chapter10 对象和类

1）类规范

【1】类声明：数据成员（描述数据部分）+ 成员函数（描述公有接口）
【2】类方法定义：描述如何实现类成员函数

ps： 何为 ”接口“?

接口是一个共享框架，供两个系统（如在计算机和打印机之间或者用户或计算机程序之间）交互时使用；例如，用户可能是您，而程序可能是字处理器。使用字处理器时，您不能直接将脑子中想到的词传输到计算机内存中，而必须同程序提供的接口交互。您敲打键盘时，计算机将字符显示到屏幕上；您移动鼠标时，计算机移动屏幕上的光标；您无意间单击鼠标时，计算机对您输入的段落进行奇怪的处理。程序接口将您的意图转换为存储在计算机中的具体信息。

对于类，我们说公共接口。在这里，公众（public）是使用类的程序，交互系统由类对象组成，而接口由编写类的人提供的方法组成。接口让程序员能够编写与类对象交互的代码，从而让程序能够使用类对象。例如，要计算 string 对象中包含多少个字符，您无需打开对象，而只需使用 string 类提供的 size( )方法。类设计禁止公共用户直接访问类，但公众可以使用方法 size( )。方法 size( )是用户和 string 类对象之间的公共接口的组成部分。通常，方法 getline( )是 istream 类的公共接口的组成部分，使用 cin 的程序不是直接与 cin 对象内部交互来读取一行输入，而是使用 getline( )。

如果希望更人性化，不要将使用类的程序视为公共用户，而将编写程序的人视为公共用户。然而，要使用某个类，必须了解其公共接口；要编写类，必须创建其公共接口。

2）类的基本成分

[1]基本示例：

```
// stock00.h -- Stock class interface
// version 00

#ifndef STOCK00_H_
#define STOCK00_H_
#include <string>
class Stock  // class declaration
{
private:
    std::string company;
    long shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    void acquire(const std::string & co, long n, double pr);
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show();
};    // note semicolon at the end
#endif
```

[2]详细介绍：
（1）class 指出这些代码给出了一个类设计
（2）要存储的数据以类数据成员 (如变量company...) 的形式出现
（3）要执行的操作以类函数成员 (方法，如 sell( ) 和 update( ))
（4）成员函数可以就地定义（ set_tot( ) ），也可以原型表示（ buy(...)；sell(...) ）

[3]访问控制：
（1）使用类对象的程序都可以直接访问公有部分
（2）但只能通过公有成员函数(或 友元函数)来访问对象的私有成员

> ps：类与结构
> 结构的默认访问类型是public，而类为private

3）实现类的成员函数

（1）定义成员函数（也叫作：方法）是用：作用域解析运算符(: :) 实现的

```
void Stock::update(double price)
[1] 意味着 update() 函数是 Stock类 的成员
[2] Stock类中的其他成员函数不必要使用 :: 就可以使用 update() 方法【这是因为它们属于同一个类，因此 update() 可见】
```

（2）方法 可以访问类的私有成员，但是非成员函数不可以（友元函数除外！）

```
void show() //是类的成员函数
{
        cout << company << shares << share_val << total_val <<endl;
}
```

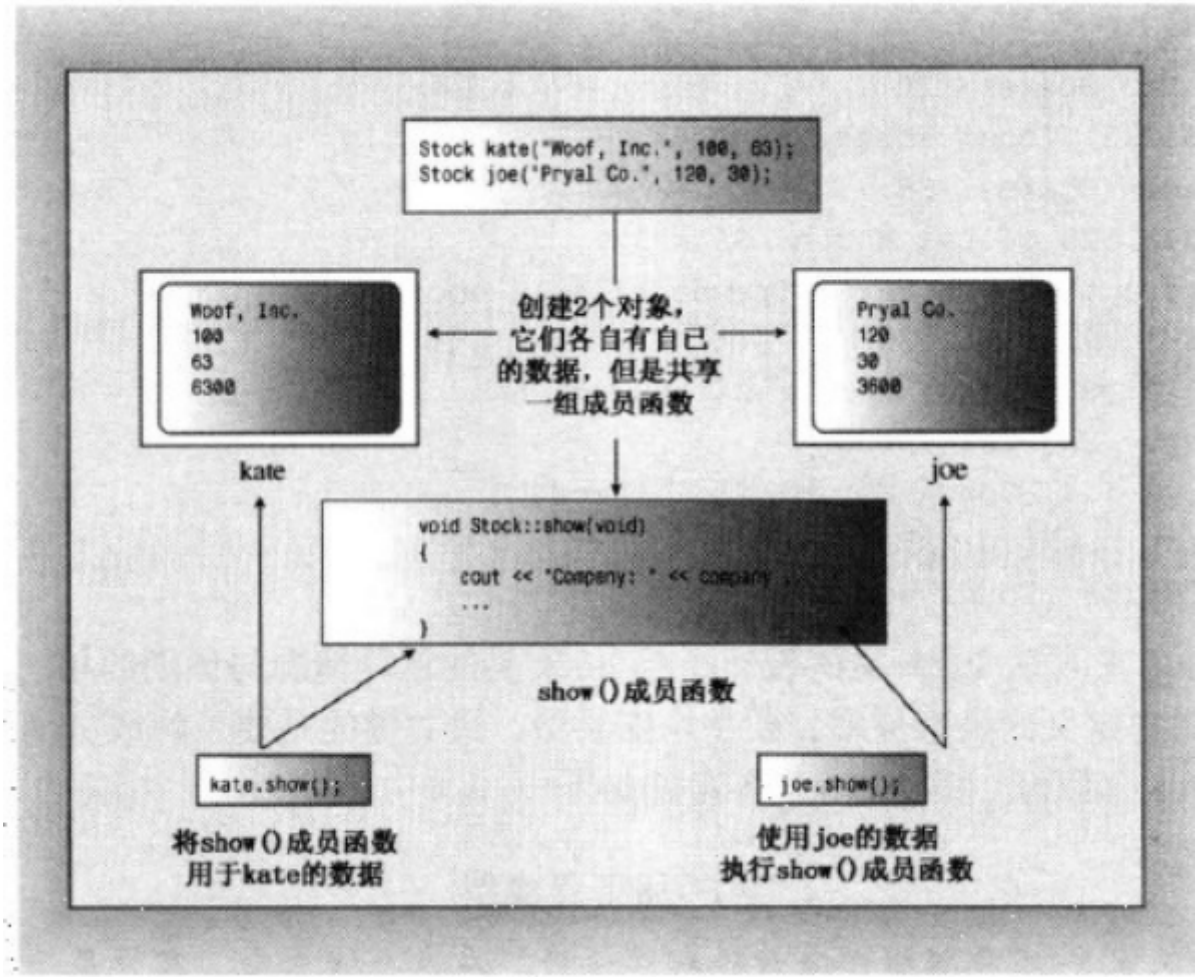
（3）内联方法
定义位于类声明的函数 都将自动成为 **内联函数**
eg：上述类Stock中的 Stock : : set_tot( )是一个内联函数

> ps : 如果愿意也可以在类声明之外定义成员函数，并使其成为内联函数
> 方式：在类实现部分定义函数时使用**inline**限定符即可

```
class Stock {
private:
        ...
        void set_tot() // declaration
public:
        ...
};

inline void Stock::set_tot() // use inline in definition
{
        total_val = shares*share_val;
}
```

（4）创建 and 使用对象
【1】所创建的每个新对象都有自己的存储空间，用于存储其内部变量和类成员
【2】同一个类的所有对象共享同一组类方法，即：每种方法只有一个副本
eg：kate 和 joe 都是Stock对象，则 kate.shares 占一个内存块， joe.shares 占另一个内存快。但是 kate.show( ) 和 joe.show( ) 都调用同一个方法，也就

是说： kate.show( ) 和 joe.show( ) 都执行同一个代码块，只是将这些代码方法用于不同的数据



4）类的构造函数和析构函数

【1】问题引入

```
class Stock {
        ......
};

struct node {
    char *pn;
    int m;
};

node amaoob = {"hhh",23};  // valid
Stock hbx = {"jjj",6};     // invalid
```

不能像上面这样初始化Stock对象，原因在于：类的数据部分访问状态是私有的，这意味着程序不能直接访问数据成员
因此需要设计合适的成员函数——构造函数

【2】构造函数格式

```
Stock::Stock(const string &co , long n = 0 , double pr = 0.0)
{
        company = co;
        ......
}
// 程序声明对象时，就将自动调用构造函数！
```

成员名和参数名

不熟悉构造函数的您会试图将类成员名称用作构造函数的参数名，如下所示：

```
// NO!
Stock::Stock(const string & company, long shares, double share_val)
{
...
}
```

这是错误的。构造函数的参数表示的不是类成员，而是赋给类成员的值。因此，参数名不能与类成员相同，否则最终的代码将是这样的：

```
shares = shares;
```

为避免这种混乱，一种常见的做法是在数据成员名中使用 m_前缀：

```
class Stock
{
private:
    string m_company;
    long m_shares;
    ...
```

另一种常见的做法是，在成员名中使用后缀_：

```
class Stock
{
private:
    string company_;
    long shares_;
    ...
```

无论采用哪种做法，都可在公有接口中在参数名中包含 company 和 shares。

【3】使用构造函数初始化对象

（1）显式地调用构造函数：

```
Stock food = Stock("world", 250, 1.1);
```

原理：允许调用构造函数来创建一个临时对象，然后将该临时对象复制到stock2中，并丢弃它【会为临时对象调用析构函数】

（2）隐式地调用构造函数：

```
Stock food ("world", 250, 1.1);
```

直接初始化

（*poi*）创建类对象 + new动态内存分配：

```
Stock *food = new Stock("world", 250, 1.1);

过程：
1）创建一个Stock对象，将其初始化为参数提供的值
2）并将该对象地址赋给food指针
```

【4】默认构造函数

方法1：给已有的构造函数的所有参数提供默认值

```
Stock(const string& co = "error" , int n = 0, double pr = 0.0);
```

方法2：函数重载定义另一个构造函数

```
Stock();

Stock::Stock()    // default constructor
{
        company = "no name";
        shares = 0;
        share_val = 0.0;
        total_val = 0.0;
}
```

eg:

```
Stock first;                // implicitly(隐式具体化)
Stock first = Stock();      // explicitly(显式具体化)
Stock *prelief = new Stock; // implicitly

Stock first("bcgsadkhcfa"); // calls constructor
Stock second;               // default constructor(默认构造函数)
```

【5】析构函数

（1）引入：析构函数负责完成清理工作
eg：如果构造函数使用new来分配内存，则析构函数将使用delete来释放这些内存

（2）格式：类名前加上~【跟构造函数一样，没有返回值和声明类型】

> poi：析构函数没有参数
> eg：Stock类的析构函数原型，必须是： ~Stock()

【6】综合示例分析

```
// stock1.cpp 
#include <iostream>
#include "stock10.h"
// constructors (verbose versions)
Stock::Stock()        // default constructor
{
    std::cout << "Default constructor called\n";
    company = "no name";
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}

Stock::Stock(const std::string & co, long n, double pr)
{
    std::cout << "Constructor using " << co << " called\n";
    company = co;
    if (n < 0)
    {
        std::cout << "Number of shares can't be negative; "
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}

// class destructor
Stock::~Stock()        // verbose class destructor
{
    std::cout << "Bye, " << company << "!\n";
}

// other methods
void Stock::buy(long num, double price)
```

```
{
     if (num < 0)
    {
        std::cout << "Number of shares purchased can't be negative. "
             << "Transaction is aborted.\n";
    }
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}

void Stock::sell(long num, double price)
{
    using std::cout;
    if (num < 0)
    {
        cout << "Number of shares sold can't be negative. "
             << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        cout << "You can't sell more than you have! "
             << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show()
{
    using std::cout;
    using std::ios_base;
    // set format to #.###
    ios_base::fmtflags orig =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    std::streamsize prec = cout.precision(3);
    cout << "Company: " << company
        << "  Shares: " << shares << '\n';
    cout << "  Share Price: $" << share_val;
    // set format to #.##
    cout.precision(2);
    cout << "  Total Worth: $" << total_val << '\n';
    // restore original format
    cout.setf(orig, ios_base::floatfield);
    cout.precision(prec);
}
```

```
// usestok1.cpp -- using the Stock class
// compile with stock10.cpp
#include <iostream>
#include "stock10.h"
int main()
{
  {
    using std::cout;
    cout << "Using constructors to create new objects\n";
    Stock stock1("NanoSmart", 12, 20.0);            // syntax 1
    stock1.show();
    Stock stock2 = Stock ("Boffo Objects", 2, 2.0); // syntax 2
```

```
    stock2.show();

    cout << "Assigning stock1 to stock2:\n";
    stock2 = stock1;
    cout << "Listing stock1 and stock2:\n";
    stock1.show();
    stock2.show();

    cout << "Using a constructor to reset an object\n";
    stock1 = Stock("Nifty Foods", 10, 50.0);    // temp object
    cout << "Revised stock1:\n";
    stock1.show();
    cout << "Done\n";
  }
    // std::cin.get();
    return 0;
}
```

说明：
1）可以将一个对象赋给另一个对象：

```
stock2 = stock1;
```

> **默认情况下，给类对象赋值时，将把一个对象复制给另一个**
>
> > 在上面这个例子当中，stock2的原内容将被覆盖！

2）构造函数不仅可用于新对象，也可以用于旧对象的新值赋予

```
stock1 = Stock("hhh", 10, 50.0);

stock1已存在，这句语句不是对stock1初始化，而是将新值赋给它
原理：让构造函数创建一个新的、临时的对象，然后将内容 复制 给stock1来实现，随后程序调用析构函数，以删除该临时对象
```

3）由函数栈原理：局部变量消失的顺序是：LIFO（自动变量被放在栈中，最后创建的对象最先被删除）

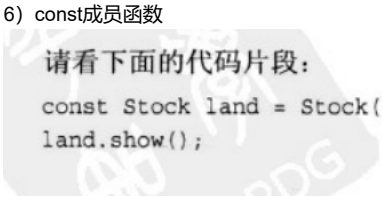
4）区分初始化 与 赋值：

```
Stock hbx2 = Stock("jjj", 2, 2.0);  // 初始化，它创建有指定值的对象  有可能会产生临时变量

hbx1 = Stock("jjj",3,5.0);          // 赋值，在赋值语句中使用构造函数，必然会产生一个临时对象
```

5）对类对象使用 初始化列表
举例说明即可

```
对于构造函数：
Stock::Stock(const std::string &co , long n = 0 , double pr = 0.0);

初始化：
Stock hop = {"cbhgkjajdgbs" , 2 , 2.0};   =>   hop("cbhgkjajdgbs" , 2 , 2.0);
Stock jum {"dshakuhas" , 5};              =>   jum("dshakuhas" , 5 , 0.0);
Stock lll {"dshakuhas"};                  =>   jum("dshakuhas", 0 , 0.0);
Stock jjj{};                              =>   将与默认构造函数匹配！
```

6）const成员函数



对于当前的 C++来说，编译器将拒绝第二行。这是什么原因呢？因为 show( )的代码无法确保调用对象不被修改——调用对象和 const 一样，不应被修改。我们以前通过将函数参数声明为 const 引用或指向 const 的指针来解决这种问题。但这里存在语法问题：show( )方法没有任何参数。相反，它所使用的对象是由方法调用隐式地提供的。需要一种新的语法——保证函数不会修改调用对象。C++的解决方法是将 const 关键字放在函数的括号后面。也就是说，show( )声明应像这样：

```
void show() const;          // promises not to change invoking object
```

同样，函数定义的开头应像这样：

```
void stock::show() const    // promises not to change invoking object
```

以这种方式声明和定义的类函数被称为 const 成员函数。就像应尽可能将 const 引用和指针用作函数形参一样，只要类方法不修改调用对象，就应将其声明为 const。从现在开始，我们将遵守这一规则。

5）this指针

【1】问题引入：

> 定义一个成员函数，它查看两个Stock对象，并返回股价较高的那个对象的引用
> 假设将该方法命名为 topval() 则函数调用stock1.topval( ) 将访问stock1对象的数据
> 如果希望该方法能对两者进行比较，则应该将第二个对象作为参数传递给它
> 出于效率，按照引用传递参数！因此：topval( ) 方法将使用一个类型为 const Stock& 的参数

```
写法：const Stock& topval (const Stock& s) const;

(1)括号内的const：该函数不会修改被显式访问的对象
(2)括号后的const：该函数不会修改被隐式访问的对象
(3)返回值类型的const：由于该函数返回了2个const对象之一的引用，因此返回类型也应该是const引用
```

> 假设要对stock1与stock2进行比较，可以采用：

```
(方法1) top = stock1.topval(stock2); // 显式访问stock2，隐式访问stock1
(方法2) top = stock2.topval(stock1); // 显式访问stock1，隐式访问stock2
```

> 现在问题又来了，注意topval()的实现：

```
const Stock& Stock::topval(const Stock& s) const
{
        if(s.total_val > total_val) return s;
        else return ?????;
}

[1] s.total_val是 作为参数的传递对象 的总值；total_val是 用来调用该方法的对象 的总值
[2] 问题在于如何称呼 ?????
```

【2】解决方法 及 知识引入：

> this指针：指向用来调用成员函数的对象
> 所有的类方法都将this指针设置为 调用它的对象 的地址

eg：
这样的话，stock1.topval (stock2); 中，将this设置为stock1对象的地址，使得这个指针可用于 topval() 方法
函数中的 total_val 也只是 this->total_val 的简写

> 每个成员函数（包括构造函数与析构函数）都有一个this指针，指向调用对象
> （1）要引用整个调用对象：使用表达式 ***this***
> （2）函数后面的const将this限定成const，这样的话就不能使用this来修改对象的值

```
const Stock& Stock::topval(const Stock& s) const
{
        if(s.total_val > total_val) return s;
        else return *this*;
}
```

6）对象数组

方案：
（1）使用默认构造函数创建数组元素
（2）花括号内的构造函数将创建临时对象【数组对象的构造函数将包括对每个元素单独调用的构造函数】
（3）将临时对象中的内容复制到相应的元素中
因此想要创建 类对象数组 ， 则这个类必须要有默认构造函数

```
Stock hbx[4];-
-------------------------------------------------
hbx[0].update();
hbx[3].show();
const Stock& tops = hbx[2].topval( hbx[1] );
-------------------------------------------------
const int STKS = 4;
Stock stocks[STKS] = {
        Stock("jhdsjds",12.5,20),
        Stock("asdnlj",200,2.0),
        Stock("adafs",130,1.25),
        Stock("jhfgjfncg",60,6.5)
};
```

7）类作用域

【1】在类中定义的名称（类数据成员名 and 类成员函数名）的作用域为整个类，它们只在该类中是已知的，在类外是不可知的

【2】类作用域意味着不能从外部直接访问类的成员。要想调用公有成员函数，必须通过对象

```
Stock sleeper("assad",100,0.25);  // create object
sleeper.show();  // use object to invoke a member_function
show();  // invalid !  ---cannot call method directly
```

【3】定义成员函数时，必须使用类作用域解析运算符

```
void Stock::update(double price)
{
        ...
}
```

【4】在类声明or成员函数定义中，可以使用未修饰的成员名称 [因为名称的作用域在类内，这几个一家亲]

```
class IK
{
private:
        int fuss;
public:
    IK(int f = 9) {fuss = f;}  //类内直接定义成员函数
    void ViewIK() const;       //类外定义成员函数【在类内只声明！】
};

void IK::ViewIK() const        //类外定义成员函数【要加类作用域解析符】
{
        cout << fuss <<endl;
}
......

int main()
{
        ......
}
```

【5】如何在类中创建一个由所有对象共享的常量?
（1）问题引入：

```
class Bakery
{
private:
        const int M = 12;
```

```
        double costs[M];
......
}
```

这样不行！因为声明类只是描述了对象的形式，并没有创建对象。因此在创建对象之前，将没有用于存储值的空间！

（2）问题解决：

```
class Bakery
{
private:
        static const int M = 12;
        double costs[M];
......
}

这将创建一个名为 M 的变量，该变量将与其他静态变量存储在一起，而不是存储在对象中
因此这个 M 常量 将被所有Bakery对象共享
```

8）抽象数据类型（ADT）

以stack作例子：

```
// stack.h -- class definition for the stack ADT
#ifndef STACK_H_
#define STACK_H_
typedef unsigned long Item;
class Stack
{
private:
    enum {MAX = 10};    // constant specific to class
    Item items[MAX];    // holds stack items
    int top;            // index for top stack item
public:
    Stack();
    bool isempty() const;
    bool isfull() const;
    // push() returns false if stack already is full, true otherwise
    bool push(const Item & item);   // add item to stack
    // pop() returns false if stack already is empty, true otherwise
    bool pop(Item & item);          // pop top into item
};
#endif
```

```
// stack.cpp -- Stack member functions
#include "stack.h"
Stack::Stack()    // create an empty stack
{
    top = 0;
}

bool Stack::isempty() const
{
    return top == 0;
}

bool Stack::isfull() const
{
    return top == MAX;
}

bool Stack::push(const Item & item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
```

```
        return false;
}

bool Stack::pop(Item & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
```

```
// stacker.cpp -- testing the Stack class
#include <iostream>
#include <cctype>  // or ctype.h
#include "stack.h"
int main()
{
    using namespace std;
    Stack st; // create an empty stack
    char ch;
    unsigned long po;
    cout << "Please enter A to add a purchase order,\n"
        << "P to process a PO, or Q to quit.\n";
    while (cin >> ch && toupper(ch) != 'Q')
    {
        while (cin.get() != '\n')
            continue;
        if (!isalpha(ch))
        {
            cout << '\a';
            continue;
        }

        switch(ch)
        {
             case 'A':
             case 'a': cout << "Enter a PO number to add: ";
                       cin >> po;
                       if (st.isfull())
                           cout << "stack already full\n";
                       else
                           st.push(po);
                       break;
             case 'P':
             case 'p': if (st.isempty())
                           cout << "stack already empty\n";
                       else {
                           st.pop(po);
                           cout << "PO #" << po << " popped\n";
                       }
                       break;
        }
        cout << "Please enter A to add a purchase order,\n"
             << "P to process a PO, or Q to quit.\n";
    }
    cout << "Bye\n";
    return 0;
}
```