***类间关联***

**1．贼(Thief)偷窃行人，每个行人都有随身带的钱，贼偷窃一个行人，他的钱就增加相应的钱包里的钱数。 请给出Thief类和Walker类。**

本例使用了四个文件：main.cpp, walker.h, thief.h, thief.cpp

***// walker.h***

#ifndef WALKER\_H

#define WALKER\_H

**class Walker**

{

public:

Walker(int m):money(m) {}

int getMoney() const {return money;}

void zeroMoney() { money =0; }

private:

int money;

};

#endif // WALKER\_H

***// thief.h***

#ifndef THIEF\_H

#define THIEF\_H

**class Walker;**

**class Thief**

{

public:

Thief(int m=0):money(m) {}

void **stole(Walker& walker)**;**//引用，参数，依赖**

int getMoney() const { return money;}

void zeroMoney() { money =0; }

private:

int money;

};

#endif // THIEF\_H

***//thief.cpp***

#include "Walker.h"

#include "Thief.h"

void Thief::stole(Walker& walker)

{

money += walker.getMoney();

walker.zeroMoney();

}

***//main.cpp***

#include <iostream>

using namespace std;

#include "Thief.h"

#include "Walker.h"

**int main()**

{

Thief t1;

Walker w1(100),w2(500),w3(800);

cout<<"被偷前:"<<endl;

cout<<"小偷的钱="<<t1.getMoney()<<endl;

cout<<"w1的钱 ="<<w1.getMoney()<<endl;

cout<<"w2的钱 ="<<w2.getMoney()<<endl;

cout<<"w3的钱 ="<<w3.getMoney()<<endl;

cout<<endl<<"w1被偷后:"<<endl;

t1.stole(w1);

cout<<"w1的钱 ="<<w1.getMoney()<<endl;

cout<<"小偷的钱="<<t1.getMoney()<<endl<<endl;

cout<<endl<<"w2被偷后:"<<endl;

t1.stole(w2);

cout<<"w2的钱 ="<<w2.getMoney()<<endl;

cout<<"小偷的钱="<<t1.getMoney()<<endl<<endl;

cout<<endl<<"w3被偷后:"<<endl;

t1.stole(w3);

cout<<"w3的钱 ="<<w3.getMoney()<<endl;

cout<<"小偷的钱="<<t1.getMoney()<<endl<<endl;

return 0;

}

**2.** **警察局有多名警察，每个警察抓获一名贼，警察局的声望就增加1点，该警察的奖金就增加100元，贼的金钱减为0。**

**请实现相关的警察局、警察、贼的类。**

**用例可为：警察局S有警察p1,p2,p3,贼有t1,t2,t3,t4, p1抓或t2,t3,p2抓获t4,p3没抓获任何贼。t1的初始金钱为500，**

**t2的初始金钱为800，t3的初始金钱为300，t4的初始金钱为1000，S的初始声望为100，警察的初始奖金为0。**

**输出最终S的声望，每个警察的奖金数。**

**先判断警察局、警察、贼这三个类间的关系。**

**警察与贼，只存在抓获被抓的关系----简单的依赖关系即可。**

**警察局与贼之间，不存在直接关系，而是通过警察才关联起来。**

**警察局和警察之间：**

**现实情况是双向关联，警察局包括多名警察，每名警察知道其所属的警察局。**

**但在本例中，警察局不需要知道有哪些警察属于本局（没有列出全部警察等类似的功能），**

**只需要警察知道其所属警察局即可。**

**即本例警察局和警察只需单向关联即可。**

***实现1：（警察局和警察之间，单向关联,thief类同上）***

***//police.h***

#ifndef POLICE\_H

#define POLICE\_H

class PoliceStation;

class Thief;

**class Police**

{

public:

Police(PoliceStation& s);

void catchThief(Thief& theThief);

int getPrize() const { return prize;}

private:

PoliceStation& station;

int prize;

};

#endif // POLICE\_H

***//police.cpp***

#include "police.h"

#include "Thief.h"

#include "PoliceStation.h"

Police::Police(PoliceStation& s)

:station(s),prize(0)

{

}

void Police::catchThief(Thief& theThief)

{

prize += 100;

theThief.zeroMoney();

station.addHonor(1);

}

***//policestation.h***

#ifndef POLICESTATION\_H

#define POLICESTATION\_H

**class PoliceStation**

{

public:

PoliceStation(int h):honor(h) {}

int getHonor() const { return honor;}

void addHonor(int h) { honor += h;}

private:

int honor; //声望

};

#endif // POLICESTATION\_H

***//main.cpp***

#include <iostream>

using namespace std;

#include "Thief.h"

#include "Walker.h"

#include "PoliceStation.h"

#include "police.h"

**int main()**

{

//用例可为：警察局S有警察p1,p2,p3,贼有t1,t2,t3,t4, p1抓或t2,t3,p2抓获t4,p3没抓获任何贼。t1的初始金钱为500，

//t2的初始金钱为800，t3的初始金钱为300，t4的初始金钱为1000，S的初始声望为100，警察的初始奖金为0。

//输出最终S的声望，每个警察的奖金数。

PoliceStation s(100); //警察局S初始声望

Police p1(s),p2(s),p3(s); //建立3个警察对象

Thief t1(500),t2(800),t3(300),t4(1000); //建立4个贼对象

//p1抓或t2,t3

p1.catchThief(t2);

p1.catchThief(t3);

//p2抓获t4

p2.catchThief(t4);

cout<<"警察局的声望为："<<s.getHonor()<<endl;

cout<<"警察p1的Prize = "<<p1.getPrize()<<endl;

cout<<"警察p2的Prize = "<<p2.getPrize()<<endl;

cout<<"警察p3的Prize = "<<p3.getPrize()<<endl;

cout<<"贼t1的Money = "<<t1.getMoney()<<endl;

cout<<"贼t2的Money = "<<t2.getMoney()<<endl;

cout<<"贼t3的Money = "<<t3.getMoney()<<endl;

cout<<"贼t4的Money = "<<t4.getMoney()<<endl;

return 0;

}

***实现2：（警察局和警察之间双向关联,thief类同上,双向关联可有多种实现，本例只是一种）***

***//policestation2.h***

#ifndef POLICESTATION2\_H

#define POLICESTATION2\_H

#include <iostream>

using namespace std; //便于演示listPolices()功能

class Police;

**class PoliceStation2**

{

public:

PoliceStation2(int h):honor(h)

{

//初始时，警察局无警察

for(int i=0;i<TOTAL;i++)

{

polices[i] = 0;

}

}

int getHonor() const { return honor;}

void addHonor(int h) { honor += h;}

bool addPolice( Police \* plc )

{

for(int i=0;i<TOTAL;i++)

{

if (polices[i] == plc)

{

return true;

}

if (polices[i] == 0)

{

polices[i] = plc;

return true;

}

}

return false; //警察添加到警察局失败

}

bool removePolice(Police \* plc)

{

for(int i=0;i<TOTAL;i++)

{

if (polices[i] == plc)

{

polices[i] = 0;

return true;

}

}

return false; //警察离开警察局失败

}

void listPolices() const

{

for(int i=0;i<TOTAL;i++)

{

if (polices[i] != 0)

{

cout<<"有一个警察"<<endl;

}

}

}

private:

static const int TOTAL = 30;***//注意：只有静态常整形才可类内初始化***

int honor; //声望

Police \* polices[TOTAL];

};

#endif // POLICESTATION2\_H

***//police2.h***

#ifndef POLICE\_H

#define POLICE\_H

class PoliceStation2;

class Thief;

**class Police**

{

public:

Police(**PoliceStation2& s**);

void catchThief(Thief& theThief);

int getPrize() const { return prize;}

private:

**PoliceStation2&** station;

int prize;

};

#endif // POLICE\_H

***//policestation2.cpp***

#include "police2.h"

#include "Thief.h"

#include "PoliceStation2.h"

Police::Police(PoliceStation2& s)

:station(s),prize(0)

{

**s.addPolice(this); //添加到警察局**

}

void Police::catchThief(Thief& theThief)

{

prize += 100;

theThief.zeroMoney();

station.addHonor(1);

}

***//main.cpp***

#include <iostream>

using namespace std;

#include "Thief.h"

#include "PoliceStation2.h"

#include "police2.h"

**int main()**

{

PoliceStation2 s(200); //警察局S初始声望

Police p1(s),p2(s),p3(s); //建立3个警察对象

Thief t1(500),t2(800),t3(300),t4(1000); //建立4个贼对象

//p1抓或t2,t3

p1.catchThief(t2);

p1.catchThief(t3);

//p2抓获t4

p2.catchThief(t4);

cout<<"警察局的声望为："<<s.getHonor()<<endl;

cout<<"警察p1的Prize = "<<p1.getPrize()<<endl;

cout<<"警察p2的Prize = "<<p2.getPrize()<<endl;

cout<<"警察p3的Prize = "<<p3.getPrize()<<endl;

cout<<"贼t1的Money = "<<t1.getMoney()<<endl;

cout<<"贼t2的Money = "<<t2.getMoney()<<endl;

cout<<"贼t3的Money = "<<t3.getMoney()<<endl;

cout<<"贼t4的Money = "<<t4.getMoney()<<endl;

***s.listPolices();***

return 0;

}

**3．在2的基础上，再加上1中的类，试一试。**

**（略）1+2就够了。**

**具体：对2而言，多了小偷偷路人（可不考虑小偷是否被抓）**

**4.用简单双向关联和关联类的形式分别实现男人(Man)和女人(Woman)间的一对一关系。**

**一个未婚男人可以和一个未婚女人结婚；**

**一个已婚男人可以其妻子离婚；**

**一个未婚女人可以和一个未婚男人结婚；**

**一个已婚女人可以其丈夫离婚；**

**一个已婚男人可以“知道”其妻子；**

**一个已婚女人可以“知道”其丈夫；**

***1)用简单双向关联实现***

**(顺便体会一下，以外联方式实现.h的好处。不信的话，可以试一试内联实现是否可行)**

***//man.h***

#ifndef MAN\_H

#define MAN\_H

**class Woman;**

**class Man**

{

***friend class Woman;***

*//这里使用友元的目的，是为了不使用Public的void setWife(Woman \* w);*

public:

Man(char \* str);

~Man();

Woman \* getWife() const {return wife;}

void marray(Woman \* w);

void divorce();

void display() const;

***//void setWife(Woman \* w);***

***//由于使用了友元类，该方法不用了。这样也就从全局角度，***

***//禁止了其它类(除了友元类woman)写访问wife成员。***

private:

Woman \* wife;

char \* name;

};

#endif // MAN\_H

***//man.cpp***

#include <iostream>

using namespace std;

#include <string.h>

#include "man.h"

#include "woman.h"

Man::Man(char \* str)

{

name = new char[strlen(str)+1];

strcpy(name,str);

wife = 0;

}

Man::~Man()

{

delete[] name;

}

void Man::marray(Woman \* w)

{

if (w!=0 && wife ==0 && **w->hasband**==0)

{

wife = w;

***wife->hasband = this;*** ***//如果不使用友元，这样的操作是不允许的***

}

}

void Man::divorce()

{

if( wife != 0) {

***wife->hasband = 0; //如果不使用友元，这样的操作是不允许的***

wife = 0;

}

}

void Man::display() const

{

cout<<"名字: "<<name<<endl;

if (wife)

{

cout<<"妻子: "<<**wife->name<**<endl<<endl;

}

else

{

cout<<"妻子: 无"<<endl<<endl;

}

}

***//woman.h***

#ifndef WOMAN\_H

#define WOMAN\_H

class Man;

**class Woman**

{

**friend class Man; //目的同man.h**

public:

Woman(char \* str);

~Woman();

Man \* getHasband() const {return hasband;}

void marray(Man \* m);

void divorce();

void display() const;

private:

Man \* hasband;

char \* name;

};

#endif // WOMAN\_H

***//woman.cpp***

#include <iostream>

using namespace std;

#include <string.h>

#include "woman.h"

#include "man.h"

Woman::Woman(char \* str)

{

name = new char[strlen(str)+1];

strcpy(name,str);

hasband = 0;

}

Woman::~Woman()

{

delete[] name;

}

void Woman::marray(Man \* m)

{

if (m!=0 && hasband ==0 && m->wife == 0)

{

hasband = m;

***hasband->wife = this; //如果不使用友元，这样的操作是不允许的***

}

}

void Woman::divorce()

{

if( hasband != 0)

{

**hasband->wife = 0;**

hasband = 0;

}

}

void Woman::display() const

{

cout<<"名字: "<<name<<endl;

if (hasband)

{

cout<<"丈夫: "<<**hasband->name**<<endl<<endl;

}

else

{

cout<<"丈夫: 无"<<endl<<endl;

}

}

***//main.cpp***

#include <iostream>

using namespace std;

#include "man.h"

#include "woman.h"

int main()

{

Man m1("管仲"),m2("乐毅"),m3("张良"),m4("韩信");

Woman f1("沉鱼"),f2("落雁"),f3("闭月"),f4("羞花");

cout<<"4个帅哥：--------------------------"<<endl;

m1.display();

m2.display();

m3.display();

m4.display();

cout<<"4个美女:---------------------------"<<endl;

f1.display();

f2.display();

f3.display();

f4.display();

m1.marray(&f1);

m2.marray(&f2);

m3.marray(&f3);

cout<<"成婚3对后：-------------------------"<<endl;

m1.display();

m2.display();

m3.display();

m4.display();

f1.display();

f2.display();

f3.display();

f4.display();

m1.divorce();

f2.divorce();

f3.divorce();

cout<<"分手2对后:--------------------------"<<endl;

m1.display();

m2.display();

m3.display();

m4.display();

f1.display();

f2.display();

f3.display();

f4.display();

return 0;

}

***2)以关联类的方式实现***

**(关联类，可以说即是一个关联，也是一个类。)**

**(本例中，只考虑二者婚姻关系中的结婚年份一个属性，**

**该属性放Man或Woman中都不合适，而适合放在关联类中。)**

***(另： 关联的实现，方法不唯一，本例只是一种实现方式)***

***//man.h***

#ifndef MAN\_H

#define MAN\_H

class Couple;

class Woman;

**class Man**

{

**friend class Couple;**

public:

Man(char \* str);

~Man();

char \* getName() const { return name;}

Woman \* getWife() const;

void display() const;

private:

char \* name;

**Couple \* couple;**

};

#endif // MAN\_H

***//man.cpp***

#include <iostream>

using namespace std;

#include <string.h>

#include "man.h"

#include "woman.h"

#include "couple.h"

Man::Man(char \* str)

{

couple = 0;

name = new char[strlen(str)+1];

strcpy(name,str);

}

Man::~Man()

{

delete[] name;

}

Woman \* Man::getWife() const

{

if (couple) return couple->getWife();

return 0;

}

void Man::display() const

{

cout<<"名字: "<<name<<endl;

if (couple)

{

cout<<"妻子: "<<couple->getWife()->getName()<<endl;

cout<<"成婚日期："<<couple->getYear()<<endl<<endl;

}

else

{

cout<<"妻子: 无"<<endl<<endl;

}

}

***//woman.h***

#ifndef WOMAN\_H

#define WOMAN\_H

class Couple;

class Man;

***class Woman***

{

**friend class Couple;**

public:

Woman(char \* str);

~Woman();

char \* getName() const { return name;}

Man \* getHasband() const;

void display() const;

private:

char \* name;

**Couple \* couple;**

};

#endif // WOMAN\_H

***//woman.cpp***

#include <iostream>

using namespace std;

#include<string.h>

#include "man.h"

#include "woman.h"

#include "couple.h"

Woman::Woman(char \* str)

{

couple = 0;

name = new char[strlen(str)+1];

strcpy(name,str);

}

Woman::~Woman()

{

delete[] name;

}

Man \* Woman::getHasband() const

{

if (couple)

return couple->getHasband();

return 0;

}

void Woman::display() const

{

cout<<"名字: "<<name<<endl;

if (couple)

{

cout<<"丈夫: "<<couple->getHasband()->getName()<<endl;

cout<<"成婚日期："<<couple->getYear()<<endl<<endl;

}

else

{

cout<<"丈夫: 无"<<endl<<endl;

}

}

***//couple.h***

#ifndef COUPLE\_H

#define COUPLE\_H

class Man;

class Woman;

***class Couples;***

***class Couple***

{

***friend class Couples;***

***private:***

Couple( Man \* h, Woman \* w,int y,int m,int d);

~Couple();

public:

Man \* getHasband() const { return hasband; }

Woman \* getWife() const { return wife; }

int getYear() const { return year; }

private:

Man \* hasband;

Woman \* wife;

int year;

int month;

int day;

};

#endif // COUPLE\_H

***//couple.cpp***

#include "couple.h"

#include "man.h"

#include "woman.h"

***//私有的构造函数***

Couple::Couple(Man \* h,Woman \* w,int y,int m,int d)

{

hasband = h;

wife = w;

***hasband->couple*** = this ;***//需要友元类支持***

***wife->couple = this; //需要友元类支持***

year = y;

month = m;

day = d;

}

Couple::~Couple()

{

hasband->couple = 0; ***//需要友元类支持***

wife->couple = 0; ***//需要友元类支持***

}

/***/couples.h***

#ifndef COUPLES\_H

#define COUPLES\_H

class Couple;

class Man;

class Woman;

**class Couples**

{

public:

static Couples& getManager()

{

static Couples mgr;

return mgr;

}

public:

Couple \* marry(Man \* h,Woman \* w,int y = -200,int m = 0,int d=0);

void divorce(Couple \* couple);

***private://私有，为实现单件***

***Couples();***

***~Couples();***

***Couples(const Couples&);***

private:

**static const int** MAXCOUNT = 100;

Couple \* couples[MAXCOUNT];

};

#endif // COUPLES\_H

***//couples.cpp***

#include "couples.h"

#include "man.h"

#include "woman.h"

#include "couple.h"

Couple \* Couples::marry(Man \* h,Woman \* w,int y,int m,int d)

{

for(int i=0;i<MAXCOUNT;i++)

{

if (couples[i] == 0 )

{

couples[i] = ***ne*w Couple**(h,w,y,m,d);***//Couple是私有构造，需Couples为友元类***

return couples[i];

}

}

return 0;

}

void Couples::divorce(Couple \* couple)

{

for(int i=0;i<MAXCOUNT;i++)

{

if (couples[i] == couple )

{

delete couple;

couples[i] = 0;

}

}

}

Couples::Couples()

{

for(int i=0;i<MAXCOUNT;i++)

{

couples[i] = 0;

}

}

Couples::~Couples()

{

for(int i=0;i<MAXCOUNT;i++)

{

delete couples[i];

}

}

***//main.cpp***

#include <iostream>

using namespace std;

#include "man.h"

#include "woman.h"

#include "couple.h"

#include "couples.h"

**int main()**

{

Man m1("管仲"),m2("乐毅"),m3("张良"),m4("韩信");

Woman f1("沉鱼"),f2("落雁"),f3("闭月"),f4("羞花");

Couples& mgr = Couples::getManager();

cout<<"4个帅哥：--------------------------"<<endl;

m1.display();

m2.display();

m3.display();

m4.display();

cout<<"4个美女:---------------------------"<<endl;

f1.display();

f2.display();

f3.display();

f4.display();

Couple \* c1 = mgr.marry(&m1,&f1,-199);

Couple \* c2 = mgr.marry(&m2,&f2,-198);

Couple \* c3 = mgr.marry(&m3,&f3,-197);

cout<<"成婚3对后：-------------------------"<<endl;

m1.display();

m2.display();

m3.display();

m4.display();

f1.display();

f2.display();

f3.display();

f4.display();

cout<<"分手2对后:--------------------------"<<endl;

mgr.divorce(c1);

mgr.divorce(c2);

m1.display();

m2.display();

m3.display();

m4.display();

f1.display();

f2.display();

f3.display();

f4.display();

return 0;

}

**5.现有类A和类B**

***class A***

{

public:

A(int num):mData(num){ }

~A( ) { }

int GetData( ) const { return mData; }

void SetData(int data) { mData = data; }

private:

int mData;

};

***class B***

{

public:

B(int num=0):pa(new A(num)) { }

~B( ) {delete pa;}

B(const B& rhs) //深拷贝

{

pa=new A(\*rhs.pa);

}

B& operator=(const B& rhs) //深赋值

{

if ( this!=&rhs )

{

delete pa;

pa=new A(\*rhs.pa);

}

return \*this;

}

A\* **operator->**( ) const {return pa;}

***void*** GetData() const { return pa->GetData();}***//错误，返回值应为int***

void SetData(int data) { pa->SetData(data); }

private:

**A\* pa;**

};

a）现需要以***引用计数***的方法，重新实现类B，要求类A不得做任何修改。

b）请在***a）的基础上，以Copy On Write的方式***修改类B的实现，使得B类对象可以访问B类的成员GetData()和成员SetData(int);(即可以修改Ｂ类对象中pa指针指向的A类对象的数据成员。也就是说，使用B类对象时，对于以**只读方式访问A类**的成员，使用引用计数；对于**以写方式访问A类**的成员，要先进行深复制，然后再写数据)。

/***/本例实现中，为了理解方便，将类A改名为Share，将类B改名为User***

***//user.h***

#ifndef USER\_H

#define USER\_H

**class Share**

{

public:

Share(int num):mData(num) {}

int getData() const {return mData;}

void setData(int data) {mData = data;}

private:

int mData;

};

**class User**

{

public:

User(**int num = 0**);

~User();

User(const User& rhs);

User& **operator=(**const User& rhs);

**void display(const char \* objName) const;**

int getData() const;

**void setData(int data);**

private:

**void addRef();**

**void releaseRef();**

private:

**Share \* objShared;**

**int \* refCount;**

};

#endif // USER\_H

***//user.cpp***

#include <iostream>

using namespace std;

#include "user.h"

User::User(int num) :**objShared(new Share(num)**)

{

**refCount = new int(1);**

}

User::~User()

{

**releaseRef();**

}

void User::addRef()

{

**++(\*refCount);**

}

void User::releaseRef()

{

if (**--(\*refCount)** **== 0**) //先减，看有多少个对象，无对象时则释放

{

**delete** **~~refCount~~**;

**delete** **~~objShared~~**;

}

}

//

User::User( const User& rhs )

{

**objShared = rhs.objShared;**

**refCount = rhs.refCount;**

**addRef();//拷贝构造一个新的对象，引用值加1**

}

User& User::**operator=(** const User& rhs)

{

if (this != &rhs)

{

***releaseRef();*//原对象已经存在，被赋另一个对象的值，原对象引用计数值要减1**

objShared = rhs.objShared;

refCount = rhs.refCount;

**addRef();//被赋新值后，等号右端对象引用计数值加1**

}

return \*this;

}

int User::**getData()** const

{

return objShared->getData();

}

**//注意：此时如果SetData不改，即void SetData(int data) { pa->SetData(data); }，则只要有**

**//一个对象改了值，所有对象的值跟着改，因为所有对象的指针pa指向同一个Share对象**

**//按Copy On Write执行，即：当写数据时，先作拷贝，只改变当前对象Share值，其它对**

**//象的Share值不改变**

**void User::setData(int data)**

{

if(\*refCount >= 2 )

{

**releaseRef();**

objShared = **new** Share(**data**);

refCount = **new** int(1);

}

else **//只有一个对象时直接改变值即可**

{

objShared->setData(data);

}

}

void User::display(const char \* objName) const

{

cout<<"对象"<<objName<<"的属性为:"<<endl;

cout<<"RefCount="<<\*refCount<<endl;

cout<<"**共享对象的mData值** = "<< **objShared->getData()**<<endl<<endl;

}

***//main.cpp***

#include <iostream>

using namespace std;

#include "user.h"

int main()

{

User a(100);

a.display("a");

cout<<"执行拷贝构造: User b(a); 后-----------------"<<endl;

User b(a);

a.display("a");

b.display("b");

cout<<"又执行拷贝构造: User c(b); 后-----------------"<<endl;

User c(b);

a.display("a");

b.display("b");

c.display("c");

cout<<"新建对象d和e： User d(500),e(999); 后-----------------"<<endl;

User d(500),e(999);

d.display("d");

e.display("e");

cout<<"执行赋值语句： d = b; 后-----------------"<<endl;

d = b;

a.display("a");

b.display("b");

c.display("c");

d.display("d");

e.display("e");

cout<<"按Copy On Write执行修改语句： c.setData(888);, 后-------"<<endl;

c.setData(888);

a.display("a");

b.display("b");

c.display("c");

d.display("d");

e.display("e");

cout<<"按Copy On Write执行修改语句： e.setData(250);, 后-------"<<endl;

e.setData(250);

a.display("a");

b.display("b");

c.display("c");

d.display("d");

e.display("e");

cout<<"-------------Over--------------"<<endl;

return 0;

}

6.若程序中需要频繁地用new创建、用delete销毁A类对象，请在A类中通过

**重载operator new和operator delete**，实时**统计创建A类对象的个数、销毁的个数**、

累计分配的字节数、还在使用的字节数。

若还可能**使用new[],delete[],如何也能完成上边的统计工作**。

***参考：定义类A***

#include <iostream>

using namespace std;

**class A**

{

public:

**static void \*** **operator new**(size\_t sz)

{

++newCount;

newBytes += sz;

return **::operator new**(sz);

}

**static** **void** **operator delete**(**void \* p**, size\_t sz)

{

++deleteCount;

deleteBytes += sz;

**::operator delete**(p);

}

**static** **void \*** **operator new[]**(size\_t sz)

{

++newCount;

newBytes += sz;

return **::operator new**(sz);

}

**static void** **operator delete[]**(**void \*** p, size\_t sz)

{

++deleteCount;

deleteBytes += sz;

**::operator delete**(p);

}

public:

**static int newCount;**

**static int newBytes**;

**static** **int deleteCount**;

**static int deleteBytes;**

private:

int n1;

int n2;

int n3;

};

int A::newCount = 0;

int A::newBytes = 0;

int A::deleteCount = 0;

int A::deleteBytes = 0;

**int main()**

{

A a1,a2;

A \* pa3= new A;

A \* pa4= new A;

A \* ps = new A[5];

delete[] ps;

delete pa3;

delete pa4;

cout<<A::newCount<<endl;

cout<<A::newBytes<<endl;

cout<<A::deleteCount<<endl;

cout<<A::deleteBytes<<endl;

return 0;

}

**注：上述代码，在VC6.0下会有内部差异。**

**vc6.0中内部没有实现::operator new[]和::operator delete[],**

**完全用::operator new和::operator delete替代了。**

**注:上述代码在vs2017和CodeBlocks下运行结果不同！**

7.通过分析**二元运算符的交换律**，以及**左操作数的限制**，理解重载自定义类型的操作符时：

为什么重载+，一般用自由函数形式？

为什么重载+=，一般用成员函数形式？

为什么重载=，必须用成员函数形式？

为什么重载<<，必须用自由函数形式？

1)**+应满足交换律，所以a+100和100+a都应该合法**，

若以成员函数形式重载operator+，那么

a+100将编译成 a.operator+(100),编译器将尝试将100转换成A类对象。

而100+a不可能编译成 100.operator+(a)，只会尝试将a转换成可与100相加

的类型如int或float。所以operator+以自由函数形式更合适。

2)**+=不应满足交换律**，而且左操作数必为类型A，所以成员函数形式更合适。

3)=**原因同上**，而且，应禁止类似100=a；这样的写法，所以只能是成员函数形式。

4)<<,也是二元运算符，但**两个操作数的类型不一样**，左操作数是流类型，右操作数才是类型A，所以只能使用自由函数的形式。

**8.实现分页器类(Paginate),不用实现输入指定页的部分**。

如：

对于如下主函数：

int main()

{

Paginate pager(13);

for(int i=1;i<=13;++i)

{

//i当前页，13总页数

pager.**setPage(i)**.show();

}

cout<<"start move...."<<endl;

pager.setPage(5).show();

pager.next().show();

pager.prev().show();

//直接翻5页

pager.nextN().show();

pager.next().show();

pager.prevN().show();

return 0;

}

其输出为：(+表示是当前页)

上页 1+ 2 3 4 5 … 13 下页

上页 1 2+ 3 4 5 … 13 下页

上页 1 2 3+ 4 5 … 13 下页

上页 1 2 3 4+ 5 … 13 下页

上页 1 2 3 4 5+ … 13 下页

上页 1 … 6+ 7 8 9 10 … 13 下页

上页 1 … 6 7+ 8 9 10 … 13 下页

上页 1 … 6 7 8+ 9 10 … 13 下页

上页 1 … 6 7 8 9+ 10 … 13 下页

上页 1 … 6 7 8 9 10+ … 13 下页

上页 1 … 9 10 11+ 12 13 下页

上页 1 … 9 10 11 12+ 13 下页

上页 1 … 9 10 11 12 13+ 下页

start move....

上页 1 2 3 4 5+ … 13 下页

上页 1 … 6+ 7 8 9 10 … 13 下页

上页 1 2 3 4 5+ … 13 下页

上页 1 … 6 7 8 9 10+ … 13 下页

上页 1 … 9 10 11+ 12 13 下页

上页 1 … 6+ 7 8 9 10 … 13 下页

**// paginate.h**

#ifndef PAGINATE\_H

#define PAGINATE\_H

**//#define MY\_DEBUGGING**

**class Paginate**

{

public:

Paginate(int theTotal);

**Paginate& setPage(int theNum );**

Paginate**&** prev();

Paginate**&** next();

Paginate**&** prevN();

Paginate**&** nextN();

Paginate**&** first();

Paginate**&** last();

void showRange() const;

void show() const;

protected:

private:

**static const int** N = **5**;

int num;

int total;

**int rangeStart;**

**int rangeEnd;**

};

#endif // PAGINATE\_H

**// paginate.cpp**

**#include <algorithm>**

#include <iostream>

**#include <iomanip>**

using namespace std;

#include "paginate.h"

Paginate::Paginate(int theTotal)

:total(theTotal)

{

setPage(1);

}

Paginate & Paginate::**setPage(int theNum)**

{

num = max(1, theNum); //num至少为1

if(total <= N+4)

{

rangeStart = 1;

rangeEnd = total;

}

else

{

rangeStart = ((num - 1) / N) \* N + 1;

rangeEnd = rangeStart + N-1;

if(rangeEnd > total)

{

rangeEnd = total;

rangeStart = rangeEnd - N+1;

}

}

return **\*this**;

}

Paginate & Paginate::prev()

{

return setPage(num-1);

}

Paginate & Paginate::next()

{

return setPage(num+1);

}

Paginate & Paginate::prevN()

{

return setPage(num-N);

}

Paginate & Paginate::nextN()

{

return setPage(num+N);

}

Paginate & Paginate::first()

{

return setPage(1);

}

Paginate & Paginate::last()

{

return setPage(total);

}

void Paginate::**showRange()** const

{

cout<<"RangeStart="<<rangeStart<<"\tRangeEnd="<<rangeEnd<<endl;

}

void Paginate::**show()** const

{

cout<<"上页";

**if(rangeStart>1) cout<<setw(5)<<1<<setw(5)<<"…";**

for(int i=rangeStart;i<=rangeEnd;++i)

{

if(i==num) cout<<setw(4)<<i<<"+";

else cout<<setw(5)<<i;

}

**if(rangeEnd<total)cout<<setw(5)<<"…"<<setw(5)<<total;**

cout<<setw(5)<<"下页";

cout<<endl;

}