数据成员或者成员函数
9.      company = co; //类内，数据成员，可以
10.     set_tot(); //类内，成员函数，可以
11. }

12. int main(){
13.     Stock fluffy_the_cat;
14.     fluffy_the_cat.show( ); //类外，公有，可以
15.     fluffy_the_cat.company; //类外，私有，不可以！
16. }
```

封装

- 类设计核心原则：将公有接口与实现细节分离
  - 公有接口：描述类的抽象行为（"做什么"）
  - 实现细节：隐藏于私有部分（"怎么做"）
- 这种分离称为**封装**，C++ 实现方式：
  - 数据隐藏：数据成员置于 private，类外不可直接访问
  - 实现隐藏：函数实现置于 private 或 .cpp 文件，类外不可见
  - 分文件存放：声明（.h）与实现（.cpp）分离
- 【注】用户只需掌握接口，无需了解内部实现

数据隐藏的好处

- 数据隐藏不仅防止直接访问数据，也使用户无需了解数据的内部表示
  - 例如，show() 显示股票总价格，该值：
    - 可存储于对象中，也可按需计算得到
    - 使用者只需知道函数的参数与返回值，无需关心实现方式
- 将实现细节与接口分离
  - 改进实现细节时无需修改接口，程序更易维护

OOP 和 C++

- OOP 是一种编程风格，可用于多种语言（包括 C）
- C++ 内置了专门支持 OOP 的语言特性：
  - 将数据表示与函数原型放入类声明，形成整体描述
  - 将数据表示设为私有，只允许授权函数访问

类成员函数公有还是私有?

- 隐藏数据是 OOP 的主要目标之一
  - 数据项通常置于私有部分，只在成员函数中使用
- 作为公有接口的函数设为 public
- 不提供访问控制时默认值：
  - class 默认 private
  - struct 默认 public

2.3 实现类成员函数

P10.2 stock00.cpp

- 类定义第二步：实现成员函数
  - 用作用域解析运算符 (::) 标识函数所属的类
  - 类方法可访问 private 成员【类内】

```
1.  #include <iostream>
2.  #include "01-stock00.h“

3.  void Stock::show()
4.  {
5.      std::cout << "Company: " << company
6.                << "  Shares: " << shares << '\n'
7.                << "  Price: $" << share_val
8.                << "  Total: $" << total_val << '\n';
9.  }
```

函数成员说明

- 数据私有化 + 限公有函数访问，使得：
  - 数据只能通过接口访问，可加安全防护
- set_tot()：私有内部函数，供类开发者调用，不对外暴露
- 内联函数（可选）
  - 写在类声明中的函数默认为 inline
  - inline void，而非 void inline
  - inline 未必会被编译器执行

调用方法时，使用哪个对象？

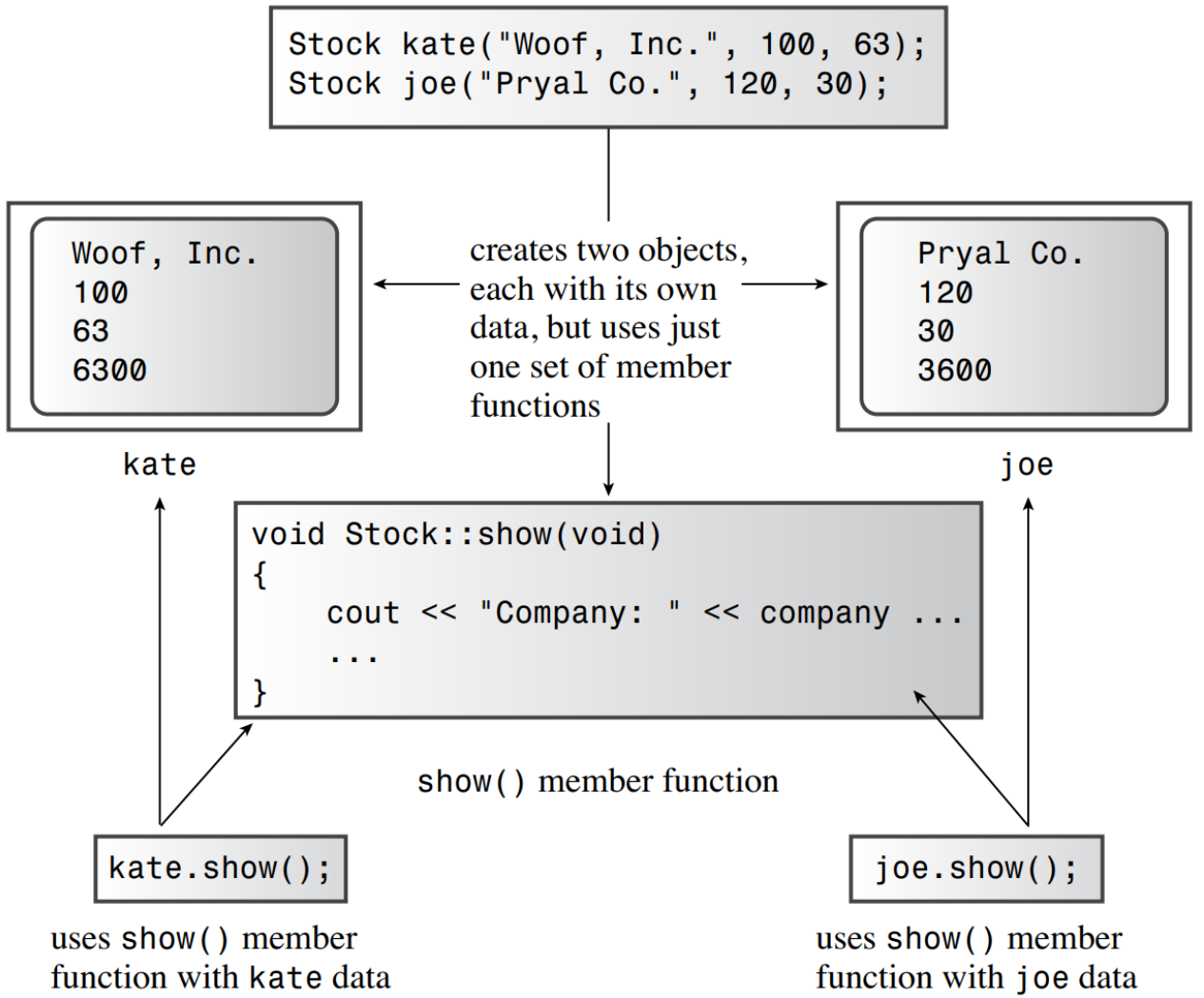
- 类方法应用于哪个对象？

  shares += num;

  - 使用的是哪个对象的 shares？

    ```
    Stock kate, joe;
    obj.show(); // show()中的成员属于 obj
    ```

- 类成员存储方式
  - 数据成员：各对象独立存储
  - 成员函数：所有对象共享同一份



2.4 使用类

P10.3, usestock0.cpp

- 引入类声明头文件
  - #include …
- 实例化对象
  - Stock fluffy_the_cat;
- 通过对象调用公有接口
  - fluffy_the_cat.show();

```
1. #include <iostream>
2. #include "01-stock00.h"

3. int main(){
4.     Stock fluffy_the_cat;
5.     fluffy_the_cat.acquire("Nano", 20, 12.50);
6.     fluffy_the_cat.show();
7.     fluffy_the_cat.buy(15, 18.125);
8.     fluffy_the_cat.show();
9.     fluffy_the_cat.sell(400, 20.00);
10.    fluffy_the_cat.show();
11.    fluffy_the_cat.buy(300000,40.125);
12.    fluffy_the_cat.show();
13.    fluffy_the_cat.sell(300000,0.125);
14.    fluffy_the_cat.show();
15.    return 0;
16.}
```

类比：服务器-客户端模型

- 服务器-客户端模型
  - 服务器：类声明（含类方法），提供可用资源
  - 客户：使用类的程序
  - 接口：客户与服务器交互的方式
- 交互机制
  - 客户只能通过公有接口使用服务器
    - 客户的责任：了解接口
  - 服务器的责任：按接口可靠执行，只改实现不改接口

2.5 修改实现

- 修改show的内部实现
- 对cout不做过多要求
  - `std::cout.precision(3); // save preceding value for precision`

2.6 小结

- 类设计的第一步：提供类声明
- 类设计的第二步：实现类成员函数

```
class className{
private:
  data member declarations
public:
  member function prototypes
};
```

类的构造函数和析构函数

3 类的构造函数和析构函数

- C++ 目标：使用类对象像使用标准类型一样
- 问题：类对象无内置初始化机制
  - 如：int year = 2001;（数据成员部分私有，无法直接初始化）
- 解决：构造函数
  - 专用于构建新对象并初始化数据成员的特殊成员函数
  - C++ 通过特定命名语法自动调用

3.1 声明和定义构造函数

- 构造函数定义
  - 函数名 = 类名，无返回值（连 void 也没有）
  - 可重载，参数由初始化需求决定
- 形参名不可与数据成员名相同，避免混淆
  - 常用命名区分：前缀 m_、后缀 _，或使用 this->

```
1.  class Stock{
2.  public:
3.      Stock(); // default constructor
4.      Stock(const std::string &c, long n = 0, double p
    = 0.0);
5.      ~Stock(); // noisy destructor
6.  };

7.  // constructors (verbose versions)
8.  Stock::Stock() // default constructor
9.  {
10.     std::cout << "Default constructor called\n";
11. }

12. Stock::Stock(const std::string &c, long n, double p)
13. {
14.     std::cout << "Constructor using " << c << "\n";
15. }
```

3.2 使用构造函数

- 构造函数调用（创建对象时自动调用）
  - 显式调用

```
Stock garment = Stock("Furry Mason", 50, 2.5);
```

  - 隐式调用

```
Stock garment("Furry Mason", 50, 2.5);
```

  - 动态创建（new）

```
Stock *pstock = new Stock("Electroshock Games", 18, 19.0);
```

- 注意：不能直接调用构造函数（必须通过创建对象）

```
Stock::Stock("Furry Mason", 50, 2.5); // 错误!
Stock *garment; // 没有创建对象
```

3.3 默认构造函数

- 未定义任何构造函数时，编译器自动提供默认构造函数：
  - 无参数，不做任何事
- 特别提醒
  - 只要定义了任意构造函数，编译器不再提供默认构造函数
- 良好习惯：定义类时显式提供默认构造函数

3.4 析构函数

- 析构函数：对象撤销/释放时自动调用的特殊成员函数，用于清理
- 形式：~类名()，无参数，无返回值
- 未提供则编译器自动补上默认析构函数
  - 若类中有动态内存申请，析构函数不可省略！

3.5 改进stock类

(P10.4, stock10.h, P10.5, stock10.cpp, P10.6, usestock.cpp)

- 增加了构造函数和析构函数
- 初始化 vs 赋值
  - 初始化：创建对象时赋初值
  - 赋值：已有对象后再赋值
- const 对象只能调用 const 方法

3.6 构造函数和析构函数小结

- 构造函数，在创建类对象时被调用
  - 用于初始化对象
  - 可以重载
  - 自动调用

- 析构函数，当对象被删除时被调用
  - 用于对象数据清理
  - 自动调用

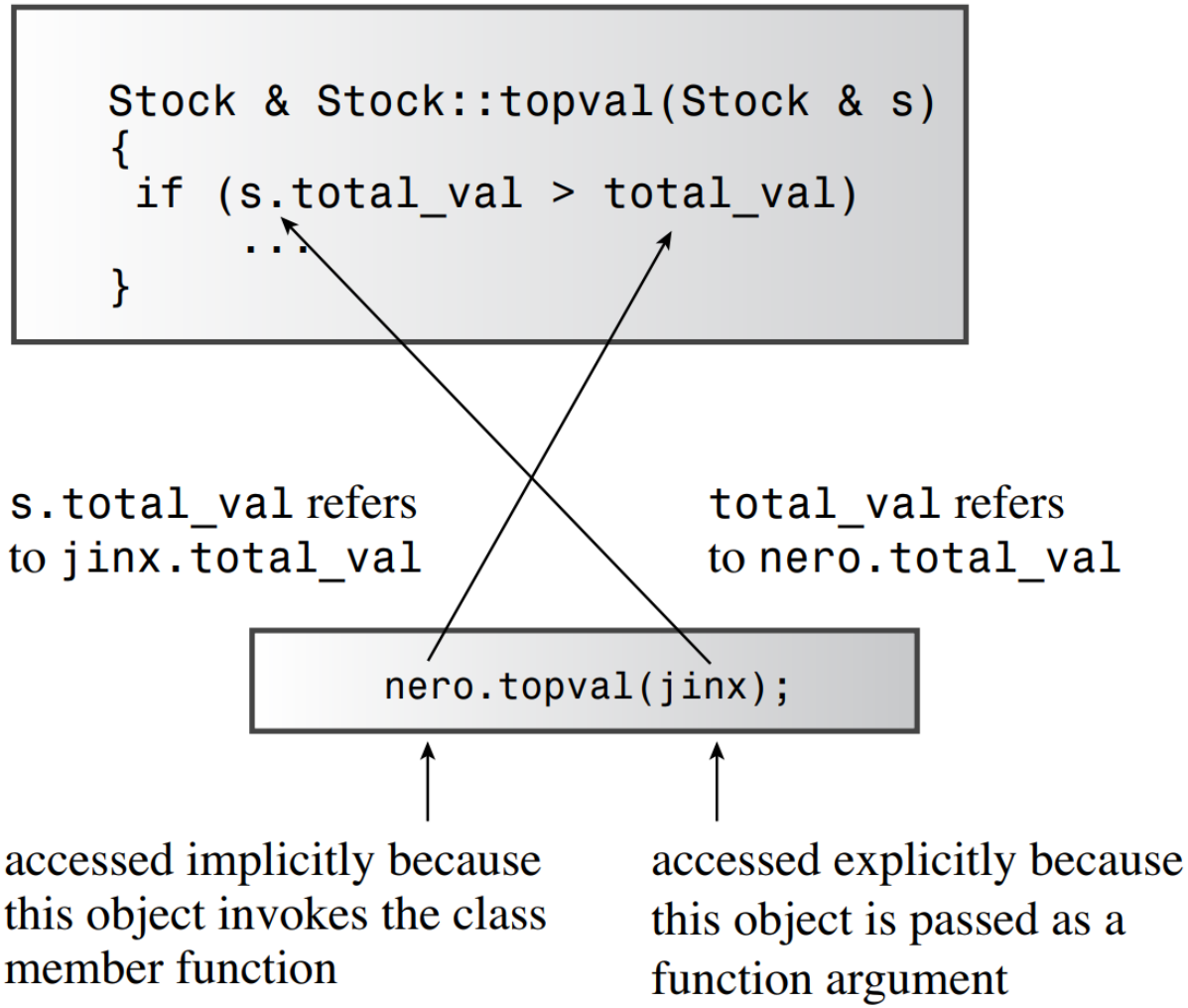
this指针

4 this指针

- 成员函数只涉及一个对象
  - 函数中的类成员均属于发起调用的对象

```
1.  top = stock1.topval(stock2);
2.  const Stock &Stock::topval(const Stock &s) const
3.  {
4.      if (s.total_val > total_val)
5.          return s;     // argument object
6.      else
7.          return ?????; // invoking object
8.  }
```

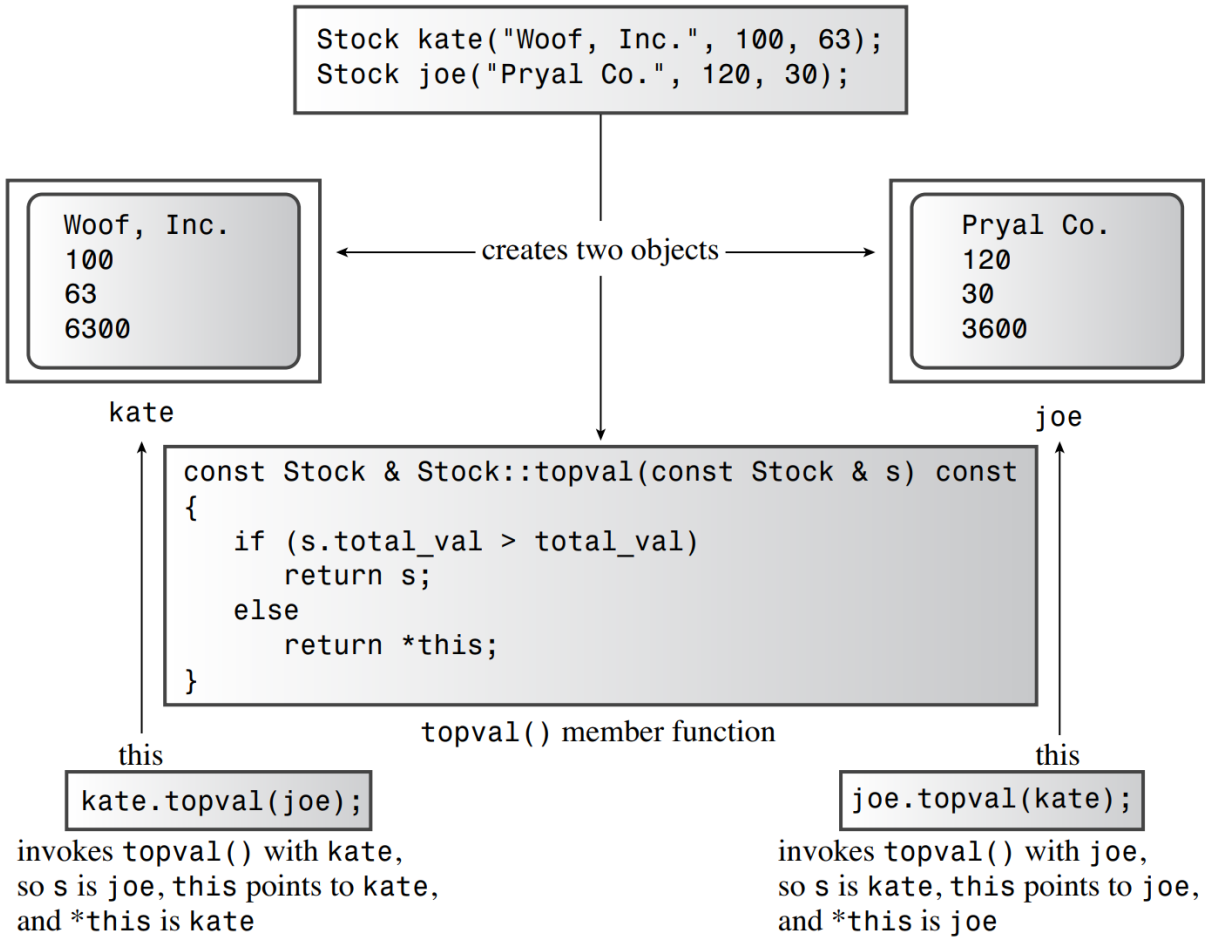
- 返回股价总值最高的对象
  - 涉及多个对象；topval内需返回调用对象(stock1)的引用
    - ?????表示发起调用的对象(stock1)的指针（类内不可见）



this指针

- C++提供了this指针机制
  - this 指针指向用来调用成员函数的对象
    - this被作为隐藏参数传递给方法
    - stockl.topval(stock2)将this设置为stockl对象的地址
  - 编译器在编译类成员函数时，自动为该函数添加一个隐含形参this
    - 在该类成员函数中，有关类成员的访问均通过this指针进行，即this->成员

```
1. const Stock &Stock::topval(const Stock &s) const
2. {
3.     if (s.total_val > this->total_val)
4.         return s; // argument object
5.     else
6.         return *this; // invoking object
7. }
```



对象数组

5 对象数组

声明对象数组的方法与声明标准类型数组相同(P10.9 usestock2.cpp)

- 数组初始化
  - 和一般数组声明类似
- 初始化
  - 必须为每个元素调用构造函数

```
1.  const int STKS = 4;
2.  int main(){
3.      // create an array of initialized objects
4.      Stock stocks[STKS] = {
5.          Stock("NanoSmart", 12, 20.0),
6.          Stock("Fleep Enterprises", 60, 6.5)
7.          };
8.      std::cout << "Stock holdings:\n";
9.      for (auto st = 0; st < STKS; st++)
10.         stocks[st].show();

11.    // set pointer to first element
12.     const Stock * top = &stocks[0];
13.     for (auto st = 1; st < STKS; st++)
14.         top = &top->topval(stocks[st]);

15.     return 0;
16. }
```

类作用域

6 类作用域

- 作用域：名称在文件中可见/使用的范围
  - 全局（文件）作用域：在所属文件任何地方可见
  - 局部（代码块）作用域：仅在所属代码块中可见
  - 函数名作用域：只能是全局的
- C++引入类作用域
  - 类中定义的名称（变量/函数），作用域为整个类
    - 类成员需带类名限定才有意义
  - 通过对象访问类成员：.（直接）、->（间接）、::（作用域解析）

```
1.  class Ik{
2.  private:
3.      int fuss; // fuss has class scope
4.  public:
5.      Ik(int f = 9) { fuss = f; } // fuss is in scope
6.      void ViewIk() const;  // ViewIk has class scope
7.  };
8.  void Ik::ViewIk() const // places ViewIk into Ik scope
9.  {
10.     cout << fuss << endl; //fuss in scope within class
11. }

12. int main()
13. {
14.     Ik *pik = new Ik;
15.     Ik ee = Ik(8);  // constructor in scope
16.     ee.ViewIk();    // brings ViewIk into scope
17.     pik->ViewIk();  // brings ViewIk into scope
18. }
```

6.1 作用域为类的常量

- 在类中
  - 不可以const
  - 可以enum
  - 可以static const int Months = 12;

```
1.  class Bakery{
2.  private:
3.      const int Months = 12;
4.      double costs[Months];
5.  };

6.  class Bakery{
7.  private:
8.      enum{Months = 12};
9.      double costs[Months];
10. };

11. class Bakery{
12. private:
13.     static const int Months = 12;
14.     double costs[Months];
15. };
```

6.2 作用域内枚举 - PASS

- 作用域内枚举

抽象和类

7 抽象数据类型

- 抽象数据类型(abstract data type, ADT)
  - 在抽象层次上，分析和描述数据类型
  - 可以认为是一种逻辑上的类型，和具体编程语言无关
  - 可以当成一种逻辑上的客观存在，与怎么用具体的方式描述无关
- 堆栈，应包含的操作
  - 可创建空栈
  - 可将数据项添加到堆顶(压入)
  - 可从栈顶删除数据项(弹出)
  - 可查看栈否填满。 可查看栈是否为空
- 然后，在具体的程序设计语言中实现堆栈

示例

- (P10.10 stack.h)
- 支持多类型可用typedef，但：
  - 一次只能支持一种类型
  - 换类型需重新编译
  - 更好方案：模板！
- 实现细节
  - 固定长度数组实现；极端情况：空时pop、满时push

```
1.  typedef unsigned long Item;
2.  class Stack{
3.  private:
4.      enum {MAX = 10};     // constant specific to class
5.      Item items[MAX];     // holds stack items
6.      int top;             // index for top stack item
7.  public:
8.      Stack();
9.      bool isempty() const;
10.     bool isfull() const;
11.     // push() returns false if stack already is full,
    true otherwise
12.     bool push(const Item & item); // add item to stack
13.     // pop() returns false if stack already is empty
14.     bool pop(Item & item);  // pop top into item
15. };
```

8 总结

- OOP方法：先描述数据与程序的接口，再设计类实现该接口
- 类声明分两部分：类声明（含函数原型）放.h，成员函数定义放方法文件
- 类是用户定义类型，对象是类的实例；各对象存储自己的数据，共享类方法
- 成员函数需操作多个对象时，可将额外对象作为参数传入
- 类适合描述ADT：公有成员函数提供接口服务，私有部分和方法实现对客户隐藏