1.考虑下边的简单代码：

int main()

{

int \*ptr = new int(0);

return 0;

}

就如上边程序，我们有可能一不小心就忘了释放掉已不再使用的内存，从而导致资源泄漏（resoure leak，在这里也就是内存泄漏）。

1. 考虑另一简单代码：

int main()

{

nt \*ptr = new int(0);

delete ptr;

return 0;

}

我们可能会心想，这下程序应该没问题了？可实际上程序还是有问题。上边程序虽然最后释放了申请的内存，但ptr会变成空悬指针（dangling pointer，也就是野指针）。空悬指针不同于空指针（nullptr），它会指向“垃圾”内存，给程序带去诸多隐患（如我们无法用if语句来判断野指针）。

上述程序在我们释放完内存后要将ptr置为空，即:

ptr = nullptr;

除了上边考虑到的两个问题，上边程序还存在另一问题：如果内存申请不成功，new会抛出异常，而我们却什么都没有做！所以对这程序我们还得继续改进（也可用try...catch...）：

1. 考虑最后一段代码

#include <iostream>

using namespace std;

int main()

{

int \*ptr = new (nothrow)int(0);

if(!ptr)

{

cout <<”new fails”<<endl;

return 0;

}

//假定hasException函数原型是bool hasException()

if(hasException())

throw exceotion();

delete ptr;

ptr = nullptr;

return 0;

}

当我们的程序运行到“if(hasException())”处且“hasException()”为真，那程序将会抛出一个异常，最终导致程序终止，而已申请的内存并没有释放掉。

当然，我们可以在“hasException()”为真时释放内存：

if(hasException())

{

delete ptr;

ptr = nullptr;

throw exception();

}

但，我们并不总会想到这么做。而且，这样子做也显得麻烦，不够人性化。

如果，我们使用智能指针，上边的问题我们都不用再考虑，因为它都已经帮我们考虑到了。

因此，****我们使用智能指针的原因至少有以下三点：****

1）智能指针能够帮助我们处理资源泄露问题；

2）它也能够帮我们处理空悬指针的问题；

3）它还能够帮我们处理比较隐晦的由异常造成的资源泄露。

自C++11起，C++标准提供两大类型的智能指针：

**1. Class shared\_ptr实现共享式拥有（shared ownership）概念。多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用（reference）被销毁”时候释放。为了在结构复杂的情境中执行上述工作，标准库提供了weak\_ptr、bad\_weak\_ptr和enable\_shared\_from\_this等辅助类。**

**2. Class unique\_ptr实现独占式拥有（exclusive ownership）或严格拥有（strict ownership）概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露（resourece leak）——例如“以new创建对象后因为发生异常而忘记调用delete”——特别有用。**

**注：C++98中的Class auto\_ptr在C++11中已不再建议使用。**

## **shared\_ptr**

**几乎每一个有分量的程序都需要“在相同时间的多处地点处理或使用对象”的能力。为此，我们必须在程序的多个地点指向（refer to）同一对象。虽然C++语言提供引用（reference）和指针（pointer），还是不够，因为我们往往必须确保当“指向对象”的最末一个引用被删除时该对象本身也被删除，毕竟对象被删除时析构函数可以要求某些操作，例如释放内存或归还资源等等。**

**所以我们需要“当对象再也不被使用时就被清理”的语义。Class shared\_ptr提供了这样的共享式拥有语义。也就是说，多个shared\_ptr可以共享（或说拥有）同一对象。对象的最末一个拥有者有责任销毁对象，并清理与该对象相关的所有资源。**

**shared\_ptr的目标就是，在其所指向的对象不再被使用之后（而非之前），自动释放与对象相关的资源。**

#include <iostream>

#include <string>

#include <vector>

#include <memory>

using namespace std;

int main()

{

//two shared pointers representing two persons by their name

shared\_ptr<string> pNico(new string(“nico”);

shared\_ptr<string> pJutta(new string(“jutta”),

//deleter(a lambda function)

[](string \*p)

{

cout << “delete”<<\*p<<endl;

delete p;

});

//capitalize person names

(\*pNico)[0] = ‘N’;

pJutta->replace(0,1,”J”);

//put them multiple times in a container

vector<shared\_ptr<string>> whoMadeCoffee;

whoMadeCoffee.push\_back(pJutta);

whoMadeCoffee.push\_back(pJutta);

whoMadeCoffee.push\_back(pNico);

whoMadeCoffee.push\_back(pJutta);

whoMadeCoffee.push\_back(pNico);

//print all elements

for (auto ptr:whoMadeCoffee)

cout <<\*ptr<<endl;

//overwitre a name again

\*pNico = “Nicolai”;

//print all elements

for(auto ptr:whoMadeCoffee)

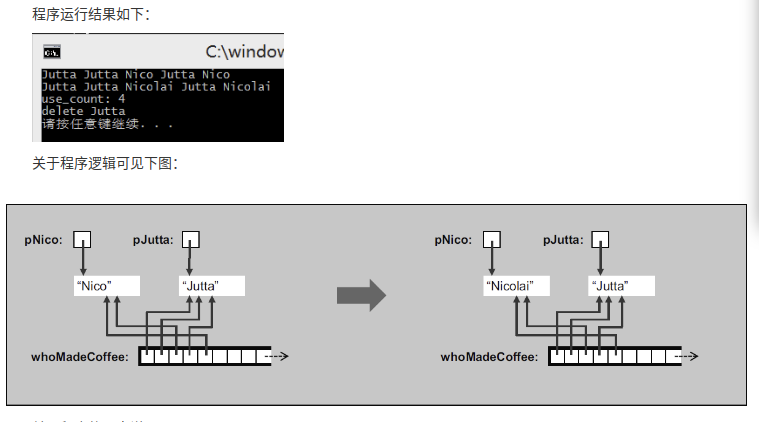
cout <<\*ptr<<endl;

//print some internal data

cout <<”use\_count”<<whoMadeCoffee[0].use\_count<<endl;

return 0;

}



**关于程序的几点说明：**

**1）对智能指针pNico的拷贝是浅拷贝，所以当我们改变对象“Nico”的值为“Nicolai”时，指向它的指针都会指向新值。**

**2）指向对象“Jutta”的有四个指针：pJutta和pJutta的三份被安插到容器内的拷贝，所以上述程序输出的use\_count为4。**

**4）shared\_ptr本身提供默认内存释放器（default deleter），调用的是delete，不过只对“由new建立起来的单一对象”起作用。当然我们也可以自己定义内存释放器，就如上述程序。不过值得注意的是，默认内存释放器并不能释放数组内存空间，而是要我们自己提供内存释放器，如：**

shared\_ptr<int> pJutta2(new int[10]),

[](int \*p)

{

delete p;

});

或者使用为unique\_ptr而提供的辅助函数作为内存释放器，其内调用delete[]：

shared\_ptr<int> p(new int[10], default\_delete<int[]>());