**第十五章之（一）友元**

**让一个类成为另一个类的友元：**

当面临两个类对象，这两个类首先是互相独立且没有共同点的。

因此，不能使用继承（无论是公有还是私有）。

但这两个类之间有联系。例如第一个类的对象，可以操纵并影响第二个类的对象的某些功能（书上的例子是电视机和电视机的遥控器）。

就以书上提供的电视机和遥控器为例，先自行定义类定义：

电视机的（被操控一方）：

#include<iostream>

//类定义

class TV

{

bool ison; //开关

int channelmax; //最大频道数

int channel; //当前频道

int volume; //当前音量（最大100，最小0）

bool model; //模式，有线1或者无线0

bool AVTV; //AV/TV模式，AV 1，TV 0

public:

TV(); //构造函数

void OnOff(); //开关机按钮

void AddChannel(); //当前频道加1

void MinusChannel(); //频道减1

void AddVolume(); //音量加1

void MinusVolume(); //音量减1

void ChangeModel(); //更改有线无线模式

void ChangeAVTV(); //更改AVTV模式

};

//类方法

TV::TV()

{

ison = false; //开关

channelmax = 10; //刚开始只有10个频道

channel = 1; //初始频道为1

volume = 1; //音量为1

model = 1; //有线

AVTV = 0; //TV模式

}

void TV::OnOff() //自定义的时候是分开的，后来想到应该是一个按钮，故合并

{

if (ison == true)ison = false;

else ison = true;

}

void TV::AddChannel()

{

if (ison == false)return; //如果关机则没反应

channel++;

if (channel > channelmax)channel = 1; //如果比最大频道大，则跳回初始频道

}

void TV::MinusChannel()

{

if (ison == false)return; //如果关机则没反应

channel--;

if (channel > channelmax)channel = channelmax; //如果比最大频道大，则跳回最后频道

}

void TV::AddVolume()

{

if (ison == false)return; //如果关机则没反应

if (volume < 100)volume++; //达到100则没不能再加了

//理论上这里应该还有一个显示音量的画面

}

void TV::MinusVolume()

{

if (ison == false)return; //如果关机则没反应

if (volume > 0)volume--;

//理论上这里应该还有一个显示音量的画面

}

void TV::ChangeModel()

{

if (ison == false)return; //如果关机则没反应

if (model == 1)model = 0;

else model = 1;

}

void TV::ChangeAVTV()

{

if (ison == false)return; //如果关机则没反应

if (AVTV == 1)AVTV = 0;

else AVTV = 1;

}

然后，因为遥控器的类要求能操控电视机，假设遥控器的类名为Control，因此，需要在TV的public部分加入代码：

public:

friend class Control; //使Control成为友元类

于是TV类的完成了，下来制作遥控器类：

class Control //遥控器类

{

public:

void OnOff(TV&t) { return t.OnOff(); } //开关机按钮

void AddChannel(TV&t) { return t.AddChannel(); } //当前频道加1

void MinusChannel(TV&t) { return t.MinusVolume(); } //频道减1

void AddVolume(TV&t) { return t.AddVolume(); } //音量加1

void MinusVolume(TV&t) { return t.MinusVolume(); } //音量减1

void ChangeModel(TV&t) { return t.ChangeModel(); } //更改有线无线模式

void ChangeAVTV(TV&t) { return t.ChangeAVTV(); } //更改AVTV模式

void SetChannel(int c, TV&t); //选择某个频道，友元类在这个函数里起作用了（可以直接调用私有成员）

};

void Control::SetChannel(int c, TV&t) //比最大大则自动为最大，比1小则自动为1

{

if (c < 1)c = 1;

else if (c>t.channelmax)c = t.channelmax;

t.channel = c;

}

若在类中声明另一个类为其友元，则另一个类可以使用当前类的私有成员（如在TV中声明Control类为TV类的友元，于是在Control类中的SetChannel()方法中，就可以使用TV类的私有成员channel）。

为了方便展示，故在TV类中再添加一类方法show()，用于输出当前状态（开关，频道，音量，最大频道等：

void TV::show()

{

using std::cout;

using std::endl;

cout << "开关：";

if (ison == true)cout << "开";

else cout << "关";

cout << "，最大频道数：" << channelmax << "，当前频道：" << channel << "，当前音量：" << volume << "，有线无线模式：";

if (model == true)cout << "有线";

else cout << "无线";

cout << "，AV/TV：";

if (AVTV == true)cout << "AV";

else cout << "TV";

cout << endl;

}

附测试程序

int main()

{

using namespace std;

TV m, n; //两个TV

Control one; //遥控器

cout << "输出2个电视（m和n）目前情况" << endl;

m.show();

n.show();

cout << "首先开启m和n";

m.OnOff();

n.OnOff();

cout << "设置m的当前频道：";

int p;

cin >> p;

one.SetChannel(p, m);

cout << "设置n的当前频道：";

cin >> p;

one.SetChannel(p, n);

cout << "m增加5次音量" << endl;

for (int i = 0; i < 5; i++)

m.AddVolume();

cout << "m增加5次音量，减少1次音量" << endl;

for (int i = 0; i < 5; i++)

n.AddVolume();

n.MinusVolume();

cout << "m增加3次频道" << endl;

for (int i = 0; i < 3; i++)

m.AddChannel();

cout << "n减少3次频道" << endl;

for (int i = 0; i < 3; i++)

n.MinusChannel();

cout << "m切换有线无线模式" << endl;

m.ChangeModel();

cout << "n切换AVTV模式" << endl;

n.ChangeAVTV();

cout << "重新查看m和n的情况" << endl;

m.show();

n.show();

system("pause");

return 0;

}

显示：

输出2个电视（m和n）目前情况

开关：关，最大频道数：10，当前频道：1，当前音量：1，有线无线模式：有线，AV/TV：TV

开关：关，最大频道数：10，当前频道：1，当前音量：1，有线无线模式：有线，AV/TV：TV

首先开启m和n设置m的当前频道：10

设置n的当前频道：5

m增加5次音量

m增加5次音量，减少1次音量

m增加3次频道

n减少3次频道

m切换有线无线模式

n切换AVTV模式

重新查看m和n的情况

开关：开，最大频道数：10，当前频道：3，当前音量：6，有线无线模式：无线，AV/TV：TV

开关：开，最大频道数：10，当前频道：2，当前音量：5，有线无线模式：有线，AV/TV：AV

请按任意键继续. . .

**让一个类的某个类方法，成为另一个类的友元函数：**

让一个类的某个类方法，成为另一个类的友元，和让一个类成为另一个类的友元很相似。

在另一个类中，使用**friend 另一个类的函数头**。

例如：

friend void Control::SetChannel(int c, TV&t); //使Control成为友元类

但这样带来一个问题，假如Control类在TV类之前，那么在其类方法中不能使用内联函数（例如void OnOff(TV&t) { return t.OnOff(); } //开关机按钮这样），而且也有一个问题是，Control类中使用了TV类作为参数

如果TV类在Control类之前，那么TV类不知道Control类是什么，自然也没办法把他的某个类方法作为内联函数了。

①为了解决后一个问题，那么应该将Control类放在TV类前面；

②为了让TV类成员（引用）能够作为Control的参数，因此，应该使用**前向声明（forward declaraton）**，即在Control类之前加入语句：class TV；

③但又存在Control类中不能使用内联函数的问题，因此，应该在Control类定义中，不要使用内联函数（但可以在后面类方法定义时，使用inline关键字使其内联），把其函数定义，放于TV类定义之后。

于是，整个结构应该是这样的：

**class TV; （类的前向声明）**

**class Control{类定义};**

**class TV{ 类定义};**

**TV类和Control类的类方法定义**

这样就可以顺利运行了。

另外，如果相反的话（Control，TV，Control）是不行的。

原因在于，在TV类里面声明Control的某个方法为其友元方法。

因此，在编译的时候，编译器要知道这个方法是什么（即之前声明过），而若只声明某个类，那是不行的（因为编译器只知道有这么一个类，但不知道这个类里面有什么类方法）。

另外，又因为类方法的声明只能在类定义中（类方法的定义可以放其他地方），因此也不能单独把其拿出来，放在类TV的定义之前。

于是，若**友元的是类方法**，其顺序只能是：

**①被友元类的前置声明——》**

**②友元（包含友元函数的方法所在的）类的声明（不能有内联函数）——》**

**③被友元类的声明——》**

**④两个类的类定义**

而若友元的是类，顺序则无要求。

代码如下：

//类定义

class TV; //前置，以使用TV类作为参数

class Control //遥控器类

{

public:

void OnOff(TV&t); //开关机按钮

void AddChannel(TV&t); //当前频道加1

void MinusChannel(TV&t); //频道减1

void AddVolume(TV&t); //音量加1

void MinusVolume(TV&t); //音量减1

void ChangeModel(TV&t); //更改有线无线模式

void ChangeAVTV(TV&t); //更改AVTV模式

void SetChannel(int c, TV&t); //选择某个频道，友元类在这个函数里起作用了（可以直接调用私有成员）

};

class TV

{

bool ison; //开关

int channelmax; //最大频道数

int channel; //当前频道

int volume; //当前音量（最大100，最小0）

bool model; //模式，有线1或者无线0

bool AVTV; //AV/TV模式，AV 1，TV 0

public:

friend void Control::SetChannel(int c, TV&t); //使Control的类方法成为友元

TV(); //构造函数

void OnOff(); //开关机按钮

void AddChannel(); //当前频道加1

void MinusChannel(); //频道减1

void AddVolume(); //音量加1

void MinusVolume(); //音量减1

void ChangeModel(); //更改有线无线模式

void ChangeAVTV(); //更改AVTV模式

void show(); //输出当前状态

};

//类方法

略。。。

**两个类互为友元的情况：**

假如Control类中，需要使用TV类的数据成员，我们是已知方法的。

那么在这个的基础上，TV类也要使用Control类的数据成员，则需要做更多。

首先，在两个类的类定义中，添加 **friend class 友元类名** 这行代码，并且，添加的位置，要位于该类使用友元类对象作为参数的类方法之前。

其次，参数应该是友元类（不过是否能用友元类的某个类方法(使用函数指针)作为参数，不确定。我倾向于是不行的，因为之前没有声明该方法的原型，另外也感觉怪怪的，好像哪里不对）。

最后，类方法要放于两个类定义之后，且类定义中不能使用友元的方法在内联函数中（但是在类定义后的类方法中，可以加关键字inline使用内联函数作为替代）。

如代码：

#include<iostream>

//类定义

class M

{

int a;

public:

friend class N; //这样要在使用其类的类方法之前

M() :a(0) {}

void add(N&n); //N是友元，故能直接引用N作为参数

void show();

};

class N

{

int b;

public:

friend class M;//这样要在使用其类的类方法之前

N():b(5){} //默认构造函数，初始值不同

void add(M&m); //M是友元，故能直接引用N作为参数

void show();

};

//类方法定义

void M::add(N&n)

{

a++;

n.b++;

}

void M::show() //显示其值，也可以将N类对象作为参数传递，用于显示N类对象的数据成员的值

{

std::cout << "a:" << a << std::endl;

}

void N::add(M&m) //两个类的方法，对数值操纵不同，以区分

{

m.a += 10;

b++;

}

void N::show()

{

std::cout << "b:" << b << std::endl;

}

//测试代码

int main()

{

using namespace std;

M one;

N two;

one.show();

two.show();

cout << "M对象add方法" << endl;

one.add(two);

one.show();

two.show();

cout << "N对象add方法" << endl;

two.add(one);

one.show();

two.show();

system("pause");

return 0;

}

运行结果是：

a:0

b:5

M对象add方法

a:1

b:6

N对象add方法

a:11

b:7

请按任意键继续. . .

**共同的友元：**

假如在一个函数中，需要同时使用两个类的私有成员。

使用一个类的私有成员，有两个办法：

①是该类的类方法（继承是不能直接调用其私有成员的）；

②是该类的友元函数。

而这个函数若要使用两个类的私有成员，

首先，①+①的方法是不可能的，因为不可能同时是两个类的类方法。

其次，①+②或者②+①的方法组合，是可以的，上面已经说明了。

最后，②+②的方法，也是可行的。即一个函数，既是第一个类的友元函数，也是第二个类的友元函数。

②+②的方法很简单，两个类的类对象都是其参数，然后在其函数内部使用friend声明其是友元函数（就像使用friend ostream& operator<<(ostream&, 类名& m)这个运算符重载函数一样）。

但由于要在第一个类中，声明友元函数，且友元函数的参数使用第二个类对象（该函数的参数包括第一个类的类对象和第二个类的类对象）。因此，第二个类应该前向声明其类声明。

但在第二个类中就无需这么做了（因为第一个类已经进行定义过了）

如代码：

#include<iostream>

class N;

class M

{

int a = 1;

public:

friend void show(M&m, N&n);

};

class N

{

int b = 2;

public:

friend void show(M&m, N&n);

};

void show(M&m, N&n)

{

std::cout << "M:" << m.a << ", N:" << n.b << std::endl;

}

int main()

{

M a;

N b;

show(a, b);

system("pause");

return 0;

}

输出内容如下：

M:1, N:2

请按任意键继续. . .

**第十五章之（二）嵌套类**

在C++中，可以将类声明放在另一个类中。

在另一个类中声明的类，被称为 **嵌套类（nested class）** 。

它通过提供新的类型类作用域类避免名称混乱。

包含类的成员函数，可以创建和使用被嵌套类的对象。

而仅当声明位于公有部分时，才能在包含类的外面使用嵌套类。而且必须使用作用域解析运算符（旧版本可能不允许使用嵌套类，或无法完全实现这种概念）。

不懂，感觉是一个类A的类声明在另外一个类B里（此时，这个A类的声明，即可能在B的公有部分，也可能在私有部分，还可能在保护部分。），只有当类A的声明在类B的公有部分，类A才能在类B外面使用（即类似数据成员位于公有部分）。并且要加作用域解析运算符才能调用A。

有点类似在类中声明一个结构。

对类进行嵌套，与包含并不相同。包含意味着将类对象作为另一个类的成员（但需要在类定义里面进行声明，并且声明几个就是几个，除非使用new来分配，但这需要一个指针）。而对类进行嵌套不创建类成员，而是定义了一种类型。该类型仅在包含嵌套类声明的类中有效（也就是说，只能在该类方法中声明此类）。

**嵌套和包含的区别：**

从以上说明来看：

**包含**一般是固定数量的类对象，或者使用new来动态分配，但就算使用new来动态分配，其指针的数量是一定的（在类定义时确认）。但由于在每个类方法中都可以使用该指针，因此在不同类方法中，可以共享同一个数据。

**嵌套**，可以自由在类方法中声明类对象，如果该类是自动变量的话，那么离开该类方法时，会自动销毁（自动变量的特性）。因此，一个类方法中声明的嵌套类的对象，不能在另一个类方法中使用（因为其作用域是该类方法）。

假如在一个类中声明另一个类，文中使用的是队列模拟（特点是队首删除，队尾添加）。之前使用的是结构形式（一个成员为项目，即队列里的东西，另一个成员为指针，指向下一个结构的位置）

现就将结构改为类的形式，进行推断：

①首先，数据成员保持不变，即一个数据成员为项目，另一个数据成员为指针。但此时，指针的类型更改，变为该类的指针；

②当添加新对象时（队尾添加），则需要创建一个新的项目，因此需要通过类对象来创建。

由于项目在类中，因此可以调用类的构造函数，在创建类对象的时候，同时创建该项目。又因为项目类型不定（比如说可能是某个类），因此，在构造函数中使用成员初始化列表。（这样，假如项目是类，将自动调用其对应构造函数）。

因此，我们需要自定义类的构造函数。

另外，这个构造函数，要能将指针自动指向空指针（因为新加入的便是最后一个）

③其他变化不大，每进来一个，则要new一个新的（并且赋值）。每结束一个，则调用delete（与new对应）

代码如下（用之前的ATM模拟测试）：

#include<iostream>

#include<ctime>

class Customer

{

long arrive; //到达时间

int processtime; //自身消耗时间

public:

Customer() { arrive = processtime = 0; } //默认构造函数，初始化

long when()const { return arrive; } //返回到达时间

int ptime()const { return processtime; } //返回消耗时间

void set(long when) //将到达时间作为参数传递

{

processtime = std::rand() % 3 + 1; //等待1~4单位时间

arrive = when; //到达时间等于when

}

};

typedef Customer Item;

class Queue

{

private:

//先声明一个类

class Node

{

public:

Item item;

Node\* next;

Node(const Item&it) :item(it) { next = NULL; } //默认为空指针，将项目传递给私有数据成员

};

int items = 0; //默认情况下，size（当前项目数）为0

const int MAX; //最大容纳

Node \*front; //指向第一个

Node \*rear; //指向最后一个

public:

Queue(int m = 10);

~Queue();

bool isFull()const { return items == MAX; } //如果等于最大则满

bool isEmpty()const { return items == 0; } //如果等于0则空

int QueueCount() { return items; } //返回数目

bool enqueue(const Item&it); //放入

bool dequeue(Item&it); //取出

};

Queue::Queue(int m) :MAX(m) //构造函数，指针默认为空指针，默认情况下，MAX的值为10

{

front = NULL;

rear = NULL;

}

Queue::~Queue()

{

while (front != NULL) //如果第一个指针指向的不是空（说明不是最后一个

{

Node\*temp = front; //临时指针

front = front->next; //front指向下一个

delete temp; //删除当前对象（new出来的）

}

}

bool Queue::enqueue(const Item&it)

{

if (isFull())return false;

Node\*add = new Node(it); //new一个动态对象（否则就成了自动变量了）

if (front == NULL) //如果front是空，说明这是加入的第一个，否则front必然指向第一个（那就不是空）

rear = front = add; //如果是新对象，那么rear和front都应该指向当前对象

else //否则加入的不是第一个

{

rear->next = add; //上一个Node的指针指向新创建的对象（最后一个）

rear = add; //指向最后一个的指针，指向新创建的

}

items++;

return true;

}

bool Queue::dequeue(Item&it)

{

if (isEmpty())return false;

Node\*temp = front;

front = front->next; //指向第一个的指针，指向下一个对象（删除后的第一个）

delete temp; //删除原来的第一个对象

if (front == NULL) //如果front是空指针（说明刚刚删除的是最后一个对象）

rear = front; //那么rear也应该是空指针

items--;

return true;

}

const int MIN\_PER\_HR = 60;

bool newcustomer(double x);

int main()

{

using namespace std;

srand(time(0));

cout << "Case Study: Bank of Heather Automatic Teller\n";

cout << "Enter maximum size of queue: ";

int qs;

cin >> qs;

Queue line(qs);

cout << "Enter the number of simulation hours: ";

int hours;

cin >> hours;

long cyclelimit = MIN\_PER\_HR\*hours;

cout << "Enter the average number of customer per hour: ";

double perhour;

cin >> perhour;

double min\_per\_cust;

min\_per\_cust = MIN\_PER\_HR / perhour;

Item temp;

long turnaways = 0;

long customers = 0;

long served = 0;

long sum\_line = 0;

int wait\_time = 0;

long line\_wait = 0;

for (int cycle = 0; cycle < cyclelimit; cycle++)

{

if (newcustomer(min\_per\_cust))

{

if (line.isFull())

turnaways++;

else

{

customers++;

temp.set(cycle);

line.enqueue(temp);

}

}

if (wait\_time <= 0 && !line.isEmpty())

{

line.dequeue(temp);

wait\_time = temp.ptime();

line\_wait += cycle - temp.when();

served++;

}

if (wait\_time > 0)

wait\_time--;

sum\_line += line.QueueCount();

}

if (customers > 0)

{

cout << "customers accepted: " << customers << endl;

cout << " customers served: " << served << endl;

cout << " turnaways: " << turnaways << endl;

cout << "average queue size: ";

cout.precision(2);

cout.setf(ios\_base::fixed, ios\_base::floatfield);

cout << (double)sum\_line / cyclelimit << endl;

cout << " average wait time: " << (double)line\_wait / served << " minutes\n";

}

else

cout << "No customers!\n";

cout << "Done!\n";

system("pause");

return 0;

}

bool newcustomer(double x)

{

return (std::rand()\*x / RAND\_MAX < 1);

}

测试结果与之前相同，说明代码正常运行。

**调用嵌套类中的方法：**

假如Node是一个普通的类，当我们调用它的方法时，是这样做的：Node::isFull();

而此时，Node类在Queue类中，因此，我们调用Node类方法时，则不能像之前那样，但方法类似，加上新的作用域解析运算符，即：Queue::Node::isFull();即可。（前提是，Node类在Queue类的public部分声明）

同样，假如需要在类外声明一个在类内public区域声明的类，需要在类型名前加上作用域解析运算符。例如Queue::Node one; 这样。

但在private和protected区域声明的类，则无法做到这点（即无法在类外使用）。

protected可以在派生类使用，而private只能在当前类使用。

**将嵌套类放在模板类之中：**

将嵌套类放在模板类之中，很简单，和使用typedef类似。

即使用模板类的<class XX>中的XX作为其类型名。

例如Node类的声明不变（其项目类型为Item），修改Queue类声明：

template<class Item>

class Queue

{

class Node

{

Item item;

...

};

....

};

之后，按照使用模板类的方法，逐个修改Queue类的类方法的函数定义即可。

**第十五章之（三）RTTI**

注：跳过了第十五章的**异常**。

所谓RTTI，是运行阶段类型识别（Runtime Type Identification）的简称，这是添加到C++中的特性之一。

很多老式的实现不支持，另一些实现可能包含开关RITI的编译器设置。

RTTI旨在为程序在运行阶段确定对象的类型提供一种标准方式。很多类库已经为其类对象提供了实现这种功能的方式，但是由于C++内部并不支持，因此各个厂商的机制通常互不兼容。创建一种RTTI语言标准将使得未来的库能够彼此兼容。

**RTTI的用途：**

假如有一个类层次结构，其中的类都是从同一个基类M派生而来的（他们可能之中可能有二代、三代甚至更多的派生类，但都不是多重继承，并且基类是M或者是M的派生类）。因此，可以让基类指针（M\*）指向其中任何一个类的对象（因为基类指针可以指向派生类对象）。

然后，基类指针调用这样一个函数：这个函数的功能是处理一些信息后，选择一个类，并创建该类的对象（例如A类方法内，创建一个B类对象；而在C类方法内，创建一个D类对象。但此时，我们并不知道这个基类指针调用的是A类的方法，还是C类的方法，因此自然也不知道创建的什么类的对象了）。

然后返回创建的新对象的地址，并将这个对象的地址赋给基类指针（这是可以的，这个基类指针指向了另一个派生类的对象）。

那么问题来了，在这个时候，这个基类指针是指向B类对象，还是D类对象呢？

当我们需要解决某些问题的时候，我们需要知道这件事，因此这也就是RTTI的用途。

**为什么要知道指针指向的类型：**

当我们需要知道类型时，在以上的基础上，有三种情况：

①我们可能需要调用类方法的正确版本。

例如：如果我们需要调用某一个类方法时。

（1）该类方法都是虚方法，且需要调用对应类的类方法，则无需知道，指针会根据指向对象的类型调用对应的类方法。

（2）该类方法都是虚方法，但需要调用基类的类方法，需要知道类型。因为基类A可能是派生类C的基类B的私有成员（即B是A类的派生类，且是私有继承，而C类又是B类的基类），若是这种情况，则无法使用基类的方法（因为不能访问基类对象，更不要说基类的方法了）。

②需要使用不同的方法。例如假如是派生类B，则使用a方法，如果是派生类c，则使用b方法。那么自然需要知道类型，才能正确的使用计划之中的方法。

③可能出于调试目的，想跟踪生成的对象的类型。看看生成的对象，是不是自己计划中想要生成的，会不会出现想要生成一个A类对象，却生成了一个B类对象。因此，需要通过知道类型，来检测。

可以看出，以上三种情况，都需要知道类型。因此，需要RTTI。

**RTTI的工作原理：**

C++有3个支持RTTI的元素：

①如果可能的话，dynamic\_cast运算符将使用一个指向基类的指针来生成一个指向派生类的指针。如果做不到，则该运算符返回一个空指针（0）。

②typeid运算符返回一个指出对象的类型的值。

③type\_info结构存储了有关特定类型的信息。

只能将RTTI用于包含虚函数的类层次结构，原因在于，只有在这种情况下，才应该将派生类地址赋给基类指针。

**警告：**RTTI只适用于包含虚函数的类。

也就是说，如果想使用RTTI，基类和派生类必须要有虚函数，否则无法使用。

**dynamic\_cast运算符：（总的来说，有的迷糊）**

dynamic\_cast运算符的作用在于，其用于确定和回答“是否可以安全的将对象的地址赋给特定类型的指针”。

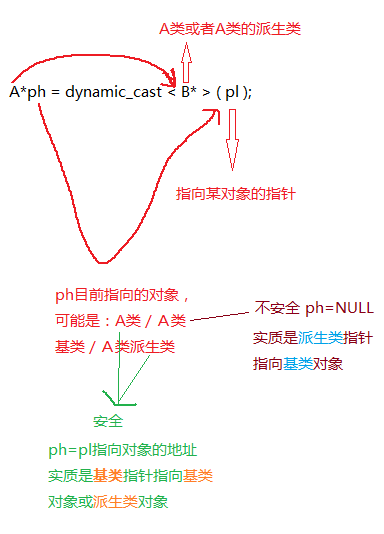
其格式为：**A类\*ph = dynamic\_cast <B类\*> ( pl ) ;**

解释：

①一般A类是派生类，B类是A类或者A类的派生类（如果B类是A类的派生类的话，意思是将指向派生类对象的地址(因为将其进行类型转换了)的指针pl赋值给基类指针ph，因为基类指针指向派生类对象的地址是安全的，反过来则不安全）；

②**作用是：将一个指针pl**（其类型待定，也不确定其指向对象是基类还是派生类），**将其类型转换为B类**（类型不定）**后赋值给A类的**（A类是B类的基类或者是B类，但总之是基类的派生类）**指针ph**。

此时， A类的指针，指向了另一个对象的地址，而这个对象的类型有三种可能：**A类**／**A类的基类**／**A类的派生类**。只有当对象是A类和A类的派生类时，这种转换才是安全的。



③使用dynamic\_cast的前提是多态**（但不太明白为什么）**。

④之所以这么做，有一种可能是基类指针虽然可能指向派生类对象，但只能调用基类的方法。如果想要调用派生类方法，则需要转换，但强制转换不一定是安全的（因为我们不确定其是否指向派生类对象），因此需要这个命令来告诉我们是否是安全的。

⑤这个过程不转换pl的类型

举个例子，例如A类是基类，B类是A类的派生类，C类是B类的派生类（A->B->C）。

①当A类指针a，指向C类对象时，这是可以的（基类指针可以指向派生类对象）。注意，A类指针a的值，是C类对象的地址。

那么将这个A类指针a，强制类型转换为C类指针是否可以呢，答案是可以的。

因为这样做的结果是，C类指针c，指向一个C类对象（强制类型转换后，其值相同，只不过类型变了）。

之所以说是安全的，因为不会产生那种派生类指针调用方法时，调用的是派生类方法，却因为指向基类对象而导致在基类中不存在派生类的同名方法，从而出现调用失败。（见下面代码）（基类指针指向派生类对象之所以安全，是因为基类方法必然被派生类所继承，必然存在继承而来的方法或者同名方法）。

②那么假如A类指针a，指向A类对象，被强制转换为C类指针并被赋值给C类指针c（此时，派生类指针指向一个基类对象）

那么这种做法就是不安全的。

假如c调用的方法是基类有的，那么可能不出现问题。

但是假如c调用的方法，是基类A没有的（这是可能的，因为派生类指针理应可以使用派生类有方法，所以编译器无法检测出这种错误）；

那就会出现问题（预料之外的问题）。

如代码：

#include<iostream>

class B

{

int a = 0;

public:

virtual void add() { a++; }

};

class A :private B

{

int c = 1;

public:

virtual void add() { B::add(); c ++; }

void mm() { std::cout << c << std::endl; }

};

int main()

{

using namespace std;

B b;

B\*ph = &b; //基类指针指向基类的地址

A\*pl = (A\*)ph; //将基类指针强行转换为派生类指针，并将地址赋值给派生类指针。此时派生类指针指向了一个基类对象的地址

pl->mm(); //指向基类对象的派生类指针调用派生类方法。因此方法调用会输出错误的结果

pl = dynamic\_cast<A\*>(ph); //使用dynamic\_cast进行转换，如果是安全的，pl则不是空指针

if (pl == nullptr)cout << "qqqq" << endl; //如果是空指针，则输出qqqq

A\*pll = new A; //派生类指针指向派生类对象

pll->mm(); //因此方法调用是正常的

delete pll;

A pp;

A\* qq = &pp; //派生类指针指向派生类对象

pll = dynamic\_cast<A\*>(qq); //使用dynamic\_cast进行转换，如果是安全的，则不是空指针

if (pll == 0)cout << "3333" << endl; //如果输出了3333，则说明是空指针，上一行代码的转换并不安全

system("pause");

return 0;

}

其输出结果是：

-858993460

qqqq

1

请按任意键继续. . .

可以从输出结果看到，正常指向派生类对象的mm方法输出的内容是1（因为c被初始化为1），而指向基类对象的mm方法输出的内容是错误的，甚至可能运行出错。

**因此，指向基类对象的派生类指针，是不安全的。**

**而一个指向派生类对象的基类指针，是安全的；**

**并且，将该指针转换为一个不高于其派生类层次的（即是该派生类或该派生类的基类的类型的）指针，并赋值给同样不高于该派生类层次的指针，也是安全的。**

因为有区别，所以这是dynamic\_cas的存在目的。

上代码，用代码表示其作用：

//程序目的：随机创造一个类型的对象，并对其初始化，然后将地址以A类形式返回，并赋给A类指针，然后用A类指针调用show函数（虚函数，三个类都有）

//之后，再尝试用A类指针用dynamic\_cast转化为B类指针，查看是否安全，如果安全，则调用take函数（虚的，只有B和C有）

//在第二步的时候，如果不安全，则输出一个提示信息，如果安全，因为take函数是虚的，因此会根据其指向的对象，输出对应类的take函数。

#include<iostream>

#include<ctime>

using namespace std;

class A

{

public:

virtual void show() { cout << "This is A show." << endl; }

};

class B :public A

{

public:

virtual void show() { cout << "This is B show." << endl; }

virtual void take() { cout << "B take out one thing." << endl; }

};

class C :public B

{

public:

virtual void show() { cout << "This is C show." << endl; }

virtual void take() { cout << "C take out one thing." << endl; }

};

A\* GetOne() //随机创造A、B、C对象之一，并返回其地址（以指针形式，类型为A\*）

{

int i = rand() % 3;

A\* one;

if (i == 0)

one = new A;

else if (i == 1)

one = new B;

else one = new C;

return one;

}

int main()

{

A\* pp;

srand(time(0));

for (int i = 0; i < 5; i++)

{

pp = GetOne(); //随机创造一个类对象，返回其地址，用基类（A）指针指向它

pp->show(); //基类指针根据指向对象输出对应的虚方法

B\* pa = dynamic\_cast<B\*>(pp); //B类指针

if (pa == NULL)cout << "错误的转换，pa是空指针" << endl; //如果不安全，则输出提示信息

else pa->take(); //如果安全，则输出对应的方法

cout << endl; //空一行表示间隔

delete pp;

}

system("pause");

return 0;

}

显示：

This is B show.

B take out one thing.

This is A show.

错误的转换，pa是空指针

This is B show.

B take out one thing.

This is C show.

C take out one thing.

This is A show.

错误的转换，pa是空指针

请按任意键继续. . .

**总结：**

①可以发现，在转换并不安全时（即B类指针pa指向的是A类对象时，基类调用派生类方法是不正确的），返回NULL。因此可以鉴别。

②程序可能不支持RTTI，或者支持，但是关闭了这个功能。如果是后者，那么可能编译的时候没问题，但是运行的时候出现问题。

③另外，dynamic\_cast也可以用于**引用**。只不过，由于没有与NULL对应的引用值，因此当请求不正确时，dynamic\_cast将引发类型为bad\_cast的异常。该异常是从exception类派生而来的，他是在头文件typeinfo定义的。书上另外给了一个例子，但是因为我跳过了异常章节，所以看不懂。

**typeid运算符和type\_info类：**

typeid运算符使得能够确定 **两个对象** 是否为 **同种类型** 。

格式为：**typeid ( 对象A ) == typeid ( 对象B );**

解释：它与sizeof有些相像，可以接受两种参数（**即对象A和对象B所在的位置**）：

①类名；

②结果为对象的表达式。（如对象名，或者是指向对象的指针使用了解除引用运算符。

这段看到这里是不懂的，只知道可以用“==”和“!=”来判断左右两边类型是否相同==>>typeid运算符返回一个对type\_info对象的引用（这句话的意思可以看后面的解释）。其中，type\_info是在头文件**typeinfo**（以前是typeinfo.h）中定义的一个类。type\_info类重载了“==”和“!=”运算符，以便可以使用这些运算符来对类型进行比较。

另外：不能使用cout<< typeid( xxx)来输出，只能使用==和!=

例如，如果pg指向的是一个C类对象，则下述表达式的结果为boo值true，否则为false：

**typeid (C) ==typeid (\*pg);**

注意：如果pg是一个空指针NULL，将引发bad\_typeid异常。该异常类型是从exception类派生而来的，是在头文件typeid中声明的。

type\_info类的实现随厂商而异，但包含一个name()成员，该函数返回一个随实现而异的字符串：通常（但并非一定）是 **类的名称**。

其格式为：**typeid ( 某类型对象).name();**

如：

cout << "当前类型为：" << typeid(\*pp).name() << endl; //输出pp指针指向对象的类型

修改上面代码的主函数部分为：

int main()

{

A\* pp;

srand(time(0));

for (int i = 0; i < 5; i++)

{

pp = GetOne(); //随机创造一个类对象，返回其地址，用基类（A）指针指向它

pp->show(); //基类指针根据指向对象输出对应的虚方法

B\* pa = dynamic\_cast<B\*>(pp); //B类指针

if (pa == NULL)cout << "错误的转换，pa是空指针" << endl; //如果不安全，则输出提示信息

else pa->take(); //如果安全，则输出对应的方法

if (typeid(B) == typeid(\*pp))cout << "对象类型为B" << endl; //指针指向对象的类型是否为B，如果是B返回true，否则false

else cout << "对象类型不为B" << endl;

cout << "当前类型为：" << typeid(\*pp).name() << endl; //输出pp指针指向对象的类型

cout << endl; //空一行表示间隔

delete pp;

}

int mm;

cout << "int类型变量mm的类型为：" << typeid(mm).name() << endl; //调用name()方法输出int类型的变量的类型

system("pause");

return 0;

}

输出结果：

This is C show.

C take out one thing.

对象类型不为B

当前类型为：class C

This is C show.

C take out one thing.

对象类型不为B

当前类型为：class C

This is B show.

B take out one thing.

对象类型为B

当前类型为：class B

This is A show.

错误的转换，pa是空指针

对象类型不为B

当前类型为：class A

This is A show.

错误的转换，pa是空指针

对象类型不为B

当前类型为：class A

int类型变量mm的类型为：int

请按任意键继续. . .

总结：

①问题：书上说需要头文件typeinfo，但是我不加这个头文件也可以使用，不知道为什么。我的编译器是VS2015。

②之前说过，typeid运算符返回的是一个对type\_info对象的引用。

例如typeid(\*pp) 之所以能调用name()方法，正是因为其表示的是一个类对象，name方法是该类的类方法。

③编译器不同时，name()方法输出的内容有可能不同（例如我目前使用的vs2015，输出的内容是class xx）。

**误用RTTI的例子：**

判断是否安全，应使用dynamic\_cast，判断类型是否一样，用typeid。

所谓的误用，我大概概括了一下，就指的是错误的使用，反而让代码更复杂和繁琐了。

假如需要当指针指向不同类型时，调用对应类型对象的同名方法，则应该是使用dynamic\_cast和虚函数（因为是同名方法，使用虚函数会根据指针指向的对象类型而决定选择哪个，如果是NULL，则不调用）。

例如可以将以下代码修改：

B\* pa = dynamic\_cast<B\*>(pp); //B类指针

if (pa == NULL)cout << "错误的转换，pa是空指针" << endl; //如果不安全，则输出提示信息

else pa->take(); //如果安全，则输出对应的方法

修改为：

B\* pa; //B类指针

if (pa = dynamic\_cast<B\*>(pp))pa->take(); //如果安全，则输出对应的方法

else cout << "错误的转换，pa是空指针" << endl; //如果不安全，则输出提示信息

if判断语句中的pa=dynamic\_cast<B\*>(pp) 表达式，假如是空指针，则表达式的结果为0（将0赋值给pa），执行else；如果不是空指针，则执行pa->take()。

问题：

书上说，如果放弃dynamic\_cast和虚函数，将代码改为类似这样的：

int main()

{

srand(time(0));

for (int i = 0; i < 5; i++)

{

A \*pp = GetOne(); //随机创造一个类对象，返回其地址，用基类（A）指针指向它

if (typeid(C) == typeid(\*pp))

cout << "1" << endl;

else if (typeid(B) == typeid(\*pp))

cout << "2" << endl;

else

cout << "3" << endl;

cout << endl; //空一行表示间隔

delete pp;

}

system("pause");

return 0;

}

可以起作用（即根据pp指向的对象决定输出哪个）。

但是我实际操作中，假如不使用虚函数，pp的类型则只为A类，而不会成为B类和C类。

只有加上虚函数后，代码才会和书上的运行结果相同。

我的编译器是VS2015，不知道是不是因为编译器的原因。

按书上的说法，假如if else句式中使用了typeid，则应该考虑是否应该使用dynamic\_cast。

备注：

这小节我学的还是有点头晕，可能是因为没有和实际结合使用的原因。也许遇见了实际情况，就知道该如何使用了。

**第十五章之（四）类型转换运算符**

类型转换运算符，共计有4个：

①dynamic\_cast;

②const\_cast;

③static\_cast;

④reinterpret\_cast;

第一种dynamic\_cast，用于将派生类指针转化为基类指针，前提是派生类指针指向的对象，对于基类是安全的。

**const\_cast：**

第二种是const\_cast，该运算符用于执行只有一种用途的类型转换，即改变值为const或者volatile（前者是不允许被改变，后者是允许在const里被修改）。

简单来说，假如一个const指针指向了一个对象，那么使用这个指针是不能修改对象的值的（但是可以修改指向的对象）；

在这种情况下，假如用另外一个指针，指向当前指针，会发生两种情况：

（1）使用非const指针指向这个const指针，即使类型一样，也无法成功赋值（编译器提示错误）；

（2）使用另一个const指针指向他，可以实现。

但假如，因为某种需求（比如说，需要通过另一个指针来指向这个const指针，以修改const指针指向的地址的值），那么，显然只能用非const指针了（只有非const指针才能修改指向的值），因此用一个非const指针指向这个const指针。

矛盾出现了，非const指针不能指向一个const指针，因此必须使用const\_cast运算符。

其格式为：

**类型\* 指针名 = const\_cast <类型\*> (另一个被限定的指针名）;**

例如：

int a=1;

const int \*b = &a;

int \*c = const\_cast<int\*>(b);

这段代码的作用，即是让非const指针指向了一个const类型的int指针b；

另外，也可以让一个const指针，指向一个volatile指针，如代码：

int a=1;

volatile int \*b = &a;

const int \*c = const\_cast<const int\*>(b);

**适用于类对象：**

同样，不仅可以用于基本类型，还可以用于自定义的类。

例如有一个Student类，使用一个const限定的Student类指针pr，指向了一个创建的Student对象。那么，可以通过const\_cast运算符，创建另一个指针pd，指向pr指向的Student类对象。便可以通过pd来修改pr指向的对象的值。

**不能修改const限定的对象的值：**

如代码：

const int a=1;

const int \*b = &a;

int \*c = const\_cast<int\*>(b);

\*c = 2;

cout << a << endl;

虽然编译不会提示错误，但其输出的结果依然为1，而不是2。

**不适用于不同类对象：**

例如，const指针pr指向Student类对象，而指针pd是Teacher类，那么，就无法将pr通过const运算符赋值给pd。

**static\_cast：**

第三种是static\_cast运算符。

按照搜索得到的资料，static\_cast运算符和dynamic\_cast运算符类似。

假如有基类A，其派生类B，那么代码：

A\* a;

B \*b1 = dynamic\_cast<B\*>(a); //前提是类A和类B必须有多态（即虚函数）

B \*b2 = static\_cast<B\*>(a);

区别①：b1那行代码成立的前提是，多态的存在（即虚函数，至少A有一个虚函数），而b2那行代码则不需要。

区别②：dynamic\_cast运算符，如果转换失败，则返回的是一个空指针（可以用代码进行判断），static\_cast，如果转换失败，编译时不会出错，但运行时会出错（不像dynamic\_cast那样返回空指针）

区别③：dynamic\_cast的开销比static\_cast的开销大；

区别④：dynamic\_cast在编译时执行转化，而static\_cast在运行时执行转换。

**static\_cast运算符出错的条件：**

①假如A类和B类毫无关系（不存在派生和继承关系），那么会出错；

②假如指针a，指向的不是一个B的实例（B类对象），那么也会出错；

问题：假如A类是基类，B类和C类分别是A类的派生类，那么，

A\* a = new C;

B \*b1 = dynamic\_cast<B\*>(a); //前提是类A和类B必须有多态（即虚函数）

B \*b2 = static\_cast<B\*>(a);

为什么这两行代码都不会出错呢？

**static\_cast的其他用途：**

①将空指针转化为其他类型的空指针，如代码：

void\* a = 0;

int \*b = static\_cast<int\*>(a);

②将任何类型的表达式转换为void类型；

int\* a = 0;

void \*b = static\_cast<void \*>(a);

③用于基本类型的转换（但需要程序员自己来保证安全）；

补充：我查的某个资料说，这个运算符，好像并没有什么用。。

**reinterpret\_cast：（这个不明白）**

第四种是reinterpret\_cast

作用：是用来处理无关类型之间的转换；它会产生一个新的值，这个值会有与原始参数（expressoin）有完全相同的比特位。

不懂，反正感觉意思就是，跟bit相关的，但具体不明白。

特点是，转换时，不允许删除const，但会进行一些敏感的操作（容易出错的操作）。依赖于底层，通常不可移植。

以书上给的例子为例：

struct m

{

short a;

short b;

};

long q = 0xA224B118;

m\*p = reinterpret\_cast<m\*>(&q);

cout << hex << p->a;

输出结果为：b118（即q的前4位——》低位的四位）；

**何时使用：**

①从指针类型（因为指针存的是地址）到一个足够大的整数类型（否则存不下）

②从整数类型或者枚举类型到指针类型（位数少到位数多是没损失的）

③从一个指向函数的指针到另一个不同类型的指向函数的指针（不懂为什么用）

④从一个指向对象的指针到另一个不同类型的指向对象的指针（同上）

⑤从一个指向类函数成员的指针到另一个指向不同类型的函数成员的指针

⑥从一个指向类数据成员的指针到另一个指向不同类型的数据成员的指针

以上都不太明白

**其作用：**

可以无视类型之间的隔离（例如不同类型之间，很可能是不能转换的。例如不同类），但使用这个就可以（可以编译和运行）。

唯一的问题是，程序员需要对其进行负责，保证其一定能用，否则在运行时会出错。

参考链接：

http://blog.csdn.net/jfkidear/article/details/7718651

类型转换运算符有四个：

①dynamic\_cast

其格式如下：

**基类指针 = dynamic\_cast<基类类型\* > (派生类指针);**

假设有High类和Low类，而ph和pl的类型分别为High\*和Low\*。

当且仅当Low是High的可访问基类（直接或者间接）时，下面的语句才将一个Low\*指针赋给pl：

pl=dynamic\_cast<Low\*>ph; //ph是High类指针，将其试图转换为Low类指针

如果Low类满足以上条件，那么该语句将空指针赋给pl。

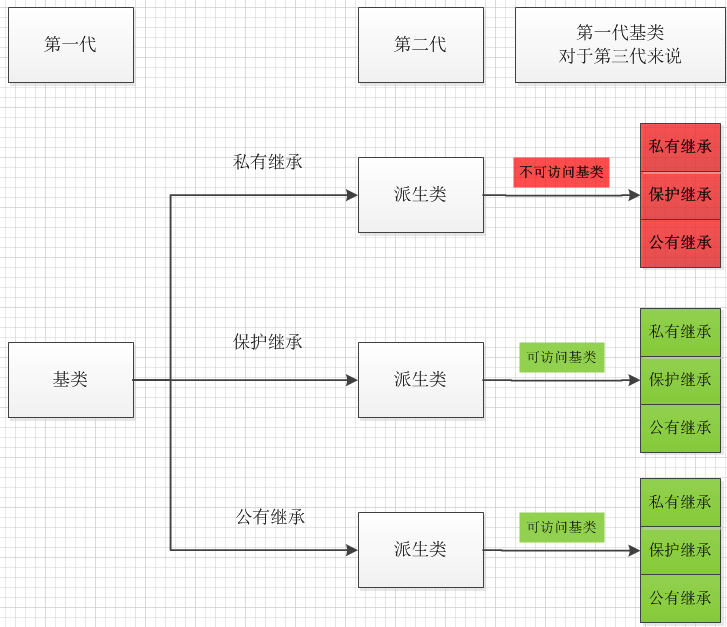
即：要将一个A类的指针赋给另一个B类，前提是A类是B类的派生类（并且B类是A类的可访问基类），这时可以正常转换，否则，被赋值的对象的值是空指针。

所谓的可访问基类，根据书上所说的来看，是这个意思（推测，但应该没问题）：

（1）假如B类是A类的基类，那么A类（是B类的派生类，但不确定是几代）要能访问到B类这个基类；

（2）如果A类要能访问到B类这个基类，那么B类是A类的公有/保护/私有成员；

（3）但B类不能是A类的基类（即B是基类，A类是第三代或者以上）的私有成员（派生类不能访问基类的私有成员，只能访问公有和保护成员）；

（4）但B类可以是A类的私有成员（即A类可以通过私有继承派生出A类，B类是第一代A类是第二代）；

红色的结果，被赋值的对象是NULL；

绿色的结果，派生类指针（ph）将被转换为基类指针并被赋值给基类指针（pl）

简单通俗的来理解：

（1）基类指针可以指向其派生类对象（这是必然的）（向下转换）（虚函数相关）；

（2）派生类指针并不能在任意情况下指向基类对象（如果基类是其私有成员则不能指向）；

（3）派生类指针指向基类对象时，那么需要使用基类的方法（如果

②