# 智能指针

auto p = make\_shared<int>(0);

坚持只使用智能指针，就可以避免所有这些问题。

不要将智能指针和内置指针一起使用

不要使用内置指针来访问内置指针管理的对象

空悬指针：指向一块曾经保存数据对象但现在已经无效的内存的指针。

....

## 1.3 shared\_ptr和new结合使用

例子：

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

解释new int(1024) 开辟了一块内存，并且返回指向这块内存的指针。p接收了这个指针，并且将它作为参数传给shared\_ptr的构造函数进行初始化。

错误例子：

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

这个例子调用了shared*ptr类重载的的赋值运算符函数。正常情况下右边的参数会先被隐式转换成`shared*ptr类型，再进行赋值操作。但是shared\_ptr`的构造函数被定义成了explicit，就是不能被隐式构造，因此这个例子是错的。

例子：

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

危险的例子：

shared\_ptr<int> p(new int(1024));
  
int \*q = p.get();
  
shared\_ptr<int> r(q);

此时，p和r是同时指向同一个内存且毫无关联的智能指针，当它们销毁时会释放同一块内存！而且指针q指向的内存块会被对象p释放，严禁对q使用delete操作符。

例子：

if (!p.unique())
  
 p.reset(new string(\*p));

借用智能指针复制一个对象，同时共享指针也重置了。

哑类：没有任何数据成员也没有虚函数的类？？

删除器函数：用于完成对shared\_ptr中的保存的指针进行释放操作，没有析构函数的对象必须有。

## 1.5 unique\_ptr指针

不同之处：同一时刻只能有一个unique\_ptr指向一个给定对象。

特点：

1. 没有类似make\_shared的函数，一般用new操作符直接构造
2. 不支持拷贝、赋值，但是可以转移

使用：

// 从指针创建一个unique\_ptr指针
  
unique\_ptr<string> p1(new string("something"));
  
// 转移一个指针
  
unique\_ptr<string> p2(p1.release());
  
// 作为参数传递
  
unique\_ptr<int> clone(int p) {
  
 return unique\_ptr<int>(new int(p));
  
}
  
// 作为局部对象返回
  
unique\_ptr<int> clone(int p) {
  
 unique\_ptr<int> ret(new int(p));
  
 return ret;
  
}
  
// 传递删除器
  
unique\_ptr<objT, delT> p(new objT, fcn);
  
// 删除器类型可由编译器推断
  
unique\_ptr<objT, decltype(fcn)\*> p(new objT, fcn);

## 1.6 weak\_ptr指针

特点：弱指针，需要绑定到shared\_ptr上使用，不控制所指向对象生存周期的智能指针。

用途：伴随指针类？？？

使用：

// 借助共享指针初始化弱指针
  
auto p = make\_shared<int>(42);
  
weak\_ptr<int> wp(p);
  
// 调用lock()返回共享指针，通过共享指针访问对象
  
if (shared\_ptr<int> np = wp.lock()) {
  
 // np不为空时条件成立
  
}

## 2.1 new分配数组

1. 类型 \*指针变量名 = new 类型[数量];
2. typedef 类型 别名[数量];

* 类型 \*指针变量名 = new 别名;

以上两种分配方式等价。

**初始化**：（类型必须有默认构造函数）

int \*pia0 = new int[10]; //默认初始化
  
int \*pia1 = new int[10](); //值初始化
  
int \*pia2 = new int[10]{0,1,2,3,4,5,6,7,8,9}; //初始化器

**释放**：delete [] 指针变量名;

**智能指针**：unique\_ptr<类型[]> 变量名(new 类型[数量]);

shared\_ptr不支持动态数组管理。

## 2.2 allocator类

用途：new将内存分配和对象构造组合在了一起，delete将对象析构和内存释放组合在了一起，而allocator类能够将它们分开，在内存管理方面更灵活，有更好的性能。

特点：提供类型感知的内存分配方法，分配的内存是原始的，未构造的。

使用：

// 开辟内存
  
allocator<string> alloc;
  
auto const p = alloc.allocate(n);
  
// 初始化对象
  
auto q = p;
  
alloc.construct(q++);
  
alloc.construct(q++, 10, 'c');
  
alloc.construct(q++, "hi");
  
// 必须初始化以后才能使用
  
// 销毁对象
  
while （q != p）
  
 alloc.destroy(--q);
  
// 释放内存
  
alloc.deallocate(p, n);