12、动态内存与智能指针 #include <memory>

智能指针 shared_ptr类 => 模板

```
shared_ptr<string> pl;  // 创建一个指向string的智能指针
//默认初始化的智能指针保存着一个空指针
shared_ptr<list<string>> p2;    // the second example
if (pl && Pl->empty())  // 因此先判断指针是否为空，然后再看string是否为空
```

shared_ptr 和 unique_ptr都支持的操作	
shared_ptr sp	空的智能指针
unique_ptr up	
p.get()	返回p中保存的指针，不会使共享指针数增加
swap(p,q)	相互交换
p.swap(q)	

shared_ptr 独有的操作	**作为函数参数或者函数返回值都会使计数器增加**
make_shared<T> (args)	返回一个shared_ptr，使用args初始化T类型对象
shared_ptr<T>p(q)	p是q的拷贝，会递增q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，p减、q增，减到0，则将原内存释放
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，用于调试

```
shared_ptr<int> p1 = make_shared<int>(42);
auto p2 = make_shared<string>(10,'a');
// ps
shared_ptr<Foo> factory(T args){
  return make_shared<Foo>(args);
}
void use_factory1(T args){
  shared_ptr<Foo> p = factory(args);
  //使用了p
}   // p离开了作用域，它指向的内存会被自动释放掉
void use_factory2(T args){

  shared_ptr<Foo> p = factory(args);
```

```
  // 使用了p
  return p; // 返回了p，引用计数进行了递增
}   // p离开了作用域，它指向的内存不会被自动释放掉
// 注解：感觉用于共享内存不错
```

new & delete

```
int *p1 = new int;  // p1指向一个动态分配的、未初始化的无名对象。ps：这意味着内置类型或者组合类型的对象的值将是未定义的
int *p11 = new int(12);
string *p2 = new string;     // 这将初始化为空的string,调用了默认构造函数
string *p3 = new string(2,'a');
auto *pv = new vector<int>{0,2,3,4,5};
auto *p4 = new auto(obj);
const int *p5 = new const int(200);

delete p;   //传递给delete的指针必须指向动态分配的内存或者一个空指针
delete [] p;     // 释放数组

shared_ptr<int> p6 = new int(1024); // 错误：必须使用直接初始化形式
shared_ptr<int> p7(new int(1024));  // 正确：使用直接初始化形式

```

定义和改变shared_ptr	
shared_ptr<T> p(q)	p管理内置指针q所指对象，q必须是new分配的内存
shared_ptr<T> p(u)	p从unique_ptr u那里接管了对象所有权，将u置为空
shared_ptr<T> p(q,d)	……,p将调用对象d来代替delete
p.reset()	将p置为空
p.reset(q)	令p指向q
p.reset(q,d)	

```
p.reset( new int(1024));     // 使智能指针从新指向一个新的
```

unique_ptr: 只指向一个指针,不能拷贝

unique_ptr的一些操作	
unique_ptr<T> u1	空的unique_ptr
unique_ptr<T,D> u2;	会用一个D类型的对象代替delete
unique_ptr<T,D> u(d)	调用类型为D的对象释放它的指针
u = nullptr	释放u
u.release()	u放弃对指针的控制权，并返回指针，并将u置为空
u.reset()	释放u
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)	

不能拷贝的一个例外：

```
unique_ptr<int> clone(int i){
  unique_ptr<int> ret(new int(i));
  return ret;    // 可以返回一个局部对象的拷贝
}
```

weak_ptr: 依附于share_ptr,但不增加share_ptr计数

weak_ptr	
weak_ptr<T> w	空 weak_ptr
weak_ptr<T> w(sp)	指向shared_ptr sp相同的对象，不会增加sp计数
w = p	p可以是shared_ptr 或 weak_ptr。赋值后w和p共享对象
w.reset()	将w置为空
w.use_count()	原shared_ptr计数
w.expired()	计数为0，返回true
w.lock()	若expired为true，返回一个空shared_ptr，否则返回真的shared_ptr

动态数组

```
int *p1 = new int[10];  // 10个为初始化的int
int *p2 = new int[10]();    //初始化为0了
string *p3 = new string[10]; // 初始化为10个空string
string *p4 = new string[10](); // 同上
int *pia3 = new int[5]{1,2,3};
// 注意
unique_ptr<int[]> up(new int[10]);  // 标准库提供了这个
shared_ptr<int> sp(new int[10], [](int *p) {delete[] p});   // 使用shared_ptr管理数组，需提供自己的删除器
```

allocator : 使内存分配与对象构造分离开来

```
allocator<string> alloc;    // 可以分配string的allocator对象
auto const p = alloc.allocate(n);   // 分配n个未初始化的string
auto q = p;     // p指向最后构造元素之后的位置
alloc.construct(q++);   // 分配了一个空string，指针q往后移了一个
alloc.construct(q++,10,'a');    // 分配了一个"aaaaaaaaaa"字符串，q往后移一个
//此时q未指向任何构造内存喔
while (q != p)
    alloc.destroy(--q);     //释放构造了的string
alloc.deallocate(p,n);  // 释放内存，在此之前必须调用destroy释放构造
```

1. 疑惑：shared_ptr为什么返回的是指针
2. 疑惑：shared_ptr 返回的指针 与 指向的值放在同一个地方吗，是两个不同的变量么