# 10.3 类的构造函数和析构函数

对于Stock类，还有其他一些工作要做。应为类提供被称为构造函数和析构函数的标准函数。下面来看一看为什么需要这些函数以及如何使用这些函数。

C++的目标之一是让使用类对象就像使用标准类型一样，然而，到现在为止，本章提供的代码还不能让您像初始化int或结构那样来初始化Stock对象。也就是说，常规的初始化语法不适用于类型Stock：

int year = 200l; // 可行的初始化

struct thing

{

 char \* pn;

 int m;

};

thing amabob = {“wodger”,-23}; // 可行的初始化

Stock hot = {“Sukie’s Autos , Inc.” ,200,50.25}; // 不行，编译器报错

不能像上面这样初始化Stock对象的原因在于，数据部分的访问状态是私有的，这意味着程序不能直接访问数据成员。您已经看到，程序只能通过成员函数来访问数据成员，因此需要设计合适的成员函数，才能成功地将对象初始化（如果使数据成员成为公有，而不是私有，就可以按刚才介绍的方法初始化类对象，但使数据成为公有的违背了类的一个主要初衷：数据隐藏）。

一般来说，最好是往创建对象时对它进行初始化。例如，请看下面的代码：

Stock gift;

gift.buy(10,24.75);

就Stock类当前的实现而言，gift对象的company成员是没有值的。类设计假设用户在调用任何其他成员函数之前调用acquire()，但无法强加这种假设。避开这种问题的方法之一是在创建对象时，自动对它进行初始化。为此，C++提供了一个特殊的成员函数一—类构造函数，专门用于构造新对象、将值赋给它们的数据成员。更准确地说，C++为这些成员函数提供了名称和使用语法，而程序员需要提供方法定义。名称与类名相同。例如，Stock类一个可能的构造函数是名为Stock()的成员函数。构造函数的原型和函数头有一个有趣的特征——虽然没有返回值，但没有被声明为void类型。实际上，构造函数没有声明类型。

## 10.3.1 声明和定义构造函数

现在需要创建Stock的构造函数。由于需要为Stock对象提供3个值，因此应为构造函数提供3个参数。（第4个值，total\_val成员，是根据shares和share\_val计算得到的，因此不必为构造函数提供这个值。）程序员可能只想设置company成员，而将其他值设置为0；这可以使用默认参数来完成（参见第8章）。因此，原型如下所示：

// 适用默认参数的构造函数原型

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

第一个参数是指向字符串的指针，该字符串用于初始化成员company。n和pr参数为shares和share\_val成员提供值。注意，没有返回类型。原型位于类声明的公有部分。

下面是构造函数的一种可能定义：

// 构造函数定义

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();

}

上述代码和本章前面的函数acquire()相同。区别存于，程序声明对象时，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：

// 不可以！

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，都可以在公有接口中在参数名中包含company和shares。

## 10.3.2 使用构造函数

C++提供了两种使用构造函数来初始化对象的方式。第一种方式是显式地调用构造函数：

Stock food = Stock(“world Cabbage” , 250 , 1.25);

这将food对象的company成员设置为字符串“World Cabbage”，将shares成员设置为250，依此类推。

另一种方式是隐式地调用构造函数：

Stock garment = (“Furry Mason”,50,2.5);

这种格式更紧凑，它与下面的显示调用等价：

Stock garment = Stock（”Furry Mason”,50,2.5）;

每次创建类对象（其至使用new分配内存）时，C++都使用类构造函数。下面是将构造函数与new一起使用的方法：

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

这条语句创建一个Stock对象，将其初始化为参数提供的值，并将该对象的地址赋给pstock指针。在这种情况下，对象没有名称，但可以使用指针来管理该对象。我们将在第11章进一步讨论对象指针。

构造函数的使用方式不同于其他类方法。一般来说，使用对象来调用方法：

stock1.show(); // stock1对象调用show()方法

但无法使用对象来调用构造函数，因为在构造函数构造出对象之前，对象是不存在的。因此构造函数被用来创建对象，而不能通过对象来调用。

## 10.3.3 默认构造函数

默认构造函数是在未提供显式初始值时，用来创建对象的构造函数。也就是说，它是用于下面这种声明的构造函数：

Stock fluffy\_the\_cat; // 使用默认构造函数

程序清单10.3就是这样做的！这条语句管用的原因在于，如果没有提供任何构造函数，则C++将自动提供默认构造函数：它是默认构造函数的隐式版本，不做任何工作。对于Stock类来说，默认构造函数可能如下：

Stock::Stock(){}

因此将创建fluffy\_the\_cat对象，但不初始化其成员，这和下面的语句创建x，但没有提供值给它一样：

int x;

默认构造函数没有参数，因为声明中不包含值。

奇怪的是，当且仅当没有定义任何构造函数时，编译器才会提供默认构造函数。为类定义了构造函数后，程序员就必须为它提供默认构造函数。如果提供了非默认构造函数（如Stock(const char \*co,int n,double pr)）但没有提供默认构造函数，则下面的声明将出错：

Stock stock1; // 不可能使用当前的构造函数

这样做的原因可能是想禁止创建未初始化的对象。然而，如果要创建对象，而不显式地初始化，则必须定义一个不接受任何参数的默认构造函数。定义默认构造函数的方式有两种。一种是给已有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);

另一种方式是通过函数重载来定义另一个构造函数——一个没有参数的构造函数：

Stock();

由于只能有一个默认构造函数，因此不要同时采用这两种方式。实际上，通常应初始化所有的对象，以确保所有成员一开始就有已知的合理值。因此，用户定义的默认构造函数通常给所有成员提供隐式初始值。例如，下面是为Stock类定义的一个默认构造函数：

Stock:Stock() // 默认构造函数

{

 company = ”no．name”;

 shares = 0;

 share\_val = 0.0;

 total\_val = 0.0;

}

**提示**：在设计类时，通常应提供对所有类成员做隐式初始化的默认构造函数。

使用上述任何一种方式（没有参数或所有参数都有默认值）创建了默认构造函数后，便可以声明对象变量，而不对它们进行显式初始化：

Stock first; // 隐式调用默认构造函数

Stock first = Stock(); // 显式调用

Stock \*prelief = new Stock; // 隐式调用

然而，不要被非默认构造函数的隐式形式所误导：

Stock first("Concrete Conglomerate”) // 调用构造函数

Stock second(); // 声明一个方法

Stock third; // 调用默认构造函数

第一个声明调用非默认构造函数，即接受参数的构造函数；第二个声明指出，second()是一个返回Stock对象的函数。隐式地调用默认构造函数时，不要使用圆括号。

## 10.3.4 析构函数

用构造函数创建对象后，程序负责跟踪该对象，直到其过期为止。对象过期时，程序将自动调用一个特殊的成员函数，该函数的名称令人生畏——析构函数。析构函数完成清理工作，因此实际上很有用。例如，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。Stock的构造函数没有使用new，因此析构函数实际上没有需要完成的任务。在这种情况下，只需让编译器生成一个什么也不做的隐式析构函数即可，Stock类第一版正是这样做的。然而，了解如何声明和定义析构函数是绝对必要的，下面为Stock类提供一个析构函数。

和构造函数一样，析构函数的名称也很特殊：在类名前加上~。因此，Stock类的析构函数为~Stock()。另外，和构造函数一样，析构函数也可以没有返回值和声明类型。与构造函数不同的是，柝构函数没有参数，因此Stock析构函数的原型必须是这样的：

~Stock()

由于Stock的析构函数不承担任何重要的工作，因此可以将它编写为不执行任何操作的函数：

Stock::~Stock()

{

}

然而，为让您能看出析构函数何时被调用，这样编写其代码：

Stock::~Stock()

{

 cout << “Bye,” << company << “!\n”;

}

什么时候应调用析构函数呢？这由编译器决定，通常不应在代码中显式地调用析构函数（有关例外情形，请参阅第12章的“再谈定位new运算符”），如果创建的是静态存储类对象，则其析构函数将在程序结束时自动被调用。如果创建的是自动存储类对象（就像前面的示例中那样），则其析构函数将在程序执行完代码块时（该对象是在其中定义的）自动被调用：如果对象是通过new创建的，则它将驻留在栈内存或自由存储区中，当使用delete来释放内存时，其析构函数将自动被调用。最后，程序可以创建临时对象来完成特定的操作，在这种情况下，程序将在结束对该对象的使用时自动调用其析构函数。

由于在类对象过期时构函数将自动被调用，因此必须有一个析构函数。如果程序员没有提供析构函数，编译器将隐式地声明一个默认析构函数，并在发现导致对象被删除的代码后，提供默认析构函数的定义。

## 10.3.5 改进Stock类

下面将构造函数和析构函数加入到类和方法的定义中。鉴于添加构造函数的重大意义，这里将名称从stock00.h改为stock10.h。类方法放在文件stock10.cpp中。最后，将使用这些资源的程序放在第三个文件中，这个文件名为usestock2.cpp。

### 1．头文件

程序清单10.4列出了头文件。它将构造函数和析构函数的原型加入到原来的类声明中。另外，它还删除了acquire()函数——现在已经不再需要它了，因为有构造函数。该文件还使用第9章介绍的#ifndef技术来防止多重包含。

**程序清单10.4 stock10.h**

// stock10.h 添加了构造函数和析构函数的Stock类声明

#ifndef STOCK1\_H\_

#define STOCK1\_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:

 Stock(); // 默认构造函数

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

 ~Stock(); // 析构函数

 void buy(long num, double price);

 void sell(long num, double price);

 void update(double price);

 void show();

};

#endif

### 2．实现文件

程序清单10.5提供了方法的定义。它包含了文件stock10.h，以提供类声明（将文件名放在双引号而不是方括号中意味着编译器将源文件所在的目录中搜索它）。另外，程序清单10.5还包含了头文件iostream，以提供I/O支持。该程序清单还使用using声明和限定名称（如std::string）来访问头文件中的各种声明。该文件将构造函数和析构函数的方法定义添加到以前的方法定义中。为让您知道这些方法何时被调用，它们都显示一条消息。这并不是构造函数和析构函数的常规功能，但有助于您更好地了解类是如何使用它们的。

**程序清单10.5 stock10.cpp**

// stock1.cpp 添加了构造函数和析构函数的Stock类实现

#include <iostream>

#include "stock10.h"

// 构造函数(啰嗦的版本)

Stock::Stock() // 默认构造函数

{

 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();

}

// 析构函数

Stock::~Stock() // verbose class destructor

{

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

}

// 其他方法

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;

 // 将格式设置为#.###

 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;

 // 将格式设置为#.##

 cout.precision(2);

 cout << " Total Worth: $" << total\_val << '\n';

 // 恢复原始格式

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

 cout.precision(prec);

}

### 3．客户文件

程序清单10.6提供了一个测试这些新方法的小程序；由于它只是使用Stock类，因此是Stock类的客户。和stock10.cpp一样，它也包含了文件stock10.h以提供类声明。该程序显示了构造函数和析构函数，它还使用了程序清单10.3调用的格式化命令。要编译整个程序，必须使用第1章和第9章介绍的多文件程序技术。

**程序清单10.6 usestock2.cpp**

// usestok1.cpp -- 使用Stock类

// 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); // 第一种语法

 stock1.show();

 Stock stock2 = Stock ("Boffo Objects", 2, 2.0); // 第二种语法

 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); // 临时对象

 cout << "Revised stock1:\n";

 stock1.show();

 cout << "Done\n";

 }

 std::cin.get();

 return 0;

}

编译程序清单10.4、程序清单10.5和程序清单10.6所示的程序，得到一个可执行程序。下面是使用某个编译器得到的可执行程雇的输出：

Using constructors to create new objects

Constructor Using NanoSmart called

Company: NanoSmart Shares: 12

 Share Price: $20.00 Total Worth: $240.00

Constructor uaing Boffo Objects called

Company: Boffo Objects Shares: 2

 Share Price: $2.00 Total worth: $4.00

Assigning stockl to stock2:

Listing stockl and stock2:

Company: NanoSmart Shares: 12

Share Price: $20.00 Total Worth:$240.00

Company: NanoSmart Shares: 12

 Share Price: $20.00 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: $50.00 Total Worth: $500.00

Done

Bye, NanoSmart!

Bye, Nifty Foods!

使用某些编译器编译该程序时，该程序输出的前半部分可能如下（比前而多了一行）：

Using constructors to create new objects

Constructor using NanoSmart called

Company:NanoSmart Shares:12

 Share Price: $20.00 Total Worth: $240.00

Constructor using Boffo Objects called

Bye, Boffo Objects! // 多出的一行

Company: Boffo Objects Shares:2

 Share Price; $2.00 Total Worth:$4.00

…

下一小节将解释输出行“Bye,Boffo Objects!”

**提示**：您可能注意到了，在程序清单10.6中，main()的开头和末尾多了一个大括号。诸如stock1和stock2等自动变量将在程序退出其定义所属代码块时消失。如果没有这些大括号，代码块将为整个main()，因此仅当main()执行完毕后，才会调用析构函数。在窗口环境中，这意味着将在两个析构函教调用前关闭，导致您无法看到最后两条消息。但添加这些大括号后，最后两个析构函数调用将在到达返回语句前执行，从而显示相应的消息。

### 4．程序说明

程序清单10.6中的下述语句：

Stock::stock1(“NanoSmart” , 12 , 20.0);

创建一个名为stock1的Stock对象，并将其数据成员初始化为指定的值：

Constructor using NanoSmart called

Company： NanoSmart Shares:12

下面的语句使用另一种语法创建并初始化一个名为stock2的对象：

Stock stock2 = Stock(“Boffo Object” , 2, 2.0);

C++标准允许编译器使用两种方式来执行第二种语法。一种是使其行为和第一种语法完全相同：

Constructor using Boffo Objects called

Company： Boffo Objects Shares: 2

另一种方式是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。如果编译器使用的是这种方式，则将为临时对象调用析构函数，因此生成下面的输出：

Constructor using Boffo Objects called

Bye，Boffo Objects!

Company： Boffo Objects Shares: 2

生成上述输出的编译器可能立刻删除临时对象，但也可能会等一段时间，在这种情况下，析构函数的消息将会过一段时间才显示。

下面的语句表明可以将一个对象赋给同类型的另一个对象：

stock2 = stock1; // 对象赋值

与给结构赋值一样，一在默认情况下，给类对象赋值时，将把一个对象的成员复制给另一个。在这个例子中，stock2原来的内容将被覆盖。

**注意**：在默认情况下，将一个对象赋给同类型的另一个对象时，C++将源对象的每个数据成员的内容复制到目标对象中相应的数据成员中。

构造函数不仅仅可用于初始化新对象。例如，该程序的main()中包含下面的语句：

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

stock1对象已经存在，因此这条语句不是对stock1进行初始牝，而是将新值赋给它。这是通过让构造程序创建一个新的、临时的对象，然后将其内容复制给stock1来实现的。随后程序调用析构函数，以删除该临时对象，如下面经过注释后的输出所示：

Using a constructor to reset an object

Constructor using Nifty Foods called: // 创建临时对象

Bye , Nifty Foods! // 删除临时对象

Revised stock1:

Company:Nifty Foods Shares:10

 Share Price: $50.00 Total Worth: $500.00 // 数据被复制到stock1

有些编译器可能要过一段时间才删除临时对象，因此析构函数的调用延迟。

最后，程序显示了下面的内容：

Done

Bye, NanoSmart!

Bye, Nifty Foods!

函数main()结束时，其局部变量（stock1和stock2）将消失。这种自动变量被放在栈中，因此最后刨建的对象将最先被删除，最先创建的对象将最后被删除（“NanaSmart”最初位于stock1中，但随后被传输到stock2中；然后stock1被重置为“Nifty Food”）。

输出表明，下面两条语句有根本性的差别：

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

stock1 = Stock(“Nifty Foods", 10 ,50．0); // 临时对象

第一条语句是初始化，它创建有指定值的对象，可能会创建临时对象（也可能不会）；第二条语句是赋值。像这样在赋值语句中使用构造函数总会导致在赋值前创建一个临时对象。

**提示**：如果既可以通过初始化，也可以通过赋值来设置对象的值，则应采用初始化方式。通常这种方式的效率更高。

### 5．C++11列表初始化

在C++11中：可将列表初始化语法用于类吗？可以，只要提供与某个构造函数的参数列表匹配的内容，并用大括号将它们括起：

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

Stock jock { “Sport Age Storage Inc”};

Stock temp {};

在前两个声明中，用大括号括起的列表与下面的构造函数匹配：

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

因此，将使用该构造函数来创建这两个对象。创建对象jock时，第二和第三个参数将为默认值0和0.0。第三个声明与默认构造函数匹配，因此将使用该构造函数创建对象temp。

另外，C++11还提供了名为std::initialize\_list的类，可将其用作函数参数或方法参数的类型。这个类可表示任意长度的列表，只要所有列表项的类型都相同或可转换为相同的类型，这将在第16章介绍。

### 6．const成员函数

请看下面的代码片段：

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

land.show();

对于当前的C++来说，编译器将拒绝第一行。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; // 确保不会修改调用对象

同样，函数定义的开头应像这样：

void Stock::show() const //确保不会修改调用对象

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。

## 10.3.6 构造函数和析构函数小结

介绍一些构造函教和析构函数的例子后，您可能想停下来，整理一下学到的知识。为此，下面对这些方法进行总结。

构造函数是一种特殊的类成员函数，在创建类对象时接调用。构造函数的名称和类名相同，但通过函数重载，可以创建多个同名的构造函数，条件是每个函数的特征标（参数列表）都不同。另外，构造函数没有声明类型。通常，构造函数用初始化类对象的成员，初始化应与构造函数的参数列表匹配。例如，假设Bozo类的构造函数的原型如下：

Bozco(const char \* fname, const char \* lname); // 构造函数原型

则可以使用它来初始化新对象：

Bozo bozetta = bozo("Bozettan”,”Biggens”); // 基本形式

Bozo fufu("Fufu", "O'Dweeb"); // 短形式

Bozo \*pc = new Bozo("Popo", “Le Peu”); // 动态对象

如果编译器支持C++11，则可使用列表初始化：

Bozo bozetta = { “Bozetta”, “Biggens”};

Bozo fufu{"Fufu", “O'Dweeb"};

Bozo \*pc = new Bozo{"Popo", "Le Peu”};

如果构造函数有且只有一个参数，则将对象初始化为一个与参数的类型相同的值时，该构造函数将被调用。例如，假设有这样—个构造函数原型：

Bozco(int n);

则可以使用下面的任何一种形式来初始化对象：

Baozo dribble = bozo(44); // 基本形式

Bozo roon(66); // 第二种形式

Bozo tubby = 32; // 只有一个参数的构造函数的特殊形式

实际上，第三个示例是新内容，不属于复习内容，但现在正是介绍它的好时机。第11章将介绍一种关闭这项特性的方式，因为它可能带来令人不愉快的意外。

**警告**：接受一个参数的构造函数允许使用赋值语法将对象初始化为一个值：

Classname object = value;

这种特性可能导致问题，但正如第11章将介绍的，可关闭这项特性。

默认构造函数没有参数，因此如果创建对象时没有进行显式地初始化，则将调用默认构造函数。如果程序中没有提供任何构造函数，则编译器会为程序定义一个默认构造函数；否则，必须自己提供默认构造函数。默认构造函数可以没有任何参数；如果有，则必须给所有参数都提供默认值：

Bozo(); // 默认构造函数原型

Bistro(const char \* s = “Chez Zero”); // Bistro类的默认构造函数

对于未被初始化的对象，程序将使用默认构造函数来创建：

Bozo bubi; // 使用默认构造函数

Bozo \*pb = new Bozo; // 使用默认构造函数

就像对象被创建时程序将调用构造函数一样，当对象被删除时，程序将调用析构函数。每个类都只能有一个析构函数。析构函数没有返回类型（连void都没有），也没有参数，其名称为类名称前加上~。例如，Bozo类的析构函数的原型如下：

~Bozo(); // 类的析构函数

如果构造函数使用了new，则必须提供使用delete的析构函数。