華中科技大學

C++的类　进阶

许向阳

xuxy@hust.edu.cn

重点及难点

类的定义　访问权限　构造函数　析构函数

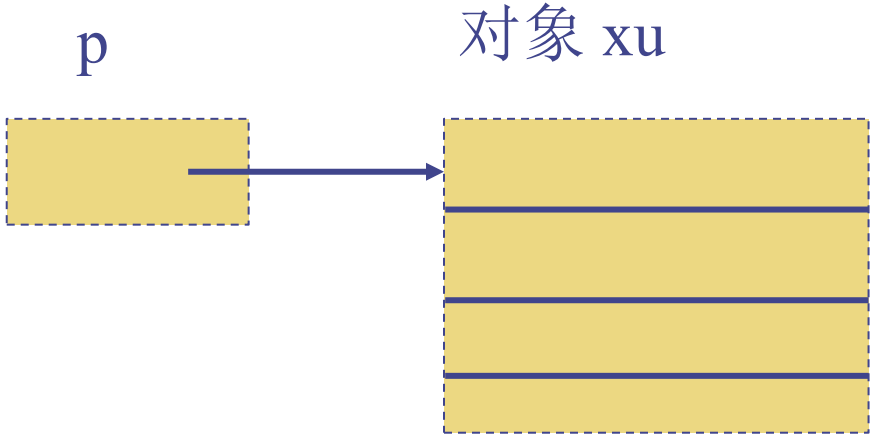
- 对象指针　（对象数组指针　VS　对象指针数组）
- 对象指针指向的空间的分配及构造、析构及释放

  （new　VS　malloc）　（delete VS free）
- 隐含函数 this 的实现机理
- 含有特定数据成员（常量、引用、对象成员）时，

  类的构造函数、对象的构造过程（数据成员的初始化顺序）、对象的析构过程（顺序）

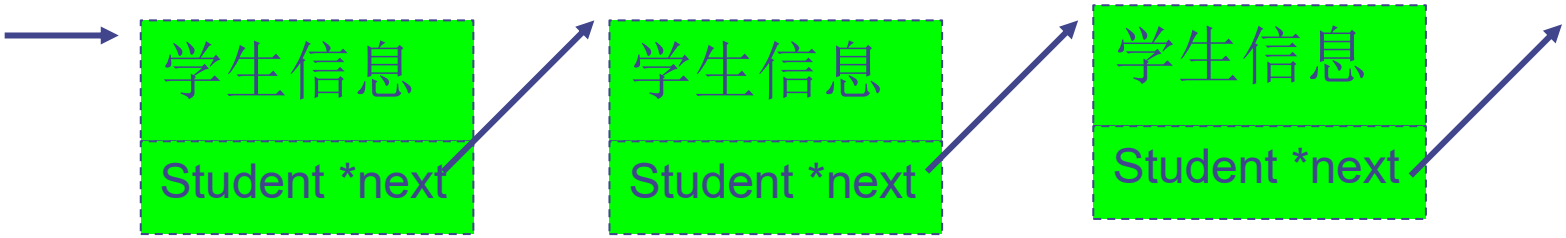
4.5 对象指针

```
Student  xu( “xuxy” ,21, 90, “very good” );
Student  *p;
p = &xu;
cout << p->name << endl;     // 注意：访问权限
```



4.5 对象指针

- 建立一个学生信息链表



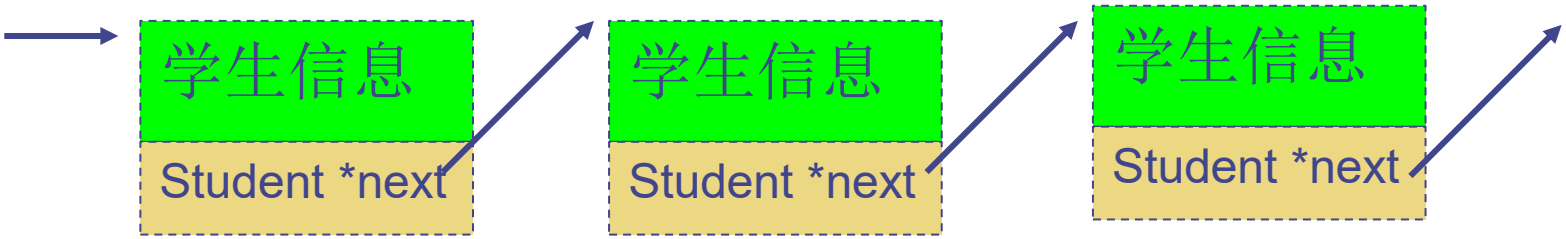
- 在 Student 类中增加了数据成员

  Student *next;　　对象指针

```
class  Student {
    ...
    Student *next;
};
```

Q：这种设计有何优点？有何缺点？

4.5 对象指针



- 增加了一个新的类 StudentNode

  含有 一个Student 对象成员

  StudentNode 对象指针

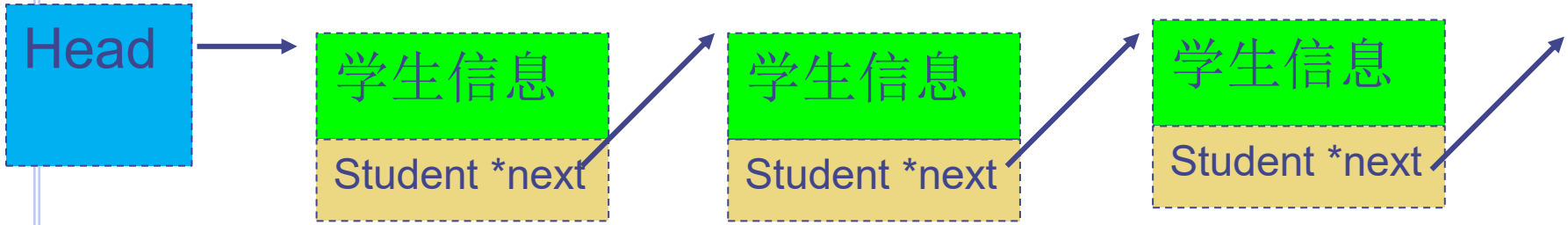
```
class  StudentNode {
    Student  s;
    StudentNode *next;
};
```

Q：这种设计有何优点？有何缺点？

Q：结点挂链的函数会是什么样的？

```
StudentNode  *head=NULL;
void insertIntoList(StudentNode * &start, StudentNode *p);
```

4.5 对象指针

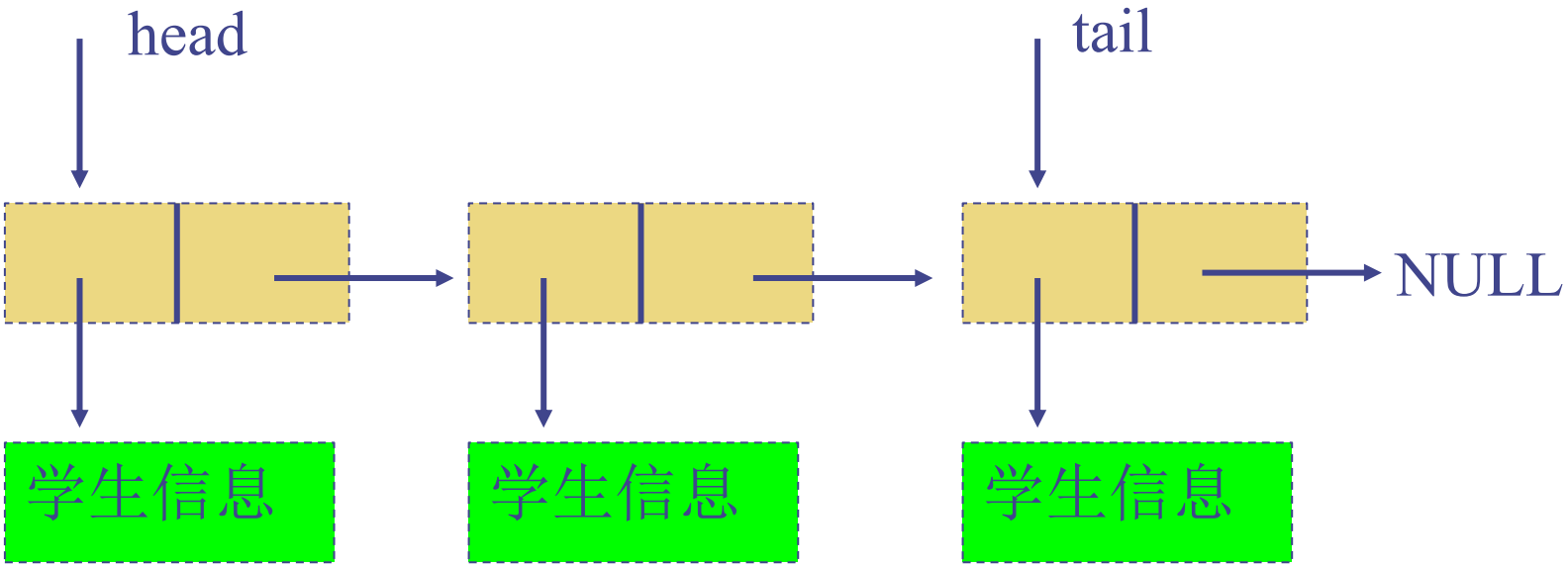



```
class  StudentList {
 StudentNode *head;
public:
  insertIntoList(StudentNode *p);
};

StudentList  class1;
StudentList  class2;
```

```
class  StudentNode {
   Student  s;
   StudentNode *next;
};
```

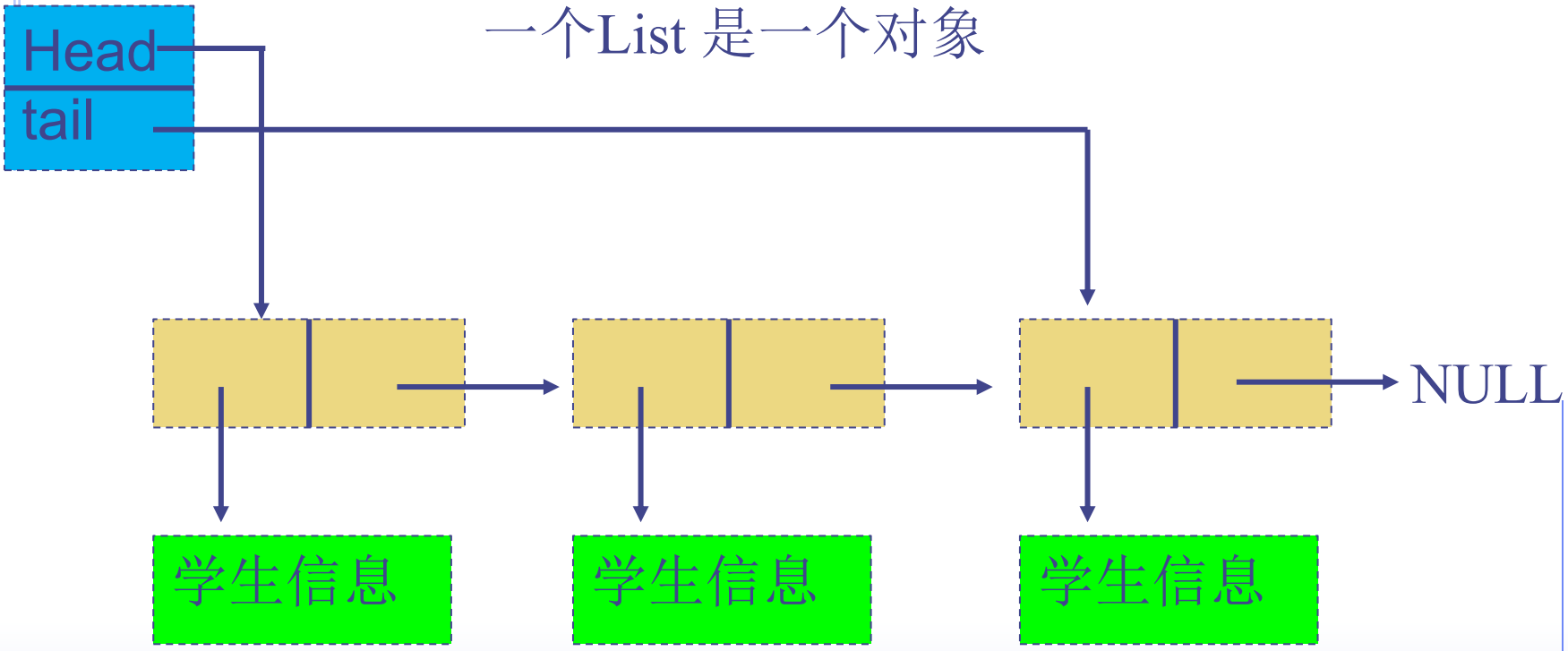
4.5 对象指针

- 建立一个学生信息链表



4.5 对象指针

- 建立一个学生信息链表



4.5 对象指针

Student.h

```
#pragma once
class Student {
    private: char  name[10];
             short age;
             float score;    char* remark;
    public:
        Student(const char* name, short age,
                float score, const char* remark);
        Student();
        Student(const Student& s);
        void displayRemark();
        int  getAge();
        char* getName();
        Student& operator =(const  Student& s);
        void studentInfo();
        ~Student();
};
```

4.5 对象指针

Node.h

```
// #pragma once
#ifndef  _NODE_H
#define  _NODE_H
#include "Student.h"

class Node {
    private:
      Student* ps;
      Node* next;
    public:
      Node(Student* p);
      ~Node();
      Node *getNext();
      void setNext(Node* pn);
      Student* getStudent();
};
#endif
```

4.5 对象指针

StudentList.h

```
#pragma once
#include "Node.h"
class StudentList {
private:
        Node *head;
        Node *tail;
public:
        StudentList();
        ~StudentList();
        void InsertToList(Node* p);
        void ListInfo();
        Student* SearchByName(const char* name);
};
```

4.5 对象指针

Q：怎样产生指针指向的对象？

使用 malloc 可以申请空间；

但空间中的内容没有初始化。

构造函数是用来初始化对象空间内容的，

但是只能自动调用，不能在程序中写调用语句。

4.6 new和delete运算符

new ： 申请空间 （step 1）
自动调用构造函数初始化 （step2）
delete ： 自动调用析构函数
释放空间

malloc: 申请空间
free ： 释放空间

4.6 new和delete运算符

- 简单类型的内存管理
- 单个对象的内存管理
- 对象数组的内存管理

简单类型的内存管理

- 使用 malloc

```
#include <malloc.h>
 char   *remark;
 remark = (char *)malloc(15);
```

- 使用 new

```
 remark = new char[15];
```

new 与 malloc 等同

free 与 delete 等同

可混用

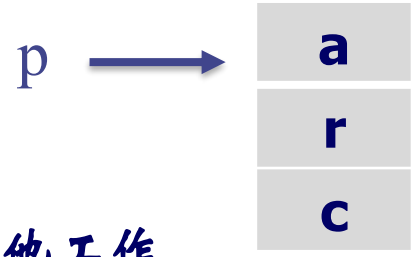
```
 free(remark);
 delete  remark;
 delete  [ ]remark;
```

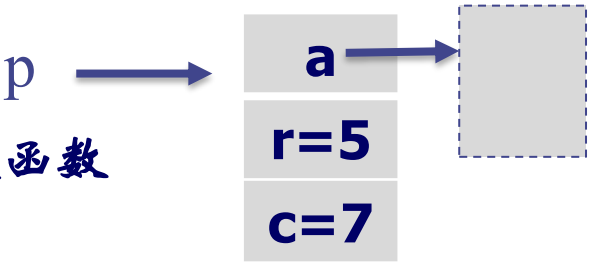
单个对象的内存管理


```
class  ARRAY{
private:
    int  r , c;              // 行数, 列数
    int  *a,                 //  数组元素存放区
public:
    ARRAY(int x, int y)   {
        r=x;  c=y;
        a=new int[x*y];          // int型，可用malloc
    }
   ~ARRAY( )     {
        if  (a)  {
           delete [ ]a;   //可用free, 也可用delete a
           a=0;
        }
        r = c = 0;
    }
};
```

单个对象的内存管理


```
int main(void){
    ARRAY  *p;
    p = (ARRAY *)malloc(sizeof(ARRAY));
            //只为p分配了指向的空间，未进行其他工作
    free(p);

    p=new ARRAY(5, 7);
            //  分配了空间，并调用ARRAY的构造函数
    delete  p;     // 调用析构函数，释放了空间
    return 0;
}
```





p	0x00380830
a	0xfeeefeee
r	-17891602
c	-17891602

p	0x00380830
a	0x00380ac0
r	5
c	7

单个对象的内存管理，注意 malloc 与 new的差别

free 与 delete 的差别

单个对象的内存管理

new 类型

```
int  *p = new int;
ARRAY  *q = new ARRAY;
```

调用无参数的构造函数

new 类型(参数)

```
int  *p = new int(10) ;
ARRAY  *q = new ARRAY(5,7);
```

对象数组的内存管理

- **new 类型[size]　size个 指定类型的对象数组**

```
int   *p;
int   *pa;
pa = (int *)malloc(sizeof(int) * 10);
```

数组指针

pa[0], pa[1],.....,pa[9]

```
ARRAY  *qa = new ARRAY[5];
```

数组指针

如果有有参数的构造函数，则必须同时要有无参的构造函数

qa[0], qa[1], qa[2], qa[3], qa[4] 都是ARRAY对象

对象数组的内存管理

new 类型[size]{{第0个对象的构造参数},

{第1个对象的构造参数}……}

```
ARRAY  *q = new ARRAY[10]{{3,4},ARRAY(5,6),{…}…};
```

delete运算符

```
ARRAY  *aq = new ARRAY[10];  // 对象数组指针
ARRAY  *sq = new ARRAY(5,7); // 对象指针
```

- delete <指针>　　delete sq;
  - 指针指向非数组的单个实体
  - 如sq指向对象，则自动调用析构函数，再释放对象所占的内存。

- delete [ ] <数组指针>　　delete [] aq;
  - 指针指向任意维的数组时使用
  - 对所有对象(元素) 自动调用析构函数。
  - 若数组元素为简单类型，则可用delete <指针>代替。

4.6 new和delete运算符

```
A* p = new A[5];
delete p;
```

C6278: “p”使用数组 new [] 分配，但使用标量 delete 删除。 将不会调用析构函数。

编译时，有警告！但是，仍能生成执行程序！
运行时，出现异常！

```
A* p = new A[5];
delete []p;
```

Microsoft Visual Studio 调试控制台

```
construct no parameter
construct no parameter
construct no parameter
construct no parameter
construct no parameter
deconstruct
deconstruct
deconstruct
deconstruct
deconstruct
```

DeepSeek :

A* p = new A[5];
delete p; 为什么运行时会出错？

4.6 new和delete运算符

DeepSeek: 使用对象指针有何缺点？

- 内存泄漏：如果使用new分配了内存，但没有使用delete释放，就会导致内存泄漏。在复杂的程序中，确保每一个new都有对应的delete可能会很困难，尤其是在异常发生时。
- 悬空指针：当指针指向的内存已经被释放，但指针仍然保留着原来的地址，这时就产生了悬空指针。如果后续通过这个悬空指针访问内存，会导致未定义行为。
- 重复释放：同一个内存块被释放两次，这会导致运行时错误。
- 内存分配与释放不匹配：使用new分配单个对象应该用delete释放，使用new[]分配数组应该用delete[]释放。如果混用，会导致未定义行为。
- 所有权不明确：当多个指针指向同一个对象时，很难确定哪个指针负责释放内存。这可能导致重复释放或者内存泄漏。

4.6 new和delete运算符

DeepSeek: 使用对象指针有何缺点？

- 异常不安全：如果在new和delete之间发生异常，可能会导致delete没有被执行，从而内存泄漏。
- 代码复杂性：使用裸指针需要程序员手动管理内存，这增加了代码的复杂性和出错的可能性。
- 资源管理分散：资源（如内存）的分配和释放可能分布在代码的不同地方，使得资源管理变得困难。
- 性能开销：动态内存分配（使用new）通常比栈上分配要慢，而且可能引起内存碎片。
- 不利于RAII：裸指针不能自动管理资源生命周期，而RAII（资源获取即初始化）是C++中管理资源的重要理念。使用裸指针需要额外的工作来实现RAII。

4.6 new和delete运算符

引用的本质是“不能移动的” 指针

Q:如何使用引用代替指针？

```
class A {
public:
    int x;
    A() { cout << "construct no parameter" << endl; }
    A(int x) { A::x = x;  cout << "construct with parameter" << endl;}
    ~A() { cout << "deconstruct " << x<< endl; }
};
int main()
{
    A  &p = *new A(10);
    delete  &p;
    A  &q = *new A[3];
    (&q)[0].x = 10;
    (*(&q+1)).x = 20;
    (&q)[2].x = 30;
    delete []&q;
}
```

```
construct with parameter
deconstruct 11
construct no parameter
construct no parameter
construct no parameter
deconstruct 30
deconstruct 20
deconstruct 10
```

4.6 new和delete运算符

```
int     *pa[10];        // 指针数组，有10个指针排在一起
ARRAY   *qa[10];        // 指针数组
```

指针数组　VS　数组指针

4.6 new和delete运算符


```
int     *pa[10];  // 指针数组，有10个指针排在一起
int     (*q)[10]; // 数组指针，指向一个数组
```

(1) 定义了一个变量 q

(2) q 是一个指针

(3) 将（*q）视为 A，则有 int A[10],这是一个数组

(4) q是指向长度为10的整型数组的指针

```
q = new  int[3][10];
```

分配的字节数为 3*10*4 。

q=q+1; // q 增加 40个字节。将A[10] 视为一个整体

4.6 new和delete运算符

对于 简单类型(没有构造、析构函数)指针分配和释放内存

- new和malloc、delete和free没有区别
- 可混合使用，如new分配的内存用free释放。

对于 复合类型(类)指针分配和释放内存

- new，先申请空间，再自动调用构造函数初始化空间
- delete ，先自动调用析构函数，在释放new申请的空间。

delete 用法讨论

Q：对于 一个对象，能否用 delete 释放空间？

```
A   a={….};
delete  &a;
```

编译未报错
运行异常

Q：编译时未报错，为什么？

因为存在如下的等价写法，编译器难以检查！

```
A   a={ …};
A   *p = &a;
delete  p;
```

但执行时出现错误。

只有用 malloc、new 申请的空间才能用delete 释放

delete 用法讨论

设计实验，验证析构与释放对象本身的空间无关

A a={ …};

a.~A( );

在此可以继续对 a 进行操作，…….

可以显示 a 的信息

a.~A( ) ; 多次调用析构函数

A *p = new A(……);

p-> ~A( );

之后，可以继续通过 p 对指向的对象进行操作；

delete p; p=nullptr;

delete 用法讨论

对于一个指针，释放其指向的空间，但该指针变量本身的值 没有变，指针所占的空间（4/8个字节）也不会释放。

指针的合适用法

(1) 在定义指针时，就赋给初值

```
A  *p = new A(……);
A  *p = nullptr;
```

(2) 在释放指针指向的空间后，立即将其置为 空指针

```
delete  p;
p=nullptr;
```

4.6 new和delete运算符

用 malloc、new分配的空间一定要用free、delete释放

否则造成内存泄露

```
#define CRTDBG_MAP_ALLOC      // 加在文件开头
#include <crtdbg.h>
```

有 #include <iostream> 时，不需要加上面的语句

```
_CrtDumpMemoryLeaks();    // 在 main函数结束前
```

智能指针

对象数组初始化


```
ARRAY   q[5];    //   对象数组   q[0],q[1],q[2]…
                 //   使用无参数的构造函数
```

```
int     x[5]={0,1,2,3,4};
ARRAY   q[5]={{对象q[0]的构造参数},……};
```

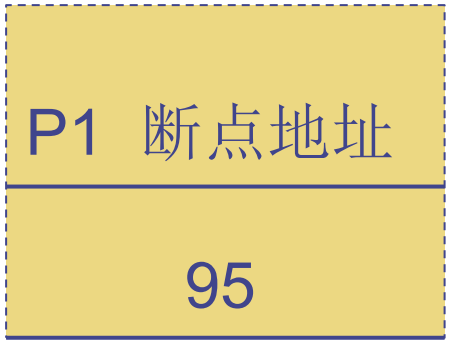
```
int     x[5] {0,1,2,3,4};
ARRAY   q[5] {{对象q[0]的构造参数},……};
```

```
ARRAY   q[5]={{对象q[0]的构造参数}};
        q[1],…,q[4] 要采用无参数的构造函数
```

4.7 隐含参数this的实现机理


```
Student(const char* name, short age, float score,
        const char* remark);
void Student::setScore(float score)
{  this->score=score;   }
```

```
Student  yang("yang", 22,95,"talent");
Student  xu("xuxy",,21,90,"good");

    xu.setScore(95);
P1:
    yang.setScore(100);
P2:
```



xu的地址 →ecx

有两个对象，是修改哪个对象中的变量？

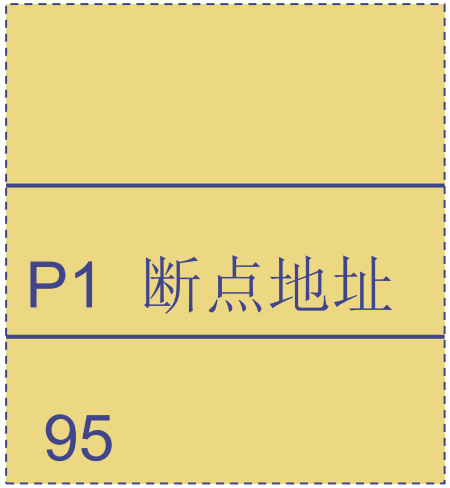
4.7 隐含参数this的实现机理

```
void Student::setScore(float score) {
    this->score=score;
   // (*this).score=score; 等价语句
   // Student::score=score;
}
```

使用this访问数据成员，区分函数参数的访问

```
    xu.setScore(95);
P1:

 lea        ecx,[xu]
 call       Student::setScore
 ……
 mov        dword ptr [this],ecx
 //  mov    dword ptr [ebp-8],ecx
```

score



4.7 隐含参数this的实现机理

- this是一个隐含的指针常量，不能移动或对该指针赋值。

  类名 * const this;

  this =p; // =左操作数必须是左值

- 普通函数成员的第一个参数，指向调用该函数成员的对象。*this代表当前被指向的对象。
- 当对象调用函数成员时，对象的地址作为函数的第一个实参，通过这种方式将对象地址传递给隐含参数this。

- 构造函数和析构函数的this参数类型固定。由于析构函数的参数表必须为空，this参数又无类型变化，故析构函数不能重载。
- 类的静态函数成员没有隐含的this指针。

4.7 隐含参数this的实现机理

華中科技大學

成员函数名后加 const 的意义！

```
class  TREE{
    int   value;
    TREE  *left, *right;
public:
    TREE (int);
    ~TREE( );                  //this类型: TREE * const this
     const TREE *find(int)const; //this类型: const TREE * const this
};
TREE::TREE(int value){   //隐含参数this指向要构造的对象
    this->value=value;     //等价于TREE::value=value
    left=right=0;          //C++提倡空指针NULL用0表示
}
const TREE* TREE::find(int v) const { //this指向调用对象
    if(v==value) return this;                  //this指向找到的节点
    if(v<value) return  left!=0?left->find(v):0;  //查左子树
    return  right!=0?right->find(v):0;     //查右子树，调用时新this=left
}
```

在树中找节点值为v 的节点

this 相当于树的根指针

4.8 对象的构造与析构

定义一个对象时，自动调用构造函数，构造对象

对象的生命周期结束时，自动调用析构函数

构造函数不能写语句调用

Q：如果一个类中 含有 常量成员 (只读成员)、引用成员、静态成员、对象成员，如何初始化？按什么顺序初始化？

4.8 对象的构造与析构

类中的各种数据成员如何初始化？

```
Class A {   ……};
Class  B
{       int  x;
        const int y;     // 只读成员
        int  &z;         // 引用成员
        static int u;    // 静态成员
        A    a;          // 对象成员
        A    *p;
        A    &q;         // 引用成员，引用的是一个对象
        B(……) { ……..}
};
```

Q:能够都在构造函数中{……}初始化吗？初始化的顺序？

4.8 对象的构造与析构

方法1：在定义变量时，直接初始化

在类的定义中，可以对非静态数据成员进行初始化。

```
class B {
  public:
     int   x=10;              // 初始化 x=10
     const  int   y=20;   //  初始化 y= 20
     int   &z = x;            //  z 中的内容为 x的地址
     A    a{…..};             //   类似于构造函数，要用{ } 代替( )
     A    *p = new A(…);
     A    &q=a;
};
```

4.8 对象的构造与析构

方法1：在定义变量时，直接初始化

```
class A {
  public:
     int   x {10};      // 初始化 x=10; 可写成 int  x=10;
     int   &y {x};     //  y 中的内容为 x的地址
    const int  z{20};
    int   u=30;
    A     a{……};
};
```

类成员不能像一般的变量那样用 ( ) 来初始化

```
int  x2(10);     // 非类的成员变量，等效 x2=10;
```

4.8 对象的构造与析构

```
Class A {   ……};
Class  B
{     int  x;
      const int y;
      int  &z;
      static int u;
      A    a;
      A    *p;
      A    &q;
     B(int t1,int t2,……): y(t),  z(t2), a(…),  q(…)
        { ……..}
};
```

Q: 在一个对象中，有些成员是常成员(引用成员、对象)。但不同的对象，这些常成员是不同的。如何初始化？

在**构造函数体前**初始化：

只读成员、引用成员、对象成员

- 对象指针可在构造函数中初始化
- 静态成员在类外初始化

4.8 对象的构造与析构

- 如果常变量、引用变量在定义时未初始化，则在构造函数前进行初始化 。

  类比：普通的常变量、引用变量（即不是一个类的数据成员），是应该在定义时就初始化的。因而不应该在构造函数内初始化的。 特殊！

- 类中有数据成员是一个对象时，也要在构造函数前进行初始化。Why?

  构造函数只能自动调用，不能写调用构造函数语句。

  数据成员是对象指针时，则可在构造函数中初始化。

4.8 对象的构造与析构


```
Class A {   ……};
Class  B
{     B(int t1,int t2,……): y(t),  z(t2), a(…),  q(…)
        { ……..}
};
```

在构造函数体前初始化：只读成员、引用成员、对象成员

- 函数说明之后
- { } 之前
- ： 分隔
- 各数据成员以逗号分隔
- 用( )、{ }形式给各变量赋初值， 如 y(t), y{t}
- 不能采用 = 来初始化　　: y=t　　// error

4.8 对象的构造与析构

在类的定义中，可以对非静态数据成员进行初始化。

```
class A {
  public:

     static  int  v =30;   // 错误语句，静态数据成员独立于
                           //  对象而存在
    static const int w=40;  // 有const 约束，可赋值
    const static int  t =50;
};

int  A::v=30;
```

4.8 对象的构造与析构

组合类对象的初始化

```
class  Date
{
private:
        int  day, month, year;
public:
        Date(int dd, int mm, int yy)
        {
            day = dd;
            month = mm;
            year = yy;
        }
        void Print()
        {
            cout<<year << " - " << month << " - " << day<<endl;
        }
};
```

4.8 对象的构造与析构

类中含有其他类的对象
组合类对象的初始化

```
class Student
{
private:
        int  number;
        char name[15];
        Date birthday;   // 何时初始化 birthday ?
        float score;
public:
        Student(int number1,char *name1,float score1,int dd,int
mm, int yy);
        ~Student();
        void Modify(float score1);
        void Print();
};
```

4.8 对象的构造与析构


```
Student::Student(int number1,char *name1,float score1,int
dd,int mm, int yy) : birthday(dd,mm,yy)
{
        number = number1;
        strcpy_s(name,name1);
        score=score1;
}
```

构造函数体之前初始化，冒号分隔

4.8 对象的构造与析构

数据成员初始化方法：

- 在定义数据成员时赋初值，等价于在构造函数体前赋初值；
- 在构造函数中赋初值；
- 在构造函数体前赋初值；【体前与定义同时赋值，以体前赋值为准】
- 在定义对象时，自动调用构造函数初始化；
- 按定义的先后次序初始化，与出现在初始化位置列表的次序无关；
- 普通数据成员没有出现在初始化位置时，若所属对象为全局、静态或new的对象，将具有缺省值0；
- 基类和非静态对象成员没有出现在初始化位置时，此时自动调用无参构造函数初始化对其初始化；
- 如果类仅包含公有成员且没有定义构造函数，则可以采用同C兼容的初始化方式，即可使用花括号初始化数据成员；联合类型的对象只须初始化一个成员(共享内存)；

4.8 对象的构造与析构

- 0个 或多个构造函数；
- 在没有定义构造函数时，默认有一个无参数的构造函数，为对象分配相应的空间；该函数为定义类时，赋了值的常成员、引用成员、对象成员进行构造；
- 定义了构造函数，则不会默认无参构造函数；
  但有缺省的以对象为参数的构造函数；
- 并非所有的数据成员都要在构造函数中出现；
- 定义常规的数据成员时，可以赋初值；
- 在构造函数中的初始化会代替定义时的初值。
- 在类中有数据成员是 const、引用、对象成员时，除非定义时就初始化，否则一定要有构造函数。

4.8 对象的构造与析构

析构函数

对象撤销时，释放体外空间或其他处理

- 函数名与类名相同
- 函数名为类名前加 ~
- 没有返回类型
- 无参数函数
- 只能有一个析构函数
- 可以自动调用，也可以在程序中显式调用
- 对象的生命周期结束时，被自动调用

4.8 对象的构造与析构

- 对象的生命周期结束时，自动调用析构函数
- 类中的对象成员，按照后定义先析构的原则，逐一自动析构；
- 即使不定义析构函数，也会有一个默认的析构函数，会析构类中的对象成员；
- 析构函数并不释放对象内部空间，一般用于释放体外申请的空间，或者做其他善后事项；
- 对象本身的空间，是依靠堆栈栈顶指针变化自动实现；
- 构造函数也不分配对象本身的空间，它只是初始化对象空间中的内容。

4.8 对象的构造与析构

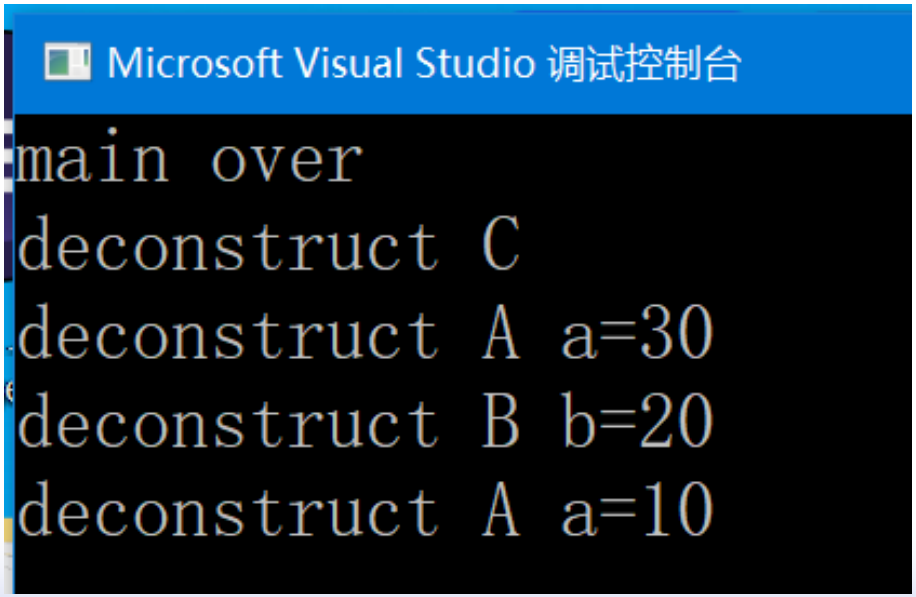
析构对象成员

```
class A {
    int a;
public:    A(int t) { a = t; }
    ~A() { cout << "deconstruct A a=" << a <<endl; }
};
class B {
    int b;
public:     B(int t) { b = t; }
    ~B() { cout << "deconstruct B b=" << b <<endl; }
};
class C {
    A  oa;        B  ob;     A  oaa;
public:
    C(int v1,int v2,int v3):oa(v1),ob(v2),oaa(v3){}
    ~C() { cout << "deconstruct C " << endl; }
};
```

4.8 对象的构造与析构

析构对象成员

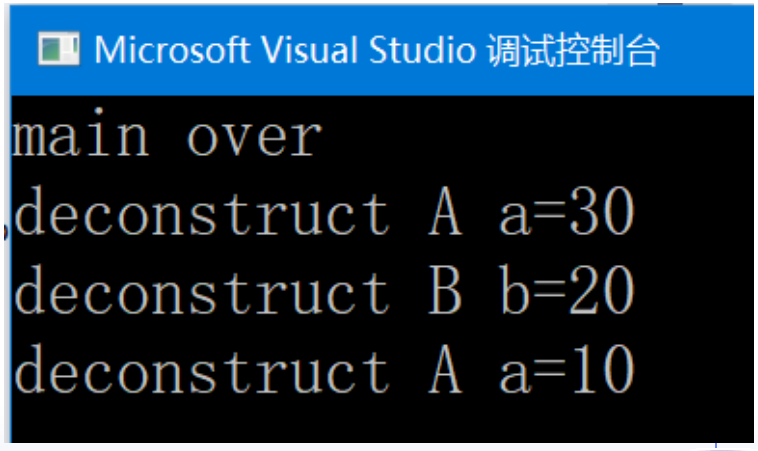
```
int main()
{
    C  c(10, 20, 30);
    cout << "main over" << endl;
    return 0;
}
```



4.8 对象的构造与析构

即使删除了 C类的析构函数，依然看得到对象成员的析构

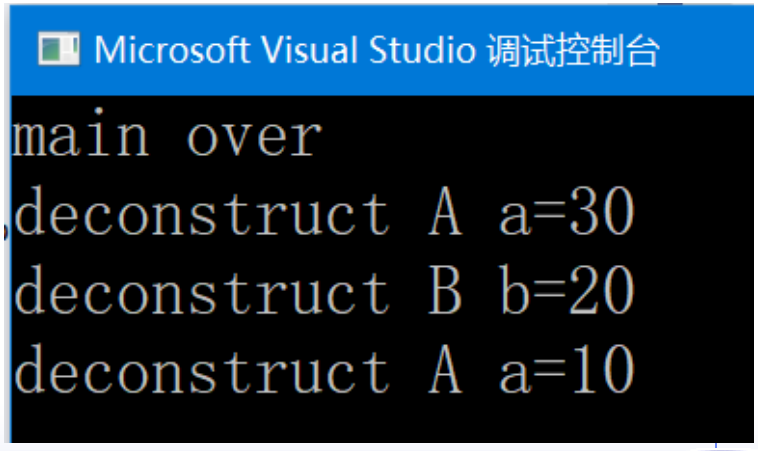
```
class C {
    A  oa;        B  ob;     A  oaa;
public:
    C(int v1,int v2,int v3):oa(v1),ob(v2),oaa(v3){}
};


int main()
{
    C  c(10, 20, 30);
    cout << "main over" << endl;
    return 0;
}
```



4.8 对象的构造与析构

即使删除了 C类的析构函数，依然看得到对象成员的析构

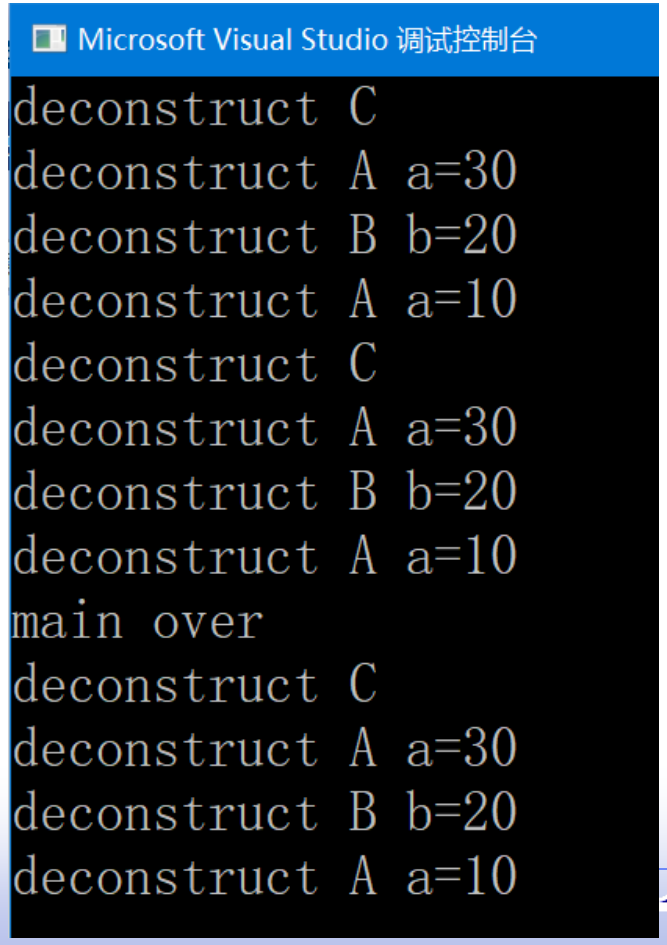
```
class C {
    A  oa;        B  ob;     A  oaa;
public:
    C(int v1,int v2,int v3):oa(v1),ob(v2),oaa(v3){}
};


int main()
{
    C  c(10, 20, 30);
    cout << "main over" << endl;
    return 0;
}
```



4.8 对象的构造与析构

析构并不释放对象本身的空间

```
int main()
{
    C  c(10, 20, 30);
    c.~C();
    c.~C();
    cout << "main over" << endl;
    return 0;
}
```



4.8 对象的构造与析构

总结

- 构造函数和析构函数与类名相同
- 两者的访问权限一般应为 public，否则无法自动调用
- 可多个构造函数，一个析构函数
- 析构函数无参数
- 都无返回类型
- 构造函数只能在定义对象时自动调用
- 析构函数可以自动和手动调用

何时自动析构，何时要手动析构？

类对象生命周期结束时，会自动调用析构函数。

4.9 类及对象的内存布局

对象的存储空间 与编译有关、与计算机硬件有关!

指明各数据成员的地址对齐方式

对象数组中对象之间的对齐方式

4.9 类及对象的内存布局

默认是松散方式：

一个 int 类型的成员，其地址被 4整除；

一个 double 类型的成员，其地址被 8整除；

一个对象的大小能够被成员的最大长度整除。

```
#pragma  pack(n)     // n=1 为紧凑方式
alignas( n )
```

4.9 类及对象的内存布局


```
struct MESSAGE {
    char   flag;        // 0
    int    size;        // 4
    char   buff[200];   // 8
    double  sum;        // 208
};
```

```
struct MESSAGE1 {
    double  sum;        // 0
    int    size;        // 8
    char   buff[200];   // 12
    char   flag;        // 212
};
```

```
cout << sizeof(MESSAGE)  << endl;
cout << sizeof(MESSAGE1) << endl;
cout << "MESSAGE  : offset of sum :" << offsetof(MESSAGE, sum) << endl;
cout << "MESSAGE1 : offset of flag :" << offsetof(MESSAGE1, flag);
```

```
Microsoft Visual Studio 调试控制台
216
216
MESSAGE  : offset of sum :208
MESSAGE1 : offset of flag :212
```

松散模式

4.9 类及对象的内存布局


```
struct alignas(16) MESSAGE2 {
    double  sum;     // 0
    int     size;    // 8
    char    buff[190]; //12
    char    flag;      //202
};
```

```
struct MESSAGE1 {
    double  sum;                 // 0
    alignas(16) int   size;      // 16
    char    buff[190];           // 20
    char    flag;                // 210
};
```

```
cout << sizeof(MESSAGE1)  << endl;
cout << sizeof(MESSAGE2) << endl;
cout << "MESSAGE1 : offset of flag :" << offsetof(MESSAGE1, flag);
cout << "MESSAGE2 : offset of flag :" << offsetof(MESSAGE2, flag);
```

```
Microsoft Visual Studio 调试控制台
224
208
MESSAGE1 : offset of flag :210
MESSAGE2 : offset of flag :202
```

4.10 内联、匿名类及位段

函数成员的内联 inline

- 什么是内联?
- 引入内联的目的是什么?
- 内联实现的原理是什么?
- 如何内联?
- 内联何时会失效?

4.10 内联、匿名类及位段

为何内联？

- 引入内联函数目的的是为了优化性能；
- 内联优化原理

  宏调用　VS　子程序调用
- 将被调用函数的函数体代码直接地整个插入到该函数被调用处，而不是通过call语句进行；
- 编译器进行“内联”时，不只是进行简单的代码拷贝，还需要做很多细致的工作。要处理被内联函数的传入参数、自己的局部变量，以及返回值等等。

4.10 内联、匿名类及位段

怎样内联?

- 在类体内定义的任何函数成员都会自动内联;
- 在类内或类外 使用inline保留字说明函数成员;

```
class   COMPLEX {
      double  r, v;
public:
      COMPLEX(double rp, double vp=0) {
            r= rp;    v=vp;
      }       //  自动称为内联函数
      inline double getr( );   //类内有 inline
      double getv( )
};
inline  double getv( ) {   return  v;   }   //类外有inline
```

4.10 内联、匿名类及位段

何谓内联失败？
为何失败？

内联失败：

如果函数有如下情况，则不会内联；

即使有 inline , 编译器也对inline 视而不见。

- 有分支类型语句

  分支、循环、开关、函数调用等；

- 在定义函数体之前，就已被调用的函数；
- 被定义为虚函数或者纯虚函数。

4.10 内联、匿名类及位段

华中科技大学

匿名类

定义类时，没有给出类名。

```
struct {
      int  x=0;
      int  randon( ) {
            return  x=(23*x+19)%101;
      }
} r={1};
```

- 不可能在类体外定义成员函数；无法写 类名::函数名
- 无法定义构造函数和析构函数；函数名与类名相同
- 定义匿名类的对象，对象的使用与非匿名类相同

4.10 内联、匿名类及位段

无对象的匿名联合

- 联合 → union
- 匿名 → 定义union时，未给出名字
- 无对象 → 定义union 时，未定义相应的对象

```
union A {  // 有名
    int x;
    int y;
}a;   // 有对象
```

```
static union {  // 匿名
    int x;
    int y;
};  // 无对象
```

```
union {  // 匿名
    int x;
    int y;
} temp;  // 有对象
```

4.10 内联、匿名类及位段

华中科技大学

```
static union { // 匿名
    int x;
    int y;
}; // 无对象
```

无对象的匿名联合

- 各个数据成员共享存储空间：
  即地址相同，类型可以不同
- 成员与联合本身的作用域相同
  函数内定义的 union：成员相当于函数内的局部变量；
  函数外定义的 union：成员相当于模块内的静态变量；

```
x=10;   cout<<y<<endl;  //输出 10
```

4.10 内联、匿名类及位段

```
static union {  // 匿名
    int x;
    int y;
};  // 无对象
```

无对象的匿名联合

类似于：

```
static int x;
static int &y=x;
```

- 各个数据成员共享存储空间；
- 成员与联合本身的作用域相同；
- 只能定义公开数据成员（即权限为 public）；
- 函数外的无对象的匿名联合，存储特性是static，union 前必须有 static；
- 函数内的无对象的匿名联合，union 可以有 static，也可以无static；数据成员分别对应 静态局部变量、局部变量。

位段 Bit field

交通路口有 红灯、绿灯、黄灯。如何表示出指示灯的状态？

定义三个变量，有何优点，有何缺点？

```
bool  red;          char  red;          int  red;
bool  green;        char  green;        int  green;
bool  yellow;       char  yellow;       int  yellow;
```

定义一个变量，有何优点，有何缺点？

```
int  trafficlights;
     #define   LIGHT_RED     0x01
     #define   LIGHT_GREEN   0x02
     #define   LIGHT_YELLOW  0x04
enum LIGHT { LIGHT_RED = 0x01,
        LIGHT_GREEN = 0x02, LIGHT_YELLOW = 0x04};
```

位段 Bit field

在结构体或者联合体中以位为单位定义成员变量所占的空间

```
struct LIGHT {
    int red:1    ;
    int green:1  ;
    int yellow:1 ;
} trafficlights;

trafficlights.red = 1;
trafficlights.green= 0;
```

位段 Bit field

在结构体或者联合体中以位为单位定义成员变量所占的空间

```
struct A{
    int i:3;          //i为位段成员
    int j:4;          //j为位段成员
    int k;
} a;                  // sizeof(a) =8;
a.i = 6;
a.j = 9;              //  一个字节中存放的信息
```

4.10 内联、匿名类及位段

位段：有几个二进制位来表示某种信息

```
class  SWITCH{
public:
   int   power:3;
   int   water:5;
   int   gas:4;
}
```

```
SWITCH temp;
temp.power = 6;
temp.water = 15;
temp.gas = 8;
```

7E

有何优点？

VS 三个独立变量 VS 一个变量

局部类

- 函数体中定义的类，只在定义它们的函数作用域内可见。

```
void outerFunction() {
    class LocalClass {     // 局部类定义
    private:          int data;
    public:
        LocalClass(int d) : data(d) { }
        void display() {
            cout << "Local class data: " << data << endl;
        }
    };
    LocalClass obj(42);   // 在函数内部使用局部类
    obj.display();
}
```

补充说明

在 .c 文件中 与 .cpp 中 struct 用法上的差别

- 在.cpp 文件中，struct 用法 与class 相同

  可以有构造函数、析构函数、权限说明等；

  差别：class 默认的访问权限是 private;

  struct 默认的访问权限是 public;

- 在 .c 文件中

  struct 中不能有函数；不能有权限说明；

  在定义结构变量时，要写成 struct *** x 之类的形式；

  当然可以用typedef , 将 struct *** 赋予新的名字。

构造函数的特殊形式

构造函数及示例

```
Student() = delete;
```

禁止产生构造函数

```
Student(const Student& s) = default;
```

使用缺省的构造函数

```
Student yang;      // error, 不得再使用无参构造函数
Student ma(xu);    // 使用缺省的以参数为对象的构造函数
                   // 不得再定义以对象为参数的构造函数
```

成员访问权限的突破方法

用强制类型转换
方法修改常变量

```
int xx;
cin >>xx;
const int yy = xx;
cout << “yy=” << yy<<endl;
*(int *)&yy = 30;
cout << “yy=” << yy<<endl;
// 显示30
```

用强制类型转换方法
访问私有成员？

```
int main(int argc, char* argv[])
{     Student  stu1, stu2;
      stu1.number = 123;
//    error C2248: ‘number' : cannot access private member declared in class 'Student'
}
*(int *)&stu1 =123;                  // 用强制类型转换 number
*((float *) &stu1+1)=99.5;           // 未直接私有成员number
                                     // 访问 score
```

成员访问权限的突破方法

用非正规方法访问私有成员

定义一个结构，字段与类相同，然后转换为该结构类型

```
struct  TROJAN_HORSE {
     int  number;
    float score;
    char  name[15];
};

((TROJAN_HORSE *) &stu1) ->score = 99.9;
```

有意识地绕开了编译器对访问权限的检查

总结

- 类的声明及定义
- 数据成员、函数成员的声明和定义
- 访问权限 private, protected, public
- 构造函数与析构函数
- new和delete
- 对象的构造与析构
- 隐含参数this

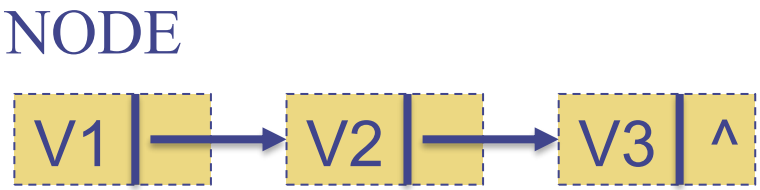
类的设计思考

栈（整型数的栈）的设计：

要求：用链表来存放各元素；

进栈、出栈、构造、析构等等操作

```
class  STACK {
        ????          *head;
public:
        STACK() { head = 0; }
        ~STACK();
        int push(int v);
        int pop(int& v);
};
```



类的设计思考


```
class NODE {
public:          // 在STACK中，要访问各成员
        int  val;
        NODE* next;
        NODE(int v);   // 在push 时，要生成一个新节点
};
 class  STACK {                    NODE
        NODE        *head;
 public:
        STACK() { head = 0; }
        ~STACK();
        int push(int v);
        int pop(int& v);
 };
```

问题：NODE中的
信息未能隐藏

类的设计思考


```
class  STACK {
      class NODE {
      public:
            int  val;
            NODE* next;
            NODE(int v);
      };                        // 可写成 class NODE {…..} *head;
      NODE     *head;
public:
            STACK() { head = 0; }
            ~STACK();
            int push(int v);
            int pop(int& v);
};
```

类的设计思考

实现信息隐藏

在非STACK函数中，不能直接使用 NODE

```
void f () { NODE  p(10)    // NODE 未申明的标识符
            STACK::NODE p(10);   // 无法访问私有类
           // 若有 public:  class NODE …… 则可访问
```

```
class  STACK {
     class NODE {
     public:
          int  val;    NODE* next;        NODE(int v);
     };
    NODE     *head;
public:
          STACK() { head = 0; }      ~STACK();
          int push(int v); int pop(int& v);  };
```

练习

试设计一个队列类 Queue, 并测试各项功能。
队列是一个先进先出(FIFO：First In First out) 的数据结构。
队列元素（整数）的存储：
　　用一个整数型的数组；
　　数组环形使用
对一个队列常用的操作有：
　　在队列尾增加一个元素
　　在队列头取一个元素
　　判断队列是否为空
　　判断队列是否已满
　　依次显示队列所有元素等