Class and object

董洪伟
http://hwdong.com

为什么需要类？

- 一个类型是一个概念在语言中的实现。如基本类型 float 等是数学中实数概念的一个具体近似。
- 内在类型表示的概念层次很低，难以直观表达现实中的许多概念
- 用户定义类型提供了表达实际概念的能力。

  如工资处理程序中的**工资单**、**员工**，学生成绩表中的**学生成绩**

考虑表示一个日期概念

- 定义一个Date 结构和一组对这种结构的基本操作函数

```
struct Date{ // representation
    int day, month, year;
};
void init_date(Date& d, int, int, int) ; // 初始化 d
void add_year(Date& d, int n) ; // 增加 d.year
void add_month(Date& d, int n) ; //增加 d.month
void add_day(Date& d, int n) ; //增加 d.day
```

struct Date

```
void init_date(Date& d, int day, int m, int y) {
     d.day = day;  d.month = m; d.year = y;
}
void add_year(Date& d, int n) {
    d.year += n;
}
void add_month(Date& d, int n) {
    d.month += n;
}
void add_day(Date& d, int n) {
    d.day+= n;
}
```

struct Date

- .成员访问运算符，通过.访问一个结构对象（变量）的成员，如

  d.year　　*d.month*　　*d.day*

- ->间接访问运算符。通过对象的指针可以用->间接访问对象的成员。如

```
Date d,*p = &d;
p -> year = 1980;
```

使用Date

```
Date d;
void fun(){
    init_date(d,6,6,1900);
    d.day = 23;
    d.year = 2010;
    add_day(d,2);
}
```

- 在任何函数中都可以直接访问或修改结构对象的成员。

使用Date

```
Date d;
void fun(){
    init_date(d,6,6,1900);
    d.day = 23;
    d.year = 2010;
    add_day(d,2);
}
```

- 假如日期变量d.year变成负数，如何查错？

任何访问d的地方都可能产生这个错误！不管是否无意还是故意的。

struct Date-类

- 关键字struct定义的是一个新的数据类型-**类**类型. 在C语言中叫做结构，c++中叫做类.

```
struct Date{
        int day, month, year;
};
```

- Date 和内在类型int 一样是一个类型.可以定义Date类型的变量(对象)：

```
Date  date, d;
Date.day = 23;  Date.year = 2013;
```

struct Date：数据和操作的分离

- 上述定义的*struct Date有一些缺点：*

1) 日期类型和处理函数之间没有任何显式的联系，即数据表示与操作分离。

2) 在程序中如果定义了一个日期Date 类型的变量，那么在该变量的包围作用域中任何程序语句都可以（无意或有意）修改该变量，特别是如果是全局变量，则程序中的任何函数、语句都可以操作、修改它。这就会带来安全隐患，也许不知道在什么地方破坏了这个Date 变量;此外，维护麻烦，保持Date变量的一致性和准确性, 需要查找所有可能修改它的代码，哎，够累的！

struct Date：包含数据和操作

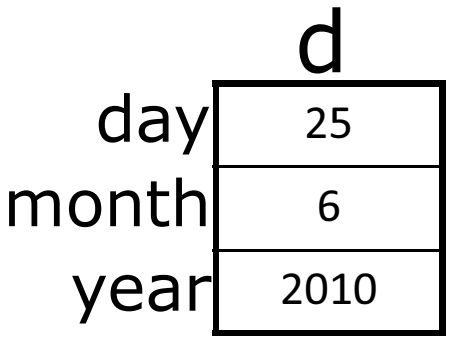
```
struct Date{
   int day, month, year;

   void init( int d, int m, int y)
          { day = d;  month = m; year = y; }
   void add_year( int n)  {  year += n; };
   void add_month( int n) { month +=n; };
   void add_day( int n) { day +=n; };
}
```

类中的函数称为类的成员函数，对类中数据进行访问或修改

新旧struct的使用对比

```
Date d;
void fun(){
    d.init(6,6,1900);
    d.day = 23;
    d.year = 2010;
    d.add_day(2);
}
```

```
Date d;
void fun(){
    init_date(d,6,6,1900);
    d.day = 23;
    d.year = 2010;
    add_day(d,2);
}
```



类的对象（变量）

- 类类型的变量和内在类型的变量定义方式是一样的

  int i= 3；

  string s=“Li Ping”;

  Date d;

- 每个类对象有自身的数据成员

  Date d,e;

  d.init(25,6,2010);

  e.init(2,2,1900);

d

day	25
month	6
year	2010

e

day	2
month	2
year	1900

类的对象（变量）

- 类的数据成员对于每个对象来说都有自身的独立一份。而不是整个类只有一份数据。
- 但类的函数成员对于整个类只有一份，类的(非静态)函数成员只能通过类对象来调用，类的函数成员是用来访问或修改类对象的数据，而不是整个类的数据。如

  d .init(6,6,1900); //对

  Date .init(6,6,1900); //错

类的对象（变量）

- *d.init(6,6,1900);* 用类Data的init函数修改变量d的三个数据成员(day,month,year)。
- 为什么类成员函数init能知道其函数体中修改(day,month,year)就是d的数据成员呢？因为该函数是通过d.来调用的。

```
struct Date{
   void init( int d, int m, int y)
                { day = d;  month = m; year = y;}
   //…
};
```

类的对象（变量）

- 类的成员函数中都隐含一个指向调用该函数的对象的指针this. 即类成员函数的this指针总是指向调用该函数的那个对象，因此那些类数据成员实际上是该对象的数据成员

```
struct Date{
   void init( int d, int m, int y) {
         this->day = d;   this-> month = m;
         this-> year = y;}
   //…
};
```

类的对象（变量）

- init(int d,int m,int y)被编译器翻译成：

*init(Date*this,int d, int m, int y);*

```
void init(Date*this,int d, int m, int y) {
        this->day = d;   this-> month = m;
        this-> year = y;}
       //…
 };
```

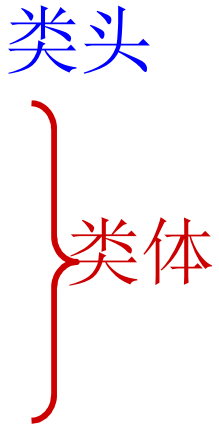
- d.init(6,6,1900)等价于init(&d,6,6,1900)

*e.init(2,2,1900)*等价于*init(&e,2,2,1900)*

定义类的关键字struct，class

- 我们可以用struct或class任何一个关键字定义一个类，就像定义C语言的结构一样。
- 定义类的格式如下：

```
struct或class 类名
{
    <类的成员>
};
```



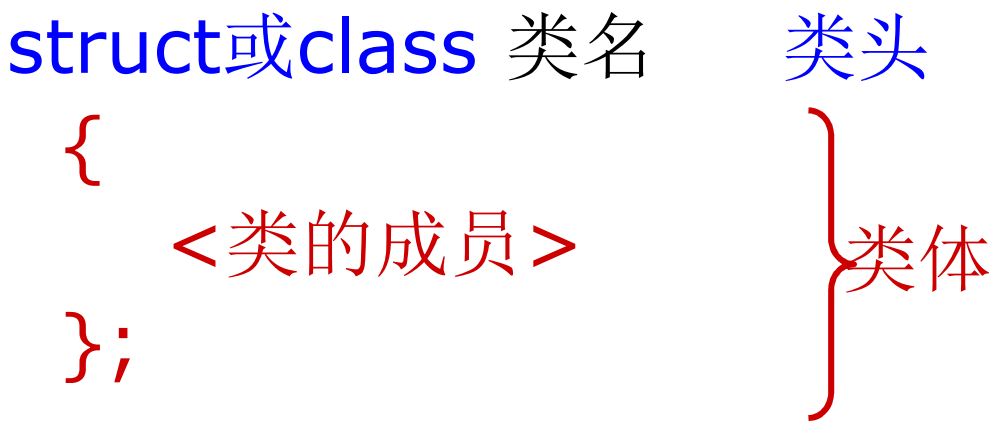
注意：最后的分号不能少！

类的定义

- struct或class+类名 构造了类头。如

  struct Date

- 由{和};围起的包含类成员的部分成为类体.
- 类是用户定义类型，包含数据成员和函数成员(成员函数).



类定义又是也被说成类声明

- 因为：它和其他不是定义的声明一样出现在头文件中，可被多个源文件多次包含.
- class Date; //类声明
- class Date{//类定义，也被称为类声明

  …

  };

回顾：声明可以多次，但定义只能一次。

成员函数

- 类的成员函数既可以在类体内定义，也可以在类体外定义，在类体外定义类成员函数时,但需要给出类作用域，如

```
struct Date{
   int day, month, year;
  void init( int d, int m, int y) ;
  void add_year( int n)
     {  year += n; };
 void add_month( int n)
     { month +=n; };
 void add_day( int n)
     { day +=n; };
}
```

```
void Date:: init( int d, int m, int y)
 {
    day = d ; month = m ;
    year = y;
 }
```

公开的成员

- 上述定义中的Data的所有数据和函数成员都是公开的，即其作用域中的任何语句都可访问它。仍然没有解决安全性问题！

```
struct Date{
    int day, month, year;

   void init( int d, int m, int y)
          { day = d;  month = m; year = y;}
   void add_year( int n)  {  year += n; };
   void add_month( int n) { month +=n; };
   void add_day( int n) { day +=n; };
}
```

信息隐蔽的方法-访问控制

- 重要的私有数据隐蔽起来，只对外展示公开接口。

```
struct Date{
  void init( int d, int m, int y)
            { day = d;  month = m; year = y;}
  void add_year( int n)  {  year += n; };
 void add_month( int n) { month +=n; };
 void add_day( int n) { day +=n; };
 private:
    int day, month, year; //私有成员
}
```

访问控制

```
Date  d;
int main(){
    d.init(24,3,2011);//ok,init是Date的公开成员
    d.day = 10;//错！ main无权限访问d的私有成员day
    return 0;
}
```

- private说明其后定义的成员数类的私有成员，不能被非类成员访问.
- struct定义的类其成员默认是公开成员，即任何函数都可以访问它们。除非用private等访问控制符修改了其访问控制属性。

访问控制

- 设计类时要确定哪些成员是公开的、私有的、保护的.
- 用struct定义的类成员默认都是公开的,即非成员函数也能访问该类对象的公开成员；而用class定义的类成员默认都是私有的，即只能被类成员函数访问。
- 用public、private、protected可以改变类成员的访问控制属性。使成员为公开、私有或保护成员。

访问控制

```
struct Date{
    int d, m, y;
    void init( int, int, int) ; // initialize d
    …
};
    Date today;
    today.init(16,10,1996);
    today.add_day(2);
    today.d+=2;
```

访问控制

```
class Date{
        int d, m, y;
       void init( int, int, int) ; // initialize d
       …
   };
    Date today;
    today.init(16,10,1996);   //错 !
    today.d += 2;          //错 !
```

访问控制

- 数据成员常是私有或保护的，而某些成员函数是公开的。
- 公开的成员就是该类对外的接口,非成员函数只能通过接口访问该类对象，而该类的成员函数能访问自身的任何成员。

访问控制

```
class Date{ // representation
    int d, m, y;
public:
   void init( int, int, int) ; // initialize d
   void add_year( int n) { y+=n; }; // add years
   void add_month( int n) { m+=n; }; // add months
   void add_day( int n) { d+=n; }; // add days
}
```

成员选择符. 和指针操作符->

- . 用于访问对象的成员
- ->访问对象指针指向的对象成员

```
void f(){
  Date d,*p;
  p = &d; //p是指向对象d的指针变量
  p->add_year(5); //通过p间接访问d的成员
  d. add_day(3);
}
```

成员选择符. 和指针操作符->

- . 用于访问对象的成员
- ->访问对象指针指向的对象成员

```
void f(){
  Date d,*p;
  p = &d; //p是指向对象d的指针变量
  p->add_year(5); //通过p间接访问d的成员
  d. add_day(3);
}
```

练习：Timer

```
#include <iostream>
using namespace std;

class Time {
 private:
 int hours_,   minutes_, seconds_;
 public:
 void set(int h, int m, int s) {
            hours_ = h; minutes_ = m; seconds_ = s; return;}
 void increment(void);
 void decrement(void);
 void display(void);
};
```

练习：Timer

```
void Time::increment (void) {
  seconds_++;
  minutes_ += seconds_/60;
  hours_ += minutes_/60;
  seconds_ %= 60;
  minutes_ %= 60;
  hours_ %= 24;
  return;
}
```

练习：Timer

```
void Time::display() {
 cout << (hours_ % 12 ? hours_ % 12:12) << ':' <<
 (minutes_ < 10 ? "0" :"") << minutes_ << ':' <<
 (seconds_ < 10 ? "0" :"") << seconds_ <<
 (hours_ < 12 ? " AM" : " PM") << endl;
}
```

练习：Timer

```
int main(void) {
  Time timer; timer.set(23,59,58);
  for (int i = 0; i < 5; i++) {
    timer.increment();
    timer.display();
    cout << endl;
  }
}
```

练习：Timer

```
void Time::decrement (void) {
  seconds_--;
  if (seconds_ < 0) {
    seconds_ += 60;     minutes_--;
  }
  if (minutes_ < 0) {
    minutes_ += 60;     hours_--;
  }
  if (hours_ < 0) {
     hours_ += 24;
  }
  return;
}
```

构造函数

- 用init初始化对象，既不优美也容易出错。如可能忘记初始化。
- 构造函数是与类名同名的成员函数。使得在定义对象时自动调用该构造函数初始化对象的数据。
- 构造函数没有返回值

```
class X{
    X();
};
```

构造函数

```
class Date{ // representation
    int d, m, y;
public:
    Date( int d0, int m0, int y0) {  构造函数没有返回值
      d = d0; m = m0; y = y0 ;
    }
  void add_year( int n) { y+=n; }; // add years
  void add_month( int n) { m+=n; }; // add months
  void add_day( int n) { d+=n; }; // add days
}
```

构造函数

- 定义类的变量（对象）时，自动调用合适的构造函数对该变量进行初始化。如定义

  Date d(6,6,1900);

  时，程序会自动调用Date的构造函数调用Date(int d0, int m0, int y0)对d的数据成员day,month,year进行初始化。
- 如找不到合适的构造函数，编译器会报错！

  Date d; //错！Date构造函数需要三个参数

构造函数

- 可以定义多个构造函数。

```
class Date {
   int d, m, y;
public:
   ...
   Date(int, int, int) ; // day, month, year
   Date(int, int) ; // day, month, today's year
   Date(int) ; // day, today's month and year
   Date() ; // default Date: today
   Date(const char*) ; //
};
```

构造函数

- 构造函数在定义类对象时自动被调用，进行对象的初始化。
- 如果有多个构造函数，则根据函数签名调用合适的构造函数
- 如果没有定义显式的构造函数，则编译器自动生成一个默认的构造函数

```
void f(){
    Date today(4) ;
    Date july4("July 4, 1983") ;
}
```

构造函数->默认参数

```
class Date {
    int d, m, y;
public:
    Date(int dd =0, int mm =0, int yy =0) ;
    //…
};
Date: :Date(int dd, int mm, int yy){
   d = dd；     m = mm;   y = yy;
}

Date d1,d2(2), d3(2,3),d2(2,3,1900);
```

默认构造函数

- 如果一个构造函数，其参数都有默认的缺省值，则该构造函数就是默认构造函数。
- 一个类只有一个默认构造函数，如果类没有构造函数，则编译器会自动生成一个默认构造函数。
- 如果类没有默认构造函数，编译器会自动生成一个默认构造函数？

//错！

默认构造函数

```
class Date {
    int d, m, y;
public:
    Date(int dd , int mm , int yy ) {
        //…
    }
    //…
};
```

```
void main(){
   Date d;
}
```

构造函数的初始化成员列表

- 可提高效率

```
class Date {
    int d, m, y;
public:
    Date(int dd =0, int mm =0, int yy =0)
    ： d(dd),m(mm),y(yy) { }
    //…
};
```

类的静态成员

- 类的静态成员通过关键字static定义。
- 静态成员是类的一部分，而不是类对象的一部分。类似于全局变量，但比全局变量安全。
- 类的静态成员只能通过类作用域访问.

类的静态成员

```
class Date {
    int d, m, y;
    static Date default_date;
public:
    Date(int dd =0, int mm =0, int yy =0) ;
    static void set_default(int,int,int);
    //…
};
```

类的静态成员

- 类的静态成员必须在某处定义

```
Date Date::default_date(16,12,1770);
void Date::set_default(int d,int m,int y){
    Date::default_date = Date(d,m,y);
}
```

类的静态成员

```
Date: :Date(int dd=0, int mm=0, int yy=0)
{
  d = dd ? dd : default_date.d;
  m = mm ? mm : default_date.m;
  y = yy ? yy : default_date.y;
  // check that the Date is valid
}
```

类的静态成员

- 和全局变量一样，要防止重定义。

Date.h

```
class Date {
   int d, m, y;
   static int s; //仅仅是声明
public:
   Date(int dd,int mm,int yy ) {
       //…
   }
   //…
};
```

Date.cpp

```
int Date::s = 0;
```

这一句放到头文件会如何？

内联(inline)成员函数

- 在类定义内部声明的函数自动为内联(inline)函数,在外部定义的成员函数则是一般的成员函数
- 外部定义的成员函数声明时如有inline修饰符，则为内联函数。

  class Date { *inline int day();};

  int Date::day() {return d;}
- 短小的函数设为内联函数可提高效率。

常量成员函数

- 函数声明的参数表后面的const，指明该函数是常量成员函数。即不会修改类对象的数据

```
class Date {
    int d, m, y;
public:
    int day() const {return d;}
    //…
};
int Date ::year() const {return y;}
```

常量成员函数

- const或非const对象都可以调用const成员函数，但非const成员函数只能对非const对象调用。

```
void f ( Date &d, const Date &cd ) {
        int i = d.year();    //ok
        d.add_year(2);     //ok
        int j = cd.year();  //ok
        cd.add_year(2);   //bad!
}
```

类对象的复制

- 初始化复制:定义一个复制(拷贝)构造函数

  X::X(const X&)

  Date d = today;

- 赋值复制: 赋值运算符=

  Date d,today(3,10,2009);

  d = today;

自引用

- 可修改状态更新函数*add_year()*、*add_month()*、*add_day()*的返回值，使返回被更新对象自身的引用，以便对对象的操作可以串起来。如流的<<和>>。

```
class Date{
  //…
  Date& add_year(int n);
  Date& add_month(int n);
  Date& add_day(int n);
};
```

自引用

- 在所有成员函数中，关键字this表示一个对象指针，指向这次函数调用的对象。*this 当然就是该对象了。

```
Date& Date::add_year(int n){
    y+=n;
   return *this;
}
```

```
Date d;
d.add_year(3).add_year(5);
```

自引用

- 在所有成员函数中，关键字this表示一个对象指针，指向这次函数调用的对象。

```
Date& Date::add_year(int n){
    this->y+=n;
    return *this;
}
```

析构函数

- 构造函数在初始化时可能要申请一些资源，如内存，那么在对象销毁时，需要释放这些资源，析构函数在对象销毁时可被自动调用完成资源的释放工作。
- 析构函数定义格式:

```
class X{
    ~X();
    //…
}
```

- 析构函数不带任何参数,也没有返回值

对象已经完蛋了，带参数 还有什么用？

析构函数

```
class String{
        char* buf;
public:
        String (const char* str = 0);
        ~String(){ delete[] buf;}
} ;
String:: String (const char* str ){
    int len = strlen(str)+1;
    buf = new char[len];
    strcpy(buf, str);
}
```

析构函数

```
class Name{
   const char* s;
   //…
};
class Table{
     Name *p;     int sz;
public:
      Table(int s=15) {  p = new Name [sz=s]; }
      ~Table(){ delete [ ] p; }
      bool insert(Name *);
};
```

例0：实现一个数学向量Vector3

具有加减、数乘、点积、叉积、求长度等操作。

```
calss Vector3{
        //.... your code
 };
//test the Vector3 class in main()
 int main(){
          Vector3 u(20,9),v(5,5,5),w = add(u,v);
          u.scale(5.0);
          Vector3 w = cross(v,u);
          double d = dot(u,v);
         double length = w.length();
         return 0;
  }
```

```
calss Vector3{
      double x_,y_,z_;
  public:
   Vector3(double x,double y,double z):x_(x),y_(y),z_(z) { };
   bool normalize();
   double length();
   double x( ){ return x_;};  double set_x(double x ){ return x_=x;};
   double y( ){ return y_;};  double set_y(double y ){ return y_=y;};
   double z( ){ return z_;};  double set_z(double z){ return z_=z;};
};
Vector3 scale(const  Vector3 u double s);
Vector3 add(const Vector3 u, const  Vector3 v);
double dot(const Vector3 u, const  Vector3 v);
Vector3 cross(const Vector3 u, const  Vector3 v);
```

例1：基于String的图书管理程序

```
class String{
  //…
};
class Book{
 //…
};
```

```
class BookVector{
   Book *books;
public:
     bool push_back();
     bool erase ( int  I );
     int size();
     Book getBook(int i);
     bool setBook ( int i, Book book );
     //…
};
```

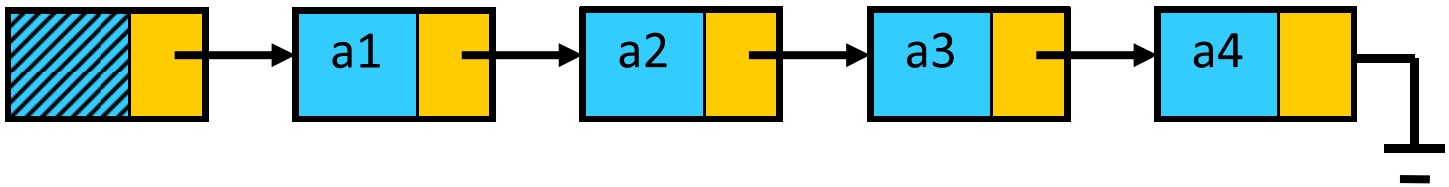
例2：实现一个矩阵类

```
class  Matrix{
   double *data_;  int rows_, cols_;
 public:
    Matrix(int rs  = 0,int cs = 0,double *data =0)  ;
    Matrix( const Matrix& m); //复制构造函数
    bool CopyTo( Matrix& m);
    Matrix& add(const Matrix& m) ;
    Matrix& mul(const Matrix& m);
    double getElem(int r,int c);
    double setElem(int r,int c);
};
 Matrix add( const Matrix& m, const Matrix& n) ;
 Matrix mul(const Matrix& m, const Matrix& n);
```

例3：链表存储学生数据

- 编写从文件中读入学生成绩数据并在屏幕上逐行显示出来。要求学生数据用链表存储。

```
struct student{
  string name;  double score;
};
struct LNode{
  student data; Lnode *next;
};
```



例3：链表存储学生数据

- 只要一个指向首结点的指针变量就可以掌握整个链表。

```
class List{
   LNode *head;
 public:
   List();  ~List();
   void insert(student d);
   int length();
   LNode* find(string name);
};
```

例3：链表存储学生数据

```
List::List(){
    head = new LNode();
    head->next = 0;
}
void List::insert(student d){
    LNode *p = new LNode();
    p->data = d;
    p->next = head->next;
    head->next = p;
}
```

例3：链表存储学生数据

```
LNode* List::find(string name){
    LNode * p = head=>next;
    while (p) {
      if( p->data.name == name) return p;
      p = p->next;
    }
    return 0;
}
```

对象(变量)

- 对象的建立有多种方式，如作为全局变量、局部变量、静态局部变量、其他类对象的成员、对象数组、动态创建、临时对象等。
- 变量（对象）的不同建立方式决定了其生命周期的不同。如全局变量在程序开始时就建立了，在程序结束后才销毁；而函数的非静态局部变量在调用该函数执行到它使才建立，函数结束后就销毁了；作为类对象成员的对象则随着其所属类对象的建立而建立，销毁而销毁...

对象

- 自动对象：程序执行时遇到它的声明时建立，离开它所在的块使销毁。

```
{
  Date d;
  //…
}
```

对象

- 自由存储对象：通过new建立，通过delete销毁

- 非静态成员对象，随着所属对象的建立而建立，销毁而销毁

```
class student{
    string name; //非静态成员对象
};
```

对象

- 一个数组元素:随所属数组的建立或销毁而建立或销毁。

  student stus[20];
- 局部静态对象：在程序执行第一次遇到它时建立，在整个程序结束时销毁。
- 全局对象、名字空间对象、类的静态成员在程序开始时建立，在整个程序结束时销毁。
- 临时对象：作为表达式的一部分建立，完整表达式计算完后就销毁。

对象-局部变量

- 每当控制离开局部变量的块时，就销毁它。
- 一组局部变量的销毁次序和构造次序相反。

```
void f(int i){
    Date aa;        Date bb;
    if ( i>0){
        Date cc;
        //…
    }
    Date dd;
}
```

问：你如何知道这些对象构造和销毁的次序？

对象-局部变量-对象的复制

- 回顾之前的String类

```
class String{
  char* buf;
public:
  const char *c_str(){return buf;}
  String (const char* str = 0);
  ~String(){ delete[] buf;}
} ;
String:: String (const char* str ){
   int len = strlen(str)+1;
   buf = new char[len];
   strcpy(buf, str);
}
```

对象-局部变量-对象的复制

```
void fun(String s){}

void main(){
    String str("Hello");
    fun(str);     //初始化复制：用str初始化s
    {String s; s = str; } //赋值复制
    std::cout<<str.c_str()<<"\n";
}
```

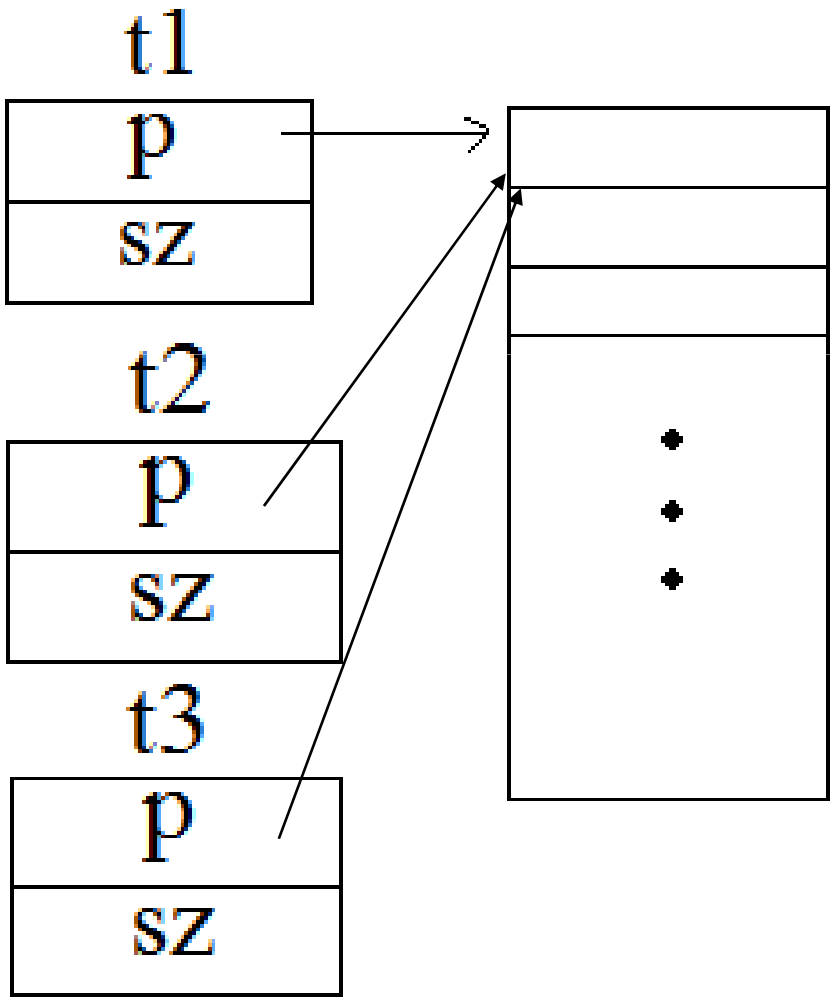
编译该程序会出错吗？

对象-局部变量-对象的复制

- 对包含指针等资源的对象复制要小心。

```
void h(){
    Table t1;
    Table t2 = t1; //复制初始化：麻烦
    Table t3;
    t3 = t1; //赋值：麻烦
}
```

对象-局部变量-对象的复制




```
void h(){
    Table t1;
    Table t2 = t1;
    Table t3;
    t3 = t1;
}
```

为什么崩溃！

对象-局部变量-对象的复制

- 解决方法：定义合适的复制构造函数和赋值运算符。

```
class String{
    const char* buf;
public:
    String(const String &);
    Stringe& operator=(const String & );
    //…
};
```

对象-局部变量-对象的复制

```
String::String(const String &str){
    const char *cs = str.c_str();
    int len = strlen(cs);
    buf = new char [len+1];
    for(int i= 0 ; i<=len;i++) buf[i] = cs[i];
}

String& String::operator=(const String &str){
    const char *cs = str.c_str();
    int len = strlen(cs);
    buf = new char [len+1];
    for(int i= 0 ; i<=len;i++) buf[i] = cs[i];    return *this;
}
```

对象-局部变量-对象的复制

- 解决方法：定义合适的复制构造函数和赋值运算符。

```
class Table{
  //…
  Table( const Table & ) ;
  Table& operator=(const Table & );
};
```

对象-局部变量-对象的复制

```
Table ::Table( const Table & t) {
    p  = new Name [ sz=t.sz ];
    for( int i = 0 ; i<sz; i++)  p[i] = t.p[i];
}
```

对象-局部变量-对象的复制

```
Table& Table ::operator= ( const Table & t) {
  if( this != &t){
     delete[] p;
     p  = new Name [ sz = t.sz];
     for(int i = 0 ; i< sz; i++)  p[ i] = t.p[ I ];
   }
   return *this;
}
```

对象-自由存储

- 多次删除同一个对象是灾难性的。
- 访问被销毁的对象也是违法的。

```
Date *p = new Date;
Date *q = new Date;
delete p;
delete p;
 p->add_day(3);
```

类对象作为成员

- 考虑一个小型组织类

```
class Club{
    string name;
    Table members;
    Date founded;
    //…
    Club( const string &n, Date fd);
};
```

类对象作为成员

- Club构造函数的参数必须提供成员构造函数所需的参数.

```
Club::Club(const string &n,Date fd)
        :name(n), members(), founded( fd ) {
    //…
}
```

- 初始化成员列表以冒号开头，以逗号分隔。

类对象作为成员

- 成员的构造函数将在容器类构造函数执行之前执行。成员的构造函数按照其在容器类中的声明次序执行。
- 如果某个成员不需要参数，则无需出现在初始化列表内。否则，必须出现在初始化列表中。

```
Club::Club(const string &n,Date fd)
    :name(n) , founded( fd ){
    //…
}
```

- 成员销毁的次序正好和构造次序相反。

类对象作为成员-必须初始化的成员

- 没有默认构造函数的、const、&引用成员的初始化式是必须的。容器类构造函数的参数表中也应该提供这些成员的初始化参数。

```
class CC{
  const int i;
  Club c;
  Club &pc
  //…
  CC(int ii,const string&n,Date &d,Club& c)
     :i(ii),c(n,d),pc(c) {}
};
```

对象-数组元素

- 如果某个类有默认构造函数，则可以定义该类对象的数组。否则，不行？

```
Club clubs[20];  //okay
CC ccs[10];      //no !
```

对象-局部静态对象

- 在第一次执行遇到它时，构造；在整个程序结束时销毁。
- 局部静态对象的销毁按照构造的相反次序销毁

```
void f(int i){
    static Table tb1;
    if(i>0){
      static Table tb2;
       //…
    }
}
```

```
int main(){
   f(0);
   f(1);
   f(2);
}
```

对象-非局部存储

- 全局对象、名字空间对象、类的static变量在main激活前构造，在退出main()之前被析构函数销毁。
- 同一个编译单元中按照定义次序构造这些对象。
- 不同编译单元中构造次序不确定。可通过一种开关技术确保对象在使用前被构造。
- 尽量少用这种类型对象。

对象-临时对象

- 临时对象通常作为表达式的结果出现，当完整表达式结束时，就不存在了。

```
w = x*y + z;
```

- 函数调用时的函数返回结果也是临时对象。

```
int add(int x,int y){return x+y;}
z = add( x,y );
```

- 完整表达式是指不是其他表达式的子表达式的表达式。

对象-临时对象

```
void f(string &s1,string &s2,string &s3){
    const char *cs = (s1+s2).c_str();
    cout<<cs;
    if( strlen(cs = (s2+s3).c_str())<8&&cs[0]!='a'){
       string s4 = cs;
    }
}
```

对象-临时对象

- 应编写清晰的代码：

```
void f(string &s1,string &s2,string &s3){
    cout<<s1+s2;
    string s = s1+s2;
    if( s.length()<8&&s[0]!='a'){
       string s4 = s;
    }
}
```

对象-临时对象

- 可以用临时对象作为const引用或命名对象的初始式。如

```
void g(const string &,const string &);
void h(string &s1,string &s2){
    const string &s = s1+s2;
    string ss = s1+s2;
    g(s,ss);  //ok
}
```

对象-临时对象

- 不能返回到局部变量的引用

```
const string& f(){
    string s; return s;
}
```

- 临时对象也不能约束到非const引用

```
string &s = s1+s2;
```

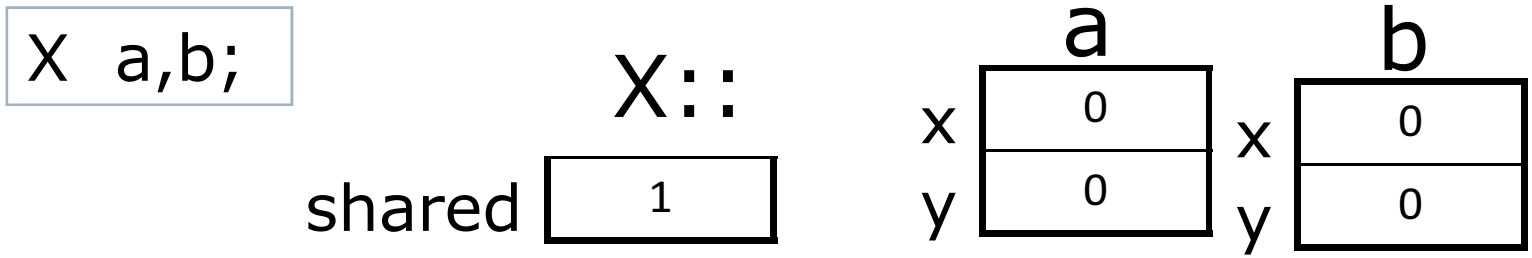
静态成员

- 每个类对象有自身的数据成员，可以对数据进行保护等。但
- 有时需要同一个类的所有对象能共享某些数据
- 全局变量可以被所有代码访问，但我们希望类的共享数据只能被该类的对象共享，并只能被类成员函数访问！
- **静态数据成员**可以解决同一类的数据共享问题
- 类的静态数据成员是所有类对象共享的，只有一份。*如某个组件类用一个计数器来共享某个资源，计数器为0时，就释放该资源。*

静态成员-说明和定义

- 在类体内说明静态数据成员，在类体外定义该静态数据成员。

```
class X{
    int x,y;
    static int shared; //静态成员说明
};
int X::shared = 1; //静态成员定义
```



静态函数成员（静态成员函数）

- 和一般函数定义基本相同，只是在类体中对静态函数成员说明前加上static.

```
class X{
    int x,y;
    static int shared; //静态成员说明
 public:
   static int get_shared(){return shared;}
    static void set_shared(int s){shared=s;}
};
 int X::shared = 1; //静态成员定义
```

- 静态成员函数只能访问静态数据成员。

静态成员-类外访问

- 类的成员函数可以访问类的所有数据（静态或非静态）。
- 类外访问类的公开静态成员：

1）不能通过类对象访问，因为静态成员不属于单独的类对象，是属于整个类的。

2）需要加上类名来访问类的静态成员。

X::shared; //写法正确，但shared不是公开的

X::set_shared(5);

友元

- 类的非公开成员不能被其他类及其成员函数、全局函数等访问。
- 有时可以在定义类时指定全局函数、其他类、某个类的成员函数为该类的友元。这些友元可以访问该类的所有数据。
- 用关键字*friend*说明哪些类或函数是该类的友元

```
class X{
  friend int fun(int c);
  friend class B;
  friend int C::mem(char c);
};
```

友元-注意点

- **友元不具有传递性**。B是A的友元， C是B的友元，不能推出：C是A的友元.
- 友元**不具有交换性**： “B是A的友元”不代表A是B的友元”.
- **友元不具有继承性**： B是A的友元， C是B的派生类，不能推出：C是A的友元.

其他注意点

- 静态整型成员，可以加上一个常量作为初始式。

```
class Curious{
  static const int c1 = 7; //ok,但要在外部定义
  static  int c2 = 7; //错，非const
  const int c3 = 6;//错，非static
  static const float f = 7;//错，非整型
};
```

其他注意点

- 初始化成员需要定义，但初始式不必重复定义

  const int Curious:c1; //ok

  const int*p = &Curious:c1;

- const,引用，无默认构造函数的成员在初始化成员列表中初始化。

常见语法错误

```
class X{
  public:
     inline void m();
     //…
};
inline void X:m(){
    //….
}
```

inline只能出现在声明内；
除非声明和定义是一个！

常见语法错误

```
class X{
  public:
     inline void m();
     //…
};
void X:m(){
    //….
}
void main(){
     X x;
     m();                    //错！ 成员函数必须通过类对象调用
     x.m() ;                 //OK
}
```

常见语法错误

```
class X{
   const int c;
  public:
     X() { c = 0; }
 };
```

//错！常量和引用成员必须在构造函数的初始化成员列表中初始化！

```
class X{
   const int c;
  public:
     X():c(0) {   }
 }
```

常见语法错误

```
class X{
public:
   void m() { /*…*/}
   static void n() { /*…*/}
 };

void main(){
   X x;
   x.m();
   x.n();
   X::m();          //错！非静态成员函数只能通过类对象调用！
   X::n();
}
```

常见语法错误

```
class X{
public:
   void m(const X& b) {
     this = &b;          //错！ This是一个常量！
     /*…*/
   }
   static void n() {
     this->count = 0;     //错！ 静态成员函数是属于类的
     /*…*/
  }
 private int count;
};
```

语法作业

- 列出你关心的语法要点和注意点。并举例

编程作业

- 实现一个字符串类
- 实现一个数学向量类Vector3,包括初始化、复制、加减、求长度、规范化、求点积、求叉积。

```
Vector3  V1(1,0,3),V2(1,1,1);
Vector3 V = add(V1,V2); //dot(V1,V2)  ,cross(V1,V2)
Vector W = V.normalize();
```

数学定义：　dot(V1,V2):= 1*1+0*1+3*1 = 4;

V1.normalize():= (1,0,3)/sqrt(10);

编程作业

- 实现一个字符串的队列类 3-8
- 实现一个基于学生成绩表（查找、插入、删除）的学生成绩分析管理（统计、显示、查找等）