**Chapter 10 Objects and Classes**

Typically, C++ programmers place the interface, in the form of a class definition, in a header file and place the implementation, in the form of code for the class methods, in a source code file.

First Stock Example

// stock00.h -- Stock class interface

// version 00

**#ifndef** STOCK00\_H\_

**#define** STOCK00\_H\_

#include <string>

**class** Stock

{

**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**

A member function can be **defined in place**, for example, *set\_tot()*, or it can be represented by a **prototype**, like the other member functions in this class. The full definitions for the other member functions come later in the implementation file, but the prototypes suffice to describe the function interfaces.

**Access Control**

Also new are the keywords ***private*** and ***public***. These labels describe **access control** for class members. Any program that uses an object of a particular class can access the public portions directly. A program can access the **private members of an object only by using the public member functions, or via a friend function**. Thus, the public member functions act as go-betweens between a program and an object’s private members. This insulation of data from direct access by a program is called data hiding.

**You don’t have to use the keyword private** in class declarations because that is the default access control for class objects:

class World

{

float mass; // private by default

char name[20]; // private by default

public:

void tellall(void);

...

};

The only difference between *structure* and *class* is that the **default access type** for a **structure** is **public**, whereas the default type for a **class** is **private**.

**Implementing Class Member Functions**

When you define a member function, you use the **scope-resolution operator (::)** to identify the class to which the function belongs. Class methods can access the private components of the class.

void Stock::update(double price)

This notation means you are defining the update() function that is a member of the Stock class. Not only does this identify *update()* as a member function, it means you can use the same name for a member function for a different class.

Other member functions of the Stock class can, if necessary, use the update() method without using the scope-resolution operator. That’s because they belong to the same class, making update() in scope. Using update() **outside the class** declaration and method definitions, however, requires special measures, which we’ll get to soon.

// **stock00.cpp** -- implementing the Stock class

// version 00

#include **<iostream>**

**#include "stock00.h"**

void Stock::**acquire**(const std::string & co, long n, double pr) // acquire() : buy a new stock

{

**company = co**; // string = string

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();**

}

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**()

{

std::cout << "Company: " << company

<< " Shares: " << shares << ‘\n’

<< " Share Price: $" << share\_val

<< " Total Worth: $" << **total\_val** << ‘\n’;

}

The *acquire()* function manages the first acquisition of stock for a given company, whereas *buy()* and *sell()* manage adding to or subtracting from an existing holding.

**Inline Methods**

Any **function** with a definition **in the class declaration** automatically becomes an **inline** function. Thus, *Stock::set\_tot()* is an inline function.

You can, if you like, define a member function **outside the class** declaration and still **make it *inline***.

class Stock

{

private:

...

**void set\_tot();** // definition kept separate

public:

...

};

**inline** void Stock::set\_tot() // use inline in definition, **注意这个函数没有参数**

{

total\_val = shares \* share\_val;

}

**The special rules for inline functions** require that they be defined in each file in which they are used. The easiest way to make sure that inline definitions are available to all files in a multi-file program is to **include the inline definition in the same header file in which the corresponding class is defined**.

Incidentally, according to the ***rewrite rule***, defining a method within a class declaration is equivalent to replacing the method definition with a prototype and then rewriting the definition as an inline function immediately after the class declaration.

**Which Object Does a Method Use?**

Stock kate, joe;

This creates two objects of the Stock class, one named *kate* and one named *joe*. Each new object you create contains **storage for its own internal variables**, **the class members**. **But all objects of the same class share the same set of class methods, with just one copy of each method**. Suppose, for example, that kate and joe are Stock objects. In that case, kate.shares occupies one chunk of memory, and joe.shares occupies a second chunk of memory. **But kate.show() and joe.show() both invoke the same method, that is, both execute the same block of code**. They just apply the code to different data.

**Using Classes**

// **usestck0.cpp** -- the client program

// compile with stock00.cpp

#include <iostream>

**#include "stock00.h"**

int main()

{

**Stock fluffy\_the\_cat**;

fluffy\_the\_cat.**acquire("NanoSmart", 20, 12.50);**

fluffy\_the\_cat.show();

fluffy\_the\_cat.buy(15, 18.125);

fluffy\_the\_cat.show();

fluffy\_the\_cat.sell(400, 20.00);

fluffy\_the\_cat.show();

fluffy\_the\_cat.buy(300000,40.125);

fluffy\_the\_cat.show();

fluffy\_the\_cat.sell(300000,0.125);

fluffy\_the\_cat.show();

return 0;

}

**Changing the Implementation**

The ostream class included member functions that control formatting. Without going into much detail, you can avoid e-notation by using the setf() method much as we did in Listing 8.8:

std::cout.setf(std::ios\_base::fixed, std::ios\_base::floatfield);

This sets a flag in the cout object instructing **cout to use fixed-point notation**. Similarly, the following statement causes cout to **show three places to the right of the decimal when using fixed-point notation**:

std::cout.precision(3);

When you change the implementation for a method, the changes should not affect other parts of the client program. The format changes just mentioned stay in place until changed again, so they could affect subsequent output in the client program. Therefore, the polite thing for show() to do is to reset the formatting information to the state that existed before show() was called. This can be done, as in Listing 8.8, using return values for the setting statements:

std::streamsize prec =

std::cout.precision(3); // save preceding value for precision

...

std::cout.precision(prec); // **reset to old value**

// **store original flags**

std::ios\_base::**fmtflags** orig = std::cout.setf(std::ios\_base::fixed);

...

// reset to stored values

std::cout.setf(orig, std::ios\_base::floatfield);

As you may recall, *fmtflags* is a type defined in the *ios\_base* class, which is defined in the std namespace, hence the rather long type name for orig. Second, ***orig* holds all the flags, and the reset statement uses that information to reset information in the floatfield section**, which includes flags for fixed-point notation and scientific notation. Third, let’s not worry too much about the details here. The main points are that the changes are confined to the implementation file and that the changes don’t affect other aspects of the program using the class.

Putting this information to use, we can replace the show() definition in the implementation file with this:

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);

}

程序中间加下划线那两行既设置又保存.

* **Class Constructors and Destructors**

One of C++’s aims is to make using class objects similar to using standard types. However, the code provided so far in this chapter doesn’t let you initialize a Stock object the way you can an ordinary int or structure. That is, **the usual initialization syntax doesn’t carry over for the Stock type:**

int year = 2001; // valid initialization

struct thing

{

char \* pn;

int m;

};

thing amabob = {"wodget", -23}; **// valid initialization**

Stock hot = {"Sukie’s Autos, Inc.", 200, 50.25}; **// NO! compile error**

**The reason you can’t initialize a Stock object this way** is because the **data parts have private access status**, which means a **program cannot access the data members directly**.

The only way a program can access the data members is through a member function. Therefore, you need to devise an appropriate member function if you’re to succeed in initializing an object.

C++ provides for special member functions, called **class constructors**, especially for constructing new objects and assigning default values to their data members. **The name is the same as the class name**. For example, a possible constructor for the Stock class is a member function called Stock(). The constructor prototype and header have an interesting property: Although the **constructor has no return value**, **it’s not declared type void. In fact, a constructor has no declared type.**

* **Declaring and Defining Constructors**

Because a Stock object has three values to be provided from the outside world, you should give the constructor three arguments. (The fourth value, the total\_val member, is calculated from shares and share\_val, so you don’t have to provide it to the constructor.) You may want to provide just the company member value and set the other values to zero; you can do this by using default arguments (see Chapter 8,“Adventures in Functions.”). Thus, the prototype would look like this:

// constructor prototype with some default arguments

Stock(const string & co, long n = 0, double pr = 0.0);

Note that there is no return type. The prototype goes in the public section of the class declaration.

// constructor definition

Stock::Stock(const string & **co**, long **n**, double **pr**)

{

company = co;

if (n < 0)

{

std::cerr << "Number of shares can’t be negative; "

<< company << " shares set to 0.\n";

shares = 0;

}

else

shares = n;

share\_val = pr;

**set\_tot();**

}

This is the same code that the *acquire()* function used earlier in this chapter. **The difference is that in this case, a program *automatically invokes the constructor* *when it declares an object*.**

**Member Names and Parameter Names**

Often those new to constructors try to use the class member names as parameter names in the constructor, as in this example:

**// NO!**

Stock::Stock(const string & **company**, long **shares**, double **share\_val**)

{

}

This is wrong. The constructor arguments don’t represent the class members; they represent values that are assigned to the class members. Thus, they must have distinct names, or you end up with confusing code like this:

shares = shares;

One common coding practice to help avoid such confusion is to use an m\_ prefix to identify data member names:

class Stock

{

private:

string m\_company;

long m\_shares;

...

Another common practice is to use an underbar suffix for member names:

class Stock

{

private:

string company\_;

long shares\_;

...

With either convention, you then can use company and shares as the parameter names in the public interface.

**Using Constructors**

C++ provides two ways to initialize an object by using a constructor.

**The first** is to call the constructor explicitly:

Stock food = **Stock**("World Cabbage", 250, 1.25);

**The second** way is to call the constructor implicitly:

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

C++ uses a class constructor whenever you create an object of that class, even when you use new for dynamic memory allocation. Here’s how to use the constructor with new:

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

This statement creates a Stock object, initializes it to the values provided by the arguments, and assigns the address of the object to the pstock pointer. In this case, the object doesn’t have a name, but you can use the pointer to manage the object.

Constructors are used differently from the other class methods. Normally, you use an object to invoke a method:

stock1.show(); // stock1 object invokes show() method

However, **you can’t use an object to invoke a constructor because until the constructor finishes its work of making the object, there is no object.** Rather than being invoked by an object, the constructor is used to create the object.

**Default Constructors**

A default constructor is a constructor that is used to create an object when you don’t provide explicit initialization values. That is, it’s a constructor used for declarations like this:

Stock fluffy\_the\_cat; // uses the default constructor

The reason this statement works is that if you fail to provide any constructors, C++ automatically supplies a default constructor. It’s an implicit version of a default constructor, and **it does nothing**. For the Stock class, the default constructor would look like this:

Stock::Stock() { }

The fact that the default constructor has no arguments reflects the fact that no values appear in the declaration.

If you provide a non-default constructor, such as Stock(const string & co, long n, double pr), and don’t provide your own version of a default constructor, then **a declaration like this** becomes an error:

Stock stock1; // not possible with current constructor

The reason for this behavior is that you might want to make it impossible to create uninitialized objects. If, however, you wish to create objects without explicit initialization, you must define your own default constructor. This is a constructor that takes no arguments.

意思是说，**如果你显式的提供了一个非默认的constructor给成员赋值，然后你又想用Stock stock1这种形式去创建一个未被初始化的实例，这是不允许的**. 因为你定义了一个显式的constructor，可能就是想禁止默认非初始化的实例创建. 这时，如果你想用非初始化方式创建实例，你需要再定义一个default constructor.

You can define a default constructor two ways.

**One is** to provide default values for ***all*** the arguments to the existing constructor:

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

**The second is** to use function overloading to define a second constructor, one that has no arguments:

Stock();

Thus, a user-provided default constructor typically provides implicit initialization for all member values. For example, this is how you might define one for the Stock class:

Stock::Stock() // default constructor

{

company = "no name";

shares = 0;

share\_val = 0.0;

total\_val = 0.0;

}

When you design a class, you should usually provide a default constructor that implicitly initializes all class members.

After you’ve used either method (no arguments or default values for all arguments) to create the default constructor, you can declare object variables without initializing them explicitly:

Stock first; // calls default constructor implicitly

Stock first = **Stock(); // calls it explicitly**

Stock \*prelief = new Stock; // calls it implicitly

However, you shouldn’t be misled by the implicit form of the non-default constructor:

Stock first("Concrete Conglomerate"); // calls constructor

Stock second(); // declares a function

Stock third; // calls **default constructor**

**The first declaration here calls the non-default constructor—that is, the one that takes arguments.** **这里并没有指定其他两个参数, 这是可以的**. The second declaration states that second() is a function that **returns a Stock object**. When you implicitly call the default constructor, you don’t use parentheses.

* **Destructors**

When you use a constructor to create an object, the program undertakes the responsibility of tracking that object until it expires. At that time, the program automatically calls a special member function bearing the formidable title ***destructor***. For example, if your constructor uses *new* to allocate memory, **the destructor should use delete to free that memory**. But if the constructor doesn’t do anything that fancy, **you can simply *let the compiler* generate an implicit, do-nothing destructor**, which is exactly what the first version of the Stock class does.

Like a constructor, a destructor has a special name: It is formed from the class name preceded by a tilde (~). Thus, the destructor for the Stock class is called *~Stock()*. Also like a constructor, *a destructor can have* ***no return value*** *and has* ***no declared type***. Unlike a constructor, a destructor **must have no arguments**. Thus, the prototype for a Stock destructor must be this:

~Stock();

Stock::~Stock() // class destructor

{

cout << "Bye, " << company << "!\n";

}

**When should a destructor be called?** The *compiler handles* this decision; **normally your code shouldn’t explicitly call a destructor**.

1. If you create a ***static storage class object***, its destructor is called automatically when the **program terminates**;
2. If you create an ***automatic storage class object***, as the examples have been doing, its destructor is called automatically **when the program exits the block of code in which the object is defined**;
3. If the object is created by using ***new***, it resides in heap memory, or the free store, and its destructor is **called automatically when you use delete to free the memory**;
4. Finally, a program can create ***temporary objects*** to carry out certain operations; in that case, the program automatically calls the destructor for the object **when it has finished using it**.

**Improving the *Stock* Class**

// **stock10.h** -- Stock class declaration with constructors, destructor added

#ifndef STOCK10\_H\_

#define STOCK01\_H\_

**#include <string>**

class Stock

{

private:

**std::string company; // without std:: will generate compiler error**

long shares;

double share\_val;

double total\_val;

void set\_tot() { total\_val = shares \* share\_val; }

public:

// two constructors

Stock(); // default constructor

Stock(const std::string & co, long n = 0, double pr = 0.0);

~Stock(); // noisy destructor

void buy(long num, double price);

void sell(long num, double price);

void update(double price);

void show();

};

#endif

This header file dispenses with the *acquire()* function, which is no longer necessary now that the class has constructors.

// stock10.cpp -- Stock class with constructors, destructor added

#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; // Here, and above, shows 2 ways to deal with 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);

}

主程序在下面：

// 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";

return 0;

}

自己机器上测试结果如下：

momo@HMI:~/C++PrimerPlus/Chapter10/UseStock1$ ./a.out

Using constructors to create new objects

**Constructor using NanoSmart called**

Company: **NanoSmart** Shares: 12

Share Price: $20.000 Total Worth: $240.00

**Constructor using Boffo Objects called**

Company: **Boffo Objects** Shares: 2

Share Price: $2.000 Total Worth: $4.00

**Assigning stock1 to stock2:**

Listing stock1 and stock2:

Company: NanoSmart Shares: 12

Share Price: $20.000 Total Worth: $240.00

Company: NanoSmart Shares: 12

Share Price: $20.000 Total Worth: $240.00

**Using a constructor to reset an object**

Constructor using Nifty Foods called

**Bye, Nifty Foods!**

Revised stock1:

Company: Nifty Foods Shares: 10

Share Price: $50.000 Total Worth: $500.00

Done

**Bye, NanoSmart!**

**Bye, Nifty Foods!**

momo@HMI:~/C++PrimerPlus/Chapter10/UseStock1$

这个程序先用Stock stock1("NanoSmart", 12, 20.0)创建并初始化了stock1, 然后用Stock stock2 = Stock ("Boffo Objects", 2, 2.0) 创建并初始化了stock2, **注意在创建并初始化一个class时对constructor的这两种用法, 并注意他们是不同的.** 后者可能会创建一个临时变量, 赋值完毕后再discard. 这里并没有, 因为stock2这是还不存在.

然后stock2 = stock1. **注意这里并没有激发constructor**, as with structure assignment, class object assignment, **by default**, copies the members of one object to the other. In this case, the original contents of stock2 are overwritten.

然后, **stock1 = Stock("Nifty Foods", 10, 50.0)**对stock1重新赋值. ***The stock1 object already exists***. **Therefore, instead of initializing stock1**, this statement assigns new values to the object. It does so by **having the constructor create a *new, temporary object* and then copying the contents of the new object to stock1**. Then the program disposes of the temporary object, invoking the destructor as it does so, as illustrated by the following annotated output:

**Bye, Nifty Foods!**

注意, ***这里不能用stock1.Stock("Nifty Foods", 10, 50.0)形式***, 因为**classname.classname()形式是专门用来初始化一个尚不存在的class的, 而直接用 class = class()只是赋值, 只不过这种赋值通过创建临时对象来实现.**

Finally, at the end, the program displays this:

Done

Bye, NanoSmart!

Bye, Nifty Foods!

When the main() function terminates, its local variables (stock1 and stock2) expires, and its destructor is called. Because such automatic variables go on the stack, the last object created is the first deleted, and the first created is the last deleted.

***The output points out that there is a fundamental difference between the following two statements:***

Stock stock2 = Stock ("Boffo Objects", 2, 2.0);

stock1 = Stock("Nifty Foods", 10, 50.0); // temporary object

**The first** of these statements invokes **initialization**; it *creates an object* with the indicated value, and it may or may not create a temporary object. **The second** statement invokes **assignment**. Using a constructor in an assignment statement in this fashion always causes the creation of a temporary object before assignment occurs.

**C++11 List Initialization**

With C++11, can you use the list-initialization syntax below:

Stock hot\_tip = {"Derivatives Plus Plus", 100, 45.0};

Stock jock {"Sport Age Storage, Inc"};

Stock temp {};

The braced lists in the first two declarations match the following constructor:

Stock::Stock(const std::string & co, long n = 0, double pr = 0.0);

For jock, the default values of 0 and 0.0 will be used for the second and third arguments. The third declaration matches the default constructor, so temp is constructed using it.

**const Member Functions**

const Stock land = Stock("Kludgehorn Properties");

land.show();

With current C++, the compiler should object to the second line. Why? **Because the code for show() fails to guarantee that it won’t modify the invoking object, which, because it is const, should not be altered**.

The *show()* method doesn’t have any arguments for const to qualify. Instead, **the object it uses is provided implicitly by the method invocation**. What is needed is a new syntax, one that says a function promises not **to modify the invoking object**. The C++ solution is to **place the const keyword after the function parentheses**. That is, the *show()* declaration should look like this:

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

Similarly, the beginning of the function definition should look like this:

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

**Constructors and Destructors in Review**

For example, suppose the Bozo class has the following prototype for a class constructor:

Bozo(const char \* fname, const char \* lname); // constructor prototype

In this case, you can use it to initialize new objects as follows:

Bozo bozetta = bozo("Bozetta", "Biggens"); // primary form

Bozo fufu("Fufu", "O’Dweeb"); // short form

Bozo \*pc = new Bozo("Popo", "Le Peu"); // dynamic object

If C++11 rules are in effect, you can use list initialization instead:

Bozo bozetta = {"Bozetta", "Biggens"}; // C++11

Bozo fufu{"Fufu", "O’Dweeb"} // C++11;

Bozo \*pc = new Bozo{"Popo", "Le Peu"}; // C++11

If a constructor has just one argument:

Bozo(int age);

Then you can use any of the following forms to initialize an object:

Bozo dribble = bozo(44); // primary form

Bozo roon(66); // secondary form

Bozo tubby = 32; // special form for one-argument constructors

Actually, the third example is a new point, not a review point, but it seemed like a nice time to tell you about it. Chapter 11 mentions a way to turn off this feature because it can lead to unpleasant surprises.

Class destructors that use delete become necessary **when class constructors use new**.

* **Knowing Your Objects: The this Pointer**

So far each class member function has dealt with but **a single object**: the object that invokes it. Sometimes, however, **a method might need to deal with *two* objects**, and doing so may involve a curious C++ pointer called ***this***.

**比如前面Stock的例子, 唯一能让你知道Stock各个成员的值的办法是调用show(); 但问题是show()只是将成员值打印出来, 程序本身并无法获得各个成员的值并加以利用**. The most direct way of letting a program know about stored data is to provide methods to return values. Typically, you use **inline code for this**, as in the following example:

class Stock

{

private:

...

double total\_val;

...

public:

double total() **const** { return total\_val; } **// defined in class itself, default as inline；**

...

};

This definition, in effect, makes *total\_val* read-only memory **through function total()** as far as a direct program access is concerned. That is, you can use the total() method to obtain the value *total\_val*.

By adding this function to the class declaration, you can let a program investigate a series of stocks to find the one with the greatest value. However, you can take a different approach, one that helps you learn about the ***this*** pointer.

**现在, 比如说我要比较两个不同的Stock的总价格并知道那个的值较大：**

**First**, how do you provide the member function with two objects to compare? Suppose, for example, that you decide to name the method *topval().* Then the function call stock1.topval() accesses the data of the stock1 object, whereas the message stock2.topval() accesses the data of the stock2 object. If you want the method to compare two objects, you have to pass the second object as an argument. **For efficiency**, **you can pass the argument by reference**.

Second, how do you communicate the method’s answer back to the calling program? The most direct way is to have the method return a reference to the object that has the larger total value.

**Thus, the comparison method should have the following prototype:**

**const Stock & topval(const Stock & s) const;**

This function is expected to **access one object** **implicitly (方法未知)** and **one object explicitly (参数那个)**, **and it returns a reference to one of those two objects**.

**对三个Const的解释：**

1. The **const in parentheses** states that the function **won’t modify the explicitly accessed object**;
2. The **const that follows the parentheses** states that the **function won’t modify the implicitly accessed object**;
3. Because the function **returns a reference to one of the two const objects**, the **return type** also has to be a const reference.

回顾前面对函数最后面的const的解释：

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

The *show()* method doesn’t have any arguments for const to qualify. Instead, **the object it uses is provided implicitly by the method invocation**. What is needed is a new syntax, one that says a function promises not **to modify the *invoking object***. The C++ solution is to **place the const keyword after the function parentheses**.

所以到目前为止, topval()通过函数参数调用了一个class, 然后他自己又会是一个class的成员, 这样他就涉及到了两个不同的class, 可以进行比较了.

You can use either of the following

statements to do so:

top = stock1.topval(stock2);

top = stock2.topval(stock1);

The first form accesses stock1 implicitly and stock2 explicitly, whereas the second accesses stock1 explicitly and stock2 implicitly (see Figure 10.3). Either way, the method compares the two objects and returns a reference to the one with the higher total value.

Meanwhile, there’s still the implementation of topval() to attend to. It raises a slight problem.

const Stock & Stock::topval(const Stock & s) const

{

if (**s.total\_val** > **total\_val**) // total\_val here is actually **this->total\_val;**

return s; **// argument object**

else

return ?????; **// how to return invoking object itself?**

}

The C++ solution to this problem is to use a special pointer called ***this***. The ***this*** pointer **points to the object used to invoke a member function**. Thus, the function call *stock1.topval(stock2)* sets ***this*** to the address of the stock1 object and makes that pointer available to the topval() method. Similarly, the function call *stock2.topval(stock1)* sets ***this*** to the address of the stock2 object.

In general, all class methods have a this pointer set to the address of the object that invokes the method. Indeed, ***total\_val*** in *topval()* is just shorthand notation for ***this->total\_val***. (Recall from Chapter 4,“Compound Types,” that you use the ***-> operator*** to access structure members **via a pointer**. The same is true for class members.)

What you want to return, however, is not this because this is the address of the object. You want to return the object itself, and that is symbolized by \*this.

const **Stock &** Stock::topval(const Stock & s) const

{

if (s.total\_val > total\_val)

return s; // argument object

else

return **\*this**; // invoking object

}

The fact that the **return type is a reference** means that the **returned object is the invoking object itself** rather than a copy passed by the return mechanism.

* **An Array of Objects**

Stock mystuff[4]; // creates an array of 4 Stock objects

Recall that a program always calls the default class constructor when it creates class objects that aren’t explicitly initialized. This declaration requires either that the class explicitly define no constructors at all, in which case the implicit do-nothing default constructor is used, or, as in this case, that an explicit default constructor be defined.

mystuff[0].update(); // apply update() to 1st element

mystuff[3].show(); // apply show() to 4th element

const Stock \* tops = **mystuff[2]**.topval(**mystuff[1]**);

// compare 3rd and 2nd elements and set tops

// to point at the one with a higher total value

You can use a constructor to initialize the array elements. In that case, you have to call the constructor for each individual element:

const int STKS = 4;

Stock stocks[STKS] = {

Stock("NanoSmart", 12.5, 20),

Stock("Boffo Objects", 200, 2.0),

Stock("Monolithic Obelisks", 130, 3.25),

Stock("Fleep Enterprises", 60, 6.5)

};

If the class has more than one constructor, you can use different constructors for different elements:

const int STKS = 10;

Stock stocks[STKS] = {

Stock("NanoSmart", 12.5, 20),

Stock(),

Stock("Monolithic Obelisks", 130, 3.25),

};

Because this declaration only partially initializes the array, the remaining seven members are initialized using the default constructor.

* **Class Scope**

Class scope means you can’t directly access members of a class from the outside world. This is true even for public function members. That is, to invoke a public member function, you have to use an object:

Stock sleeper("Exclusive Ore", 100, 0.25); // create object

sleeper.show(); // use object to invoke a member function

show(); // invalid -- can’t call method directly

In short, within a class declaration or a member function definition you can use an unadorned member name (the unqualified name), as when sell() calls the set\_tot() member function. A constructor name is recognized when it is called because its name is the same as the class name. Otherwise, you must use the direct membership operator (.), the indirect membership operator (->), or the scope-resolution operator (::), depending on the context, when you use a class member name.

**Class Scope Constants**

Sometimes it would be nice to have symbolic constants with class scope. You might think the following would be a solution:

class Bakery

{

private:

**const int Months = 12**; // declare a constant? FAILS

double costs[Months];

...

**But this won’t work** because declaring a class describes what an object looks like but doesn’t create an object. Hence, until you create an object, there’s no place to store a value.

**You can declare an enumeration** within a class. An enumeration given in a class declaration **has class scope**, so you can use enumerations to provide class scope symbolic names for integer constants.

class Bakery

{

private:

**enum {Months = 12};**

double costs[Months];

...

Note that **declaring an enumeration in this fashion *does not create a class data member***. That is, each **individual object does not carry an enumeration in it**. Rather, *Months* is just a symbolic name that the compiler replaces with 12 when it encounters it in code in class scope.

Because the Bakery class uses the enumeration merely to create a symbolic constant, with no intent of creating variables of the enumeration type, **you needn’t provide an enumeration tag**.

**C++ has a second way** of defining a constant within a class, using the keyword ***static***:

class Bakery

{

private:

**static const int Months = 12;**

double costs[Months];

...

**This creates a single constant** called Months that is ***stored with other static variables*** rather than in an object. Thus, **there is only one Months** constant shared by all Bakery objects.

**Scoped Enumerations (C++11)**

Traditional enumerations have some problems. One is that enumerators from two different *enum* definitions can conflict.

enum egg {Small, Medium, Large, Jumbo};

enum t\_shirt {Small, Medium, Large, Xlarge};

This won’t work because the *egg Small* and the *t\_shirt Small* would both be in the same scope, and the names conflict. C++11 provides a new form of enumeration that avoids this problem by having class scope for its enumerators. The declarations for this form look like this:

enum class egg {Small, Medium, Large, Jumbo};

enum class t\_shirt {Small, Medium, Large, Xlarge};

Alternatively, you can use the keyword struct instead of class. In either case, you now need to use the enum name to qualify the enumerator:

egg choice = egg::Large; // the Large enumerator of the egg enum

t\_shirt Floyd = t\_shirt::Large; // the Large enumerator of the t\_shirt enum

C++11 also tightens up type security for scoped enumerations. ***Regular enumerations*** **get converted to integer types automatically in some situations**, such as assignment to an int variable or being used in a comparison expression, but ***scoped enumerations* have no implicit conversions to integer types**:

enum **egg\_old** {Small, Medium, Large, Jumbo}; // unscoped

enum **class t\_shirt** {Small, Medium, Large, Xlarge}; // scoped

egg\_old one = Medium; // unscoped

t\_shirt rolf = t\_shirt::Large; // scoped

int king = one; // **implicit type conversion for unscoped**

int ring = rolf; // **not allowed**, no implicit type conversion

if (king < Jumbo) // allowed

std::cout << "Jumbo converted to int before comparison.\n";

if (king < **t\_shirt::Medium**) // **not allowed**

std::cout << "Not allowed: < not defined for scoped enum.\n";

But you can do an explicit type conversion if you feel you have to:

int Frodo = int(t\_shirt::Small); // Frodo set to 0

Enumerations are represented by some underlying integer type, and **under C98** that choice was **implementation-dependent**. Thus, a structure containing an enumeration might be of different sizes on different systems. **C++11 removes that dependency for scoped enumerations**. By default, the underlying type for **C++11 scoped enumerations is *int***. Furthermore, there’s a syntax for indicating a different choice:

// underlying type for pizza is short

enum class : short pizza {Small, Medium, Large, XLarge};

Under C++11, you also can use this syntax to indicate the underlying type for an unscoped enumeration, but if you don’t choose the type, the choice the compiler makes is implementation-dependent.

* **Abstract Data Types**

**Example implementing a Stack:**

// **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’) **// first drop other following inputs if any**

continue;

if (!isalpha(ch))

{

cout << ‘\a’;

continue; **// Jump to top level while()**

}

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;

}

[Summary]

**const Member Functions**

const Stock land = Stock("Kludgehorn Properties");

land.show();

With current C++, the compiler should object to the second line. Why? **Because the code for show() fails to guarantee that it won’t modify the invoking object, which, because it is const, should not be altered**.

问题是, 如果我们想把show()函数定义成不会改变invoking object的, 该怎么定义呢? Show()本身没有任何参数. What is needed is a new syntax, one that says a function promises not **to modify the invoking object**. The C++ solution is to **place the const keyword after the function parentheses**. That is, the *show()* **declaration** should look like this:

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

Similarly, the beginning of the **function definition** should look like this:

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

**The this Pointer**

如果要比较两支不同股票(两个不同的Stock的object)的总值并返回较大的那个呢？

第一种办法是在Class定义中增加一个member:

class Stock

{

private:

...

double total\_val;

...

public:

double total() **const** { return total\_val; } // defined in class itself, default as inline；

...

};

然后再定义一个额外的函数, 以两个Object的Reference为参数, 比较后返回较大的那个.

第二种办法是直接把这个函数定义为Class的member. 但是how do you provide the member function with two objects to compare? 比如说这个函数叫*topval().*If you want the method to compare two objects, you have to pass the second object as an argument (the other one is the invoking object). **For efficiency**, **you can pass the argument by reference**.

How do you communicate the method’s answer back to the calling program? The most direct way is to have the method return a reference to the object that has the larger total value.

**const Stock & topval(const Stock & s) const;**

This function is expected to **access one object** **implicitly (the invoking one, 方法未知)** and **one object explicitly (参数那个)**, **and it returns a reference to one of those two objects**.

**对三个Const的解释:**

1. The **const in parentheses** states that the function **won’t modify the explicitly accessed object**;
2. The **const that follows the parentheses** states that the **function won’t modify the implicitly accessed object, which is the invoking object**;
3. Because the function **returns a reference to one of the two const objects**, the **return type** also has to be a const reference.

这个函数可以定义成这样:

const **Stock &** Stock::topval(const Stock & s) const

{

if (s.total\_val > total\_val) // 2nd total\_val here is actually **this->total\_val;**

return s; // argument object

else

return **\*this**; // invoking object

}

The ***this*** pointer **points to the object used to invoke a member function**. What you want to return, is not *this* because this is the address of the object. You want to return the object itself, and that is symbolized by *\*this*. The fact that the **return type is a reference** means that the **returned object is the invoking object itself** rather than a copy passed by the return mechanism.

所以真正的比较可以这么用:

top = stock1.topval(stock2);

top = stock2.topval(stock1);

The first form accesses stock1 implicitly and stock2 explicitly, whereas the second accesses stock1 explicitly and stock2 implicitly.

Stock mystuff[4]; // creates an array of 4 Stock objects

mystuff[0].update(); // apply update() to 1st element

mystuff[3].show(); // apply show() to 4th element

**const Stock \*** tops = mystuff[2].**topval**(**mystuff[1]**);

// compare 3rd and 2nd elements and set tops to point at the one with a higher total value

// **这里应该是个错误, 左边是指针, 右边是对象数据本身, 应该加取址符;**

**An Array of Objects**

You can use a constructor to initialize the array elements. In that case, you have to call the constructor for each individual element:

const int STKS = 4;

Stock stocks[STKS] = {

Stock("NanoSmart", 12.5, 20),

Stock("Boffo Objects", 200, 2.0),

Stock("Monolithic Obelisks", 130, 3.25),

Stock("Fleep Enterprises", 60, 6.5)

};

If the class has more than one constructor, you can use different constructors for different elements:

const int STKS = 10;

Stock stocks[STKS] = {

Stock("NanoSmart", 12.5, 20),

Stock(),

Stock("Monolithic Obelisks", 130, 3.25),

};

Because this declaration only partially initializes the array, the remaining seven members are initialized using the default constructor.

**Class Scope Constants**

Sometimes it would be nice to have symbolic constants with class scope. You might think the following would be a solution:

class Bakery

{

private:

**const int Months = 12**; // declare a constant? FAILS

double costs[**Months**];

...

**But this won’t work** because declaring a class describes what an object looks like but doesn’t create an object. Hence, until you create an object, there’s no place to store a value.

**You can declare an enumeration** within a class. An enumeration given in a class declaration **has class scope**, so you can use enumerations to provide class scope symbolic names for integer constants.

class Bakery

{

private:

**enum {Months = 12};**

double costs[**Months**];

...

Note that **declaring an enumeration in this fashion *does not create a class data member***. That is, each **individual object does not carry an enumeration in it**. Rather, *Months* is just a symbolic name that the compiler replaces with 12 when it encounters it in code in class scope.

Because the Bakery class uses the enumeration merely to create a symbolic constant, with no intent of creating variables of the enumeration type, **you needn’t provide an enumeration tag**.

**C++ has a second way** of defining a constant within a class, using the keyword ***static***:

class Bakery

{

private:

**static const int Months = 12;**

double costs[Months];

...

**This creates a single constant** called Months that is ***stored with other static variables*** rather than in an object. Thus, **there is only one Months** constant shared by all Bakery objects.