**一.使用普通的动态内存开辟存在的问题**

我们在使用动态内存开辟一个空间的时候，需要释放掉这个空间，不然就容易出现内存泄漏。

比如下面的程序

**情况一：**

#include<iostream>

using namespace std;

int errorTest()

{

int flag = 0;

cin >> flag;

int\* p = new int;

if (flag)

{

cout << "success" << endl;

return 0;

}

else

{

delete p;

cout << "error" << endl;

return 1;

}

}

如果我们的程序是正确的条件，那么这个时候直接是返回的，此时一种情况就是我们动态开辟的空间没有释放掉

**情况二：**

void errorTest2()

{

int\* p;

try

{

p = (int\*)malloc(1000);

}

catch (...) //如果捕获到了一个异常，程序就会跳转了，然后就没有下面的delete了

{

throw;

}

delete p;

}

如果我们捕获了一个异常，那么这个时候我们的程序就直接跳转了，下面的代码是不执行的，这个时候我们的资源也没有释放掉

有的同学可能就会说了，这里我开辟的资源少，对我的系统影响很少的，还有的同学可能也会给出另外的一种解释就是，我们开辟的资源是在堆上面开辟的，当我们的进程退出之后，我们的资源就释放了，但是如果是这样的一种情况呢，我们的服务器是7\*24小时一直运行着的，而且我们的这一段代码可能 还是循环执行着的，这样一点点占据资源，我们的内存迟早有点会被吃完，然后我们的服务器就崩溃掉了。

**二. 解决办法**

我们想到C++类中有一些函数是自动的调用的，就是构造函数和析构函数，我们想利用这个来进行动态内存的管理，就是让一个动态空间在使用结束后就直接释放掉了。

这里首先想到的就是一个定义一个类，用这个类实例化的对象来维护我们的动态空间。

**这里我们要用到一种技术是RAII技术，就是资源获得即初始化**

**管理权的转移**

**auto\_ptr**

这里首先提出的一个方法是auto\_ptr，下面我们就来模拟实现这个类

先看一个简单的代码

template<class T>

class AutoPtr

{

public:

AutoPtr(T\* ptr)

:\_ptr(ptr)

{}

~AutoPtr()

{

delete \_ptr;

}

//这里应该返回的是一个对象，这里T就是一个对象，然后我们需要改变里面的内容，所以这里返回的应该是T&

T& operator\*()

{

return \*\_ptr;

}

T\* operator->() //这里相当于是放回了一个指针，然后在使用这个指针指向一个内容

{

return \_ptr;

}

private:

T\* \_ptr;

};

下面我们还需要实现的一个内容是拷贝构造函数还有赋值运算符的重载

拷贝构造函数的时候就不得不想的一个问题就是，如果我们多个指针维护一个空间的话，当一个指针结束后，释放了，那么当其他的指针结束之后再次释放，程序就会崩溃掉了，因为之前的内容已经释放掉了

我们首先想到的方法是深拷贝，就是每次拷贝的时候都给这个指针开辟一个空间，但是这种设计是不符合我们的使用要求，我们既然要求它像指针一样使用，那么当我们的一个指针使用\*p改变空间内容的时候只有当前的内容 改变了，但是其他的内容没有改变，所以这种是不符合我们的要求的

这个时候我们想解决的办法就是在设计这个类的时候我们可以多设计一个BOOL型变量，对于动态空间的释放 权只能 有一个指针有权限管理，其他的指针没有这个权限，而且我们规定只有最后一次拷贝构造的那个指针有这个权限去释放指针。

按照这个思路我们的拷贝构造函数还有我们的赋值运算符的重载应该设计成下面的形式

#include<iostream>

using namespace std;

struct A

{

int a;

int b;

};

template<class T>

class AutoPtr

{

public:

AutoPtr(T\* ptr = NULL)

:\_ptr(ptr)

,\_owner(true)

{}

~AutoPtr()

{

if (\_owner == true)

delete \_ptr;

}

//这里应该返回的是一个对象，这里T就是一个对象，然后我们需要改变里面的内容，所以这里返回的应该是T&

T& operator\*()

{

return \*\_ptr;

}

T\* operator->() //这里相当于是放回了一个指针，然后在使用这个指针指向一个内容

{

return \_ptr;

}

AutoPtr(AutoPtr<T>& p)

:\_ptr(p.\_ptr)

, \_owner(p.\_owner)

{

p.\_owner = false;

}

AutoPtr operator=(AutoPtr<T>& p)

{

if (&p == this)

{

return \*this;

}

else

{

if (\_owner = true)

{

delete \_ptr;

}

\_ptr = p.\_ptr;

\_owner = p.\_owner;

p.\_owner = false;

return \*this;

}

}

private:

T\* \_ptr;

bool \_owner;

};

void AutoTest()

{

AutoPtr<int> p(new int);

\*p = 10;

cout << \*p << endl;

AutoPtr<int> p2(p);

\*p2 = 20;

cout << \*p2 << endl;

AutoPtr<A> pa(new A);

AutoPtr<int> p3;

p3 = p2;

\*p3 = 30;

cout << \*p3;

pa->a; //这里实际上是这个样子的pa.operator\*()->a;

}

auto\_ptr问题分析

管理权转移了，但是我们还存在着另外的一个问题就是，如果在一个if语句中 ，我们把使用了拷贝构造函数构造一个p3，这个时候管理权在p3中，因为某些原因p3释放了这个空间，但是我们在if的外面还想使用cout<<\*p2<<endl;这个时候是不是就出现了问题了呢

第二种方式当我们使用拷贝构造函数的时候，比如我们使用了p2(p1)，这个时候直接使p1 = NULL

**scoped\_ptr**

为了防止我们的程序出现那样的问题，我们的标准模板库又设计了一种简单粗暴的方式就是防拷贝，这个时候就是只能 一个指针管理一个存储空间

代码实现如下（其中只实现了拷贝构造函数部分）

#include<iostream>

using namespace std;

template<class T>

class ScopedPtr

{

public:

ScopedPtr(T\* ptr = NULL)

:\_ptr(ptr)

{}

~ScopedPtr()

{

delete \_ptr;

}

private:

ScopedPtr(ScopedPtr &p); //这里采用的是只声明不定义的方式，那么这个如果我们在使用的时候，编译的时候就会出错

//这里我们还应该注意的一点就是，我们的拷贝构造函数要放置在private中，如果放置在公有中，这个时候是很有可能被攻击

//如果被攻击了，然后别人写了一个函数，对我们的程序不力，这个时候就容易造成麻烦

private:

T\* \_ptr;

};

void ScopedPtrTest()

{

ScopedPtr<int> p(new int);

ScopedPtr<int> p1(p);

}