本文作者：黄邦勇帅

学习本文首先你应熟悉 C++中的构造函数，基本的类的声明及怎样初始化类，关于这些问题，请参看本人所作的《C++构造函数,复制构造函数和析构函数》一文，在这篇文章中作了详细的介绍。
掌握 C++类中的各种成员，是学习好 C++的基础，因此对于本文的内容应全部熟练掌握，本文主要集中介绍 C++类中的各种成员，这些成员分别是：类中的静态(static)成员变量，成员函数；const(常量)成员变量，成员函数和 const 对象；const static(常量静态)数据成员；对象数组；类中的对象成员；类成员指针；嵌套类；友元；this 指针以及.*和->*运算符共 12 种内容。本文内容全面，简单易懂，是学习 C++不错的选择。
本文内容完全属于个人见解与参考文现的作者无关，其中难免有误解之处，望指出更正。

主要参考文献：

1、C++.Primer.Plus.第五版.中文版 [美]Stephen Prata 著 孙建春 韦强译 人民邮电出版社 2005 年 5 月
2、C++.Primer.Plus.第四版.中文版 Stanley B.Lippman、Barbara E.Moo 著 李师贤等译 人民邮电出版社 2006 年 3 月
3、C++.Primer.Plus.第三版.中文版 Stanley B.Lippman 等著 潘爱民 张丽译 中国电力出版社 2002 年 5 月
4、C++入门经典 第三版 [美]Ivor Horton 著 李予敏译 清华大学出版社 2006 年 1 月
5、C++参考大全 第四版 [美]Herbert Schidt 著 周志荣 朱德芳 于秀山等译 电子工业出版社 2003 年 9 月
6、21 天学通 第四版 C++ [美]Jesse Liberty 著 康博创作室 译 人民邮电出版社 2002 年 3 月

类中的各种成员

const ,static,[],*p,const static,对象成员，常量对象，mutable

1. **类中的静态成员变量 static：**被类中的所有对象所共享，静态成员属于整个类不属于某个对象。静态成员变量和全局变量差不多，只是他的作用范围为定义他的类及类的对象所知。
   - **1.1.** 当在一个类中声明静态成员变量时并没有定义该变量即没有为他分配存储空间，所以必须在类外对静态成员变量提供全局定义，注意必须在类外，这样就可以为变量分配内存，定义的方式为使用作用域解析符，比如有类 hyong 类中有静态整型成员 a，那么在类外定义该变量的方式为 int hyong::a=9;注意必须要有在类中声明的类型，如果定义时没有初始化则自动初始化为 0。
   - **1.2.** 静态成员变量如果是类中的公有成员则可以在程序中使用语句 hyong::a 来访问该变量，当然也可以通过对象名来访问该变量，如果变量是私有或者保护的，则只能用静态的成员函数来访问该变量。
   - **1.**3 不能用初始化列表来初始化静态成员变量。
2. **静态成员函数 static，**静态成员函数没有 this 指针，静态成员函数只能引用这个类的其他类静态成员，当然全局函数和数据可以访问，因为类的函数都要用 this 指针来访问成员，因为静态成员函数没有 this 指针，所以不能访问除静态成员之外的成员。同一个函数不可以有静态和非静态两种版本，静态成员函数不可以是虚函数，静态成员函数不能是 const 和 volatile。静态成员函数使用是有限的，通常使用静态成员函数来访问类中的私有静态成员变量。在类外定义的形式为 int hyong::g(){}注意，这里没有 static 说明符。
3. **const 成员变量参看第四节初始化列表。**
4. **const 成员函数，**形式为 int f() const{}注意 const 在括号后，如果把 const 放在 int 前就成为一个反回 const int 类型的函数了，把函数声明为 const 后就使得 this 可以当做一个 const 指针，从而使得函数不能修改调用它的对象也就是说不能改变任何一个类的成员变量的值。如果想让 const 函数能修改某一部分成员，可以把该成员声明为 mutable 类型，例如 mutable int a。在类外定义的形式为 int hyong::f() const{}
5. **const static 常量静态数据成员，**这种类型的成员变量可以直接在类的定义中直接初始化，这也是唯一一种可以在类中初始化的类型，如果没有在类中初始化，在类外的初始化方式为 const int hyong::a=2;注意 const 和类型都必须有。
6. **const 常量对象：**即把对象声明为常量，即 const hyong m，常量对象不能调用可能改变对象的值的函数，因此常量对象只能调用类中的 const 常量函数，因为不是 const 的函数都有可能改变对象的值。
   - 6.2. 常量对象可以调用类中的公有成员，如 m.a 就是正确的如果 a 是公有的。
   - 6.4. 不能对常量对象的公有成员重新赋值，如 m.a=3 就是错误的。但可以对类中的公有静态成员变量重新赋值，因为静态成员变是不属于这个常量对象，他是属于整个类的。
7. **对象数组：**对象数组即数组中的每个成员都是一个对象，例如 hyong a[3];其中 a[0]，a[1]，a[2]都是一个 hyong 类型的对象。对象数组的初始化，如果有默认构造函数则语句 hyong a[3]将调用默认构造函数初始化 3 个对象，如果对象数

组带有一个参数的构造函数则可以这样初始化 hyong a[3]={1,2,3}，如果对象数组带有多个参数的构造函数，则初始化方法为 hyong a[3]={hyong(1,2),hyong(3,4),hyong(4,5)}。

例：各种类成员,对象数组,常量对象的使用。

```
class hyong
{  static int e; //私有静态成员变量，声明静态成员变量时没有为他分配存储空间必须在类外部定义。
public: int a;
         static int b; //静态成员变量
         const static int c=9; //常量静态成员变量，只有这种类型的变量才能在类中声明时初始化，也可以在类外部初始化。
         static int g(); //静态成员函数，一般用这个函数访问类中的私有静态成员变量，其他地方用处不大。
         int f() const;  //常量成员函数，注意const的位置。
         hyong(){a=b=e=4;} //注意，如果在类外部定义了静态成员变量，则可以在构造函数中重新设置他们的值。否则就会错误。
         hyong(int i){a=b=e=i;}       hyong(int i,int j){a=i;b=e=j;}       void ff(){cout<<a;}    };
```

//static静态成员变量定义的注意事项

```
//hyong::hyong():a(1),b(1),e(1){}  //错误，不能用初始化列表初始化静态成员变量。
  int hyong::b=8;      int hyong::e=7;
 //定义静态成员变量，静态成员变量必须在内外定义，以便分配存储空间，如果不提供初值将被初始化为0；
//const int hyong::c=9; //静态常量成员变量也可以在类外初始化，注意其用法，没有static关见字。
```

//static静态成员函数定义的注意事项

```
int hyong::g() {cout<<e<<"\n";return e;
 //cout<<a; a=1; //错误，静态成员函数只能访问类中的静态成员变量，既使不改变变量的值也是不行的，静态成员函数一般用来访问类中的私有静态成员变量}
```

//常量函数定义的注意事项

```
int hyong::f() const {cout<<a<<b<<c<<e<<"\n"; return a;//注意除了静态成员函数外其他函数同样可以访问类中的私有静态成员变量。
                      //a=1;   //错误，const常量成员函数不能改变任何成员变量的值。}
int main()
```

{//static静态成员函数和变量的应用

```
hyong m;  cout<<m.a<<m.b<<m.c<<"\n";
// cout<<m.e;   //错误，静态成员变量e是私有的，对象m不能访问，可以通过静态成员函数来访问e，当然也可以用其他成员函数来访问e。
cout<<m.g()<<"\n";  //调用静态成员函数访问类中的私有静态成员变量e
//cout<<hyong::e;  //错误，静态成员变量e是私有的，只能通过函数来访问
cout<<hyong::b<<"\n";  //因为静态成员变量是属于整个类的，而不属于类的某个对象，所以可以通过作用域解析符来直接访问。
hyong n; cout<<n.b<<"\n";
m.b=5;   cout<<n.b<<m.b<<"\n";//输出两个五，注意这里用对象m改变静态成员变量b的值，但对于对象n来说对象n并没有改变他的值，然而b的值却改变了，这就说明静态成员变量属于整个类，不属于某个对象。
```

//对象数组的应用

```
hyong m2[2]; //声明一个常量数组，这里将调用两次默认构造函数以初始化两个对象m2[0],m2[1]
cout<<m2[0].a<<m2[1].b<<"\n"; //输出两个四，m2[0],m2[1]被默认初始化函数初始化了。
hyong m3[2]={1,2}; //调用带有一个形参的构造函数初始化对象数组的成员
cout<<m3[0].a<<m3[1].a<<"\n"; //输出一和二。
hyong m4[2]={hyong(3,4),hyong(5,6)}; //调用带两个形参的构造函数，初始化对象数组的形式。
cout<<m4[0].a<<m4[1].a<<"\n"; //输出三和五。
```

//const常量对象的应用

```
const hyong m1;//声明一个常量对象，常量对象不能调用可能改变对象值的任何函数。m1调用默认构造函数初始化他，一但被初始化常量对象的值将不能被改变。
m1.f(); //调用m1的常量成员函数，输出四四九四。
//m1.ff(); //错误，m1是常量对象，不能调用除了const之外的任何成员函数，因为他们都有可能改变常量对象m1的值。
//m1.a=2;  //错误，常量对象不能改变成员变量的值。
m1.b=2;//正确，常量对象虽然不能改变成员变量的值，但静态成员变量是个例外，因为他不属于对象m1，他属于整个类，不属于某个对象。
cout<<m1.b<<"\n";}
```

8. 类中的对象成员：即把对象作为另一个类的成员。比如 class hyong1{public: hyong x;}，这时如果声明了一个 hyong1 的对象则调用 hyong 的默认构造函数初始化对象 x，而不管 hyong1 的构造函数有没有初始化对象 x，如果没有声明 hyong1 的对象，则不会初始化对象 x。

2. 如果要在用带有参数的构造函数初始化类中的对象成员，则对象成员必须在初始化列表中初始化。否则将发生错误。
3. 如果在初始化列表中显式对 hyong1 的对象成员 x 初始化了，则用初始化列表的构造函数初始化对象 x。不会再调用对象成员 x 的默认构造函数初始化对象 x 了。
4. 如果再在 hyong1 的构造函数里对 x 重新赋值，即有语句 x=hyong()，此语句不是对对象变量 x 初始化，而是对对象 x 重新赋值，将调用赋值操作符函数。

例：类中的对象成员的注意事项

```
class hyong
{ public: int a,b;   hyong(){a=b=0;cout<<"mogou"<<"\n";}     hyong(int i){a=b=i;cout<<"onegou"<<"\n";}
hyong(int i,int j){a=i;b=j; cout<<"twogou"<<"\n";}  };
class hyong1
{public: int a,b;
hyong x;   //把另一个类对象做为类的成员。且调用hyong的默认构造函数初始化对象x
```

```
//hyong y(9); //错误，不能这样调用hyong的带一个参数的构造函数初始化类对象y;要调用带参数的构造函数初始化类对象成员只能在初始化列表中进行。
hyong1(){a=b=2;cout<<"mogou1"<<"\n";}    hyong1(int i){a=b=i; cout<<"onegou1"<<"\n";}   hyong1(int i,int j);};
//如果要使用带参数的构造函数初始化类中的对象成员，只能在初始化列表中进行
hyong1::hyong1(int i,int j):a(i),b(j),x(3){} //用初始化列表重新初始化类对象成员x，此时就不会调用hyong的默认构造函数初始化类中的对象x了。
int main()
{ hyong m; //输出mogou，此时没有创造hyong1的对象，所以不会初始化hyong1中的类对象x。
  hyong1 n; //此时调用hyong的默认构造函数以初始化hyong1的类对象x。
  hyong1 n1(4,5); //此时用n1的带两个参数的构造函数初始化对象n1，并调用初始化列表的带有一个参数的hyong的构造函数初始化hyong1类的成员变量x，注意，此时不会调用hyong的默认构造函数初始化对象成员x
  n.x=hyong(6,7); //对类中的成员对象x重新初始化，这里是赋值将会调用默认赋值操作符，不会调用默认复制构造函数。
  cout<<n.x.a<<n.x.b; }//访问类中的对象成员的成员的方法。
```

9．类成员指针和.*，->*运算符

9.1．声明类成员指针的方式为：int hyong::*p1 声明了一个指向类中整型成员的指针 p1。 int (hyong::*p2)()注意括号，声明一个指向反回类型为 int 的无参数的函数的指针 p2

9.2. 对类成员指针的初始化方式为：p1=&hyong::a，p2=&hyong::f，注意初始化指向类函数的指针时不能省掉&地址运算符。

.9.3. .*运算符：.*运算符的左侧必须是一个类的对象，而右侧则必须是类类型的成员指针。比如 hyong m;则运用方式为 m.*p1

9.4. ->*运算符：运算符的左侧必须是一个指向对象的指针，而右侧则必须是一个类类型的成员指针。比如有 hyong *p=&m，则运用方式为 p->*p1。

9.5. 类成员指针即指向类中成员的指针注意是直接指向类中的成员而不是指向对象的某一成员的指针，即与指针 p=&m.a 是不一样的。类成员指针提供的是成员在类中的对象的偏移量，不是一个真正的指针。因为不是一个真正的指针所以不能通过指针来访问类中的成员，而只能通过特殊的运算符.*或->*来访问指针指向的成员。比如*p1=2 ，hyong::*p1=2 是错误的，不能对类成员指针指向的类成员直接赋值。cout<<*p1<<hyong::*p1 也是错误的，不能直接用类成员指针来访问类中的成员。

例：类成员指针和.*与->*运算符

```
class hyong
{public: int a,b;   int f(){cout<<"fhanshu"<<a<<"\n";return a;}    hyong(){a=1,b=2;}   hyong(int i){a=5,b=i;}  };
int main()
{ int *p;   //声明一个常量指针
int hyong::*p1; //声明一个类成员指针，注意这种指针是指向类中的成员的，不是指向对象中的某个成员变量
int (hyong::*p2)(); //声明一个类成员指针函数
hyong *p3;  //声明一个指向对象的指针

hyong m;   p=&m.a; cout<<*p<<"\n";  //注意指针p是指向的对象中的某个成员变量。
p1=&hyong::a;            //注意赋地址的格式，注意指针p1是指向的类中的某个成员，和指针p是不同的。
p2=&hyong::f;           //指针p2指向类中的某个函数，注意语句格式，虽然函数名就是函数的地址，但这里必须得有&地址运算符。
cout<<m.*p1<<m.a<<"\n"; //用.*运算符访问对象中的成员变量，.*运算符的左侧必须是类类型的对象，右侧必须是该类型的成员指针。其实m.*p1和m.a都是指的对象m中的成员a。只是m.*p1是通过指针来访问的。
//hyong::*p1=3;*p1=3; //错误，虽然p1是指向类成员变量a的指针，但不能直接使用该指针对该指针指向的成员变量进行访问，只能使用.*和->*运算符才能访问类中的成员变量。
//cout<<hyong::*p1;  cout<<*p1;  //错误，和以上情况相同，只能通过.*或者->*运算符来访问类成员变量。
m.*p1=2; cout<<m.*p1<<endl; //把类成员指针指向的类的成员变量a的值赋2，并输出其值。

cout<<(m.*p2)()<<m.f()<<"\n"; //访问成员函数
p3=&m;    cout<<p3->*p1<<(p3->*p2)()<<"\n"; //用->*运算符访问对象中的成员，->*运算符左侧必须是类类型的指针，右侧必须是该类类型的成员指针。
cout<<p3->a<<(p3->f)()<<"\n";}
```

10．this 指针：this 指针是所有成员函数的隐含指针，每次调用成员函数时，this 指针就指向调用此函数的对象。可以在成员函数类 部使用显使用this 指针。友元函数不是类的成员函数，所以友元函数没有 this 指针。静态成员函数也没有 this 指针。this 指针默认是* const this 类型，即 this 是一个常量指针，不能改变this 指针指向的地址。

例：this指针的使用

```
class hyong     {public: int a,b;      hyong(){a=1,b=2;}      hyong(int i){a=b=i;}
               hyong f(hyong &k){if (this->a<k.a) return k; else return *this;}
                          //this=&k;//错误，this指针是常量指针，不能改变this指针指向的地址。}};
int main()    { hyong m(2),n(3),l;    l=m.f(n);     cout<<l.a<<l.b;   }
```

例：对各种与类有关的指针的综合使用，整个程序有12个输出语句，总共输出12个2。

```
class hyong  {public: int a,b,*c;  hyong(){a=1,b=2;}    hyong(int i){a=b=i;c=&a;}};
//记住一条定律：点运算符的左边必须是对象，而箭头->运算符的左边必须是一个指针。
int main()
{ hyong m(2),n(3),*p=&m;     int *pr=&m.a;    int hyong::*p1=&hyong::a;
```

//指向对象的指针p的调用方法

```
cout<<p->a;   //用->箭头运算符调用成员变量a。
cout<<(*p).a;  //用.运算符调用成员变量a。点运算符要求左边必须是对象，而*p就代表指针指向的对象。
```

//调用类中的指针变量的方法

```
cout<<*(m.c)<<*m.c;  //注意调用方法，m.c表示调用对象m中的指针成员c的地址，加上*则表示调用对象m中的指针成员c的值。点运算符的优先级高于*指针运算符，所以可用也可不用括号，点运算符的左边是对象m
//cout<<*p.a<<*(p.a); //错误,因为p是指向对象的指针,点的运算符高于*运算符，所以点运算符的左边是一个指针p不是对象而发生错误。
```

//指向对象中的成员的指针的用法

```
cout<<*pr;  //pr只是一个整型指针，他指向的是对象中的成员变量a的地址。
//cout<<(*pr).a<<*pr.a<<*(pr.a)<<pr.a;  //错误，pr不是一个对象，他只是一个指向对象中的成员变量的整型指针
```

//使用指向对象的指针调用对象中的指针成员的方法

```
cout<<*(*p).c<<*((*p).c); //注意点运算符优先级高于*运算符，*p必须括起来,*p表示的是指针p所指向的对象,(*p).c就表示调用p指向对象的指针成员c的地十，所以最后要用*来表示指针c的值，即*((*p).c)就表示p所指向的对象中的指针成员变量c的值。
cout<<*p->c<<*(p->c);  //使用简头运算符调用对象中的指针成员c的方法，因为->运算符要求左边必须是指针，所以p->c就表示调用指针p所指向的指针成员c的地址，*(p->c)就表示调谓调用的指针p所指向的对象的指针成员变量c的值。
```

//类类型的指针的调用方法

```
cout<<m.*pl;  //.*运算符左侧必须要是对象名，右侧必须要是一个类类型的指针
cout<<p->*pl;  //->*运算符左侧必须要求是一个指针，右侧也必须要求是一个类类型的指针
cout<<(*p).*pl;} // 因为.*左侧必须要求是一个对象名，所以(*p)表示的就是指针 p 所指向的对象。
```

11、嵌套类：即类中的类。

11.1、 **嵌套的类的声明：**比如 class A{public: class B;}即在类 A 中声明了一个类 B。

11.2、 **嵌套类和外围类是两个互相独立的类：**也就是说嵌套类中的成员属于嵌套类外围类不知道这个成员，外围类的成员属于外围类而嵌套类不知道外围类的成员。如果要在 嵌套类中访问外围类的成员则必须以外围类的指针，引用或对象的 形式访问外围类中的成员。同样，如果要在 外围类中访问嵌套类中的成员时也必须以嵌套类的指针，引用或对象的 形式来访问嵌套类中的成员。比如 class A{public: int a; class B{public:int b; void gb(){a=3}};};就是错误的，因为外围类的成员在嵌套类中是未知的，正确的方法为 class A{public:int a;class B{public:int b; void gb(){A m;m.a=3}};};即在 嵌套类中声明一个对象 m 再用这个对象对访问 外围类的成员

11.3、 **嵌套的类的定义：**嵌套的类即可以在外围类中定义也可以在外围类外定义，比如 class A{public: class B{};};即在外围类 A 中定义了一个什么也不做的嵌套的类 B。而要在 外围类A 的外面定义嵌套类 B 的方法为：class A::B{};在外围类外定义嵌套类时要在嵌套类的前面使用作用域解析运算符以指出这个嵌套类来是外围类 A，如果不指 定就是重新定义一个新的同名类 B。

11.4、 **在嵌套类外定义嵌套类中的成员：不能在外围类中定义嵌套类的成员，**因为外围类和嵌套类是互相独立的。要在嵌套类外定义嵌套类的成员必须在外围类外定义。且必须使用作用域解析运算符以指定这个成员是哪个外围类的，比如嵌套类 B 中有成员 void g()则在外围类 A 外定义的方法为：void A::B::g(){}第一个作用域解析运算符指出嵌套类 B 的外围类是 A，第二个作用域解析运算符指出函数 g()是嵌套类 B 的成员函数。

11.5、 **在嵌套类中可以直接访问外围类中的静态成员，类型名和枚举成员。**即在嵌套类中访问这些外围类的这些成员时可以不 使用作用域解析运算符，当然也可以使用。而在外围类的外面访问这些成员时必须使用作用域解析运算符。

11.6、**外围类的对象同样不能直接访问嵌套类中的成员，但可以用作用域解析运算符来访问，这条规则只适合于公有成员变量，对成员函数则是错误的，原因在下面说明。**比如有 外围类A 和外围类 A 的成员变量 a，嵌套类 B，和嵌套类 B 中的公有成员变量 b，在 main 函数中有语句 A m;m.b=3 则是错误的，而 m.B::b=3;就是正确的访问方式。但反过来用嵌套类 B 的对象访问 外围类B 中的成员 就不行了，因为 嵌套类不知道外围类中的成员，比如 main 函数中有语句 B n; n.A::a=2;就是错误的，因为 嵌套类B 不知道外围类 A 的存在，所以是错误的。

11.7、当使用上面的方法使用外围类访问嵌套类的非静态成员函数时，因为外围类和嵌套类的非静态成员函数都有隐藏的 this 指针，而外围类的 this 指针指向 外围类的对象，嵌套类的 this 指针指向 嵌套类的对象，这时 就存在 this 指针指向不同的对象的问题。比如 A m; m.B::g();这时 就会出现外围类的 this 指针无法 转换为嵌套类的this 指针的错误，所以不能 使用上面的方法访问嵌套类中的成员函数。

11.8、访问控制权限同样适用于嵌套类，当嵌套类为公有，私有和保护的访问权限时，嵌套类同样遵守成员变量的私有，公有和保护的特性。

11.9 **嵌套类的构造函数和析构函数：**在创建一个外围类的对象时先执行嵌套类的构造函数然后再执行外围类的构造函数，析构函数则以相反的方式执行。对于嵌套类只要知道构造函数和析构函数的执行顺序就行了，其余的问题不做介绍。

11.10 创建嵌套类的对象的方法，比如类 B 是类 A 的嵌套类，则创建类 B 的对象的方法为 A::B mab;这时只会调用 嵌套类中的构造函数，而不会调用 外围类A 的构造函数，因为这里只 创建了嵌套类B 的对象。

嵌套类举例：

```
class A{ class D{public:int dd;};//声明一个私有的嵌套类D
 public: int a,b; static int e; const static int ee=0;
//定义一个嵌套类B开始
 class B{public:int c,d;
   //在嵌套类中访问外围类的成员的方法
   void gb(){c=9;
  //a=1;//错误，在嵌套类B中不能访问外围类A中的成员。因为嵌套类的成员和外围类的成员是互相独立的。
  //A::a=1;//错误，也不能以这种方式来访问外围类A中的成员a，因为a是非静态成员。
```

```
  A m; //要访问外围类中的成员，必须以外围类的指针或者对象的方式访问。
  m.a=1;cout<<"gb() A::m.a="<<m.a<<endl;
  m.ga();}
  void gb1(); };//在嵌套类B中声明一个在外围类A定义的函数。
```

//定义嵌套类B结束

```
// void B::g1(){cout<<""<<endl;} //错误，不能在外围类A中定义嵌套类B的成员。只能在外围类A的外面定义。因为嵌套类的成员和外围类的成员是互相独立的。
 B n;//可以在外围类中声明嵌套类的对象。
//在外围类中访问嵌套类的成员的方法
 void ga(){a=9;
     //c=2; //错误，外围类A不能访问嵌套类B中的成员。因为嵌套类的成员和外围类的成员是互相独立的。
    //B::c=2; //错误，不能以这种方式来访问嵌套类中的成员，因为成员c不是静态成员。
  n.c=3; cout<<"gc() B::n.c="<<n.c<<endl; //正确，要在外围类中访问嵌套类中的成员，必须以嵌套类的指针或对象的方式访问。
 D md; md.dd=44; cout<<"md.dd="<<md.dd<<endl; } //正确，私有嵌套的类可以在外围类中使用。

class C; }; //在类A中声明嵌套类C，以便在类A的外面定义嵌套类C。
```

//在外围类外定义嵌套类的方法及访问外围类的静态成员

```
int A::e=0; //被始化外围类A中的静态成员e
class A::C //在外围类的外部定义嵌套类的方法，在外围类外定义嵌套类应使用作用域解析运算符以指定嵌套类C来自外围类A。
{public: void gc(){e=3;cout<<A::e<<ee<<A::ee<<endl;}}; //可以在嵌套类中直接该问外围类中的静态成员，静态常量成员，类型名和枚举成员，也可以在嵌套类中加上作用域解析运算符。在外围类外访问这些成员就必须要使用作用域解析符。
void A::B::gb1(){cout<<"gb1()"<<endl;} //在外围类的外部定义嵌套类中的成员必须要使用作用域解析符以指定嵌套类的外围类是A。
```

//main函数开始

```
int main()
{A ma;  //声明一个外围类的对象。构造函数的执行顺序是先构造嵌套类然后才是外围类  析构函数则按相反的顺序执行。构造函数的例子就不举例了，只要知道构造的先后顺序就行了
//ma.c=3;ma.gb(); //错误，成员c和gb()不是类A的成员，而是嵌套类B的成员。
ma.n.c=33; cout<<"class A ma.n.c="<<ma.n.c<<endl; //通过在外围类中声明的嵌套类B的对象n来访问嵌套类B中的成员。
ma.n.gb();
ma.B::c=333; cout<<"ma.B::c="<<ma.B::c<<endl; //正确，可以用作用域解析符以指定成员c是来自嵌套类的。
//ma.B::gb(); //错误，嵌套类B和外围类A的非静态成员函数的this指针指向不同的对象，在这里就不能把指向外围类A对象的this指针转换为指向嵌套类B对象的this指针，所以出现错误。

A::B mb; //外围类外声明一个嵌套类的对象的方法。在这里必须要在嵌套类B的前面用作用域解析操作符指出嵌套类B来自外围类A。这里只执行嵌套类的构造函数，不执行外围类的构造函数。因为只创建了嵌套类的对象。
mb.gb1();
//mb.gc(); //错误，gc()是外围类A的成员，不是嵌套类B的成员。
//mb.A::a=11;  //错误，外围类的对象可以这样访问嵌套类中的成员变量，但反过来就不行了。因为B是嵌套在类A中的，所以在这里嵌套类B并不知道有类A的存在，在这里就会认类外围类A不是嵌套类B的成员的错误。
//A::D md; //错误，嵌套类D是私有的，不能在外围类的外面声明私有的嵌套类的对象。
}
```

12、友元

1、声明友元的目的是让类的非成员函数或者类能访问该类对象的私有和受保护成员，一但把函数或者类声明为友元那么就可以访问该类对象的私有和受保护的成员，注意是能访问类对象的 私有和受保护成员不是类的私有和受保护成员比如class A{int a; pulbic:friend void g();} 则void g(){a=2;}就是错误的因为友元函数不能直接访问类中的私有成员只能通过类的对象来访问 私有成员 即 void g(){A m; m.a=2;}就是正确的，通过类 A 的对象 m 来访问类中的 私有成员。对于类的友元类也是同样的情况。

2、friend 出现的位置对友元来说无关紧要。即把友元声明在公有私有和受保护的位置都是一样的。

3、怎样在类中定义友元函数或友元类：要在类中 定义友元函数或者友元类那么该类或者函数必须在外围类外进行声明，注意是声明不是 定义，声明的 位置无关紧要，只要在 使用之前声明一下就行，如果在 全局范围类声明了友元函数则函数就可以在全局范围类使用，如果在 局部声明则只能在局部使用。如果既在类中定义友元函数或友元类，又在类外定义，则会 出现二重定义的错误。如果只在类中 定义友元函数或友元类而在类外不进行声明 则将 出现无法访问到该友元的情况，当然该程序能通过编译，只是无法访问 友元而已。比如有类 class A{friend void g(){cout<<”g”<<endl;}如果不在类 A 外再声明一下友元函数 g，则程序中将无法调用友元函数 g，既 A m; m.g(); A::g()或者 g();这些调用都是错误的，前两个调用是因为 g 不是类 A 的成员，后一个调用会出现无法找到标识符 g 的错误。如果再在类 外定义g 函数，比如在类 外这样定义void g(){cout<<”g1”<<endl;}则会 出现重复的 g 函数的 定义的错误。要正确使用在类中定义的友元函数或友元类就必须在类 A 的外面重新声明一下函数 g，比如 void g();声明而不 定义，这样的 话程序就只须直接调用该函数了，即语法 g();将是正 确的调用方法。

//友元示例

```
class A{int a;
 friend class B;  //friend出现的位置对友元来说无关紧要。
 public: friend void g();
 friend void f(){cout<<”f”<<endl;}}; //在类中定义友元函数
```

```
void g(){//a=2;//错误，友元不能直接访问类中的私有成员，只能访问该类的对象的私有成员。
    A m; m.a=2;cout<<m.a<<endl; //正确，要使用友元访问类中的私有成员应通过对象来访问。
     void f();  f();}  //正确，要访问在类中定义的友元函数f，则只须在使用之前先进行声明即可，声明的位置无关紧要。
class B{public:void gb(){//a=3;//错误，友元不能直接访问类中的私有成员，只能访问该类的对象的私有成员。
    A m; m.a=3;cout<<m.a<<endl;}};//正确，要使用友元访问类中的私有成员应通过对象来访问。

int main()  {g();  B n; n.gb();  void f(); f();}//访问类中定义的友元的示例，使用之前要进行声明，如果在全局范围类声明了友元函数f()则函数f()就可以在全局范围类使用。如果在局部声明则只能在局部使用。
```

作者：黄邦勇帅