**Chapter 12 Classes and Dynamic Memory Allocation**

Suppose you want to create a class with a member that represents someone’s last name. The simplest, most primitive way is to use a character array member to hold the name. But this has some drawbacks. You might use a 14-character array and then run into Bartholomew Smeadsbury-Crafthovingham. Or to be safer, you might use a 40-character array. But if you then create an array of 2,000 such objects, you’ll waste a lot of memory with character arrays that are only partly filled.

**Often it is much better to decide many matters, such as how much storage to use, when a program runs rather than when it’s compiled**.

The usual C++ approach to storing a name in an object is to use the ***new*** operator in a class constructor to allocate the correct amount of memory while the program is running. The usual way to accomplish this is to use the ***string*** class, which takes care of the memory management details for you. Introducing new to a class constructor raises several new problems unless you remember to take a series of additional steps, such as expanding the class destructor, bringing all constructors into harmony with the new destructor, and writing additional class methods to facilitate correct initialization and assignment.

* **Dynamic Memory and Classes**

// **strngbad.h** -- ***flawed string class definition***

#include <iostream>

#ifndef STRNGBAD\_H\_

#define STRNGBAD\_H\_

class StringBad

{

private:

**char \* str;** // pointer to string

int len; // length of string

***static*** int num\_strings; // number of objects

public:

StringBad(const char \* s); // constructor

StringBad(); // default constructor

~StringBad(); // destructor

// friend function

**friend** std::ostream & operator<<(std::ostream & os, const StringBad & st);

};

#endif

You should note two points about this declaration.

First, **it uses a pointer-to-char** *instead of a char array* to represent a name. This means that the *class declaration does not allocate storage space for the string itself*. Instead, it uses new in the constructors to allocate space for the string. This arrangement avoids straitjacketing the class declaration with a predefined limit to the string size.

Second, the definition declares the ***num\_strings*** member as belonging to the ***static storage*** class. ***A static class member has a special property*:** **A program creates only one copy of a static class variable, regardless of the number of objects created**. That is, a static member is shared among all objects of that class. If, say, you **create 10 StringBad objects**, there would be 10 str members and 10 len members, **but just 1 shared num\_strings member**. This is convenient for data that should be private to a class but that should have the same value for all class objects.

num\_strings member is a convenient means of illustrating static data members and as a device to point out potential programming problems. *In general, a string class doesn’t need such a member*.

// **strngbad.cpp** -- StringBad class methods

#include **<cstring>** // string.h for some

#include "strngbad.h"

using std::cout;

// initializing static class member

**int StringBad::num\_strings = 0; // outside the declaration of class**

// class methods

// construct StringBad from C string

**StringBad::StringBad(const char \* s)**

{

**len = std::strlen(s);** // set size

**str = *new* char[len + 1];** // alloc storage

**std::strcpy(str, s);** // initialize pointer

***num\_strings++;*** // set object count

cout << num\_strings << ": \"" << str << "\" object created\n"; // For Your Information

}

**StringBad::StringBad()** // default constructor

{

len = 4;

str = new char[4];

std::strcpy(str, "C++"); // default string

num\_strings++;

cout << num\_strings << ": \"" << str << "\" default object created\n";

}

**StringBad::~StringBad()** **// necessary destructor**

{

cout << "\"" << str << "\" object deleted, ";

**--num\_strings;**

cout << num\_strings << " left\n";

**delete [] str**;

}

std::**ostream &** operator<<(std::ostream & os, const StringBad & st)

{

os << st.str;

return os;

}

First, notice the following statement from Listing above:

int StringBad::num\_strings = 0;

**This statement initializes the static num\_strings member to 0**. **Note that you CAN’T initialize a static member variable inside the class declaration**. That’s because the declaration is a description of how memory is to be allocated but it doesn’t, and shouldn’t, allocate memory. *You allocate and initialize memory by creating an object* using that format. **In the case of a static class member, you initialize the static member independently, with a separate statement outside the class declaration**. **That’s because the static class member is stored separately rather than as part of an object.** Note that the initialization statement gives the type and uses the scope operator, but it doesn’t use the static keyword.

**This initialization goes in the methods file, not in the class declaration file.** That’s because the class declaration is in a header file, and a program may include a header file in several other files. That would result in multiple copies of the initialization statement, which is an error.

**The exception to the non-initialization of a static data member inside the class declaration (Chapter 10,”Objects and Classes”) is if the static data member is a *const* of integral or enumeration type.**

A static data member is declared in the class declaration and is initialized in the file containing the class methods. The scope operator is used in the initialization to indicate to which class the static member belongs. **However, if the static member is a const integral type or an enumeration type, it can be initialized in the class declaration itself.**

在constructor StringBad::StringBad(const char \* s)中, 首先计算了输入string的长度, 然后new出内存, 把输入string复制到这个内存里, 在记下它的指针;you should realize that the string is not stored in the object. The string is stored separately, in heap memory, and the object merely stores information that tells where to find the string.

Note that you **DO NOT** **use this**:

str = s; // not the way to go

This merely stores the address without making a copy of the string.

StringBad::~StringBad() // Necessary Destructor

{

cout << "\"" << str << "\" object deleted, ";

--num\_strings;

cout << num\_strings << " left\n";

delete [] str;

}

When a *StringBad* object expires, the *str* pointer expires. But the memory *str* pointed to remains allocated unless you use delete to free it. **Whenever you use new in a constructor to allocate memory, you should use delete in the corresponding destructor to free that memory.** If you use new [] (with brackets), then you should use delete [] (with brackets).

// **vegnews.cpp** -- using new and delete with classes

**//** **compile with strngbad.cpp**

#include <iostream>

using std::cout;

**#include "strngbad.h"**

void callme1(StringBad &); // pass by reference

void callme2(StringBad); // pass by value

int main()

{

using std::endl;

{

cout << "Starting an inner block.\n";

StringBad headline1("Celery Stalks at Midnight");

StringBad headline2("Lettuce Prey");

StringBad sports("Spinach Leaves Bowl for Dollars");

cout << "headline1: " << headline1 << endl;

cout << "headline2: " << headline2 << endl;

cout << "sports: " << sports << endl;

callme1(**headline1**);

cout << "headline1: " << headline1 << endl;

***callme2(headline2);***

cout << "headline2: " << headline2 << endl;

cout << "**Initialize** one object to another:\n";

***StringBad sailor = sports;* // 这一句没有任何输出, 也就说没有调用自己定义的任何constructor.**

cout << "sailor: " << sailor << endl;

cout << "**Assign** one object to another:\n";

***StringBad knot;***

***knot = headline1;***

cout << "knot: " << knot << endl;

cout << "Exiting the block.\n";

}

cout << "End of main()\n";

return 0;

}

void callme1(StringBad & rsb)

{

cout << "String passed by reference:\n";

cout << " \"" << rsb << "\"\n";

}

void callme2(StringBad sb)

{

cout << "String passed by value:\n";

cout << " \"" << sb << "\"\n";

}

在我自己的机器上的测试结果是：

momo@HMI:~/C++PrimerPlus/Chapter12/BadString$ ./a.out

Starting an inner block.

1: "Celery Stalks at Midnight" object created **// headline1**

2: "Lettuce Prey" object created **// headline2**

3: "Spinach Leaves Bowl for Dollars" object created **// sports**

headline1: Celery Stalks at Midnight

headline2: Lettuce Prey

sports: Spinach Leaves Bowl for Dollars

String passed by reference: **// callme1(headline1);**

"Celery Stalks at Midnight"

headline1: Celery Stalks at Midnight

***String passed by value:* // callme2(headline2);**

***"Lettuce Prey"***

***"Lettuce Prey" object deleted, 2 left* // 调用callme2(headline2)出现了destructor的调用**

***headline2:* // 调用完callme2(headline2)后headline2的data没了…**

Initialize one object to another:

***sailor: Spinach Leaves Bowl for Dollars***

Assign one object to another:

***3: "C++" default object created***

knot: Celery Stalks at Midnight

**Exiting the block.**

***"Celery Stalks at Midnight" object deleted, 2 left***

***"Spinach Leaves Bowl for Dollars" object deleted, 1 left***

***"" object deleted, 0 left***

\*\*\* Error in `./a.out': double free or corruption (fasttop): 0x0000000000db5060 \*\*\*

Aborted (core dumped)

momo@HMI:~/C++PrimerPlus/Chapter12/BadString$

从结果上看,这个程序有下面几个现象和问题：

* 在执行到下面语句时：

***callme2(headline2);* // passed by value**

cout << "headline2: " << headline2 << endl;

输出为：

String passed by value:

"Lettuce Prey"

"Lettuce Prey" ***object deleted, 2 left***

headline2:

这里, cout那句的输出出现了错位, 并且出现了不该出现的语句；

First, passing headline2 ***by value instead of reference*** as a function argument ***somehow causes the destructor to be called***. **所以调用callme2(headline2)出现了destructor的调用.**

Second, although passing by value is supposed to protect the original argument from change, the function messes up the original string beyond recognition. **所以调用完callme2(headline2)后headline2的data没了.**

**[原因]**

在调用callme2(headline2)时, 因为这是pass by value, 所以headline2在执行之前被copy了一遍, 然后在callme2()调用结束时这个copy出来的temporary object被destructor释放. 问题就出在这里, object里的指向data的指针str在上述过程中也被copy了, 在释放temporary object时, destructor被调用, 它会 delete str, 也就是说, 在释放temporary object的过程中, 这个temporary object中str被delete了, 但这个temporary object的str和原始的headline2对象的str指向同一个data区域, 所以造成原始数据里的data被delete了.

* 随后, **程序创建(Initialize)了第4个object: sailor**, 并把sports赋值给了它.

**StringBad sailor = sports; // 这一句没有任何输出, 也就说没有调用自己定义的任何constructor, 也没有对计**

**// 数加一**

cout << "sailor: " << sailor << endl;

这两句的输出是：

sailor: Spinach Leaves Bowl for Dollars

**从输出上看, 赋值那一句没有任何输出.** **也就是说这里的创建object以及赋值没有调用程序定义的constructor; 注意, 没有调用程序里的constructor也就不会对object的count进行加一操作;**

* 然后, 程序创建**(Assign)**了第5个object: knot, 并把headline1赋值给了它:

StringBad knot; **// 调用了default constructor**

knot = headline1; **// 这一句执行时没有任何输出**

cout << "knot: " << knot << endl;

这三句的输出是：

**3:** "C++" default object created **// 明明是第5个被创建, 但计数只是3**

knot: Celery Stalks at Midnight

从输出上看, **第一句创建object调用了default constructor**, 也就是没有参数的那一个, **输出的第一行就是这一句产生的, 注意, 这里会对object数量的count加一**; 然后赋值语句对这个object进行参数设置, 没有任何输出;

* 最后, 在程序退出main中的inner block时, 因为几个object都是在这个block里面产生的, 这时会对这几个object进行注销:

"Celery Stalks at Midnight" object deleted, 2 left **// 这里注销的是knot**

"Spinach Leaves Bowl for Dollars" object deleted, 1 left **// 这里注销的是sailor**

"" object deleted, 0 left **// 这里注销的应该是sports. Sports只给sailor赋值了一**

**// 下, 它的data就变成了””**

\*\*\* Error in `./a.out': double free or corruption (fasttop): 0x0000000000db5060 \*\*\*

Aborted (core dumped)

在程序里, 我们一共创建了5个object: headline1(Celery开头), headline2(Lettuce开头), sports(Spinach开头), sailor == sports(创建sailor时没有调用定义的constructor), 和knot == headline1(创建knot时调用的定义的default constructor).

在这里注销时, 只出现了注销knot **(不是headline1)**, sailor**(不是sports)**, 和一个未知的, 字符串为空object(**其实是sports**)的现象, 然后就是memory access的错误.

Because **automatic storage objects are deleted in an order opposite to that in which they are created**, **the first three objects deleted** are *knots*, *sailor*, and *sport*. The knots and sailor deletions look okay, **but for sport, “Spinach Leaves Bowl for Dollars” has become “”**. **The only thing the program does with sport is use it to initialize sailor*, but that act appears to have altered sport***. **And the last two objects deleted, headline2 and headline1, are unrecognizable. Something messes up these strings before they are deleted.**

**[总结]**

程序里一共创建了5个object:

StringBad headline1("Celery Stalks at Midnight");

StringBad headline2("Lettuce Prey");

StringBad sports("Spinach Leaves Bowl for Dollars");

callme1(headline1); // Pass by reference

callme2(headline2); // Pass by value, **using copy constructor**

StringBad sailor = sports; **// 这一句没有任何输出, 也就说没有调用自己定义的任何constructor.**

**// Using copy constructor.**

StringBad knot; **// 调用了default constructor**

knot = headline1; **// 这一句执行时没有任何输出**

**当Free knot时, 由于knot和headline1的str其实指向同一个地址, 所以在释放knot时headline1的str所指向的data就已经被释放了; sailor和sports的关系类似; headline2在callme2()里做copy, 在callme2()结束时释放那个copy时也一样, str被deleted了.**

**What constructor is used here**?

StringBad sailor = sports;

*Not the default constructor*, and *not the constructor with a const char \* parameter*. Remember**, initialization using this form is another syntax for the following**:

**StringBad sailor = StringBad(sports); //constructor using sports**

Because sports is type StringBad, a matching constructor could have this prototype:

StringBad(**const StringBad &**);

And it turns out that the **compiler automatically generates this constructor** (called a ***copy constructor*** because it makes a copy of an object) if you initialize one object to another. The automatic version would not know about updating the num\_strings static variable, so it would mess up the counting scheme. Indeed, all the problems exhibited by this example stem from member functions that the compiler generates automatically, so let’s look at that topic now.

**Special Member Functions**

The problems with the StringBad class stem from *special member functions*. These are member functions that are defined automatically. In particular, C++ automatically provides the following member functions:

* A default constructor if you define no constructors;
* A default destructor if you don’t define one;
* A copy constructor if you don’t define one;
* An assignment operator if you don’t define one;
* An address operator if you don’t define one;

More precisely, the compiler generates definitions for the last three items if a program uses objects in such a way as to require them. For example, if you assign one object to another, the program provides a definition for the assignment operator. It turns out that the **implicit copy constructor** and the **implicit assignment operator cause the StringBad class problems**.

C++11 provides two more special member functions—the **move constructor** and the **move assignment operator**. Chapter 18,”Visiting with the New C++ Standard”, discusses these further.

**Default Constructors**

If you fail to provide any constructors at all, C++ provides you with a default constructor.

Klunk::Klunk() { } // implicit default constructor

That is, it supplies a constructor (the defaulted default constructor) that takes no arguments and that does nothing. It’s needed because creating an object always invokes a constructor:

Klunk lunk; **// invokes default constructor**

**A constructor with arguments still can be a default constructor** ***if all its arguments have default values***. For example, the Klunk class could have the following inline constructor:

Klunk(int n = 0) { klunk\_ct = n; }

However, you can have only one default constructor. **That is, you can’t do this:**

Klunk() { klunk\_ct = 0 } // constructor #1

Klunk(int n = 0) { klunk\_ct = n; } // ambiguous constructor #2

Why is this ambiguous? Consider the following two declarations:

Klunk kar(10); // clearly matches Klunt(int n)

Klunk bus; **// could match either constructor**

**The second declaration matches constructor #1 (no argument),** but **it also matches constructor #2 (using the default argument 0).**This will cause the compiler to issue an error message.

**Copy Constructors [Important!]**

A **copy constructor** is used to **copy an object to a newly created object**.

**StringBad sailor = sports; // 这一句没有任何输出, 也就说没有调用自己定义的任何constructor**

**// 这是initialization, 用的Copy constructor.**

StringBad knot; **// 调用了default constructor. 这一句是initialization.**

knot = headline1; **// 这是assignment, 不是initialization**

That is, it’s used during ***initialization***, including passing function arguments by value and ***not during ordinary assignment***. A copy constructor for a class normally has this prototype:

**Class\_name(const Class\_name &);// Copy constructor的参数是已存在的class object的reference**

Note that it takes a constant reference to a class object as its argument.

1. **When a Copy Constructor Is Used**

A **copy constructor is invoked** whenever a new object is created and **initialized** to **an existing object of the same kind**. **Initialization**指的是在**创建一个object的同时就给它初始化**, 不能先创建, 然后初始化, 那叫assignment. **判断是不是用copy constructor的两个条件: initialization, 用的同类的object.**

This happens in several situations. The most obvious situation is when you explicitly initialize a new object to an existing object. For example, given that motto is a StringBad object, **the following four defining declarations invoke a copy constructor**:

StringBad ditto(motto); // 1. calls StringBad(const StringBad &)

StringBad metoo = motto; // 2. calls StringBad(const StringBad &)

StringBad also = StringBad(motto); // 3. calls StringBad(const StringBad &)

StringBad \* pStringBad = new StringBad(motto); // 4. calls StringBad(const StringBad &)

**5. 当函数以object本身 (不是引用或者指针) 为参数, 或者返回值是object (不是引用或者指针) 时;**

Depending on the implementation, **the middle two** declarations may use a copy constructor **directly** to create metoo and also, or they may use a copy constructor to generate *temporary objects* whose contents are then assigned to metoo and also **(以StringBad为例, 如果用了temporary object, 再把它assign给目标object以后需要对这个temporary object进行free, 这个free会调用destructor, 会不会因此把str指向的data给free掉呢?)**. **The last example initializes an anonymous object** **to motto** and **assigns the address of the new object to the pStringBad pointer**.

Less obviously, a compiler uses a copy constructor whenever a program generates copies of an object. In particular, **it’s used when a function passes an object by value** (as callme2() does in Listing above) or **when a function returns an object**. Remember, passing by value means creating a copy of the original variable. A compiler also uses a copy constructor whenever it generates temporary objects.

Compilers vary as to when they generate temporary objects, but **all invoke a copy constructor when passing objects by value and when returning them**. In particular, this function call in Listing above invokes a copy constructor:

callme2(headline2);

By the way, the fact that **passing an object by value involves invoking a copy constructor is a good reason for passing by reference instead. That saves the time of invoking the constructor and the space for storing the new object.**

1. **What a Default Copy Constructor Does**

The default copy constructor performs a **member-by-member copy** of the ***non-static members*** (memberwise copying, also sometimes called **shallow copying**).

In Listing above, the statement

StringBad sailor = sports;

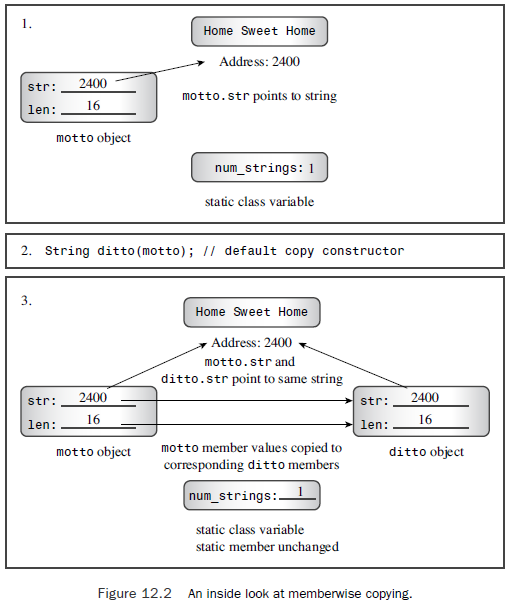
amounts to the following (**aside from the fact that it doesn’t compile because access to private members is not allowed**):

StringBad sailor;

sailor.str = sports.str;

sailor.len = sports.len;

If a member is itself a class object, the copy constructor for that class is used to copy one member object to another.



1. **Back to Stringbad: Where the Copy Constructor Goes Wrong**

在上面的程序里, **有两个地方调用了copy constructor**：callme2(headline2)和创建sailor, StringBad sailor = sports. Copy constructor不会去给object count加一; 但destructor仍然会用程序中创建的那个, 会给object count减一. This weirdness is a problem because it means the program doesn’t keep an accurate object count. **The solution is to provide an *explicit copy constructor* that does update the count**:

StringBad::StringBad(const StringBad & s)

{

num\_strings++;

... // important stuff to go here

}

**[Tips]** If your class has a static data member whose value changes when new objects are created, you should provide an explicit copy constructor that handles the accounting.

The second weirdness is the more subtle and dangerous.

void callme2(StringBad sb)

{

cout << "String passed by value:\n";

cout << " \"" << sb << "\"\n";

}

***callme2(headline2);* //** passed by value

cout << "headline2: " << headline2 << endl;

输出为：

String passed by value:

"Lettuce Prey"

"Lettuce Prey" ***object deleted, 2 left***

headline2:

callme2()**首先调用copy constructor产生一个temporary object**, 然后把headline2的值assign给这个temporary object, 其中有：

temporary object.str = headline2.str;

这样temporary object的str指针也指向headline2中的指针指向的字符串;

**当callme2()结束, destructor被调用去注销这个temporary object**; 先对object count减一, 然后delete了temporary object.str, 相当于把headline2的str指向的字符串给free了;

"Lettuce Prey" object deleted, 2 left

这一句输出是调用destructor是的输出; 因为headline2的str指向的字符串已经被delete了, 所以后面的:

cout << "headline2: " << headline2 << endl;

的输出就是空了:

headline2:

后面的

StringBad sailor = sports; **// 调用了copy constructor但没有产生temporary object；**

**// 按照Page7的说法, 这个根据compiler定;**

有一样的问题: 两个指针指向同一个字符串, 然后在程序最后注销sailor时destructor又会去free这个字符串, 就出现了最后报的错：

\*\*\* Error in `./a.out': double free or corruption (fasttop): 0x0000000000db5060 \*\*\*

Aborted (core dumped)

在程序里, 我们一共创建了5个object: headline1(Celery开头), headline2(Lettuce开头), sports(Spinach开头), sailor == sports(创建sailor时没有调用定义的constructor而是调用的copy constructor), 和knot == headline1(创建knot时调用的定义的default constructor).

在这里注销时, 只出现了注销knot **(不是headline1)**, sailor**(不是sports)**, 和一个未知的, 字符串为空object(**其实是sports, 但这时sports的字符串已经被free掉了**)的现象, 然后就是memory access的错误.

**[最后注意]**

callme2(headline2); // passed by value

**这一句首先调用copy constructor产生一个temporary object**, 然后把headline2的值assign给这个temporary object当作参数; **当callme2()结束, destructor被调用去注销这个temporary object;**

StringBad sailor = sports; **// 调用了copy constructor但没有产生temporary object；**

**// 按照Page7的说法, 这个根据compiler定;**

**Fixing the Problem by Defining an Explicit Copy Constructor**

The cure for the problems in the class design is to make a **deep copy**. That is, rather than just copy the address of the string, the ***copy constructor*** ***should*** duplicate the string and assign the address of the duplicate to the str member.

StringBad::StringBad(**const StringBad & st**)

{

num\_strings++; // handle static member update

len = st.len; // same length

str = new char [len + 1]; // allot space

**std::strcpy(str, st.str);** // copy string to new location

cout << num\_strings << ": \"" << str << "\" object created\n";

}

这里的explicit copy constructor里的copy叫deep copy; 前面的implicit copy constructor里的copy叫shallow copy.

**[Caution]**

**If a class contains members that are pointers initialized by *new*, you should define a copy constructor that copies the pointed-to data instead of copying the pointers themselves.** This is termed deep copying. The alternative form of copying (member-wise, or shallow, copying) just copies pointer values.

**More Stringbad Problems: Assignment Operators**

An overloaded **assignment operator is used** when you ***assign*** one object to another existing object:

StringBad headline1("Celery Stalks at Midnight");

...

StringBad knot; **// call default constructor;**

knot = headline1; **// then assignment operator invoked;**

An assignment operator is **not necessarily used** when ***initializing*** an object:

StringBad metoo = knot; **// use copy constructor, *possibly assignment*, too**

StringBad metoo(knot); **// same as above from compiler’s view;**

Here metoo is a newly created object being initialized to knot’s values; hence, ***the copy constructor is used***.

前一个例子里(knot那个), 先用***default*** constructor去构建一个default的object, 再利用=进行赋值; 后一个例子里, 尽管有等号, 但这个=在这里不一定有用, 因为copy constructor可以创建object并在其内部实现赋值;

**后一个例子里**: However, as mentioned before, implementations have the option of handling this statement in two steps: using the copy constructor to create a *temporary object* and then using assignment to copy the values to the new object. That is, **initialization *always* invokes a copy constructor**, and forms using the = operator may also invoke an assignment operator.

Just as ANSI C allows structure assignment, *C++ allows default class object assignment*. It does so by automatically overloading an assignment operator for a class. This operator has the following prototype:

Class\_name & Class\_name::operator=(const Class\_name &);

That is, it takes and returns a reference to an object of the class. For example, here’s the prototype for the StringBad class:

StringBad & StringBad::operator=(const StringBad &);

**Like a copy constructor, an *implicit implementation of an assignment operator performs a member-to-member copy*.**

所以, knot = headline1也会让他们俩的指针指向同一个字符串; **现象和用copy constructor一模一样, 但原因不同**.

**Fixing Assignment**

The solution for the problems created by an inappropriate default assignment operator is **to provide your own assignment operator definition, one that makes a deep copy**. The implementation is similar to that of the copy constructor, but there are some differences:

* Because the target object may already refer to previously allocated data, the function should use delete [] to free former obligations; **先delete掉target之前的数据;**
* The function **should protect against assigning an object to itself**; otherwise, the freeing of memory described previously could erase the object’s contents before they are reassigned;
* The function returns a reference to the invoking object;

*By returning an object*, the function can emulate the way ordinary assignment for built-in types can be chained. That is, if S0, S1, and S2 are StringBad objects, you can write the following:

S0 = S1 = S2;

In function notation, this becomes the following:

S0.operator=(S1.operator=(S2));

Here’s how you could write an assignment operator for the StringBad class:

**StringBad &** StringBad::operator=(const StringBad & st)

{

if (**this == *&*st**) // object assigned to itself

**return \*this;**  // all done

delete [] str; **// free old string of this**

len = st.len;

str = new char [len + 1]; // get space for new string

**std::strcpy(str, st.str); // copy the string**

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

}

The function proceeds to free the memory that str pointed to. The reason for this is that shortly thereafter str will be assigned the address of a new string. If you don’t first apply the delete operator, the previous string will remain in memory. Because the program no longer has a pointer to the old string, that memory will be wasted.

**Assignment does not create a new object**, so you don’t have to adjust the value of the ***static*** data member num\_strings.

Adding the copy constructor and the assignment operator described previously to the StringBad class clears up all the problems. 自己电脑上测试的输出结果为：

Exiting the block.

"Celery Stalks at Midnight" object deleted, 4 left

"Spinach Leaves Bowl for Dollars" object deleted, 3 left

"Spinach Leaves Bowl for Dollars" object deleted, 2 left

"Lettuce Prey" object deleted, 1 left

"Celery Stalks at Midnight" object deleted, 0 left

End of main()

在程序里, 我们一共创建了5个object: headline1(Celery开头), headline2(Lettuce开头), sports(Spinach开头), sailor == sports(创建sailor时没有调用定义的constructor而是调用的copy constructor), 和knot == headline1(创建knot时调用的定义的default constructor).

* **The New, Improved String Class**

Now that we are a bit wiser, we can revise the StringBad class, renaming it String. A useful String class would incorporate all the functionality of the standard cstring library of string functions, but we’ll add only enough to see what happens. We’ll add the following methods:

int length () const { return len; }

friend bool operator<(const String &st1, const String &st2);

friend bool operator>(const String &st1, const String &st2);

friend bool operator==(const String &st, const String &st2);

friend operator>>(istream & is, String & st);

**char & operator[](int i);**

**const char & operator[](int i) const;**

static int HowMany();

The first new method returns the length of the stored string. The next three friend functions allow you to compare strings. The operator>>() function provides simple input capabilities. *The two operator[]() functions provide array-notation access to individual characters in a string*. The static class method HowMany() complements the static class data member num\_strings. Let’s look at some details.

1. **The Revised Default Constructor**

The new default constructor merits notice. It look likes this:

String::String()

{

len = 0;

**str = new char[1]; // 不要用new char；**

**str[0] = '\0'**; // default string

}

You might wonder why the code uses

str = new char[1]; // compatible with destructor

and **NOT this**:

**str = new char;** // incompatible with destructor

**Both forms allocate the same amount of memory**. **The difference is that the first form is compatible with the class destructor and the second is not**. Recall that the destructor contains this code:

delete [] str;

Using **delete [] is compatible with pointers initialized by using new [] and with the null pointer**. So another possibility would be to replace:

str = new char[1];

str[0] = '\0'; // default string

with this:

str = 0; // sets str to the **null pointer**

The effect of using **delete []** with any pointers initialized any other way is undefined:

char \* p1 = words;

**char \* p2 = new char;**

char \* p3;

delete [] p1; // undefined, so don't do it

**delete [] p2; // undefined, so don't do it. 不是不能delete, 而是delete []不行**

delete [] p3; // undefined, so don't do it

**In C++98**, the literal 0 has two meanings—it can be the numeric value 0, and it can be the null pointer—thus making it difficult for the reader and compiler to distinguish between the two. Sometimes programmers use **(void \*) 0** to identify the pointer version.

**C++11** provides a better solution by introducing a new keyword, **nullptr**, to denote the null pointer. You still can use 0 as before—otherwise an enormous amount of existing code would be invalidated—but henceforth the recommendation is to use nullptr instead:

str = nullptr; // C++11 null pointer notation

1. **Comparison Member**

Three of the methods in the String class perform comparisons.

The **operator<()** function returns true if the first string comes before the second string alphabetically (or more precisely, in the machine collating sequence).

**bool** **operator<**(const String &st1, const String &st2)

{

if (std::**strcmp**(**st1.str**, **st2.str**) **<** 0)

return true;

else

return false;

}

Because the **built-in < operator** already returns a type bool value, you can simplify the code further to this:

bool operator<(const String &st1, const String &st2)

{

return (std::strcmp(st1.str, st2.str) **<** 0); **// here is built-in <**

}

Similarly, you can code the other two compar ison functions like this:

bool **operator>**(const String &st1, const String &st2)

{

return st2.str < st1.str; **// here is built-in <**

}

bool **operator==**(const String &st1, const String &st2)

{

return (std::strcmp(**st1.str**, **st2.str**) **== 0**); **// here is built-in ==**

}

**Making the comparison functions friends facilitates comparisons between String objects and regular C strings**. For example, suppose *answer* is a String object and that you have the following code:

**if ("love" == answer)**

This gets translated to the following:

if (operator==("love", answer))

**The compiler then uses one of the constructors to convert the code, in effect, to this:**

**if (operator==(*String("love")*, answer))**

And this matches the prototype.

1. **Accessing Characters by Using Bracket Notation**

**The bracket operator** places **one operand in front of the first bracket** and **the other operand between the two brackets.** Thus, in the expression **city[0], city is the first operand, [] is the operator, and 0 is the second operand.**

Suppose that opera is a String object:

*String opera*("The Magic Flute");

If you use the expression **opera[4]**, C++ looks for a method with this name and signature:

String::operator[](int i)

If it finds a matching prototype, the compiler replaces the expression opera[4] with this function call:

**opera.operator[](4)**

The opera object invokes the method, and the array subscript 4 becomes the function argument. Here’s a simple implementation:

**char &** String::operator[](int i)

{

**return str[i];**

}

With this definition, the statement

cout << opera[4];

becomes this:

cout << opera.operator[4];

The return value is opera.str[4], or the character 'M'. So the public method gives access to private data.

Declaring the return type as type**char &** allows you to assign values to a particular element. For example, you can use the following:

String means("might");

means[0] = 'r';

The second statement is converted to an overloaded operator function call:

means.operator[](0) = 'r';

This assigns 'r' to the method’s return value. But the function returns a reference to means.str[0], making the code equivalent to:

means.str[0] = 'r';

**This last line of code violates private access, but because operator[]() is a class method, it is allowed to alter the array contents.**

**Suppose you have a constant object:**

**const** String answer("futile");

Then, if the only available definition for operator[]()is the one you’ve just seen, the following code is labeled an error:

cout << answer[1]; **// compile-time error**

**The reason is that answer is const, and the first method doesn’t promise not to alter data.** However, C++ distinguishes between const and non-const function signatures when overloading, so you can provide a second version of operator[]() that is used just by const String objects:

// for use with const String objects

**const** char & String::operator[](int i) **const**

{

return str[i];

}

**前面的const char &代表返回的字符不能被修改; 最后面的const代表调用这个函数operator的object本身是const, 不会被修改.**

With the definitions, you have read/write access to regular String objects and read only access to const String data:

String text("Once upon a time");

const String answer("futile");

cout << text[1]; // ok, uses non-const version of operator[]()

cout << answer[1]; // ok, uses const version of operator[]()

cin >> text[1]; // ok, uses non-const version of operator[]()

cin >> answer[1]; // compile-time error

1. **Static Class Member Functions**

**It’s possible to declare a member function as being static.** The keyword static should appear in the function declaration but not in the function definition if the latter is separate. **如果函数声明和函数定义是分开的, static关键字要摆在声明前面**. This has **two important consequences**.

First, **a static member function doesn’t have to be invoked by *an object***; in fact, **it doesn’t even get a *this* pointer** to play with. If the static member function is declared in the ***public section***, it can be invoked using the **class name (not the object name)** and the **scope-resolution operator**.

For instance, you can give the String class a static member function called HowMany() with the following prototype/definition in the class declaration:

**static int HowMany() { return num\_strings; }** //这里函数声明和函数定义没分开

这里的num\_strings是在private里的static int, 函数HowMany是在public里的函数.

It could be invoked like this:

int count = **String::HowMany();** // invoking a static member **function**

**The second consequence** is that because a **static member function is not associated with a particular object, the only data members it can use are the static data members.** For example, the *HowMany()* static method can access the *num\_strings* static member, **but not *str* or *len***. **str和len是class的private data member, HowMany()不能直接访问, 也不能通过加::来访问, 必须通过public member function来访问.**

1. **Further Assignment Operator Overloading**

Suppose you want to copy an ordinary string to a String object. For example, suppose you use getline() to read a string and you want to place it in a String object. The class methods already allow you to do the following:

String name;

char temp[40];

cin.getline(temp, 40);

**name = temp; // use constructor to convert ‘temp’ to String type**

在前面章节介绍的=是用于将一个object赋值给另一个object的, 比如knot = headline1这种. **这里, 之前的=仍然可以用, 原因如下**:

1. The program uses the **String(const char \*) constructor to construct a temporary String object containing a copy of the string stored in temp**. Remember from Chapter 11,”Working with Classes”, that a **constructor with a single argument serves as a conversion function**;
2. In Listing 12.6, later in this chapter, the program uses the **String & String::operator=( const String &)** function to deep copy information from the temporary object to the name object;
3. The program calls the **~String()** destructor to **delete the temporary object**;

但这个方法太复杂了.

**The simplest way** to make the process more efficient is to **overload the assignment operator so that it works directly with ordinary strings**. **This removes the extra steps of creating and destroying a temporary object.**

String & String::operator=(const char \* s)

{

delete [] str;

len = std::strlen(s);

str = new char[len + 1];

std::strcpy(str, s);

**return \*this**;

}

**Demonstration code of new String**

// **string1.h** -- fixed and augmented string class definition

#ifndef STRING1\_H\_

#define STRING1\_H\_

#include <iostream>

using std::ostream;

using std::istream;

class String

{

private:

char \* str; // pointer to string

int len; // length of string

static int num\_strings; // number of objects

static const int CINLIM = 80; // cin input limit

public:

// constructors and other methods

String(const char \* s); // constructor

String(); // default constructor

String(const String &); // **copy constructor**

~String(); // destructor

int length () **const** { return len; }

// overloaded operator methods

String & **operator=**(const String &);

String & **operator=**(const char \*);

char & operator[](int i);

const char & operator[](int i) const;

// overloaded operator friends

friend bool operator<(const String &st, const String &st2);

friend bool operator>(const String &st1, const String &st2);

friend bool operator==(const String &st, const String &st2);

friend ostream & operator<<(ostream & os, const String & st);

friend istream & operator>>(istream & is, String & st);

// static function

static int HowMany();

};

#endif

Method definitions below:

// **string1.cpp** -- String class methods

#include <cstring> // string.h for some

#include "string1.h" // includes <iostream>

using std::cin;

using std::cout;

// initializing static class member

int **String**::num\_strings = 0;

// static method

int **String::HowMany()**

{

return num\_strings;

}

// class methods

**String::String(const char \* s)** // construct String from C string

{

len = std::strlen(s); // set size

str = new char[len + 1]; // allot storage

std::strcpy(str, s); // initialize pointer

num\_strings++; // set object count

}

**String::String()** // default constructor

{

len = 4;

str = new char[1];

str[0] = '\0'; // default string

num\_strings++;

}

**String::String(const String & st)** // **copy constructor**

{

num\_strings++; // handle static member update

len = st.len; // same length

str = new char [len + 1]; // allot space

std::strcpy(str, st.str); // copy string to new location

}

String::~String() // necessary destructor

{

--num\_strings; // required

delete [] str; // required

}

**// overloaded operator methods**

// assign a String to a String

String & String::operator=(const String & st)

{

if (this == &st)

return \*this;

delete [] str;

len = st.len;

str = new char[len + 1];

std::strcpy(str, st.str);

return \*this;

}

// assign a C string to a String

String & String::operator=(const char \* s)

{

delete [] str;

len = std::strlen(s);

str = new char[len + 1];

std::strcpy(str, s);

return \*this;

}

// read-write char access for non-const String

char & String::operator[](int i)

{

return str[i];

}

// read-only char access for const String

const char & String::operator[](int i) const

{

return str[i];

}

// overloaded operator friends, parameters are all class String

bool operator<(const String &st1, const String &st2)

{

return (std::strcmp(st1.str, st2.str) < 0);

}

bool operator>(const String &st1, const String &st2)

{

return st2 < st1;

}

bool operator==(const String &st1, const String &st2)

{

return (std::strcmp(st1.str, st2.str) == 0);

}

// simple String output

ostream & operator<<(ostream & os, const String & st)

{

os << st.str;

return os;

}

**// quick and dirty String input**

**istream &** operator>>(istream & is, String & st)

{

char temp[String::CINLIM];

is.get(temp, String::CINLIM);

if (is)

st = temp;

while (is && is.get() != '\n')

continue;

return is;

}

一个测试主程序：

// **sayings1.cpp** -- using expanded String class

// compile with string1.cpp

#include <iostream>

#include "string1.h"

const int ArSize = 10;

const int MaxLen =81;

int main()

{

using std::cout;

using std::cin;

using std::endl;

**String name**;

cout <<"Hi, what's your name?\n>> ";

cin **>>** name;

cout **<<** name << ", please enter up to " << ArSize

<< " short sayings <empty line to quit>:\n";

String sayings[ArSize]; // array of objects, **[] here is built-in one**

char temp[MaxLen]; // temporary string storage

int i;

for (i = 0; i < ArSize; i++)

{

cout << i+1 << ": ";

cin.get(temp, MaxLen);

while (cin && cin.get() != '\n') **// discard remained characters in input queen**

continue;

**if (!cin || temp[0] == '\0')** **// empty line?**

break; // i not incremented

else

**sayings[i] = temp**; // **overloaded assignment**

}

int total = i; // total # of lines read

if ( total > 0)

{

cout << "Here are your sayings:\n";

for (i = 0; i < total; i++)

cout << **sayings[i][0]** << ": " << **sayings[i]** << endl;

int shortest = 0;

int first = 0;

for (i = 1; i < total; i++)

{

if (sayings[i].length() < sayings[shortest].length())

shortest = i;

if (sayings[i] < sayings[first])

first = i;

}

cout << "Shortest saying:\n" << sayings[shortest] << endl;;

cout << "First alphabetically:\n" << sayings[first] << endl;

cout << "This program used "<< String::HowMany()

<< " String objects. Bye.\n";

}

else

cout << "No input! Bye.\n";

return 0;

}

* **Things to Remember When Using new in Constructors**

By now you’ve noticed that you must take special care when using new to initialize pointer members of an object. In particular, you should do the following:

* If you use new to initialize a pointer member in a constructor, you should use delete in the destructor;
* The uses of new and delete should be compatible. You should pair **new** with **delete** and **new []** with **delete []**;
* If there are multiple constructors, all should use new the same way - either all with brackets or all without brackets. **There’s only one destructor**, so all constructors have to be compatible with that destructor. However, it is permissible to initialize a pointer with new in one constructor and with the null pointer (0, or, with C++11, nullptr) in another constructor because **it’s okay to apply the delete operation (with or without brackets) to the null pointer**.
* **You should define a copy constructor that initializes one object to another by doing deep copying.** Typically, the constructor should emulate the following example:

String::String(const String & st)

{

num\_strings++; // handle static member update if necessary

len = st.len; // same length as copied string

str = new char [len + 1]; // allot space

std::strcpy(str, st.str); // copy string to new location

}

In particular, the copy constructor should allocate space to hold the copied data, and it should copy the data, not just the address of the data. Also it should update any static class members whose value would be affected by the process.

* **You should define an assignment operator** that copies one object to another by doing deep copying.

String & String::operator=(const String & st)

{

if (this == &st) // object assigned to itself

return \*this; // all done

delete [] str; // free old string

len = st.len;

str = new char [len + 1]; // get space for new string

std::strcpy(str, st.str); // copy the string

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

}

**Memberwise Copying for Classes with Class Members**

Suppose you use the String class, or, for that matter, the **standard string class** as a type for class members:

class Magazine

{

private:

**String** title; // String class defined before

**string** publisher; // Standard string class

...

};

String and string both use dynamic memory allocation. Does that mean you need to write a copy constructor and assignment operator for the Magazine class? **No, at least not in itself**. The default member-wise copying and assignment behavior does have some smarts. If you copy or assign one Magazine object to another, **member-wise copying uses the copy constructors and assignment operators defined for the member types**. That is, the String copy constructor will be used to copy the title member from one Magazine object to another, the String assignment operator will be used to assign the title member of one Magazine object to another, and so on. Things get more complicated, however, if the Magazine class needs a copy constructor and assignment operator for some other class member. In that case, those functions have to call the String and string copy constructors and assignment operators **explicitly**. But that’s a tale for Chapter 13,“Class Inheritance.”

* **Observations About Returning Objects**

When a member function or standalone function **returns an object**, you have choices. The function could return a reference to an object, a constant reference to an object, an object, or a constant object. 前面3个都已经见过了, 现在来看最后一个.

* **Returning a Reference to a const Object**

**The usual reason for using a const reference is efficiency**. Suppose you wanted to write a function Max() that returned the larger of two Vector objects, where Vector is the class developed in Chapter 11.The function would be used in this manner:

Vector force1(50,60);

Vector force2(10,70);

Vector max;

max = Max(force1, force2);

Either of the following two implementations would work:

**// version 1**

**Vector** Max(const Vector & v1, const Vector & v2) **// return object**

{

if (v1.magval() > v2.magval())

return v1;

else

return v2;

}

**// version 2**

**const Vector &** Max(const Vector & v1, const Vector & v2) **// return const object**

**// reference**

{

if (v1.magval() > v2.magval())

return v1;

else

return v2;

}

There are three important points here.

1. First, recall that ***returning an object* invokes the *copy constructor to build a temporary object***, whereas returning a reference doesn’t. Thus **Version 2 does less work and is more efficient**;
2. Second, the reference should be to an object that exists when the calling function is executing;
3. Third, both v1 and v2 are declared as being const references, so the return type has to be const to match; 这里是说你要返回的是V1和V2中的一个, 所以返回类型要是const, 这是针对这个例子而言的, 而不是一个宽泛的规则;

* **Returning a Reference to a Non-const Object**

Two common examples of **returning a non-const object** are **overloading the assignment operator** and **overloading the << operator** for use with cout. The first is done for reasons of efficiency, and the second for reasons of necessity.

The return value of operator=() is used for chained assignment:

String s1("Good stuff");

String s2, s3;

s3 = s2 = s1;

In this code, the return value of **s2.operator=(s1)** is assigned to s3. Returning either a String object or a reference to a String object would work, but, as with the Vector example, using a reference allows the function to avoid calling the String copy constructor to create a new String object. **In this case, the *return type is not const* because the operator=() method returns a reference to s2, which it does modify.**

// assign a String to a String

**String &** String::operator=(const String & st)

{

if (this == &st)

return \*this;

delete [] str;

len = st.len;

str = new char[len + 1];

std::strcpy(str, st.str);

return \*this;

}

The return value of operator<<() is used for chained output:

String s1("Good stuff");

cout << s1 << "is coming!";

Here, **the return value of operator<<(cout, s1) becomes the object used to display the string "is coming!".** **Here, the return type has to be ostream & and not just ostream.** Using an ostream return type would require calling the ostream copy constructor, and, as it turns out, the **ostream class does not have a public copy constructor.** Fortunately, returning a reference to cout poses no problems because cout is already in scope in the calling function.

* **Returning an Object**

If the object being returned is local to the called function, then it should not be returned by reference because the local object has its destructor called when the function terminates. Thus, when control returns to the calling function, there is no object left to which the reference can refer. In these circumstances, you should return an object, not a reference. **Typically, overloaded arithmetic operators fall into this category.**

Vector force1(50,60);

Vector force2(10,70);

Vector net;

net = force1 **+** force2;

The value being returned is not force1, which should be left unaltered by the process, nor force2, which should also be unaltered. Thus the return value can’t be a reference to an object that is already present in the calling function. Instead, the sum is a new, temporary object computed in Vector::operator+(), and the function shouldn’t return a reference to a temporary object either. Instead, it should return an actual vector object, not a reference:

Vector Vector::operator+(**const** Vector & b) **const**

{

return Vector(x + b.x, y + b.y);

}

()里的const代表+ operator的参数Vector object不会被修改; **函数最后的const代表代表调用这个+ operator的Vector object不会被修改;**

There is the added expense of calling the **copy constructor** to create the returned object, but that is unavoidable. 过程是这样的: 首先Vector(x + b.x, y + b.y)会调用constructor (not copy constructor)构建一个Vector的object, **这个object不是被return的object**; **然后再调用copy constructor把前面那个object复制作为returned object**; 然后在calling function里net **=** force1 + force2**会调用assignment operator做对这个returned object做deep copy**.

* **Returning a const Object**

The preceding definition of Vector::operator+() has a bizarre property. The intended use is this:

net = force1 + force2; // 1: three Vector objects

However, the definition also allows you to use the following:

force1 + force2 = net; // 2: dyslectic programming

cout << (force1 + force2 = net).magval() << endl; // 3: demented programming

In Statement 1, the temporary object is assigned to net. In Statements 2 and 3, net is assigned to the temporary object.

The temporary object is used and then discarded. For instance, in Statement 2, the program computes the sum of force1 and force2, copies the answer into the temporary return object, overwrites the contents with the contents of net, and **then discards the temporary object. The original vectors are all left unchanged.** In Statement 3, the magnitude of the temporary object is displayed before the object is deleted.

If you are concerned about the potential for misuse and abuse created by this behavior, you have a simple recourse: **Declare the return type as a const object**. **For instance, if Vector::operator+() is declared to have return type const Vector, then Statement 1 is still allowed but Statements 2 and 3 become invalid.**

**[Summary]**

In summary, if a method or function returns a local object, it should return an object, not a reference. In this example, **the** **program uses the copy constructor to generate the returned object**. If a method or function returns an object of a class for which there is no public copy constructor, such as the ostream class, it must return a reference to an object. Finally, some methods and functions, such as the overloaded assignment operator, can return either an object or a reference to an object. In this example, the reference is preferred for reasons of efficiency.

* **Using Pointers to Objects**

// **sayings2.cpp** -- using pointers to objects

// compile with string1.cpp

#include <iostream>

#include <cstdlib> // (or stdlib.h) for rand(), srand()

#include <ctime> // (or time.h) for time()

#include "string1.h"

const int ArSize = 10;

const int MaxLen = 81;

int main()

{

using namespace std;

String name;

cout <<"Hi, what's your name?\n>> ";

cin >> name;

cout << name << ", please enter up to " << ArSize

<< " short sayings <empty line to quit>:\n";

**String sayings[ArSize];**

char temp[MaxLen]; // temporary string storage

int i;

for (i = 0; i < ArSize; i++)

{

cout << i+1 << ": ";

cin.get(temp, MaxLen);

while (cin && cin.get() != '\n')

continue;

if (!cin || temp[0] == '\0') // empty line?

break; // i not incremented

else

**sayings[i] = temp;** **// overloaded assignment for C style string**

}

int total = i; // total # of lines read

if (total > 0)

{

cout << "Here are your sayings:\n";

for (i = 0; i < total; i++)

cout << sayings[i] << "\n";

**// use pointers to keep track of shortest, first strings**

**String \* shortest = &sayings[0];** // initialize to first object

**String \* first = &sayings[0];**

for (i = 1; i < total; i++)

{

if (sayings[i]**.**length() < shortest**->**length())

shortest = &sayings[i];

if (sayings[i] **<** \*first) // **compare values using overloaded <**

first = &sayings[i]; // assign address

}

cout << "Shortest saying:\n" << \* shortest << endl;

cout << "First alphabetically:\n" << \* first << endl;

srand(time(0));

int choice = rand() % total; // pick index at random

// use new to create, initialize new String object

**String \* favorite = new String(sayings[choice]);**

**cout << "My favorite saying:\n" << \*favorite << endl;**

**delete favorite;**

}

else

cout << "Not much to say, eh?\n";

cout << "Bye.\n";

return 0;

}

Initially, the shortest pointer points to the first object in the array. Each time the program finds an object with a shorter string, it resets shortest to point to that object. Similarly, a first pointer tracks the alphabetically earliest string. Note that ***these two pointers do not create new objects***; they merely point to existing objects. Hence they don’t require using new to allocate additional memory.

String \* favorite = new String(sayings[choice]);

Here the pointer favorite provides the only access to the nameless object created by new. This particular syntax means to initialize the new String object by using the object sayings[choice]. **That invokes the copy constructor because the argument type for the copy constructor (const String &) matches the initialization value (sayings[choice]).**

**Object Initialization with new**

In general, if *Class\_name* is a class and if *value* is of type *Type\_name*, the statement

Class\_name \* pclass = new Class\_name(value);

invokes this constructor:

Class\_name(Type\_name);

There may be trivial conversions, such as to this:

Class\_name(const Type\_name &);

Also the usual conversions invoked by prototype matching, such as from int to double, takes place as long as there is no ambiguity. An initialization in the following form invokes the **default constructor**:

Class\_name \* ptr = new Class\_name;

**Looking Again at new and delete**

String \* favorite = new String(sayings[choice]);

**This allocates space** not for the string to be stored but **for the object**, that is, for the str pointer that holds the address of the string and for the len member. **(It does not allocate space for the *num\_strings* member because it is a static member that is stored separately from the objects.)** Creating the object, in turn, calls the **constructor**, **which allocates space for storing the string and assigns the string’s address to str**. The program then uses delete to delete this object when it is finished with it. The object is a single object, so the program uses delete without brackets. Again, **this(delete operation) frees only the space used to hold the str pointer and the len member. It doesn’t free the memory used to hold the string str points to, but the destructor takes care of that final task.**

意思是说, 上面的那行只是给str指针本身分配了内存,以及len. 它并没有给str指向的字符串分配内存; 然后, 上面那行会调用constructor, 去给str指向的字符串分配内存, 然后把这段内存地址交给str;

当delete时一样, delete只是free和str指针本身和len, 至于str指向的字符串的内存是由destructor负责的;

**If an object is created by new, its destructor is called only when you explicitly use delete on the object.**

**Pointers and Objects Summary**

String \* glamour;

String \* first = &sayings[0];

String \* favorite = new String(sayings[choice]);

// invokes **default constructor**

String \* gleep = new String;

// invokes the **String(const char \*) constructor**

String \* glop = new String("my my my");

// invokes the **String(const String &) constructor**

String \* favorite = new String(sayings[choice]);

// You use the -> operator to access a class method via a pointer:

if (sayings[i].length() < shortest->length())

if (sayings[i] < \*first) // compare object values

**first = &sayings[i];** // assign object address

**Looking Again at Placement of new**

// **placenew1.cpp** -- new, placement new, no delete

#include <iostream>

**#include <string>**

**#include <new>**

using namespace std;

const int BUF = 512;

class JustTesting

{

private:

string words;

int number;

public:

JustTesting(**const string & s = "Just Testing"**, int n = 0)

{words = s; number = n; cout << words << " constructed\n"; }

~JustTesting() { cout << words << " destroyed\n";}

void Show() const { cout << words << ", " << number << endl;}

};

int main()

{

**char \* buffer = new char[BUF];** // get a block of memory

**JustTesting \*pc1, \*pc2;**

**pc1 = new (buffer) JustTesting;** **// place object in buffer**

pc2 = new JustTesting("Heap1", 20); **// place object on heap**

cout << "Memory block addresses:\n" << "buffer: "

<< **(void \*) buffer** << " heap: " << pc2 <<endl;

cout << "Memory contents:\n";

**cout << pc1 << ": "; // 这里pc1是个地址, 输出的也是个地址**

pc1->Show();

**cout << pc2 << ": "; // 这里pc2是个地址, 输出的也是个地址**

pc2->Show();

**JustTesting \*pc3, \*pc4;**

**pc3 = new (buffer) JustTesting("Bad Idea", 6);**

pc4 = new JustTesting("Heap2", 10);

cout << "Memory contents:\n";

cout << pc3 << ": ";

pc3->Show();

cout << pc4 << ": ";

pc4->Show();

**delete pc2;** // free Heap1

**delete pc4;** // free Heap2

**delete [] buffer;** // free buffer

cout << "Done\n";

return 0;

}

在自己电脑上的测试结果：

momo@HMI:~/C++PrimerPlus/Chapter12/Placenew$ ./a.out

Just Testing constructed

Heap1 constructed

Memory block addresses:

buffer: 0x9dd010 heap: 0x9dd280

**Memory contents:**

0x9dd010: Just Testing, 0

0x9dd280: Heap1, 20

Bad Idea constructed

Heap2 constructed

Memory contents:

0x9dd010: Bad Idea, 6

0x9dd300: Heap2, 10

Heap1 destroyed

Heap2 destroyed

Done

There are a couple problems with placement new as used in Listing above.

First, when creating a second object pc3, placement new simply overwrites the same location used for the first object, pc1, with a new object. Not only is this rude, **it means that the destructor was never called for the first object.** This, of course, would create real problems if, say, the class used dynamic memory allocation for its members.

Second, using delete with pc2 and pc4 automatically invokes the destructors for the two objects that pc2 and pc4 point to. **But using delete [] with buffer does not invoke the destructors for the objects created with placement new.**

One lesson to be learned here is the same lesson you learned in Chapter 9: **It’s up to you to manage the memory locations in a buffer that placement new populates**.

**To use two different locations**, you provide two different addresses within the buffer, making sure that the locations don’t overlap.

pc1 = new (buffer) JustTesting;

pc3 = new (**buffer + sizeof (JustTesting)**) JustTesting("Better Idea", 6);

**The second lesson** to be learned here is that if you use placement new to store objects, you need to arrange for their destructors to be called. But how?

For objects created on the heap, you can use this:

delete pc2; // delete object pointed to by pc2

**But you can’t use this:**

**delete pc1;** // delete object pointed to by pc1? NO!

**delete pc3;** // delete object pointed to by pc3? NO!

**The reason is that *delete* works in conjunction with *new* but not with *placement new*. The pointer pc3, for example, does not receive an address returned by new, so delete pc3 throws a runtime error**. **The pointer pc1, on the other hand, has the same numeric value as buffer, but buffer is initialized using new [], so it’s freed using delete [], not delete**. **Even if buffer were initialized by new instead of new [], delete pc1 would free buffer, not pc1.**That’s because the new/delete system knows about the 256-byte block that is allocated, but it doesn’t know anything about what placement new does with the block.

Note that the program does free the buffer:

delete [] buffer; // free buffer

As this comment suggests, delete [] buffer; deletes the entire block of memory allocated by new. But it doesn’t call the destructors for any objects that placement new constructs in the block.

**The solution to this quandary is that you must call the destructor explicitly for any object created by placement new.** Normally, destructors are called automatically; this is one of the rare cases that require an explicit call. An explicit call to a destructor requires identifying the object to be destroyed. Because there are pointers to the objects, you can use these pointers:

**pc3->~JustTesting();** // destroy object pointed to by pc3

**pc1->~JustTesting();** // destroy object pointed to by pc1

**One important fact is the proper order of deletion**. The **objects constructed by placement new should be destroyed *in order opposite that in which they were constructed***. The reason is that, in principle, a later object might have dependencies on an earlier object. And the buffer used to hold the objects should be freed only after all the contained objects are destroyed.

* **Reviewing Techniques**

**Overloading the << Operator**

ostream & operator<<(ostream & os, const c\_name & obj)

{

os << ... ; // display object contents

return os;

}

对这个操作符的重载一般是friend, 可以直接access class的private member.

**Conversion Functions**

**To convert a single value to a class type, you create a class constructor with following prototype:**

c\_name(type\_name value);

Here *c\_name* represents the class name, and *type\_name* represents the name of the type you want to convert.

**Classes Whose Constructors Use new**

You need to take several precautions **when designing classes that use the new operator to allocate memory pointed to by a class member**.

* Any class member that points to memory allocated by new should have the delete operator applied to it in the class destructor. This frees the allocated memory;
* **If a destructor frees memory by applying delete to a pointer that is a class member, every constructor for that class should initialize that pointer, either by using new or by setting the pointer to the null pointer;**
* Constructors should settle on using either new [] or new, **but not a mixture of both**. The destructor should use delete [] if the constructors use new [], and it should use delete if the constructors use new;
* You should define a **copy constructor** that allocates new memory rather than copying a pointer to existing memory. This enables a program to initialize one class object to another;
* You should define a ***class member function*** that **overloads the assignment operator** and that has a function definition with the following prototype. The following example assumes that the constructors initialize the variable c\_pointer by using new []:

c\_name & c\_name::operator=(const c\_name & cn)

{

if (this == & cn)

return \*this; // done if self-assignment

delete [] c\_pointer;

// set size number of type\_name units to be copied

c\_pointer = new type\_name[size];

// then copy data pointed to by cn.c\_pointer to

// location pointed to by c\_pointer

...

return \*this;

}

* **A Queue Simulation**

The items in the queue will be customers. A Bank of Heather representative tells you that, on average, a third of the customers will take one minute to be processed, a third will take two minutes, and a third will take three minutes. Furthermore, **customers arrive at random intervals**, but the **average number of customers per hour is fairly constant**.

* A queue holds an ordered sequence of items;
* A queue has a limit on the number of items it can hold;
* You should be able to create an empty queue;
* You should be able to check whether a queue is empty;
* You should be able to check whether a queue is full;
* You should be able to add an item to the end of a queue;
* You should be able to remove an item from the front of a queue;
* You should be able to determine the number of items in the queue;

class **Queue**

{

enum {Q\_SIZE = 10}; // Q\_SIZE is default queue size;

private:

// private representation to be developed later

public:

Queue(**int qs = Q\_SIZE**); // create queue with a qs limit

~Queue();

bool isempty() const;

bool isfull() const;

int queuecount() const;

bool enqueue(const **Item &** item); // add item to end

bool dequeue(Item & item); // remove item from front

};

The constructor creates an empty queue. By default, the queue can hold up to 10 items, but that can be overridden with an explicit initialization argument:

Queue line1; // queue with 10-item limit

**Queue line2(20);** // queue with 20-item limit

**The Queue Class Implementation**

First, you have to decide how to represent the queue data. One approach is to use *new* to dynamically allocate **an array** with the required number of elements. However, **arrays aren’t a good match to queue operations**. For example, removing an item from the front of the array should be followed up by shifting every remaining element one unit closer to the front. Otherwise, you need to do something more elaborate, such as treat the array as circular.

**Using a linked list,** however, is a reasonable fit to the requirements of a queue. For the queue in this example, each data part is a type Item value, and you can use a structure to represent a node:

struct Node

{

Item item; // data stored in the node

struct Node \* next; // pointer to next node

};

Because a queue always adds a new item to the end of the queue, it is convenient to have a data member point to the last node.

class Queue

{

private:

// class scope definitions

// Node is a nested structure definition local to this class

**struct Node** **{ Item item; struct Node \* next;};**

enum {**Q\_SIZE = 10**};

// private class members

Node \* front; // pointer to front of Queue

Node \* rear; // pointer to rear of Queue

int items; // **current number of items** in Queue

**const int qsize**; // **maximum number of items** in Queue

...

public:

Queue(**int qs = Q\_SIZE**); // create queue with a qs limit

~Queue();

bool isempty() const;

bool isfull() const;

int queuecount() const;

bool enqueue(const **Item &** item); // add item to end

bool dequeue(Item &item); // remove item from front

};

The declaration uses the C++ ability to nest a structure or class declaration inside a class. **By placing the Node declaration inside the Queue class**, you give it class scope. That is, Node is a type that you can use to declare class members and as a type name in class methods, **but the type is restricted to the class.** That way, you don’t have to worry about this declaration of Node conflicting with some global declaration or with a Node declared inside some other class.

A *structure*, a *class*, or an ***enumeration*** declared **within a class** declaration is said to be **nested in the class**. It has class scope. ***Such a declaration doesn’t create a data object.*** Rather, it specifies a type that can be used internally within the class. **If the declaration is made in the private section** of the class, then the declared type can be used only within the class. **If the declaration is made in the public section**, then the declared type can also be used out of the class, through use of the scope-resolution operator. For example, if Node were declared in the public section of the Queue class, you could declare variables of ***type*** Queue::Node outside the Queue class.

**The Class Methods**

A class constructor should provide values for the class members. Also you should set the maximum queue size qsize to the constructor argument qs. Here’s an implementation that does not work:

Queue::Queue(int qs)

{

front = rear = NULL;

items = 0;

qsize = qs; **// Not Acceptable!**

}

The problem is that **qsize is a const**, so it ***can be initialized*** to a value, but it ***can’t be assigned*** a value.

Conceptually, ***calling a constructor* *creates an object* *before the code within the brackets is executed***. Thus, **calling the Queue(int qs) constructor causes the program to first allocate space for the four member variables(对qsize而言, 这时就已经被初始化了).** Then program flow enters the brackets and uses ordinary assignment to place values into the allocated space. **Therefore, if you want to initialize a const data member, you have to do so when the object is created before execution reaches the body of the constructor.**

C++ provides a special syntax for doing just that. It’s called a member initializer list.

Queue::Queue(int qs) **: qsize(qs)** // initialize qsize to qs

{

front = rear = NULL;

items = 0;

}

The technique is not limited to initializing constants; you can also write the Queue constructor like this:

**Queue::Queue(int qs) : qsize(qs), front(NULL), rear(NULL), items(0)**

{

}

Only constructors can use this initializer-list syntax. As you’ve seen, you have to use this syntax for const class members. You also have to use it for **class members** that are declared as **references**:

class **Agency** {...};

class **Agent**

{

private:

**Agency & belong**; // must use initializer list to initialize

...

};

Agent::**Agent(Agency & a) : belong(a)**

**{...}**

**That’s because references, like const data, can be initialized only when created**. As you’ll see in Chapter 14, however, it’s more efficient to use the member initializer list for members that are themselves class objects.

**The Member Initializer List Syntax**

If Classy is a class and if mem1, mem2, and mem3 are class data members, a class constructor can use the following syntax to initialize the data members:

Classy::Classy(int n, int m) : mem1(n), mem2(0), mem3(n\*m + 2)

{

//...

}

This initializes mem1 to n, mem2 to 0, and mem3 to n\*m + 2. Conceptually, these initializations take place when the object is created and before any code within the brackets is executed. Note the following:

* This form can be used only with constructors;
* You must (at least, in pre-C++11) use this form to initialize a **nonstatic const data member**;
* You must use this form to initialize a **reference data member**;

**Data members are initialized in the order in which they appear in the class declaration, not in the order in which initializers are listed.**

**C++11 Member In-Class Initialization**

**C++11** allows you to do what would seem to be the intuitively obvious thing to do:

class Classy

{

int mem1 = 10; // in-class initialization

**const int mem2 = 20; // C++ 11 in-class initialization, default setting**

//...

};

The members *mem1* and *mem2* get initialized to 10 and 20, respectively, **unless a constructor using a member initialization list is called**. Then the actual list overrides these default initializations:

Classy::Classy(int n) : mem1(n) {...}

In this case, the constructor would use the value of n to initialize mem1, and mem2 still would be set to 20.

The code for isempty(), isfull(), and queuecount() is simple. If items is 0, the queue is empty. If items is qsize, the queue is full. Returning the value of items answers the question of how many items are in the queue.

**Adding an item to the rear of the queue** (enqueuing) is more involved. Here is one approach:

bool Queue::enqueue(const Item & item)

{

if (isfull())

**return false**;

Node \* add = new Node; // create node

**// on failure, new throws std::bad\_alloc exception**

add->item = item; // set node pointers using parameters

add->next = NULL; // or nullptr;

items++;

if (front == NULL) // if queue is empty,

front = add; // place item at front

else

rear->next = add; // else place at rear

rear = add; // have rear point to new node

**return true**;

}

rear指向的是queue里的当前最后一个成员, 所以上面加绿的两句的意思是先把目前队列的最后一个成员的next指针指向新成员, 然后再更新rear指针到这个新成员上.

**Removing an item from the front of the queue** (dequeuing) also has several steps. Here is one approach:

bool Queue::dequeue(**Item & item**) // 参数item是用来装data of queue front的

{

if (front == NULL)

return false;

**item** = front->item; // Provide the first item in the queue to the calling function

items--;

**Node \* temp = front;** **// save location of first item**

front = front->next; // reset front to next item

**delete temp; // delete former first item**

if (items == 0)

rear = NULL;

return true;

}

**Other Class Methods?**

The class constructor doesn’t use *new*, but adding objects to a queue does invoke *new* to create new nodes. It’s true that the dequeue() method cleans up by deleting nodes, but there’s no guarantee that a queue will be empty when it expires. Therefore, **the class does require an explicit destructor - one that deletes all remaining nodes.**

Queue::~Queue()

{

Node \* temp;

while (front != NULL) // while queue is not yet empty

{

temp = front; // save address of front item

front = front->next; // reset pointer to next item

**delete temp**; // delete former front

}

}

You’ve seen that classes that use *new* usually require **explicit copy constructors** and **assignment operators** that do *deep copying*. Is that the case here? The first question to answer is, “Does the default memberwise copying do the right thing?”The answer is no. Member-wise copying of a Queue object would produce a new object that points to the front and rear **of the same linked list as the original**. Adding an item to the copy Queue object changes the shared linked list. That’s bad enough. ***What’s worse*** is that only the copy’s rear pointer gets updated, essentially corrupting the list from the standpoint of the original object.

**Clearly, then, cloning or copying queues requires providing a *copy constructor* and an *assignment constructor* that do deep copying.**

Of course, that raises the question of why you would want to copy a queue. Can’t you simply ignore those concerns and use the methods you already have? Of course you can. However, at some time in the future, you might need to use a queue again, but with copying. And you might forget that you failed to provide proper code for copying. So it would seem that it’s best to provide a copy constructor and an assignment operator, even though you don’t need them now.

Fortunately, there is a sneaky way to avoid doing this extra work while still protecting against future program crashes. The idea is to define the required methods as dummy private methods:

class Queue

{

**private**:

Queue(**const Queue & q**) : qsize(0) { } // preemptive definition

**Queue &** operator=(const Queue & q) { return \*this;}//...

};

This has two effects. First, it overrides the default method definitions that otherwise would be generated automatically. Second, because these methods are private, they can’t be used by the world at large. That is, if nip and tuck are Queue objects, the compiler won’t allow the following:

Queue snick(nip); **// not allowed**

tuck = nip; **// not allowed**

Therefore, instead of being faced with mysterious runtime malfunctions in the future, you’ll get an easier-to-trace compiler error, stating that these methods aren’t accessible. **Also this trick is useful when you define a class whose objects really should not be copied.**

上面的意思是, 通过在private section里定义了copy constructor和assignment operator, 我们覆盖了compiler可能会自动产生的相关操作; 并且实际上禁止了这两个操作, 因为在private section里;

Are there any other effects to note? Yes. **Recall that a copy constructor is invoked when objects are passed (or returned) by value**. However, this is no problem if you follow the preferred practice of passing objects as references. Also a copy constructor is used to create other temporary objects. **But the Queue definition lacks operations that lead to temporary objects, such as overloading the addition operator.**

**The Customer Class**

In general, an ATM customer has many properties, such as a name, account numbers, and account balances. However, the only properties you need for the simulation are **when a customer joins the queue** and the **time required for the customer’s transaction**. When the simulation produces a new customer, the program should create a new customer object, storing in it the customer’s time of arrival and a *randomly generated value for the transaction time*. When the customer reaches the front of the queue, the program should note the time and subtract the queue joining time to get the customer’s waiting time.

class Customer

{

private:

long arrive; // arrival time for customer

int processtime; // expected processing time for customer

public:

Customer() { arrive = processtime = 0; }

void set(long when);

long when() const { return arrive; }

int ptime() const { return processtime; }

};

void Customer::set(long when)

{

processtime = std::rand() % 3 + 1; // transaction time needed

arrive = when; // std::rand() returns an integer

}

The set() member function sets the arrival time to its argument and randomly picks a value from 1 through 3 for the processing time.

**Full code of Queue Simulation**

// **queue.h** -- interface for a queue

#ifndef QUEUE\_H\_

#define QUEUE\_H\_

**class Customer**

{

private:

long arrive; // arrival time for customer

int processtime; // processing time for customer

public:

Customer() { arrive = processtime = 0; }

void set(long when);

long when() const { return arrive; }

int ptime() const { return processtime; }

};

**typedef Customer Item;**

class Queue

{

private:

struct Node { **Item** item; struct Node \* next;};

enum {Q\_SIZE = 10};

Node \* front; // pointer to front of Queue

Node \* rear; // pointer to rear of Queue

int items; // current number of items in Queue

**const** int qsize; // maximum number of items in Queue

**// preemptive definitions to prevent public copying**

Queue(const Queue & q) : qsize(0) { }

Queue & operator=(const Queue & q) { return \*this;}

public:

Queue(int qs = Q\_SIZE); // create queue with a qs limit

~Queue();

bool isempty() const;

bool isfull() const;

int queuecount() const;

bool enqueue(const Item &item); // add item to end

bool dequeue(Item &item); // remove item from front

};

#endif

// **queue.cpp** -- Queue and Customer methods

#include "queue.h"

#include <cstdlib> // (or stdlib.h) for rand()

// Queue methods

Queue::**Queue(int qs) : qsize(qs)**

{

front = rear = NULL; // or nullptr

items = 0;

}

Queue::**~Queue()**

{

Node \* temp;

while (front != NULL) // while queue is not yet empty

{

temp = front; // save address of front item

front = front->next; // reset pointer to next item

delete temp; // delete former front

}

}

bool Queue::isempty() const

{

return items == 0;

}

bool Queue::isfull() const

{

return items == qsize;

}

int Queue::queuecount() const

{

return items;

}

bool Queue::enqueue(const Item & item)

{

if (isfull())

return false;

Node \* add = new Node; // create node

// on failure, new throws std::bad\_alloc exception

add->item = item; // set node pointers

add->next = NULL; // or nullptr;

items++;

if (front == NULL) // if queue is empty,

front = add; // place item at front

else

rear->next = add; // else place at rear

rear = add; // have rear point to new node

return true;

}

// Place front item into item variable and remove from queue

bool Queue::dequeue(Item & item)

{

if (front == NULL)

return false;

item = front->item; // set item to first item in queue

items--;

Node \* temp = front; // save location of first item

front = front->next; // reset front to next item

delete temp; // delete former first item

if (items == 0)

rear = NULL;

return true;

}

**// customer method**

void Customer::set(long when)

{

processtime = std::rand() % 3 + 1;

arrive = when;

}

You now have the tools needed for the ATM simulation. The program should allow the user to enter three quantities: **the maximum queue size**, **the number of hours the program will simulate**, and the **average number of customers per hour**. The program should use a loop in which each cycle represents one minute. During each minute cycle, the program should do the following:

1. Determine whether a new customer has arrived. If so, add the customer to the queue if there is room; otherwise, turn the customer away;
2. If no one is being processed, take the first person from the queue. Determine how long the person has been waiting and set a *wait\_time* counter to the processing time that the new customer will need;
3. If a customer is being processed, decrement the *wait\_time* counter by one minute;
4. Track various quantities, such as the number of customers served, the number of customers turned away, cumulative time spent waiting in line, and cumulative queue length;

An interesting matter is how the program determines whether a new customer has arrived. Suppose that on average, 10 customers arrive per hour. That amounts to a customer every 6 minutes. The program computes and stores that value in the variable min\_per\_cust. The program uses this function to determine whether a customer shows up during a cycle:

bool newcustomer(double x)

{

return (std::rand() \* x / RAND\_MAX < 1);

}

RAND\_MAX is defined in the cstdlib file (formerly stdlib.h) and represents the largest value the rand() function can return (0 is the lowest value). Suppose that x, the average time between customers, is 6. Then the value of rand() \* x / RAND\_MAX will be somewhere between 0 and 6. In particular, it will be less than 1 one-sixth of the time, on average. However, it’s possible that this function might yield two customers spaced 1 minute apart one time and two customers 20 minutes apart another time.

意思是, 假如每小时10个guest, guest之间间隔6分钟, 那么newcustomer(6)有1/6的概率为真, 也就是说, 在1个

小时60次的测试中有10次为真, 这就实现了1个小时产生10个客人的状态.

这个函数具体的见code. 下面的是simualtion的主程序：

// **bank.cpp** -- using the Queue interface

#include <iostream>

#include <cstdlib> // for rand() and srand()

#include <ctime> // for time()

#include "queue.h"

const int **MIN\_PER\_HR = 60**;

bool newcustomer(double x); // is there a new customer?

int main()

{

using std::cin;

using std::cout;

using std::endl;

using std::ios\_base;

// setting things up

std::srand(std::time(0)); // random initializing of rand()

cout << "Case Study: Bank of Heather Automatic Teller\n";

cout << "Enter maximum size of queue: ";

int qs;

cin >> qs;

**Queue line(qs);**

cout << "Enter the number of simulation hours: ";

int hours; // hours of simulation

cin >> hours;

// simulation will run 1 cycle per minute

long cyclelimit = MIN\_PER\_HR \* hours;

cout << "Enter the average number of customers per hour: ";

double perhour; // average # of arrival per hour

cin >> perhour;

double min\_per\_cust; // average time between arrivals

**min\_per\_cust = MIN\_PER\_HR / perhour;**

**Item temp;** // new customer data

long turnaways = 0; // turned away by full queue

long customers = 0; // joined the queue

long served = 0; // served during the simulation

long sum\_line = 0; // cumulative line length

int wait\_time = 0; // time until autoteller is free

long line\_wait = 0; // cumulative time in line

**// running the simulation**

for (int cycle = 0; cycle < cyclelimit; cycle++)

**{**

if (**newcustomer(min\_per\_cust)**) // have newcomer

{

if (line.isfull())

turnaways++;

else

{

customers++;

temp.set(cycle); // cycle = time of arrival

line.enqueue(temp); // add newcomer to line

}

}

if (wait\_time <= 0 && !line.isempty())

{

line.dequeue (temp); // attend next customer

wait\_time = temp.ptime(); // for wait\_time minutes

line\_wait += cycle - temp.when();

served++;

}

if (wait\_time > 0)

wait\_time--;

sum\_line += line.queuecount();

**} // end of for loop**

// reporting results

if (customers > 0)

{

cout << "customers accepted: " << customers << endl;

cout << " customers served: " << served << endl;

cout << " turnaways: " << turnaways << endl;

cout << "average queue size: ";

cout.precision(2);

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

cout << (double) sum\_line / cyclelimit << endl;

cout << " average wait time: "

<< (double) line\_wait / served << " minutes\n";

}

else

cout << "No customers!\n";

cout << "Done!\n";

return 0;

}

bool newcustomer(double x)

{

return (std::rand() \* x / RAND\_MAX < 1);

}

程序的具体测试没做.