类与对象

Andrew Huang<bluedrum@163.com>

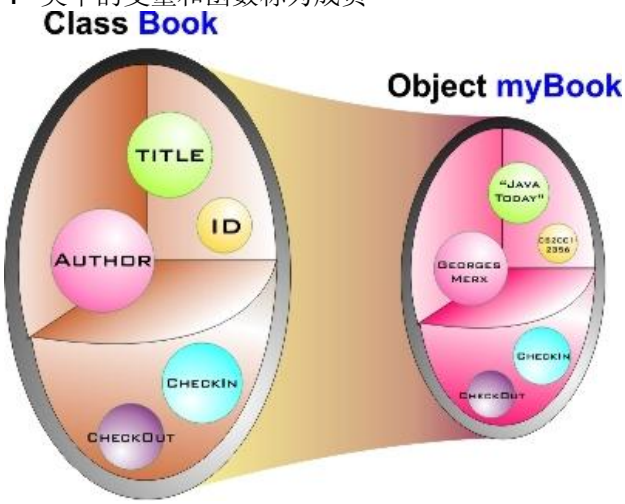
课堂练习

类的概念

- 在现实中，类是对一组客观对象的**抽象**，它将该组对象所具有的共同特征（包括属性和服务）**封装**起来，以说明该组对象的能力和性质。
- 在系统中，**类**（class）是一种用户自定义的数据类型。通过类使得现实中的抽象实体在程序中直接表示为一个标识符，并可以进行引用和操作。
- 这使得程序中的概念与应用中的概念相互比较一致和对应。

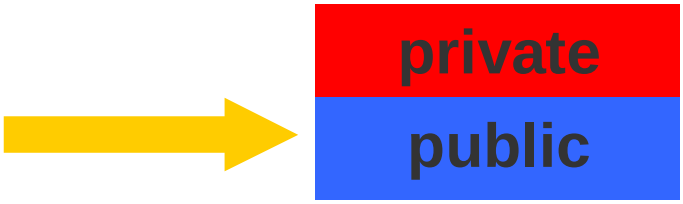
类 术语

- 定义数据的类型以及操作这些数据的函数
- 类的实例称为对象
- 类中的变量和函数称为成员



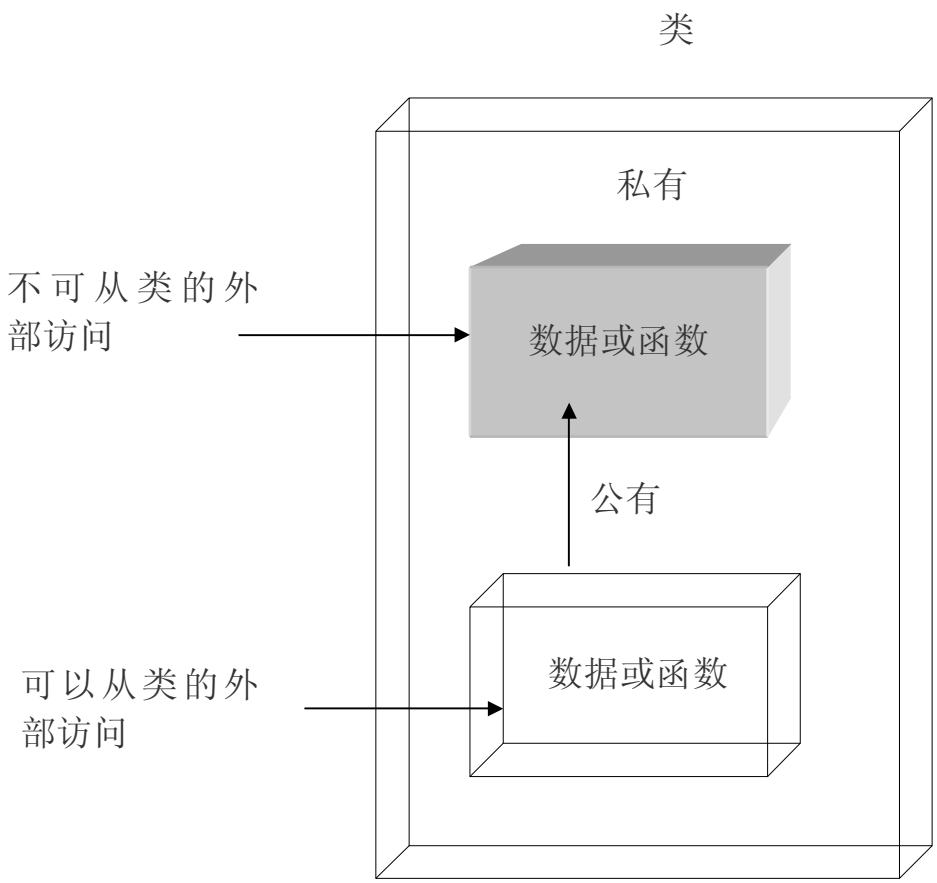
私有和公 有 2-1

- 类成员可以在类的公有或者私有部分声明
- 数据成员通常在私有部分中声明
- 在公有部分中声明的成员可以被该类外部的函数访问
- 公有部分构成类的接口



私有和公 有 2-2

- 无法从类的外部访问私有数据
- 其他类的私有数据对于当前类也是隐藏的



类定义的格式

类的定义格式一般分为**说明部分**和**实现部分**。

说明部分：是用来说明类中的成员，包含**数据成员**的说明和**成员函数**的说明。成员函数是用来对数据成员进行操作的，也称为“方法”。

实现部分：是用来对成员函数进行定义。

说明部分的一般格式为：

```
class 类类型名{
        private:
            私有成员说明 /*私有访问权限，为默认值，可缺省， 除本类中的
                          成员函数及友元外，其他类外函数不能访问*/
        public:
            公有成员说明 /*公有访问权限， 程序中的任何函数都可访问*/
        };
```

类实例

- 例如：定义一个person类，设person具有的特征：属性（姓名、年龄、性别），方法（输出一个人的属性）；则对应着有4个成员：**三个数据成员，一个成员函数**。这些数据成员一般不能由外界直接访问，随意修改。而只能通过成员函数进行访问。所以三个数据成员定义为**私有成员**，print成员函数定义为**公有成员**。则有：

```
class CPerson {                    //类名通常用C字母开始，以区别其他标识符
        private:
```

```
        char name[10];  //数据成员的说明形式与变量的定义形式相似
        int age;
        char sex;
    public:
        void print( );  //成员函数的说明形式与一般函数的说明形式一致
};
```

类实例(2)

- **类的实现部分**即定义它的成员函数，方式与定义普通函数大体相同。
- 若在类说明外部定义成员函数，则应使用**作用域限定符**::指明该函数是哪个类中的成员函数。
- 格式为：
  - **类型 类名::成员函数名（参数表）{函数体}**
  - **如：**

```
void CPerson::print( )
   {  cout<<name<<age<<sex<< endl; }
```

- 除特殊指明外，成员函数操作的是**同一对象**中的数据成员。其中如name等。
- 若在成员函数中调用非成员函数（没有类名的函数；全局函数），则可用**不带类名的**::来表示。

内联函数

当成员函数的规模较小时，语句只有 1-5 行，符合内联函数条件，则可**在类中定义成员函数**，**成为内联成员函数**。

如：

```
class CDate {
      public:
            void Set(int m, int d, int y)      // 置日期值
              {
              month=m; day=d; year=y;
               }
            void Print( );                          // 打印输出
    private:
        int month;
        int day;
        int year;
          };
```

类定义注意事项

- 1、在类体中不允许对所定义的数据成员进行初始化。

```
class TDate
 {        public:¡-
           private:
                int year(1998),month(4);
                ¡-
  };
```

- 2、类中的数据成员的类型可以为任意的。
- 3、习惯将类的定义代码放到一个头文件中,以后若要使用则用文件包含命令包含。
- 4、成员名与方法名不能重名.

一个类实例

如定义一个类：

```
class CPoint
 {     public:
              void SetPoint(int x, int y)    // 置坐标值，内联成员函数的定义
                {
                       X=x; Y=y;      //给数据成员赋值
                         }
              int Xcoord( ) {return X;}   //提供X坐标值
              int Ycoord( ) {return Y;}   //提供Y坐标值
              void Move(int xOffset,int yOffset ); // 移动点
      private:
          int X; //私有数据成员的定义
          int Y;
   };
void Cpoint::Move(int xOffset,int yOffset)
{X+=xOffset;  Y+=yOffset;  }
```

对象的定义,创建

对象定义格式

- 定义了一个类只是定义了一种类型，它并不分配空间，不能进行操作。只有用它创建对象后，系统才为对象分配存储空间。
  - 对象是类的实例。（对象是一种广义变量？）
  - 类可以用sizeof()求大小
- 定义对象之前要先定义好类。
- 对象定义格式为：类类型名 对象名表；如：

```
student s,t;      //定义两个student类的对象s和t
Cdate date1;    //定义一个Tdate类的对象date1
Cdate *Pdate=&date1;
 //定义一个指向date1对象空间的指针Pdate
Cdate date[3];
//定义一个data对象数组，其元素类型为Cdate
```

对象成员的表示方法

- 通过对象可以访问类中的**公有类型数据和成员函数**。其使用方式为：
  - 对象名 . 成员函数名
  - 对象名 . 数据成员

  例：date1.year=2002;date1.Print( );
- 通过指向类类型对象的指针访问类的公有数据成员和成员函数时。其使用方式为：
  - 指针名->数据成员 或 指针名->成员函数

  例：Cdate date1;

```
    Cdate *Pdate=&date1; // 指针指向对象date1
   Pdate1->day=30;Pdate1->Print( );
```

类和对象关系如同整型 int 与整型变量 i 之间的关系。

注意：

在成员函数中访问成员无须加对象名来作为前缀。

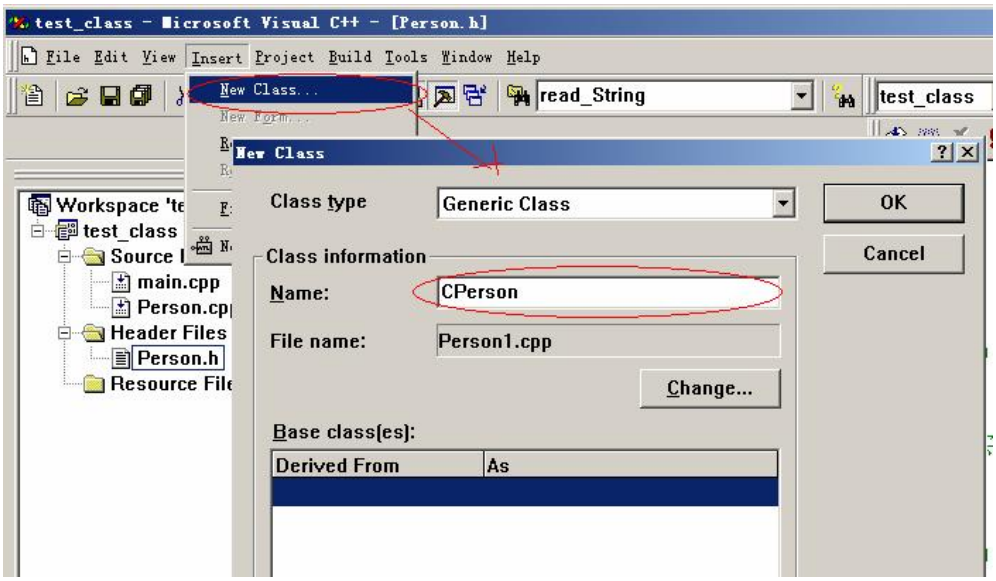
如：

```
void Cperson::print( )//自家人不客气
{
  cout<<name<<age<<sex<<endl;
}
```

- 在 name等成员之前不能加对象名，**在定义类时，成员函数是所有对象共享的代码，不被某一个对象独占**。也无法确定类的对象名。

VC++ 快速新增类方法

- 用class wizard ,自动生成类框架,包括头文件,源码和构造和析构函数



this 指针

一个类中所有对象调用的成员函数都是在同一代码段。那么成员函数如何区分调用的成员来自哪个对象呢？

原来在对象调用 s.set("li",23,'m');时，成员函数除了接受 3 个实参外，还接受了一个对象 s 的地址。

如：

CDate::Set(int m, int d, int y)

CDate::Set(CDate *this, int m, int d, int y)

这个地址被一个隐含的形参 this 指针所获取，等同于执行了 this=&s。所有对数据成员的访问都隐含加上了前缀 this->。

所以：

age=a;　　this->age=a;　　s.age=a;

this 指针

```
class person{
private:
        int age;
public:
        void display();
};
void Person::display(){
        this -> age = 25;          // 与 age = 25 一样
        cout << this -> age;      // 与 cout << age 一样
        cout << endl;
};
int main(){
        Person Jack;
        Jack.display();
        return 0;
}
```

用new 创建对象

- new操作符用于为类的对象创建内存空间
- 如果成功，返回指向所分配内存空间的指针
  - data_type * pointer_variable = new data_type
- 内存空间不足或者检测到错误，返回空
- new类似于C语言中所使用的malloc()函数

```
int *p;                    //指向 int 类型的指针
        float *f;          //指向 float 类型的指针
        p = new int;                //为 int 类型分配内存
        f = new float;     //为 float 类型分配内存
```

用new 创建对象(2)

- new最大作用是动态创建对象,它在分配空间后,自动调用对象构造函数

```
Student *stu_ptr; //指向 Student 类型对象的指针
        stu_ptr = new Student; //指向新的 Student 对象
```

delete

- delete显式销毁由new创建的对象
- 使用完内存后，使用delete将其释放
- 不要使用指向已经释放的内存的指针

```
int *ptr;
        ptr = new int;
        *ptr = 12;
        cout << *ptr;
        delete ptr;
```

delete

- 分配由变长数组组成的内存块
  使用new分配对象数组，必须在delete语句中使用[ ]
- 使用delete删除由malloc分配的内存是错误的
- 使用free释放由new分配的内存也是错误的

```
int *ptr;
                ptr = new int[100];
                delete [] ptr;
```

new/delete 常犯的错误

- new 即可以分配单个对象，也可以分配数组.
- 分配对象时可以带参数
  - obj = new a_class(12);
- 同时分配数组也可以带数字
  - ary = new char[12]; // 分配12 个char 大小的数组
- 因此最常见笔误会造成一些严重的错误
  - Ary = new char(12);
  - 这个错误不会报编译错误，因为表示实际只分配一个char 空间，并且这个char 的初值为12
  - 但实际上是本意是 Ary =new char[12]，即要分配12 个char 空间，用上面错误的定义(Ary = new char(12)) 生成的空间，对第2 个char 空间进行写操作，必然造成地址被破坏，在delete 时会报错.

对象的初始化

构造函数

- 构造函数和析构函数是在类体中说明的两种特殊的成员函数。
- 构造函数的功能是在创建对象时给对象分配内存空间 ，并可使用给定值来初始化对象。
- 一个对象可以有多个构造函数，
- 构造函数名必须跟类名一样,并且没有返回类型
- 默认构造函数是不带任何参数的构造函数

```
class CDate{
    int month, day, year;
public:
    CDate()                          //默认构造函数
    {day=1; month=1; year=1999;}
    CDate(int x)                     //仅指定日
    {day=x; month=1; year=1999;}
    CDate(int x, int y, int z) //指定年月日
    {day=x; month=y; year=z;}
};
```

析构函数

- 析构函数的功能是用来释放一个对象的已分配空间。并可在对象被清除前，完成一些清理工作。
- 构造函数与析构函数的功能正好是对应的。

- 在对象销毁时自动调用的成员函数
- 编译器会生成对析构函数的调用
- 与类同名，但是在函数名前有个波浪号 (~)
- 析构函数没有返回类型，也不带参数

```
class username {
        public:
          ~username();
        //析构函数
        };
```

构造函数和析构函数要求

- 对前面的日期类进行修改，并将定义存放在Cdate1.h文件中。

如：

```
class CDate1
 {    public:
                CDate1(int y,int m,int d);/*  函数名与类名一致，无类型但可有参数，  构造函数的说明*/
                virtual ~CDate1( );/*  名字与类名一致，前有~，无类型无参数，  析构函数的说明*/
                void Print( ); //一般成员函数的说明
      private:
                int year,month,day;
  };
CDate1::CDate1(int y,int m,int d) //构造函数的定义
 {
        year=y;month=m;day=d;
        cout<<"Constructor called.\n";
    }
CDate1::~CDate1( ) //析构函数的定义
   {
        cout<<"Destructor called.\n";
      }
void CDate1::Print( ) //一般函数的定义
    {cout<<year<<"."<<month<<"."<<day<<endl;}
```

与其他成员函数定义一样，两种函数的定义可放在类体内，也可放在类体外

- ***构造函数***的特点有：
  程序中不能直接调用构造函数，在创建对象时**系统自动调用**构造函数
  构造函数**可以重载**，即可定义多个参数个数不同的函数
  构造函数有**隐含的返回值**，并由系统内部使用
- ***析构函数***的特点有：
  一个类中只可能定义一个析构函数，即**不能重载**
  析构函数可以被调用，也可系统调用
  被自动调用的情况有两种：
  在一个函数体内定义的一个对象，当**函数结束**时
  用 **new** 运算符动态创建的一个对象，在**使用 delete 释放**时

```
#include <iostream.h>
#include  “Cdate1.h" //将含有类定义的头文件包含进来
void main( )
```

```
{
    CDate1 today(2005,4,1),tomorrow(2005,4,2);// 对象定义时，自动调用构造函数
     cout<<"today is";
     today.Print( );
     cout<<"tomorrow is";
     tomorrow.Print( );
  }                                    //函数结束时，自动调用析构
```

执行程序后，输出结果为：

```
Constructor Called.
Constructor Called.
today is 2005.4.1
tomorrow is 2005.4.2
Destructor called.
Destructor called.
```

缺省构造函数和缺省析构函数

函数体内无代码，也可自定义一个无参的构造函数来替代缺省

在类定义中可以不定义任何构造函数，这时编译器会自动生成一个不带参数的缺省构造函数。

其格式为：

<类名>::<缺省构造函数名>（）

{}（何时不能缺省？）

则在程序中定义一个对象而未指明初始化时，编译器便按缺省构造函数来初始化该对象。

这也是前面的程序中可以没有构造和析构函数的原因

同理，若一个类中未定义析构函数，编译系统也生成一个缺省析构函数。

其格式为：

<类名>::~<缺省析构函数名>（）　　{}

如：

```
class  CExample
{      public:      //自定义无参构造函数
                CExample ( ) ;
                CExample ( int   i ) ;//   重载构造函数
                CExample ( int   x , int   y ) ;
       private :
                int   member1, member2 ;
};
CExample :: CExample ( )
 {   member1 = 0;   member2 = 0;       }
CExample :: CExample ( int i )     //CExample ( int   i=8 ) ;可否？
 { member1 = i ;   member2 = 0 ;     }
CExample :: CExample ( int x , int y )
  {
      member1 = x ; member2 = y ;
    }
void main( )
{
      CExample     ex1; //不带参数创建对象，调用无参构造函数
      CExample     ex2(1);//   对应调用相应的构造函数
      CExample     ex3( 2, 3 );
        …
   }
```

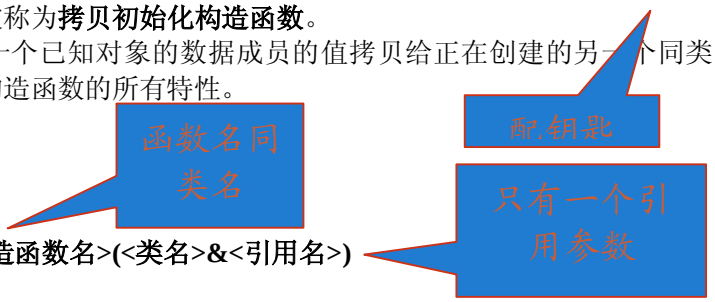
注：若有构造函数的重载，则不会生成缺省构造函数，须自定义无参构造函数

拷贝初始化构造函数

构造函数除可以用**基本数据类型**初始化对象外，还可以使用已存在的**同类型的对象**即**类类型的对象**来初始化正在创建的对象。为此，必须在类中定义一个**特殊的构造函数**来完成这个工作，这个构造函数被称为**拷贝初始化构造函数**。

它实现了在初始化时将一个已知对象的数据成员的值拷贝给正在创建的另一个同类的对象。当然，它具有一般构造函数的所有特性。

格式为：



<类名>::<拷贝初始化构造函数名>(<类名>&<引用名>)

{…}

如：**student::student(student & s) {…}**

每个类中**必须有一个**。若类中未说明，则编译系统会**自动生成缺省函数**。

例：

```
class    CMyClass
           {
              public:
                   CMyClass ( int    x = 0 ) ;//    带缺省值的构造函数
                   CMyClass ( CMyClass &    c ) ;//    拷贝初始化构造函数
                    // … …
               private:
                    int    member ;
            };
CMyClass :: CMyClass ( int    x )
     {
              member = x ;
        }
CMyClass :: CMyClass ( CMyClass &    c )
     {
         member = c . member ;
       }
```

拷贝初始化构造函数被自动调用有三种情况：

一、是用一个已知对象**初始化一个新对象**时。

二、是以值调用方式向一个函数**传递对象参数**时。

如：

```
void    fun ( CMyClass    a ) ;
void    main( )
       {
          CMyClass    a;//    调用带缺省值的构造函数
          CMyClass     b(3);//    调用带缺省值的构造函数
          CMyClass     c(b);//    调用拷贝初始化构造函数
          …
          fun ( c ) ;     //调用拷贝初始化构造函数
          }
```

结果为：

```
a.member:0
b.member:3
c.member:3
```

fun 中的形参 a 中的数据成员 a.member:3

- 三、当对象作为**函数返回值**时。

如：

```
student fn( )
  {student ms("randy"); //创建一个ms对象
    return ms;    //将ms对象作为返回值
   }
void main( )
  {student s; //创建一个s对象
    s=fn( );
    ¡-
     }
```

- 则返回的ms对象将产生一个**临时对象**。系统调用拷贝构造函数将ms拷贝到新创建的临时对象中。当fn（）返回时产生的临时对象拷贝给s后，临时对象被析构。

默认的拷贝构造函数问题

- 如果在类中没有实现拷贝构造函数,编译器会自动加入一个.其动作把所有成员的值复制到新的对象里.
- 如果数据成员有动态分配对象,只会把其地址复制新对象里,这样两个对象实际在引用同一块地址.如果两个对象都是释放这一个块空间,第二个对象就会出现内存错误.
- **原则:如果对象有动态分配空间,一定要手动编写拷贝构造函数.自己在里面作重新作内存拷贝**

类型的引用

C++ 的类型引用

- 在C++引入一种特殊的语法,即类型引用.即在一个类型后面加入&,表示对类型实例的引用.
  - int i=100; int &a = i; cout <<a
  - class a_class; a_class obj; a_class &b=obj;
  - 引用相当于是原有类型的别名,不能被置空(NULL)
- **C没有对应机制**
- 类型引用有如下特点
  - C++中,任何一个类型都能被引用.包括基本类型,结构和类.
  - 类型引用并没创建实际空间,而是指向被引用类型实例的空间,从这一点来说,它的行为象指针
  - 从引用的方法来说,跟被引用类型使用方法一致.如一个引用的结构和对方,采用.来取存取结构和对象成员,从这一点看引用又象普通类型变量
  - 引用的本质更偏向于指针,但是比指针更安全.如一个指向结构的指针,可以被修改向结构内部进行修改,但是引用是禁止做这样的操作.因此可以把引用看成一个安全指针类型.
  - 因此任何使用引用的场和,指针都能被使用.
  - 但用new创建的空间,必须是指针,无法使用引用

引用在函数中的应用.

- 在函数调用中，对象作实参如果是值传递的方法，将会在堆栈中完全创建一个与实参一规一样的中间对象．这样做有如下缺点
  - 效率低下，因为是实际上创建一个多余的中间对象进行操作．
  - 在函数对中间对象(形参) 的修改不会反应到外面对象之间
- 而且引用对象作形参则很好的解决这两个问题

引用在函数参数中的应用．

- 以下是一个常见的错识,在实现在>>重载时后,调用结束显示发现date1并未改变值
- 这里主要问题是在函数调用时,采用值传递,operator>>会在堆栈中创建一个值跟date1一模一样的中间对象date,从流中输入结果只会反应到date中,但退出函数后,date被销毁,用户输入被丢弃．
- 这里需要改为 istream &operator >>(istream& is,CDate &date),这样date实际只是指向date1一个引用,只创建一个对象指针空间．效率高,而且输入内容也会反应到date1中

```
istream &operator >>(istream& is,CDate date)
{
    is >>date.year>>date.month>>date.day;

    return is;
}

CDate date1;
cin >> date1 ;
cout <<date1;
```

引用的函数返回值的应用

- 这一个是常见问题,在返回值写的是对象本身,这样在return os ;时,将会在堆栈中临时创建一个中间对象
- 在当下句执行到<<1时,这个对象赋给了<<1,然后这个中间对象被销毁,运行效率相当低下．
- 当把函数定义改变成ostream &operator <<(ostream& os,CDate &date),在整个<<链运行只是一个引用,并未完全创建新的中间对象

```
ostream operator <<(ostream& os,CDate &date)
{
        os                                                      <<
"year:"<<date.year<<",month:"<<date.month<<",day:"<<date.d
ay; //通常不加回车符,由外界自行判断
        return os;
}

CDate date1;
cout <<1<<date1;
```

引用的函数返回值的应用

- 关于引用一个危险的用法，这一对象使用了局域对象tmp，当返回对象引用时,date1

只在调用一次才有效，然后tmp 对象立即被销毁.

```
CDate &CDate::get(){
      CDate tmp = *this;
          return tmp;
}

CDate &date,date1;
date = date.get();
date.print(); //崩溃,中间对象已经销毁.
```

引用的函数返回值的应用

- 如果的返回值是一个对象,这样在return的一瞬间,会立刻创建一个中间对象,以便能返回给调用者.
- 下例中的get()方法里将会创造二个中间对象,首先是用自行定义的tmp,然后在return tmp时系统在堆栈中再创建了一个自动中间对象,并且值等于tmp,
- 函数调用结束时,这个自动中间对象再次把值传给date1
- 由此直接返回对象开销是相当大的

```
CDate CDate::get(){
      CDate tmp = *this;
          return tmp;
}

CDate date,date1;
Date1=date.get(); //做了三次赋值操作
```

静态数据

静态数据

- 静态数据成员
- 静态成员函数

静态成员

- 在C 中，在一个函数里的静态局域变量的值会被这个函数所有调用共享
- C++ 的对象静态成员也有同样作用，为 同一个类的所有对象共享.
- 静态成员在所有类里只有一份拷贝
- 只在类的内部可见
- 生命周期贯穿整个程序

静态成员实例

```
class race_cars{
private:
        static int Count;
        int car_number;
        char name[30];
public:
        race_cars(){count++;}   //构造函数，用于增加 count
        ~race_cars(){count--;} //析构函数，用于减小 count
};
int race_cars::count=3; //在 main() 之前初始化
```

关于静态数据成员的更多内容

- 如果将一个静态成员声明为类的私有成员，则非成员函数不能访问它
- 是类的全局数据
- 不是对象的一部分，没有 this 指针

静态成员函数

- 静态成员函数的定义及用法同静态数据成员相似。
- 静态成员函数属于类，而不是对象成员（无this 指针用来存放对象的地址），对静态成员的引用不需用对象名，也不能直接引用具体对象中的非静态成员。须使用（对象名 . 成员名）来引用。

如：

```
class M{public:  M(int a)  {A=a;}
                          static void f1(M m);
                 private:
                          int A; static int B;
                };
        int M::B=0;
        void M::f1(M m){cout<<m.A<<B<<endl;}
        void main( ){M p(5); M::f1(p); }
```

```
class alpha
{
private:
   static int count;          //静态数据
public:
   alpha()                    //构造函数
   {
        count++;
   }
   static void display_count() //静态成员函数
   {
        cout << count;
        cout << endl;
   }
};
```

友元

通常，类的私有成员只能由本类的成员访问，外部函数只能访问类的成员函数，再由成员函数访问类的私有成员。

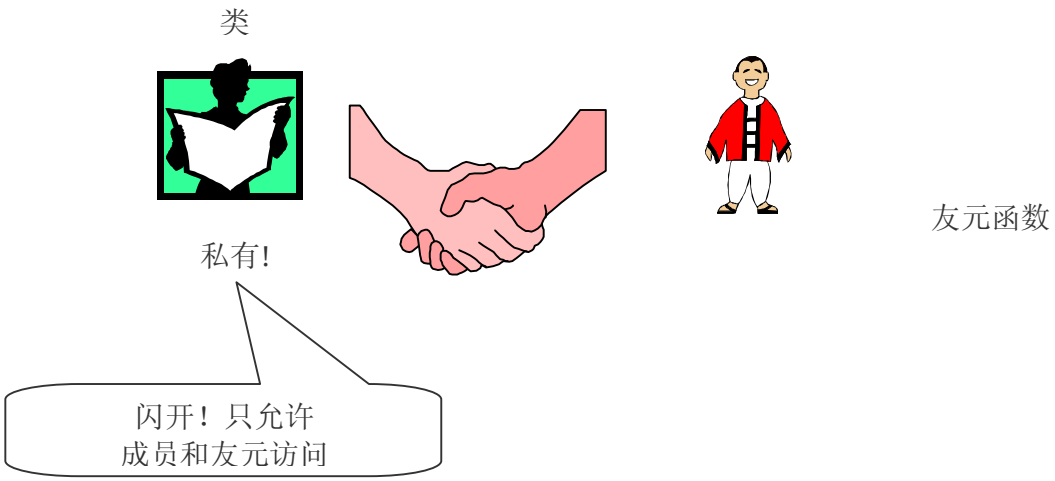
但是，如果在某个类定义中用friend说明一个外部函数后，这个外部函数就可直接访问该类的任何私有成员。则该外部函数称为这个类的友元函数。

采用友元的目的主要是为**提高效率**。显然它**破坏了类的封装性**。

注意：

在某个类中说明的**友元函数并不是该类的成员函数**。它可以是外部的一个独立函数，也可以是另外一个类中的成员函数。

友元



友元函数实例

```
class teacher
{
private:
        int a;
public:
    teacher()
    {
        a = 5;
    }
    friend int stud(teacher);
};
int stud(teacher t1)
{
    return (t1.a);     //因为 stud()是一个友元
}
```

```
class Teacher;              //前向声明
class Student {
private:
      int st_data;
public:
      void getstuddata();
      friend void display(Student abc, Teacher xyz);
};
class Teacher {
private:
      int th_data;
public:
      void getteachdata();
      friend void display(Student abc, Teacher xyz);
};
void display(Student abc, Teacher xyz) {
      //某些代码
}
```

类只有在声明之后才能被引用

我是大家的朋友

友元函数的特性

- 拥有访问类的私有部分的权限
- 没有 this 指针
- 可以置于类说明的私有或公有部分
- 定义友元函数时，不需要使用类名和作用域解析操作符作为前缀

优点

- 在接口设计的选择方面提供了自由度
- 成员函数和友元函数具有同等的特权
  - 主要的区别：
    友元函数的调用方式类似于 func(xobject) ，而成员函数的调用方式类似于 object.func( )
  - 设计者可以选择最有可读性的语法

缺点

- 提高了编程的灵活性，但是违背了面向对象编程的原则
  - 在编码中对完整性的破坏可以在一定程度上加以控制
- 必须在它要访问的类中进行声明。没有源代码是无法完成的
- 需要用友元解决的问题一般都能用其它方法解决

友元类

- 即一个类作为另一个类的友元。
- 这意味着这个类的所有成员函数都是另一个类的友元函数。
- 这只需先声明而不一定需要先定义。但这种友元关系是单向的，并只在两个类之间有效。

如：

```
class X {
      friend class Y;
      public:          ¡-
       private:        ¡-
             };
class Y {¡-}
```

- //则Y中的所有成员函数都可引用X中的任何成员。

友元类实例

```
#include <iostream.h>
class beta;          //前向声明
class alpha {
private:
      int a_data;
public:
      alpha() { a_data = 10; }
      void display(beta);
};
class beta {
private:
      int b_data;
public:
      beta() { b_data = 20; }
      friend void alpha::display(beta bb);
};
```

```
#include <iostream.h>
class beta;          //前向声明
class alpha {
private:
    int data;
public:
    friend class beta;          //beta 是一个友元类
};
class beta {
public:
    void display(alpha d) {     //可以访问 alpha
        cout << d.data;
    }
    void get_data(alpha d) {    //可以访问 alpha
        int x = d.data;
    }
};
```

课堂练习

- 1. 定义一个满足如下要求的Date 类
  - a) 用下面的格式输出日期
    - 年/ 月/ 日
  - B) 可以运行在日期上加一天操作
  - c) 设置日期操作
  - D) 年，月，日可以各用一个整数表示
- 2. 将单链表或双链表代码改造成C++ 代码
  - 需要有一个node 类，和一个list 类的
  - 链表list 可以为node 的友元
  - 对链表操作均可转换为方法

操作符重载

Andrew Huang<bluedrum@163.com>

课程内容

- 操作符重载的概念
- 操作符重载函数设计
- 几种特殊的操作符
  - >>,<<
  - =
  - [] 操作符

什么是操作符重载

- C/C++ 非常多的运算符(又称为操作符)，只能对基本数据进行运算.
- 在实际编程，用户自定义的的数据结构，如struct ,class 的实例之间也是需要进行运算的.
- 在C 的中，只能通过函数来扩展自定义结构运算

```
//C 计算两个时间之差的
#include <time.h>
double difftime(time_t timer1,time_t timer0);
//实际上相当于 timer1-timer0
```

什么是操作符重载(2)

- C++ 除了通过函数方法来扩展外，有一种更为简洁的扩展方案，即把操作符赋与新的含意，以便能处理新的类型
- 相对于扩展函数，操作符重载的形式更为简洁，易懂. 并能参与连续的链式运算.

```
如 obj3= obj1+obj2;
比 obj3.addobject(obj1,obj2);简洁
```

- 最重要一点是，在不改变原有的体系和代码情况下，操作符重载提供一个直接使用和扩展原有机制
  - 例如: 任意一个类重载了<< 操作符，即可以把自己的输出无缝加入到cout 输出

操作符重载的本质

- 重载的操作符本质上也是一个函数，名字比较特殊的函数.
  - 这样设计一个操作符重载的实际上是设计一个特殊函数.
  - 如a+b 可以理解为是一个名字为 operator+(a ，b) 的函数 ，表达式结果就是这个函数的返回值
  - 所有操作符重载函数名，是保留字¡±operator¡± 加上操作符本身构成.
    - << 的重载写成 operator <<
  - 类的操作符重载，隐藏表示当前对象是表达式的第一个操作数

- Class A::operator+(B)　相当于A+B

一个操作符重载实例

```
class temp
{
private :
      int x, y;
public :
      void operator ++(void);
};
//重载++操作符,相当是实现了一个名叫 operator ++的函数
void temp::operator ++(void)
{      x++;      y++;}
//调用
temp objTmp;
objTmp++;
}
```

操作符重载函数的设计

- 操作符有一元和二元操作符.
- 重载操作符虽然类似重载函数，但最重大的区别在
  - c++ 的运算符所能操作的操作数的个数的个数是规定好的。在重载二元运算符时只能指定两个参数，而重载一元运算符时只能指定一个参数。
  - 在重载函数时，可以指定任意的返回类型，甚至可以指定void 类型作为返回类型.但c++ 的运算符是用在表达式中，一个运算符的运算结果要供别的运算符使用，因此任何运算符都指定有非void 类型的返回类型

C++ 允许重载的操作符

+	—	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	— —	<<	>>	= =	!=	&&	||
+=	— =	*=	/=	%=	^=	&=	! =
<<=	>>=	[ ]	( )	— >	— >*	new	delete

C++ 不允许重载的操作符

运算符	运算符的含义	不允许重载的原因
? :	三目运算符	在C++中没有定义一个三目运算符的语法
.	成员操作符	为保证成员操作符对成员访问的安全性，故不允许重载
*	成员指针操作符	同上
: :	作用域运算符	因该操作符左边的操作数是一个类型名，而不是一个表达式
sizeof	求字节数操作符	其操作数是一个类型名，而不是一个表达式

一元操作符重载

一元操作符的重载

- 一元运算符
  - +(正号),-(负号),!(逻辑否),~(位取反),&(取地址)
  - ++(自增),--(自减)
- 一元表达式格式为
  - < 操作符>< 表达式> 如 !a,++a,
  - < 表达式> ++/--, 如 i++,b¡ª
- 一元表达式重载实现方法
  - 1. 用不带参数类方法重载
  - 2. 用带一个参数的全局函数实现，这个函数通常是友元函数，这个方法用的比较少
- 重载的实现(@ 表达一元操作符)
  - 表达式 object @ 重载成 object对应类的方法 operator @ ()
  - 如 object ++ 重载成
- 重载函数的定义格式
  - <类型><类名>::operator <一元运算符>(形参)

自增，自减的重载

- ++,--比较特殊,因为前缀,后缀之分.如果重载不加区别,两者名称一样,无法区别
- 因此C++规定,为了区分前缀和后缀两种形式，后缀形式的自增和自减操作符接受一个额外的int型形参。使用后缀形式的操作符时，编译器提供0作为这个形参的实参，在实现操作符时可以不对这个形参命名。
- 为了与内置操作符一致，后缀形式的操作符应返回旧值，并且，作为值返回，而不是返回引用。通常，后缀形式的实现是调用前缀形式实现的。

```
Demo& Demo::operator++() {
// prefix form
    // do something here
    return *this;
  }
```

```
Demo Demo::operator++(int) { // postfix form
    Demo ret(*this);
    ++*this;
    return ret;
  }
```

自增，自减的重载

```
//原始表达式
class MyClass;
  MyClass MyObject;
++MyObject;
MyObject++;
```

```
//实现代码
class MyClass;
  MyClass MyObject;
MyClass &Myclass::operator ++(){ }
MyClass MyClass::operator ++(int){}
```

```
//显式调用 operator 函数,等价,
class MyClass;
  MyClass MyObject;
MyObject.operator ++();
MyObject.operator ++(0);
```

二元操作符重载

二元操作符重载

- C/C++ 大部分是二元操作符
  - +,-,*,/ ...
  - 表达式 c=a+b ，表示 a 与b 相加，得到中间结果，再赋值给c
- 二元操作符的重载方法
  - 二元操作符的表达式有牵涉到3 个类型，两个参加运算的参数和一个结果．实现方法有两种
  - **第1种,用普通全局函数实现,为了方便,通常是友元函数**，这样两个参与运算的就是，友元函数两个参数，结果对象是函数返回值
  - **第2种,用类的方法实现**，表达式左边为类本身，表达式右边为重载函数唯的一参数，结果对象是函数返回值

用友元函数实现操作符重载

- 实现某一个类之间的加法

```
class CComplex
{
public:
        double m_fReal;
        double m_fImag;
        char m_szStatus[32];
        CComplex(){
                m_fReal=0;
                m_fImag=0;
        }
        CComplex(double fReal,double fImage){
                m_fReal=fReal;
                m_fImag=fImage;
        }
        friend CComplex operator+(const CComplex &cpx1,const
CComplex &cpx2);
};
```

运算符重载为类的成员函数

```
class CComplex
{
public:
        double m_fReal;
        double m_fImag;
        char m_szStatus[32];
        CComplex(){
                m_fReal=0;
                m_fImag=0;
        }
        CComplex(double fReal,double fImage){
                m_fReal=fReal;
                m_fImag=fImage;
        }
CComplex operator+(const CComplex &cpx);
};
```

常用操作符重载

- 流操作符<< 和>>
- 赋值操作符=
- 下标运算符[]

流操作符<< 和>>

- 流操作符<< 和>> 分别对应C++ 标准库中的istream 和ostream ，一般而言，应该为

大多数类重载此操作符；因为输出操作符可用于测试和调试，并方便调用库函数的上层应用者

- 要重载流操作符，通常将此操作符定义为类的友元函数，否则类本身必须使istream 和ostream 的派生类；
- 另外在重载操作符时，还通常返回istream& 和ostream& ，这样返回值可以作为下一个操作符的输入，从而实现操作符的链接，如cout << a1 << a2 << endl ，其中a1 ，a2 分别为实现了流操作符重载的类A 的实例。
- 如果是用友元函数来实现，流操作符的第一个参数时istream 和ostream 的引用，这是因为需要更新流的内部状态；而第二个参数为类的引用，这对<< 而言是为了效率（可以定义为常量引用），对>> 而言是需要接受修改的内容。

流操作符<< 和>>(2)

```
class Complex;
ostream operator<<(ostream& os, Complex& c);
istream operator>>(istream& is, Complex& c);

//输出操作符应做的格式化应尽量少。
ostream operator<<(ostream& os, Complex& c)
{
        os<<"complex value is:("<<c.real<<","<<c.imag<<")"<<endl;
        return os;
}
//输入操作符的重载应加入错误或文件结束的处理
istream operator>>(istream& is, Complex& c)
{
        cout<<"input a complex :\n"<<endl;
        is>>c.real>>c.imag ;
        return is;
}
```

赋值操作符=

- 赋值操作符的最重要问题就是类深拷贝的问题，即对指针成员，需要重新申请空间，在拷贝内容，以免两个实例的成员指向同一块地址；
- 默认赋值操作符是浅拷贝的，这时指针成员会指向同一块地址。
- 浅拷贝是指拷贝对象的普通成员数扰，深拷贝还拷贝对象内部动态分配空间
- 需要在重载时返回*this 的引用，以实现连续赋值，即a1=a2=1
- operator= 必须为成员函数。可以重载。但右边只准出现一个参数。

赋值操作符=

```
//字符串拷贝
CMyString &CMyString::operator=(char *pszData)
{
        delete []m_pszData;
        m_pszData=new char[strlen(pszData)+1];
        strcpy(m_pszData,pszData);
        return *this;
}
```

下标运算符[]

- 标准情况下，[] 运算符用于访问数组的元素。我们可以通过重载下标运算符为类运算符。使得可以象访问数组元素一样的访问对象中的数据成员。
- C++ 只允许把下标运算符重载为非静态的成员函数。
- [] 既可以做左值也可以做右值，即即可以在表达式右边(只读)，也可以在表达式左边(可写). 因此一般需要定义两个版本
  - A[1] = ¡®a¡¯ ； 左值，c = B[2]； 右值
- [] 可以用数字做下标，也能用其它类型，如字符串作下标
  - MyList[5];
  - MyObject[¡®hxy¡¯]=a;

下标运算符[]

```
char CMyString::operator[](int iIndex)
{
        if(iIndex<strlen(m_pszData))
                return m_pszData[iIndex];
        return 0;
}
```

下标运算符[]

```
class Foo
{
public:
 int & operator[] ( const size_t);
 const int & operator[] (const size_t) const;
private:
 int data[10];
}
int & Foo::operator[] (const size_t index)
{
 return data[index];
}
const int& Foo::operator[] ( const size_t index) const
{
 return data[index];
}
```

课堂练习

- 1. 将CDate 操作全部转成重载，至少要实现如下几种
  - >>,<<,=,++,--
- 2. 自行实现一个string 类，里面用new/delete 动态分配空间
  - 要求new []，来分配字符串空间.
  - 可以直接用= 进行字符串赋值

- 支持>>,<< 重载
- 支持用[] 直接访问某一个字符
- 支持用 == 来判断字符串是否相等
- 支持用+ 进行字符串连接
- 有字符串长度的方法
- 可参考 afx.h 和MFC/src/strcore.cpp 的实现代码