动态分配的对象的生存期与它们在哪里创建是无关的，只有当显式地释放时，这些对象才会被销毁。标准库定义了两个智能指针类型来管理动态分配的对象，当一个对象应该被释放时，指向它的智能指针可以确保自动地释放它。

内存包括：静态内存、栈内存、内存池（堆、自由空间）。

我们到目前为止只使用过静态内存或栈内存。分配在静态或栈内存中的对象由编译器自动创建和销毁；内存池中的生存期由程序来控制。

静态内存用来保存局部static对象、类static数据成员以及定义在任何函数之外的变量。栈内存用来保存定义在函数内的非static对象。程序堆用来存储动态分配的对象。

对于栈内存保存的对象，仅在其定义的程序块运行时才存在；static则对象在使用之前分配，在程序结束时销毁；堆中保存的对象不用时必须用代码显式地销毁他们。

# 1.动态内存与智能指针：

①new：在动态内存中为对象分配空间并返回一个指向该对象的指针，可选择对对象进行初始化。②delete：接受一个动态对象的指针，销毁该对象，并释放与之关联的内存。

①忘记释放内存：产生内存泄漏。②尚有指针引用内存的情况下释放：产生引用非法内存的指针。

新的标准库提供了两种智能指针类型来管理动态对象，是模板类。定义在memory头文件中。①shared\_ptr：允许多个指针指向同一对象。②unique\_ptr：独占所指向的对象。③weak\_ptr:伴随类，是一种弱引用，指向shared\_ptr所管理的对象。

## 1.1shared\_ptr类：

shared\_ptr<string> p;//声明时要额外提供一个指针可以指向的类型，默认初始化的智能指针中保存着一个空指针。

|  |  |
| --- | --- |
| shared\_ptr和unique\_ptr支持的操作 | |
| chared\_ptr<T> sp; | 空智能指针，可以指向类型为T的对象 |
| unique\_ptr<T> up; |
| p | 将p用作一个条件判断，若p指向一个对象，则为true |
| \*p | 解引用true，获得它指向的对象 |
| p->mem | 等价(\*p).mem |
| p.get() | 返回p中保存的指针，要小心使用，若智能指针释放了其对象，返回的指针所指向的对象也就消失了 |
| swap(p,q) | 交换了p和q中的指针 |
| p.swap(q) |

|  |  |
| --- | --- |
| shared\_ptr独有的操作 | |
| make\_shared<T>(args) | 在动态内存中分配一个对象并初始化它，返回一个shared\_ptr，指向一个动态分配的类型为T的对象，使用args初始化此对象。是最安全的分配和使用动态内存的方法。 |
| shared\_ptr<T>p(q) | p是shared\_ptr q的拷贝，此操作会递增q中的计数器，q中的指针必须能转换为T\* |
| p = q | p和q都是shared\_ptr，所保存的指针必须能相互转换。此操作会递减p的引用计数，递增q的引用计数；若p的引用计数变为0，则将其管理的原内存释放。 |
| p.unique() | 若p.use\_count()为1，返回treu，否则返回false |
| p.use\_count() | 返回与p共享对象的智能指针数量，可能很慢，主要用于调试。 |

1.1.1make\_shared函数：

shared\_ptr<int> p = make\_shared<int>();//p指向一个值初始化的int

shared\_ptr<string> p = make\_shared<string>(“efqd”);//p指向为efqd的string

auto p = make\_shared<vetor<int>>();//较简便

1.1.2shared\_ptr的拷贝和赋值：

每个shared\_ptr都有一个关联容器（成为引用计数），会记录有多少个其他shared\_ptr指向相同的对象。当用一个shared\_ptr初始化另一个shared\_ptr或将它作为参数传递给一个函数或作为函数的返回值时，它所关联的计数器就会递增。当我们给shared\_ptr赋予一个新值或是shared\_ptr被销毁时，计数器会递减。

1.1.3shared\_ptr的自动析构

一旦一个shared\_ptr的计数器变为0，就会通过析构函数自动释放自己所管理的对象完成销毁工作。析构函数用来释放对象所分配的资源。

//chared\_ptr会自动释放相关联的内存

shared\_ptr<Foo> factory(T arg)

{

return make\_shared<Foo>(arg);//适当地处理arg，释放内存

}

void use\_factory(T arg)

{

//由于p是use\_factory的局部变量，在use\_factory结束时它将被销毁，当p被销毁时，将递减其引用计数并检查它是否为0。由于p将要销毁，p指向的这个对象也会被销毁，所占用的内存会被释放。但如果有其他的shared\_ptr也指向这块内存，它就不会被释放。

chared\_ptr<Foo> p = factory(srg);

}

如果忘记了销毁程序不再需要的shared\_ptr，在无用之后不再保留就非常重要了。如果忘记了销毁程序不再需要的shared\_ptr，程序仍会正确执行，但会浪费内存。shared\_ptr在无用之后仍然保留的一种可能情况是：你将shared\_ptr保存在一个容器中，随后重排了容器，从而不再需要某些元素。在这种情况下，你应该确保用erase删除那些不再需要的shared\_ptr元素。

1.1.4使用了动态生存期的类：

程序使用动态内存处于以下三种原因之一：①程序不知道自己需要使用多少对象，②程序不知道所需对象的准确类型，③程序需要在多个对象间共享数据。

我们定义一个类，它使用动态内存是为了让多个对象能共享相同的底层数据。

#include <iostream>

#include <string>

#include<list>

#include <vector>

#include<string>

#include<memory>

#include<initializer\_list>

#include<fstream>

#include<algorithm>

#include<map>

#include<stdexcept>

using namespace std;

class strBlob

{

public:

strBlob();//构造函数

strBlob(initializer\_list<string> il);//构造函数

int size() const {return str->size();};

bool empty() const {return str->empty();};

void Push\_back(const string &t) {str->push\_back(t);};

void Pop\_back() {str->pop\_back();};

string &Front() {return \*(str->begin());};

string &Back() {return \*(str->end());};

private:

shared\_ptr<vector<string>> str;

void check(int i,const string &msg) const;//检查一个给定索引是否在合法范围内

};

void strBlob::check(int i,const string &msg) const

{

if(i >= str->size()) throw out\_of\_range(msg);

}

strBlob::strBlob():str(make\_shared<vector<string>>()){};

strBlob::strBlob(initializer\_list<string> il):str(make\_shared<vector<string>>(il)){};

int main()

{

strBlob b1;

{

strBlob b2 = {"a","an","the"};

b1 = b2;//b1和b2共享相同的元素

}//

return 0;

}

如果用vector，当拷贝一个vector时，原vector和副本vector中的元素是相互分离的，但我们想让不同拷贝之间共享相同的底层元素，当某个对象被销毁时，不能单方面地销毁底层数据。为了保证vector中的元素能继续存在，我们将vector保存在动态内存中。

## 1.2直接管理内存：

1.2.1分配和初始化对象

默认情况下，动态分配的对象是默认初始化的，这意味着内置类型或组合类型的对象的值将是未定义的，而类类型对象将用默认构造函数进行初始化。

①使用传统的构造方式（使用圆括号），②使用列表初始化（使用花括号），③值初始化。

int \*p = new int(1024);// ①

vector<int> \*p = new vector<int>{1,2,3,4,5,6,};// ②

string \*p = new string();//③ 初始化为空string

如果我们提供了一个括号包围的初始化器，就可以使用auto从此初始化器来推断我们想要分配的对象的类型。但是由于编译器要用初始化器的类型来推断要分配的类型，只有当括号中仅有单一初始化器时才可以使用auto。

auto p = new auto(obj);//p指向一个与obj类型相同的对象，用obj初始化

auto p = new auto{a,b,c};/错误，括号中只能有单个初始化器

1.2.2动态分配的const对象：

类似于其他任何const对象，一个动态分配的const对象必须进行初始化，对于一个定义了默认构造函数的类类型，其const动态对象可以隐式初始化，而其他类型的对象就必须显示初始化。

1.2.3内存耗尽：

默认情况下，如果new不能分配所要求的内存空间，它会抛出一个类型为bad\_alloc的异常。

int \*p = new (nothrow) int;

我们称这种形式的new为定位new，定位new表达式允许我们向new传递额外的参数。我们传递给它一个由标准库定义的名为nothrow对象。如果将nowthrow传递给new，我们告诉它不要抛出异常。如果这种形式的new不能分配所需内存，它会返回一个空指针。bad\_alloc和nothrow都定义在头文件new中。

1.2.4释放动态内存：

delete表达式接受一个指向动态分配的内存的指针或空指针。释放一块非new分配的内存，或者将相同的指针值释放多次，其行为是未定义的。delete表达式执行两个动作：①销毁给定的指针指向的对象，②释放对应的内存。

int I,\*pi1 = &I,\*pi2 = nullptr;

double \*pd = new double(3),\*pd2 = pd;

delete i;//错误，i不是一个指针

delete pi1;//未定义，pi1指向一个局部变量

delete pd;//正确

delete pd2;//未定义，pd2的指针已经被释放

delete pi2;//正确，可以释放空指针

返回指向动态内存的指针（而不是智能指针）的函数给其调用者增加了一个额外负担——调用者必须记得释放内存。

Foo \*factory(T arg)

{return new Foo(arg);}

由内置指针（而不是智能指针）管理的动态内存在被显式释放前一直都会存在。

动态内存的管理非常容易出错：①忘记delete内存，真正耗尽内存时，才能检测到这种错误，②使用已经释放掉的对象，通过在释放内存后将指针置空，可以检测出来这种错误，③同一块内存被释放两次，第二次自由空间可能被破坏。

当delete一个指针后，指针值就变为无效了，成为空悬指针，未初始化指针的所有缺点空悬指针也都有，我们可以在delete之后将nullptr赋予指针。在实际系统中，查找指向相同内存的所有指针是异常困难的。

## 1.3shared\_ptr和new结合使用：

我们可以用new返回的指针来初始化智能指针，仅限于自身的拷贝：

shared\_ptr<int> p2(new int(42));//p2指向一个值为42的int

接受指针参数的智能指针构造函数是explicit的，因此，我们不能将一个内置指针隐式转换为一个智能指针，一个返回shared\_ptr的函数不能在其返回语句中隐式转换一个普通指针，必须使用直接初始化形式来初始化智能指针：

shared\_ptr<int> p = new int(1024);//错误，必须使用直接初始化形式

shared\_ptr<int> p(new int(1024));//正确

shared\_ptr<int> clone(int p)

{

return new int(p); //错误

return shared\_ptr<int>(new int(p));//shared\_ptr显式绑定到一个想要返回的指针上

}

|  |  |
| --- | --- |
| 定义和改变shared\_ptr的其他方法 | |
| shared\_ptr<T> p(q) | p管理内置指针q所指向的对象；q必须指向new分配的内存，且能够转换为T\*类型 |
| shared\_ptr<T> p(u) | p从unique\_ptr u哪里接管了对象的所有权；将u置为空 |
| shared\_ptr<T> p(q,d) | p接管了内置指针q所指向的对象的所有权；q必须能转换为T\*类型，p将使用可调用对象d来代替delete |
| shared\_ptr<T> p(p2,d) | p是p2的拷贝，唯一的区别是p将用可调用对象d来代替delete |
| p.reset() | 若p是唯一指向其对象的shared\_ptr，reset会释放此对象。若传递了可选的参数内置指针q，会令p指向q，否则会将p置为空。若还传递了参数d，将会调用d而不是delete来释放q |
| p.reset(q) |
| p.reset(q,d) |

使用一个内置指针来访问一个智能指针所负责的对象是很危险的，因为我们无法知道对象何时会被销毁

void process(shared\_ptr<int> ptr)

{}

int main()

{

//函数process的局部变量ptr被销毁时，ptr指向的内存不会被释放

shared\_ptr<int> p(new(42));

process(p);

//错误，不能直接传递内置指针，要传递shared\_ptr

int \*x(new int(1024));

process(x);

//用临时shared\_ptr传递，函数process的局部变量ptr被销毁时，ptr指向的内存被释放，但x还继续指向，变成空悬指针，main函数也会二次delete

process(shared\_ptr<int>(x));

}

智能指针定义了一个名为get的函数，它返回一个内置指针，指向智能指针管理的对象。此函数是为了这样一种情况而设计的：我们需要向不能使用智能指针的代码传递一个内置指针，将指针的访问权限传递给代码。使用get返回的指针的代码不能delete此指针。不要用get初始化另一个智能指针或为智能指针赋值。

//p和q指向相同的内存，由于它们是相互独立创建的，因此各自的引用计数都为1.当q所在的程序块结束时，q被销毁，这会导致q指向的内存被释放。从而p变成了一个空悬指针，意味着当我们试图使用p时，将发生未定义的行为。而且，当p被销毁时，这块内存会被第二次delete。

shared\_ptr<int> p(new int(42));

int \*q = p.get();

{

shared\_ptr<int>(q);

}

int foo = \*p;

## 1.4智能指针和异常：

为了用shared\_ptr来管理一个类，我们必须首先定义一个函数（删除器）来代替delete，这个删除器函数必须能够完成对shared\_ptr中保存的指针进行释放的操作。

class xclass;

void disclass(xclass);

void end\_xclass(xclass \*p) {disxclass(\*p);}

void f(destination &d)

{

//当我们创建一个shared\_ptr时，可以传递一个（可选的）指向删除器函数的参数。

shared\_ptr<xclass> p(& c,end\_xclass);

}

正确使用智能指针的基本规范：①不使用相同的内置指针初始化或reset多个智能指针，②不delete get()返回的指针，③不使用get()初始化或reset另一个智能指针，④如果你使用get()返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就无效了，⑤如果你使用智能指针管理的资源不是new分配的内存，记住传递给它一个删除器。

## 1.5unique\_ptr：

某个时刻只能有一个unique\_ptr指向一个给定对象，当unique\_ptr被销毁时，它所指向的对象也被销毁。当我们

|  |  |
| --- | --- |
| unique\_ptr操作 | |
| unique\_ptr<T> u1 | 空unique\_ptr可以指向类型为T的对象，u1会使用delete来释放它的指针；u2会使用一个类型为D的可调用对象来释放它的指针。 |
| unique\_ptr<T,D> u2 |
| unique\_ptr<T,D> u(d) | 空unique\_ptr，指向类型为T的对象，用类型为D的对象d代替delete |
| u = nullptr | 释放u指向的对象，将u置为空 |
| u.release() | u放弃对指针的控制权，返回内置指针，返回的指针通常被用来初始化另一个智能指针或给一个智能指针赋值，并将u置为空。 |
| u.reset() | 释放u指向的对象，如果提供了内置指针q，令u指向这个对象；否则将u置为空 |
| u.reset(q) |
| u.reset(nullptr) |

类似shared\_ptr，初始化unique\_ptr必须采用直接初始化形式，当我们定义一个unique\_ptr时，需要将其绑定到一个new返回的指针上。由于一个unique\_ptr拥有它指向的对象，因此unique\_ptr不支持普通的拷贝或赋值操作。

unique\_ptr<int> p2(new int(42));

unique\_ptr<int> p2(p1);//错误，unique\_ptr不支持拷贝

p2 = p1;//错误，unique\_ptr不支持赋值

unique\_ptr<int> p2(p1.release());//p1被置空，p2被初始化为p1原来保存的指针

p2.reset(p1.release());//reset释放了p2原来指向的内存

管理内存的责任简单地从一个智能指针转移给另一个，我们不用另一个智能指针来保存release返回的指针，我们的程序就要负责资源的释放。

不能拷贝unique\_ptr的规则有一个例外：我们可以拷贝或赋值一个将要销毁的unique\_ptr，常见的例子是从函数返回一个unique\_ptr。

unique\_ptr<int> clone(int p)

{return unique\_ptr<int>(new int(p));}

//返回一个局部对象的拷贝：

unique\_ptr<int> clone(int p)

{unique\_ptr<int> ret(new int (p));return ret;}

对于两段代码，编译器都知道要返回的对象将要被销毁，在此情况下，编译器执行一种特殊的拷贝。

标准库的较早版本包含了一个名为auto\_ptr的类，它具有unique\_ptr的部分特性，但不是全部。特别是，我们不能在容器中保存auto\_ptr，也不能从函数中返回auto\_ptr。虽然suto\_ptr仍是标准库的一部分，但编写程序时应该使用unique\_ptr。

与shared\_ptr一样，我们可以重载一个unique\_ptr中默认的删除器，但是unique\_ptr管理删除器的方式与shared\_ptr不同。重载一个unique\_ptr中的删除器会影响到unique\_ptr类型以及如何构造（或reset）该类型的对象。与重载关联容器的比较操作类似，我们必须在尖括号中unique\_ptr指向类型之后提供删除器类型。在创建或reset一个这种unique\_ptr类型的对象时，必须提供一个指定类型的可调用对象（删除器）。

//p指向一个类型为objT的对象，并使用一个类型为delT的对象释放objT对象，它会调用一个名为fcn的delT类型对象

unique\_ptr<objT,delT> p(new objT,fcn);

## 1.6weak\_ptr：

是一种不控制所指向对象生存期的智能指针，它指向由一个shared\_ptr管理的对象，讲一个weak\_ptr绑定到一个shared\_ptr不会改变shared\_ptr的引用次数。一旦最后一个指向对象的shared\_ptr被销毁，对象就会被释放，即便有weak\_ptr指向对象，对象还是会被释放。

|  |  |
| --- | --- |
| weak\_ptr | |
| weak\_ptr<T> w | 空weak\_ptr可以指向类型为T的对象 |
| weak\_ptr<T> w(sp) | 与shared\_ptr sp指向相同对象的weak\_ptr。T必须能转换为sp指向的类型。 |
| w = p | p是一个shared\_ptr或一个weak\_ptr。赋值后w与p共享对象。 |
| w.reset() | 将w置空 |
| w.use\_count() | 与w共享对象的shared\_ptr的数量 |
| w.expired() | 若w.use\_count()为0，返回true，否则返回false |
| w.lock() | 如果expired为true，返回一个空shared\_ptr；否则返回一个指向w的对象的shared\_ptr。检查weak\_ptr指向的对象是否存在。 |

创建weak\_ptr时要用一个shared\_ptr来初始化它。由于对象可能不存在，我们不能使用weak\_ptr直接访问对象，必须先调用lock检查weak\_ptr指向的对象是否存在。

if(shared\_ptr<int> np = wp.lock())//如果np不为空则条件成立

//strBlob类的伴随指针类

class strblobptr  
{  
public:  
 strblobptr() :curr(0){}   
 strblobptr(strblob &a, size\_t sz = 0) :wptr(a.data), curr(sz){}  
 string &deref() const;  
 strblobptr& incr();//前缀递增  
private:

//若检查成功，check返回一个指向vector的shared\_ptr

shared\_ptr<vector<string>> check(size\_t, const string &) const;   
weak\_ptr<vector<string>> wptr;//指向strBlob的data成员，不会影响一个给定的strBlob所指向的vector的生存期，但是可以阻止用户访问一个不再存在的vector的企图。或者为空，或者指向一个strBlob中的vector  
size\_t curr;//保存当前对象所表示的元素的下标，在数组中的当前位置

};

# 2.动态数组：

C++语言和标准库提供了两种一次分配一个对象数组的方法：①new表达式语法分配并初始化一个对象数组，②allocator类允许我们将分配和初始化分离。使用allocator通常会提供更好的性能和更灵活的内存管理能力。

大多数应用应该使用标准库容器而不是动态分配的数组。使用容器的类可以使用默认版本的拷贝、赋值和析构操作。分配动态数组的类则必须定义自己版本的操作，在拷贝、复制及销毁对象时管理所关联的内存。

## 2.1new和数组：

分配一个数组会得到第一个元素类型的指针。

int \*pia = new int[get\_size()];//方括号中的大小必须是整型，但不必是常量

2.1.1初始化动态分配对象的数组：

①值初始化，在大小之后跟一对空括号：

int \*pia = new int[10];//10个值初始化为0的int

int \*psa = new int[10](“heheda”);//10个值初始化为”heheda”的string

②提供一个初始化器的花括号列表，如果初始化器数目大于元素数目，则new表达式失败，不会分配任何内存，抛出一个类型为bad\_array\_length的异常，类似bad\_alloc，此类型定义在头文件new中。虽然我们用空括号对数组中元素进行值初始化，但不能再括号中给出初始化器，这意味着不能用auto分配数组：

int \*pia = new int[10]{0,1,2,3,4};//剩余的进行值初始化

动态分配一个大小为0（空数组）是合法的。new返回一个合法的非空指针，此指针保证与new返回的其他任何指针都不相同，就像尾后指针一样，我们可以像使用尾后迭代器一样使用这个指针，可以用此指针进行比较操作，可以用此指针加上或减去0，但不能解引用。虽然我们不能创建一个大小为0的静态数组对象，但当n等于0时，调用new[n]是合法的：

char arr[0];//错误

char \*cp = new char[0];//正确，但cp不能解引用

2.1.2释放动态数组：

数组中的元素按逆序销毁，最后一个元素首先被销毁。空括号是必须的：它只是编译器此指针指向一个对象数组的第一个元素，如果我们在delete一个指向数组的指针时忽略了方括号（或者在delete一个指向单一对象的指针时使用了方括号），其行为是未定义的。

delete [] pa;//pa必须是指向一个动态分配的数组或为空

2.1.3智能指针和数组：

标准库提供了一个可以管理new分配的数组的unique\_ptr版本：

unique\_ptr<int[]> up(new int [10]);

up.release();//自动用delete[]销毁其指针

①当一个unique\_ptr指向一个数组时，我们不能使用点和箭头成员运算符。毕竟unique\_ptr指向的是一个数组而不是单个对象，因此这些运算符是无意义的。②当一个unique\_ptr指向一个数组时，我们可以使用下标运算符来访问数组中的元素。

|  |  |
| --- | --- |
| 指向数组的unique\_ptr特有的操作 | |
| unique\_ptr<T[]> u | U可以指向一个动态分配的数组，数组元素类型为T |
| unique\_ptr<T[]> u(p) | u指向内置指针p所指向的动态分配的数组，p必须转换为类型T\* |
| u[i] | 返回u拥有的数组中位置i处的对象 |

与unique\_ptr不同，shared\_ptr不直接支持管理动态数组，如果希望使用shared\_ptr管理一个动态数组，必须提供自己定义的删除器,如果未提供删除器，代码将是未定义的，也会影响我们下标运算符访问数组中的元素：

shared\_ptr<int> sp(new int[10],[](int \*p){delete [] p;});

sp.reset();

//shared\_ptr未定义下标运算符，而且智能指针类型不支持指针算术运算，因此，为了访问数组中的元素，必须用get获取一个内置指针，然后用它来访问数组元素

for(size\_t i = 0;i != 10;++i)

\*(sp.get() + i) = I;

## 2.2allocator类：

new有一些灵活性上的局限，它将内存分配和对象构造组合在了一起，而allocator则分开了。①分配单个对象时，通常希望用new，会导致不必要的浪费，而且没有默认构造函数的类不能动态分配数组了，②分配一大块内存时，通常计划在这块内存上按需构造对象，内存分配和对象构造分离意味着我们可以分配大块内存，但只在真正需要的时候才真正执行对象创建操作。

2.2.1allocator类：

标准库allocator类定义在头文件memory中。它提供一种类型感知的内存分配方法，它分配的内存是原始的、未构造的。当一个allocator对象分配内存时，它会根据给定的对象类型来确定恰当的内存大小和对齐位置，allocator是一个模板。

|  |  |
| --- | --- |
| 标准库allocator类及其算法 | |
| allocator<T> a | 定义了一个名为a的allocator对象，它可以为类型为T的对象分配内存 |
| a.allocate(n) | 分配一段原始的、未构造的内存，保存n个类型为T的对象 |
| a.deallocate(p,n) | 释放从T\*指针p中地址开始的内存，这块内存保存了n个类型为T的对象；p必须是一个先前由allocator返回的指针，且n必须是p创建时所要求的大小。在调用deallocate之前，用户必须对每个在这块内存中创建的对象调用destroy。 |
| a.construct(p,args) | p必须是一个类型为T\*的指针，指向一块原始内存；arg被传递给类型为T的构造函数，用来在P指向的内存中构造一个对象。args可以是0个或多个额外参数，这些额外参数必须是与构造的对象的类型相匹配的合法的初始化器。 |
| a.destroy(p) | p为T\*类型的指针，此算法对p指向的对象执行析构函数。 |

2.2.2allocator分配未构造的内存

allocator分配的内存是未构造的，我们按需要在此内存中构造对象。为了使用allocator返回的内存，我们必须用constructe构造对象。使用未构造的内存，其行为是未定义的。在早期版本的标准库中，construct只接受两个参数，指向创建对象位置的指针和一个元素类型的值。因此，我们只能将一个元素拷贝到未构造函数中，而不能用元素类型的任何其他构造函数来构造一个元素。还未构造对象的情况下就使用原始内存是错误的。

allo.construct(q++,10,’c’);//q为cccccccccc

当我们用完对象后，必须对每个构造的元素调用destroy来销毁他们，一旦元素被销毁后，就可以重新使用这部分内存来保存其它string，也可以将其归还给系统。释放内存通过调用deallocate来完成。

2.2.3拷贝和填充为初始化内存的方法：

标准库为allocator类定义了两个伴随算法，可以在为初始化内存中创建对象，它们都定义在头文件memory中。

|  |  |
| --- | --- |
| allocator算法（在给定的位置创建元素，而不是由系统内存分配） | |
| uninitialized\_copy(b,e,b2); | 从迭代器b和e指出的输入范围中拷贝元素到迭代器b2指定的未构造的原始内存中。b2指向的内存必须足够大，能容纳输入序列中元素的拷贝，且b2必须指向未构造的内存。返回递增后目的位置迭代器，指向最后一个构造的元素之后的位置。 |
| uninitialized\_copy\_n(b,n,b2); | 从迭代器b指向的元素开始，拷贝n个元素到b2开始的内存中 |
| uninitialized\_fill(b,e,t); | 在迭代器b和e指定的原始内存范围中创建对象，对象的值均为t的拷贝 |
| uninitialized\_fill\_n(b,n,t); | 从迭代器b指向的内存地址开始创建n个对象，b必须指向足够大的未构造的原始内存，能够容纳给定数量的对象。 |