@阿西拜-南昌

动态内存

C++ Primer第五版　　第12章

t

栈对象：仅在其定义的程序块运行时才存在。
static对象：在使用之前分配，在程序结束时销毁。
堆对象：动态分配的对象，在程序运行过程中可以随时建立或删除的对象。

有严格的生存期

动态内存与智能指针

动态内存的管理是通过一对运算符来完成：
new：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
delete：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

c++11

为了更安全的使用动态内存，新的标准库提供了两个智能指针：
- shared_ptr：允许多个指针指向同一个对象
- unique_ptr：“独占”所有对象

类似常规指针，区别在于它负责自动释放所指向的对象

```
//智能指针也是模板，默认初始化的智能指针中保存着一个空指针
shared_ptr<string> p1; //可以指向string
shared_ptr<list<int>> p2; //可以指向int的list

//如果p1不为空，检查它是否指向一个空string
if(p1 && p1->empty()) //如果p1指向一个空string 且p1 不为空
      *p1 = "hi"; //解引用p1，将一个新值赋予string
```

<memory> shared_ptr和unique_ptr都支持的操作

shared_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 p，获得它指向的对象
p->mem	等价于 (*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 p 和 q 中的指针
p.swap(q)	

注意是 -> 访问对象成员函数

就是指向的对象已经被销毁再使用就会很危险

shared_ptr独有的的操作

make_shared<T>(*args*)	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
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 true；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

最安全的分配和使用动态内存的方法是调用一个名为make_shared的标准库函数

```
//指向一个值为42的int的shared_ptr
shared_ptr<int> p3 = make_shared<int>(42);
//p4指向一个值为 “9999999999”的string
shared_ptr<string> p4 = make_shared<string>(10,'9');
//p5指向一个值初始化的int，即，值为0
shared_ptr<int> p5 = make_shared<int>();

//p6指向一个动态分配的空vector<string>
auto p6 = make_shared<vector<string>>();
```

引用计数

```
auto r = make_shared<int>(42);//r指向的int只有一个引用者
r = q;      //给r赋值，令它指向另一个地址
            //递增q指向的对象的引用计数
            //递减r原来指向的对象的引用计数
            //r原来指向的对象已没有引用者，会自动释放
```

```
//factory返回一个shared_ptr,指向一个动态分配的对象
shared_ptr<Foo> factory(T arg)
{
      //恰当的处理arg
      //shared_ptr负责释放内存
      return make_shared<Foo>(arg);
}

void use_factory(T arg)
{
      shared_ptr<Foo>p = factory(arg);
      //使用p
}//离开了作用，它指向的内存会被自动释放掉
```

使用动态内存的一个常见原因是允许多个对象共享相同的状态

```
vector<string> v1; //空vector

{//新作用域
      vector<string> v2 = {"a","an","the"};
      v1 =  v2;//从v2拷贝元素到v1中
}//v2被销毁，其中的元素也被销毁
//v1有三个元素，是原来v2中元素的拷贝
```

与容器不同，我们希望定义一个Blob类，对象不用拷贝直接共享相同的元素

```
Blob<string> b1; //空Blob

{//新作用域
      Blob<string> b2 = {"a","an","the"};
      b1 = b2;//b1和b2共享相同的元素
}//b2倍销毁了，但b2中的元素不能销毁
//b1指向最初由b2创建的元素
```

定义StrBlob类：由于我们还没有学习模板的实现，先定义一个管理string的类

```
class StrBlob{
public:
      typedef vector<string>::size_type size_type;
      strBlob();
      StrBlob(std::initializer_list<std::string> il);
      size_type size() const {return data->size();}
      bool empty() const {return data->empty();}
      //添加和删除元素
      void push_back(const string &t) { data->push_back(t);}
      void pop_back();
      //元素访问
      string& front();
      string& back();
private:
      shared_ptr<vector<string>> data;  date 共享指针
      //如果data[i]不合法，抛出一个异常
      void check(size_type i, const std::string &msg) const;
}
```

```
//两个构造函数都使用构造函数初始化列表
StrBlob::StrBlob(): data(make_shared<vector<string>>()){ }
StrBlob::StrBlob(initializer_list<string> il):data(make_shared<vector<string>>(il)){ }
```

使用列表初始化来初始化date

```
void StrBlob::check(size_type i, const string &msg) const{
        if(i>=data->size())
                throw out_of_range(msg);
}

string& StrBlob::front(){
        //如果vector为空，check会抛出一个异常
        check(0,"front on empty StrBlob");
        return data->front();
}

string& StrBlobk::back(){
        check(0, "back on empty StrBlob");
        return data->back();
}

void StrBlob::pop_back(){
        check(0, "pop_back on empty StrBlob");
        data->pop_back();
}
```

不改变数据成员

动态内存与智能指针：直接管理内存

默认初始化：

```
int *pi = new int; //pi指向一个动态分配的、未初始化的无名对象
string *ps = new string; //初始化为空string
```

直接初始化：

```
int *pi=new int(1024); //pi指向的对象的值为1024
string *ps=new string(10,'9'); //*ps为 “9999999999”
//vector有10个元素，值依次从0到9
vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

值初始化：

```
string *ps1 = new string; //默认初始化为空string
string *ps = new string(); //值初始化为空string
int *pi1 = new int; //默认初始化；*pil的值未定义
int *pi2 = new int(); //值初始化为0；*pi2为0
```

同类型即可

使用auto从初始化器来推断我们想要分配的对象的类型：

```
auto p1 = new auto(obj);   //p1指向一个与obj类型相同的对象
                           //该对象用obj进行初始化
auto p2 = new auto{a,b,c}; //错误：括号中只能有单个初始化器
```

用new分配const对象是合法的：

```
//分配并初始化一个const int
const int *pci = new const int(1024);
//分配并默认初始化一个const的空string
const string *pcs = new const string;
```

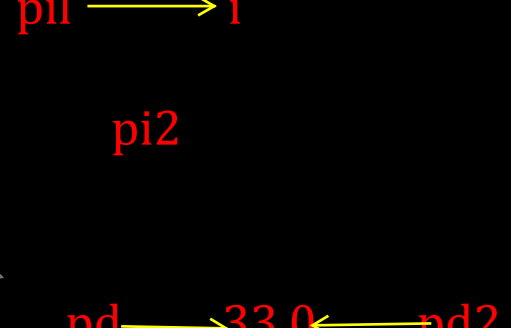
内存耗尽：

```
//如果分配失败，new返回一个空指针
int *p1 = new int; //如果分配失败，new抛出std::bad_alloc
int *p2 = new (nothrow) int; //如果分配失败，new返回一个空指针
//定位new表达式，nothrow对象告诉它不能抛出异常
```

指针值和delete：

```
int i, *pil = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete i; //错误：i不是一个指针
delete  pil;  //未定义：pil指向一个非动态内存对象
delete pd; //正确
delete  pd2;  //未定义：pd2指向的内存已经被释放了
delete  pi2;  //正确：释放一个空指针总是没有错的

const int *pci = new const int(1024);
delete pci; //正确：释放一个const对象
```



在delete之后，指针就变成了空悬指针：

```
int *p(new int(42)); //p指向动态内存
auto q = p; //p和q指向相同的内存
delete p; //p和q均变为无效
p = nullptr; //支出p不再绑定到任何对象
//q依然是空悬指针
```

shared_ptr和new结合使用：

```
shared_ptr<double> p1; //shared_ptr可以指向一个double
shared_ptr<int> p2(new int(42)); //p2指向一个值为42的int
```

智能指针构造函数是explicit的　显式调用的

```
shared_ptr<int>p1 = new int(1024); //错误：必须使用直接初始化形式不能隐式转换
shared_ptr<int>p2(new int(1024)); //正确：使用了直接初始化形式

shared_ptr<int> clone(int p){
    return new int(p); //错误：隐式转换为shared_ptr<int>
}

shared_ptr<int> clone(int p){
    //正确：显示地用int*创建shared_ptr<int>
    return shared_ptr<int>(new int(p));
}
```

默认情况下，用来初始化智能指针的普通指针，必须指向动态内存。否则必须提供操作来替代delete。

new 的对象

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 那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 shared_ptr 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)	

重置 无参数则释放

调用d 来释放q

不要混合使用普通指针和智能指针.....


```
void process(shared_ptr<int> ptr)
{
      //使用ptr       cnt = 2
}//ptr离开作用域，被销毁    脱离作用域 cnt = 1

shared_ptr<int> p(new int(42)); //引用计数为1
process(p);          //拷贝p会递增它的引用计数；在process中引用计数值为2
int i = *p;          //正确：引用计数值为1
```

风险

```
//用内置指针显式构造一个shared_ptr，这样做很可能会导致错误
int *x(new int(1024));       //危险：x是一个普通指针
process(x);                  //错误：不能将int*转换为shared_ptr<int>
process(shared_ptr<int>(x)); //合法的，但内存会被释放！
int  j =  *x;                //未定义的：x是一个空悬指针！
```

创建了 ptr 指向1024 然后脱离作用域之后 1024 的cnt为0 释放 1024

get：向不能使用智能指针的代码，传递一个内置指针

```
shared_ptr<int> p(new int(42)); //引用计数为1
int  *q =  p.get();  //正确：但使用q时要注意，不要让它管理的指针被释放
{//新程序块          get 的对象  必须是不会被删除的
//未定义：两个独立的shared_ptr指向相同的内存
shared_ptr<int>(q);
}//程序块结束，q被销毁，它指向的内存被释放
int  foo =  *p;  //未定义：p指向的内存已经被释放了
```

永远不要用get初始化另一个智能指针或为另一个智能指针赋值

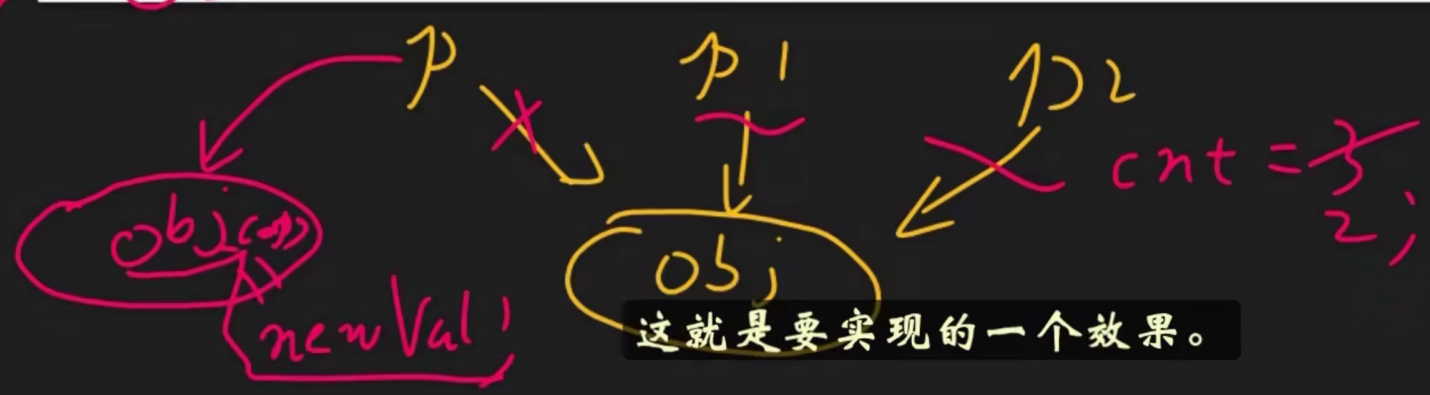
reset：更新引用计数，如果需要的话，会释放p指向的对象

shared_ptr

```
p = new int(1024); //错误：不能将一个指针赋予shared_ptr
p.reset(new int(1024)); //正确：p指向一个新对象

if(!p.unique())
      p.reset(new string(*p)); //我们不是唯一用户；分配新的拷贝
*p +=  newVal;  //现在我们知道自己是唯一的用户，可以改变对象的值
```

```
if(!p.unique())
  p.reset(new string(*p)); //我们不是唯一用户；分配新的拷贝
*p += newVal; //现在我们知道自己是唯一的用户，可以改变对象的值
```



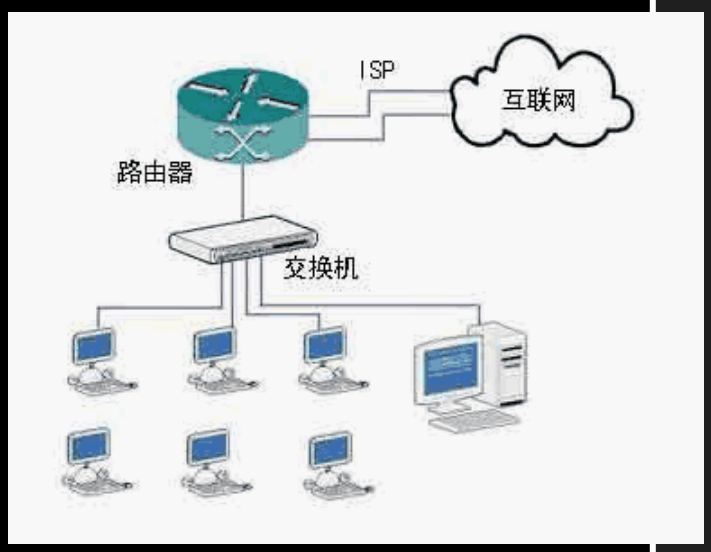
智能指针和异常


```
//如果使用智能指针，即使程序块过早结束，也能正确释放内存
void f()
{
	shared_ptr<int> sp(new int(42)); //分配一个新对象
	//这段代码抛出一个异常，且在f中未被捕获
}//在函数结束时shared_ptr自动释放内存

//直接管理内存，则不会
void f()
{
	int *ip = new int(42); //动态分配一个对象
	//这段代码抛出一个异常，且在f中未被捕获
	delete ip; //在退出之前释放内存
}
```

使用类似的技术来管理不具有良好定义的析构函数的类

```
struct destination; //表示我们正在连接什么
struct connection; //使用连接所需的信息
connection connect(destination*); //打开连接
void disconnect(connection); //关闭给定的连接
void f(destiantion &d/*其他参数*/)
{
	//获得一个连接；记住使用完后要关闭它
	connection c = connect(&d);
	//使用连接
	//如果我们在f退出前忘记调用disconnect，就无法关闭c了
}
```



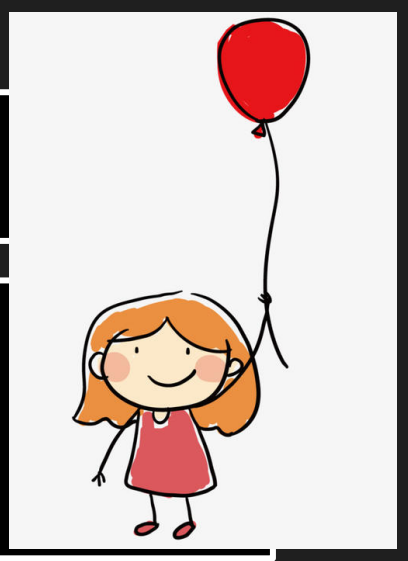
使用我们自己的释放操作

```
void end_connection(connection *p) { disconnect(*p);}

void f(destination &d /*其他参数*/)
{
	connection c = connect(&d);
	shared_ptr<connection> p(&c,end_connection);
	//使用连接
	//当f退出时（即使是由于异常而退出），connection会被正确关闭
}
```

动态内存与智能指针

unique_ptr：“拥有”所指向的对象



```
unique_ptr<double> p1; //可以指向一个double的unique_ptr
unique_ptr<int> p2(new int(42)); //p2指向一个值为42的int
```

```
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1); //错误：unique_ptr不支持拷贝
unique_ptr<string> p3;
p3 = p2; //错误：unique_ptr不支持赋值
```

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，返回指针，并将 u 置为空
u.reset()	释放 u 指向的对象
u.reset(q)	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(nullptr)	

虽然不能拷贝或赋值unique_ptr，但可以通过release或reset将指针的所有权从一个（非const）的转义给另一个：

```
unique_ptr<string> p2(p1.release()); //release将p1置为空
unique_ptr<string> p3(new string("Trex"));
//将所有权从p3转移给p2
p2.reset(p3.release()); //reset释放了p2原来指向的内存
```

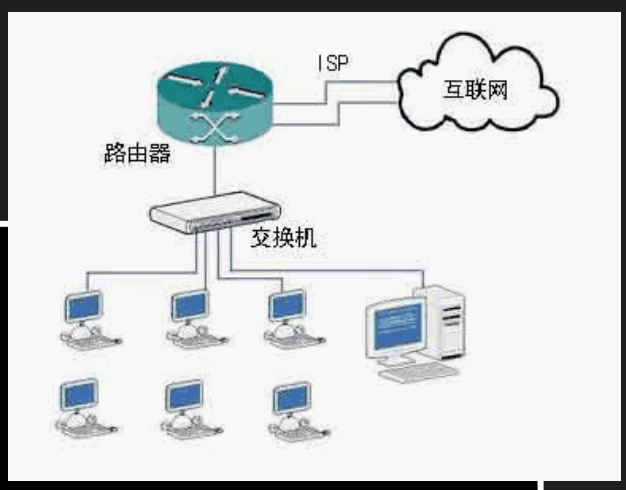
auto p = p2.release( ); // 需要delete

不能拷贝的例外：可以拷贝或赋值一个将要被销毁的unique_ptr

```
unique_ptr<int> clone(int p){
    //正确：从int*创建一个unique_ptr<int>
    return unique_ptr<int>(new int(p));
}

unique_ptr<int> clone(int p){
    unique_ptr<int> ret(new int(p));
    // ...
    return ret; //正确：返回的是即将销毁的局部对象
}
```

向unique_ptr传递删除器
我们可以重载一个unique_ptr中默认的删除器。




```
void f(destination &d /*其他需要的参数*/)
{
    connection c = connect(&d); //打开连接
    //但p被销毁时，连接将会关闭
    unique_ptr<connection, decltype(end_connection)*> p(&c,end_connection);
    //使用连接
    //当f退出时（即使是由于异常而退出），connection会被正确关闭
}
```

动态内存与智能指针
weak_ptr：绑定到shared_ptr，不会改变引用计数

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 可以是一个 shared_ptr 或一个 weak_ptr。赋值后 w 与 p 共享对象
w.reset()	将 w 置为空
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cnt = 0 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

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p); //wp弱共享p；p的引用计数未改变

//不能使用weak_ptr直接访问对象，必须调用lock
if(shared_ptr<int> np = wp.lock()){//如果np不为空则条件成立
    //在if中，np与p共享对象
}
```

检查指针类:我们将为StrBlob类定义一个伴随指针类


```
//对于访问一个不存在元素的尝试，StrBlobPtr抛出一个异常
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;
	//保存一个weak_ptr，意味着底层vector可能会被销毁
	weak_ptr<vector<string>> wptr;
	size_t curr; //在数组中的当前位置
};                    容器

shared_ptr<vector<string>> StrBlobPtr::check(size_t i,const string &msg) const{
	auto ret = wptr.lock(); //vector还存在吗？

	if(!ret)
		throw runtime_err("unbound StrBlobPtr");
	if(i>=ret->size())
		throw out_of_range(msg);
	return ret; //否则，返回指向vecotr的shared_ptr
}
string& StrBlobPtr::deref() const{
	auto p = check(curr,"dereference past end");
	return (*p)[curr]; //（*p)是对象所指向的vector
}
StrBlobPtr& StrBlobPtr::incr(){
	//如果curr一级指向容器的尾后，就不能递增了
	check(curr,"increment past end of StrBlobPtr");
	++curr; //推荐当前位置
	return *this;//返回递增后的对象
}
```

```
//对于StrBlob中的右元声明来说，此前置声明是必要的
class StrBlobPtr;
class StrBlob{
	friend class StrBlobStr;
	//...
	//返回指向首元素和尾后元素的StrBlobPtr
	StrBlobPtr begin(){ return StrBlobPtr(*this);}
	StrBlobPtr end() {auto ret = StrBlobPtr(*this, data->size()); return ret;}
};
```

动态数组

new和数组

```
//调用get_size确定分配多少个int
int *pia = new int[get_size()]; //pia指向第一个int
//方括号中的大小必须是整数，但不必是常量

typedef int arrT[42]; //arrT表示42个int的数组类型
int *p = new arrT; //分配一个42个int的数组；p指向第一个int
                   //编译器会执行：int *p = new int[42];
```

动态数组并不是数组类型

分配一个数组会得到一个元素类型的指针

初始化动态分配对象的数组

```
int *pia = new int[10]; //10个未初始化的int
int *pia2 = new int[10](); //10个值初始化为0的int
string *psa = new string[10]; //10个空string
string *psa2 = new string[10](); //10个空string

//10个int分别用列表中对应的初始化器初始化
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
//10个string，前4个用给定的初始化器初始化，剩余的进行值初始化
string *psa3 = new string[10]{"a","an","the",string(3,'x')};
```

动态分配一个空数组是合法的

```
size_t n = get_size(); //get_size返回需要的元素的数目,可以为0
int* p = new int[n]; //分配数组保存元素
for(int *q = p; q!=p+n; ++q)
/*处理数组*/

char  arr[0];//错误：不能定义长度为0的数组
char *cp = new char[0];//正确：但cp不能解引用
```

释放动态数组

```
delete  p;  //p必须指向一个动态分配的对象或为空
delete  []  pa;  //pa必须指向一个动态分配的数组或为空
//数组中的元素按逆序销毁
```

智能指针和动态数组

```
//标准库提供了一个可以管理new分配的数组的unique_ptr版本
//up指向一个包含10个未初始化int的数组
unique_ptr<int[]> up(new int[10]);
up.release(); //自动用delete[]销毁其管理的指针

for(size_t i = 0; i != 10; ++i)
      up[i] = i; //为每个元素赋予一个新值
```

unique_ptr指向一个数组时，不能用点和箭头运算符，毕竟指向的是一个数组而不是单个对象

指向数组的unique_ptr	
指向数组的 unique_ptr 不支持成员访问运算符（点和箭头运算符）。	
其他 unique_ptr 操作不变。	
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 必须能转换为类型 T*
u[i]	返回 u 拥有的数组中位置 i 处的对象 u 必须指向一个数组

shared_ptr不直接支持管理动态数组

```
//如果希望使用shared_ptr管理，必须自定义删除器
shared_ptr<int> sp(new int[10],[](int*p){delete[] p;});
sp.reset(); //使用我们提供的lambda释放数组，它使用delete[]

//shared_ptr未定义下标运算符，并且不支持指针的算术运算
for(size_t i=0; i!=10; ++i)
      *(sp.get() + i) = i; //使用get获取一个内置指针
```

allocator类：将内存分配和对象构造分离
分配大块内存，但只在真正需要时才执行指向对象创建操作

```
//将内存分配和对象构造组合在一起可能导致不必要的浪费
string *const p = new string[n]; //构造n个空string
string s;
string *q = p; //q指向第一个string
while( cin >> s && q != p+n)
      *q++ = s; //赋予*q一个新值
const size_t size = q - p; //记住我们读取了多少个string
//使用数组
delete[] p; //p指向一个数组；  记得用delete[]来释放
```

1. 我们可能不需要n个string
2. 每个对象都赋予了两遍值
3. 没有默认构造函数的类就不能动态分配数组了

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 allocate 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
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

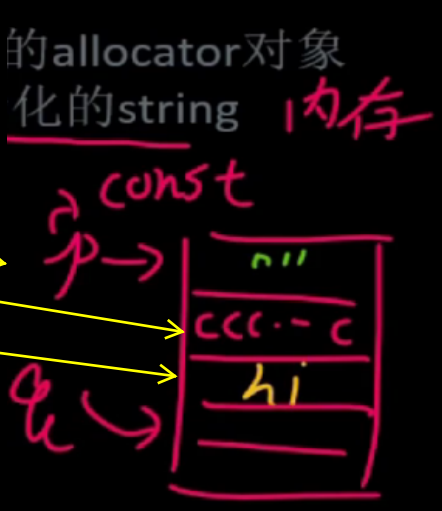
allocator类分配的内存是原始的、未构造的

```
allocator<string> alloc;                 //可以分配strng的allocator对象
auto const p = alloc.allocate(n);        //分配n个未初始化的string

auto q = p; //q指向最后构造的元素之后的位置
alloc.construct(q++);                    //*q为空字符串
alloc.construct(q++,10,'c');             //*q为ccccccccc
alloc.construct(q++,"hi");               //*q为hi!

cout<<*p<<endl; //正确：使用string的输出运算符
cout<<*q<<endl; //灾难：q指向未构造的内存

while(q != p)          注意：是先 --操作
    alloc.destroy(--q); //释放我们正在构造的string
                        //我们只能对真正构造了的元素进行destroy操作
alloc.deallocated(p,n);//释放内存
```



标准库为allocator类定义了两个伴随算法，可以在未初始化内存中创建对象

allocator算法	
这些函数在给定目的位置创建元素，而不是由系统分配内存给它们。	
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
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

假定有一个int的vector，希望将其内容拷贝到动态内存中

```
//分别比vi中元素所占用空间打一倍的动态内存
auto p = alloc.allocate(vi.size() *2);
//通过拷贝vi中的元素来构造从p开始的元素
auto q = uninitialized_copy(vi.begin(),vi.end(),p);
//前两个参数表示输入序列，第三个参数为目的地空间
//与copy不同，uninitialized_copy在给定的位置构造元素
//q指向最后一个构造的元素之后的位置

//将剩余元素初始化为42
uninitialized_fill_n(q,vi.size(),42);
```

使用标准库：文本查询程序

查询单词在文件中出现的次数，以及所在行的列表。

```
element occurs 112 times
      (line 36) A set element contains only a key;
      (line 158) operator creates a new element
      (line 160) Regardless of whether the element
      (line 168) When we fetch an element from a map, we
      (line 214) If the element is not found, find returns
```

接下来还有大约 100 行，都是单词 element 出现的位置。

标准库的运用：

使用vector<string>来保存整个输入文件的一个拷贝	输入文件中的每一行保存为vector中的一个元素。当需要打印一行时，可以用行号作为下标来提取行文本
使用istringstream	将每行分解为单词
使用set来保存每个单词在输入文本中出现的行号	这保证了每行只出现一次且行号按升序保存
使用一个map来将每个单词与它出现的行号set关联起来	这样我们可以方便地提取任意单词的set

用map防止重复

数据结构：

TextQuery	• 包含一个vecotr<string>：保存输入文本 • 包含一个map<string,set<string>>：关联单词和行号
QueryResult	保存查询结果、print函数

在类直接共享数据：

TextQuery QueryResult	两个类共享了数据，使用shared_ptr来反映数据结构中的这种共享关系

```
void runQueries(ifstream &infile)
{
      // infile is an ifstream that is the file we want to query
      TextQuery tq(infile);  // store the file and build the query map
      // iterate with the user: prompt for a word to find and print results
      while (true) {
            cout << "enter word to look for, or q to quit: ";
            string s;
            // stop if we hit end-of-file on the input or if a 'q' is entered
            if (!(cin >> s) || s == "q") break;
            // run the query and print the results
            print(cout, tq.query(s)) << endl;
      }
}

// program takes single argument specifying the file to query
int main(int argc, char **argv)
{
      // open the file from which user will query words
      ifstream infile;
      // open returns void, so we use the comma operator XREF(commaOp)
      // to check the state of infile after the open
      if (argc < 2 || !(infile.open(argv[1]), infile)) {
            cerr << "No input file!" << endl;
            return EXIT_FAILURE;
      }
      runQueries(infile);
      return 0;
}
```

TextQuery	• 包含一个vecotr<string>：保存输入文本 • 包含一个map<string,set<string>>：关联单词和行号
QueryResult	保存查询结果、print函数

```
class QueryResult; // declaration needed for return type in the query function
class TextQuery {
public:
	TextQuery(std::ifstream&);
	QueryResult query(const std::string&) const;
private:
	std::shared_ptr<std::vector<std::string>> file; // input file
	// maps each word to the set of the lines in which that word appears std::map<
	std::string,
		std::shared_ptr<std::set<line_no>>> wm;
};
```

```
// read the input file and build the map of lines to line numbers
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
	string text;
	while (getline(is, text)) {	// for each line in the file
		file->push_back(text);	// remember this line of text
		int n = file->size() - 1;	// the current line number
		istringstream line(text);	// separate the line into words
		string word;
		while (line >> word) {	// for each word in that line
			// if word isn't already in wm, subscripting adds a new entry
			auto &lines = wm[word]; // lines is a shared_ptr
			if (!lines) // that pointer is null the first time we see word
				lines.reset(new set<line_no>); // allocate a new set
			lines->insert(n);	// insert this line number
		}
	}
}
```

```
class QueryResult {
friend std::ostream& print(std::ostream&, const QueryResult&);
public:
	QueryResult(std::string s,
		    std::shared_ptr<std::set<line_no>> p,
		    std::shared_ptr<std::vector<std::string>> f):
	sought(s), lines(p), file(f) { }
private:
	std::string sought;  // word this query represents
	std::shared_ptr<std::set<line_no>> lines; // lines it's on
	std::shared_ptr<std::vector<std::string>> file;  //input file
};
std::ostream &print(std::ostream&, const QueryResult&);
```

```
QueryResult TextQuery::query(const string &sought) const{
	// we'll return a pointer to this set if we don't find sought
	static shared_ptr<set<line_no>> nodata(new set<line_no>);
	// use find and not a subscript to avoid adding words to wm!
	auto loc = wm.find( sought);

	if (loc == wm.end())
		return QueryResult(sought, nodata, file);  // not found
	else
		return QueryResult(sought, loc->second, file);
}
```

```
ostream &print(ostream & os, const QueryResult &qr)
{
	// if the word was found, print the count and all occurrences
	os << qr.sought << " occurs " << qr.lines->size() << " "
		<< make_plural(qr.lines->size(), "time", "s") << endl;

	// print each line in which the word appeared
	for (auto num : *qr.lines) // for every element in the set
		// don't confound the user with text lines starting at 0
		os << "\t(line " << num + 1 << ") "
			<< *(qr.file->begin() + num) << endl;

	return os;
}
```