C++类中如果有指针成员，为了安全的管理指针成员，通常采用两种方法：

1. 使用智能指针。
2. 类采取值型行为。指针所指向的对象是唯一的，由每个类对象独立管理。

使用智能指针：

// 计数类

class U\_Ptr

{

friend class HasPtr;

// 计数类的所有成员都为私有型，只允许HasPtr类访问

private:

U\_Ptr(int\* p) : ip(p), use(1) {} // 初始化时，计数为1

~U\_Ptr() {delete ip; ip = 0;}

private:

int\* ip;

size\_t use;

};

// 使用智能指针

#include "counter.cpp"

#include <iostream>

class HasPtr

{

public:

HasPtr(int\* p, int i) : ptr(new U\_Ptr(p)), val(i) {}

// 拷贝构造函数

HasPtr(const HasPtr& orig) : ptr(orig.ptr), val(orig.val)

{

++ptr->use;

}

// 赋值运算符重载

HasPtr& operator=(const HasPtr&);

~HasPtr()

{

if (--ptr->use == 0)

{

delete ptr;

ptr = 0;

}

}

public:

int\* get\_ptr() const {return ptr->ip; }

int get\_int() const {return val; }

void set\_ptr(int\* p) {ptr->ip = p; }

void set\_int(int i) { val = i; }

int get\_ptr\_val() const {return \*ptr->ip; }

void set\_ptr\_val(int i) {\*ptr->ip = i; }

private:

U\_Ptr\* ptr; // 计数类指针

int val;

};

// 重载赋值运算符

HasPtr& HasPtr::operator=(const HasPtr& rhs)

{

++rhs.ptr->use;

if (--ptr->use == 0)

{

delete ptr;

ptr = 0;

}

ptr = rhs.ptr;

val = rhs.val;

return \*this;

}

int main(int argc, char\* argv[])

{

int\* p = new int(42);

HasPtr obj(p, 10);

HasPtr copy(obj);

copy.set\_ptr\_val(32); // 通过副本对象改变指针中保存的值，会同时改变obj中指针所保存的值

std::cout << obj.get\_ptr\_val() << std::endl; // 输出为32

//delete p; // 不能手动删除指针p，编译器会在析构对象obj时，自动删除

//p = NULL;

return 0;

}

HasPtr类的构造函数接受一个int指针和int值，利用这个int指针形参来创建一个新的U\_Ptr对象，在该U\_Ptr对象中存储给定的int指针。U\_Ptr中的计数为1，表示只有一个HasPtr对象指向它，如下图1所示。

执行拷贝构造函数完毕后，新创建的HasPtr对象和原有的HasPtr指向同一个U\_Ptr对象，该U\_Ptr对象的引用计数加1，如下图2所示。

在析构函数中，首先将U\_Ptr对象的引用计数减1，如果引用计数为0，则这是最后一个指向该U\_Ptr对象的HasPtr对象，在这种情况下，析构函数删除U\_Ptr指针，这会引起对U\_Ptr析构函数的调用，U\_Ptr析构函数会删除保存的int指针。

在重载赋值运算符时，首先将右操作数（即赋值运算符”=”右边用来赋值的HasPtr对象）的引用计数加1，然后加左操作数（即赋值运算符”=”左边被赋值的HasPtr对象）的引用计数减1（这样做的原因是：赋值运算符会把右操作数的U\_Ptr对象复制给左操作数，所以在复制之前，要先判断左操作数是否是最后一个指向左操作数中的U\_Ptr对象的对象，如果是，就需要删除其指向的U\_Ptr对象），然后将右操作数中的U\_Ptr复制给左操作数。

智能指针的使用过程如下

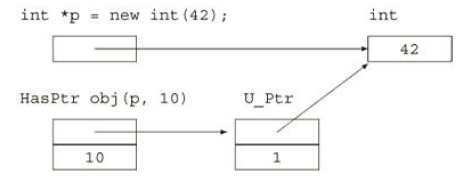


图1

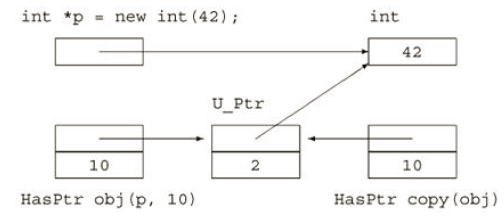


图2

复制HasPtr对象时，副本和原对象中的指针仍指向同一个基础对象（U\_Ptr对象），对基础对象的改变将影响通过任一HasPtr对象所看到的值。

定义值型类：

复制对象时，得到一个不同的新副本，对副本所做的改变不会反映在原因的对象上。

例：程序class\_test11

// 定义值型类

#include <iostream>

class HasPtr

{

public:

HasPtr(const int& p, int i): ptr(new int(p)), val(i)

{

std::cout << "调用构造函数" << std::endl;

}

HasPtr(const HasPtr& orig): ptr(new int(\*orig.ptr)), val(orig.val)

{

std::cout << "调用拷贝构造函数" << std::endl;

}

HasPtr& operator=(const HasPtr&);

~HasPtr() {delete ptr; ptr = 0;}

public:

int get\_ptr\_val() const {return \*ptr;}

int get\_int() const {return val;}

void set\_ptr(int\* p) {ptr = p; }

void set\_int(int i) {val = i; }

int\* get\_ptr() const {return ptr; }

void set\_ptr\_val(int i) {\*ptr = i; }

private:

int\* ptr;

int val;

};

HasPtr& HasPtr::operator=(const HasPtr& rhs)

{

std::cout << "赋值运算符重载" << std::endl;

\*ptr = \*rhs.ptr;

val = rhs.val;

return \*this;

}

int main(int argc, char\* argv[])

{

int i = 42;

HasPtr obj(i, 10);

HasPtr copy(obj); // 调用拷贝构造函数

HasPtr copy1 = obj; // 调用拷贝构造函数

HasPtr copy2(i, 32);

copy2 = obj; // 赋值运算符重载

return 0;

}

拷贝构造函数不再复制指针，它将分配一个新的int对象，并初始化该对象以保持与被复制对象相同的值。因为每个对象保持自己的副本，所以析构函数直接删除指针。

重载赋值运算符时，必须给其int指针所指向的对象赋值，而不是给int指针本身赋值。