



华中科技大学

第4章 C++的类

许向阳

xuxy@hust.edu.cn





学习内容

- 4.1 类的声明及定义
- 4.2 成员访问权限及突破方法
- 4.3 内联、匿名类及位段
- 4.4 new和delete运算符
- 4.5 隐含参数this
- 4.6 对象的构造与析构
- 4.7 类及对象的内存布局





难点

思维的转变：

由习惯 C 语言的表达过渡到 C++ 的表达
习惯隐含参数 this

对象空间分配与 对象构造 的概念

对象析构 与 对象空间释放 的概念

new 与 malloc, delete 与 free 的差别





4.1 类的声明及定义

日常生活中表达方式的差异

XX号，YY栋，喻园小区，华中科技大学，洪山区，武汉市
武汉市，洪山区，华中科技大学，喻园小区，YY栋，XX号

2021年10月1日

Xu Xiangyang

我和你

1号/10月/2021年

Xiangyang Xu

You and I

把你的手伸给我，拉你起来
握住我伸给你的手，拉你起来

xu，上课了别废话
上课了别废话，xu

核心要素不变，但表达的顺序不同

强调的重点不同， 动作 or 完成动作的人





4.1 类的声明及定义

从 C 到 C++ 的思维转变

任务1: 初始化信息,

xu的信息是, “xuxy”, 21, 90, “very ...”

VS: xu, 初始化信息

“xuxy”, 21, 90, “very ...”

任务2: 显示评价, xu

VS xu, 显示评价

任务3: 复制信息, 把zhang复制xu中

VS xu, 复制zhang的信息

任务4: 释放remark指向的空间, xu

VS xu, 释放remark指向的空间

```
struct Student {  
    char    name[10];  
    short   age;  
    float   score;  
    char*   remark;  
};  
struct Student xu;  
struct Student zhang;
```

对象: xu

接收消息者: xu

消息: 初始化 显示评价

复制信息 释放空间

参数:





4.1 类的声明及定义

任务1: 初始化信息, xu, “xuxy”, 21, 90, “very ...”

```
init(&xu, “xuxy”, 21, 90, “very ...”);
```

VS: xu, 初始化信息

```
xu. init( “xuxy”, 21, 90, “very ...” );
```

C: 更强调操作

init, 初始化

assign, 复制

任务2: 显示评价, xu display_remark(&xu);

VS xu, 显示评价 xu. display_remark();

C++:

任务3: 复制信息, 把zhang复制xu中 assign(&xu, &zhang);

VS xu, 复制zhang的信息 xu. assign(&zhang);

强调对象

听到指令者

任务4: 释放remark指向的空间, xu freespace(&xu);

VS xu, 释放remark指向的空间 xu. freespace();

消息的接收者

xu, 可以初始化、显示评价、抄(复制)别人的信息、释放空间

以对象为中心, 操作是对象的行为





4.1 类的声明及定义

任务1: 初始化信息, xu, “xuxy”, 21, 90, “very ...”
init(&xu, “xuxy”, 21, 90, “very ...”);
init(“xuxy”, 21, 90, “very ...”, &xu);
init(&zhang,);
.....

VS: xu, 初始化信息
xu. init(“xuxy”, 21, 90, “very ...”);
zhang. init(.....);

C: 更强调操作, 核心词是 init

后面的参数可以有多种排列顺序, 体现不出参数之间的重要程度。

C++: 更强调对象 (消息的接收者, 即听到指令的人)

操作 可以看成是对象能够的任务, 对象的行为。

xu, 填上信息……; zhang, 填上信息……





4.1 类的声明及定义

类（类型）	对象（类型的实例）
数据成员	实例数据成员 静态数据成员
函数成员	实例函数成员 静态函数成员
构造函数	自定义的各种构造函数 默认的非参构造函数 默认的以对象有址引用为参数的构造函数 默认的以对象无址引用为参数的构造函数
析构函数	自定义的析构函数 默认的析构函数
赋值函数	自定义的赋值函数 默认的以对象有址引用为参数的赋值函数 默认的以对象无址引用为参数的赋值函数
隐含的参数	this





4.1 类的声明及定义

从 .c 到 .cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
#include <malloc.h>
struct Student {
    char    name[10];
    short   age;
    float   score;
    char*   remark;
};
```

```
struct Student xu;
struct Student zhang={0};
```

Q: sizeof(struct Student)= 20;
sizeof(xu) =?

在定义结构变量 xu 时，分配了 20 个字节，因而用 &xu 取地址，但空间中的内容没有初始化。

zhang 中的内容全部初始化为 0

name 10 字节	00 00 00 00
Age 2 字节	00 00
Score 4 字节	00 00 00 00
Remark 4 字节	00 00 00 00

xu

zhang





4.1 类的声明及定义

Q: 如何将 xu 中的数据置为:

“xuxy”, 21, 90, “very good student” ?

能否 `struct Student xu = { “xuxy”, 21, 90, “very ...” };` ?

`‘xuxy’,0,.....`

`21`

`90`



`“very good student”,0`

结构变量空间的初始化

对象空间的初始化

在只读数据区, 存在隐患
若通过 `xu.remark` 修改串, 后崩溃。

Q: 正确的做法是要自己申请remark空间。如何实现?

.c 文件可以; .cpp 文件不可。 .cpp 增加了const 检查





4.1 类的声明及定义

Q: 要自己申请remark空间, 如何实现?

xu: “xuxy”, 21, 90, “very good student” ?

```
void Student(struct Student *s, char* name, short age,
             float score, char* remark)
```

```
{
    int len;
    strcpy(s->name, name);
    s->age = age;
    s->score = score;
    len = strlen(remark);
    s->remark = (char *)malloc(len+1);
    strcpy(s->remark, remark);
}
```

C 的写法

```
Student(&xu, “xuxy”, 21, 90, “very good student”);
Student(&zhang, “zs”, 22, 95, “success”);
```





4.1 类的声明及定义

在类中定义**构造函数**，在定义对象时，**自动调用构造函数**。

```
struct Student {  
    char name[10];  
    short age;  
    float score;  
    char* remark;  
    Student(const char* name1, short age1, float score1,  
const char* remark1)  
    { int len;  
      strcpy(name, name1);  
      age = age1;          score = score1;  
      len = strlen(remark1);  
      remark = (char *)malloc(len + 1);  
      strcpy(remark, remark1);  
    }  
};
```

C++ 的写法

- 参数前加 const
- 与 C 的写法有哪些差异。

```
Student xu = { "xuxy", 21, 90, "very good student" };  
Student zhang = { "zs", 22, 95, "success" };
```





4.1 类的声明及定义

在C++中，在类中定义构造函数，在定义对象时，自动调用构造函数。

```
struct Student {  
    char name[10];  
    short age;          float score;  
    char* remark;  
    Student(const char* name, short age, float score,  
const char* remark)  
    { int len;  
      strcpy(this->name, name);  
      (*this).age = age;  
      Student::score = score;  
      len = strlen(remark);  
      this->remark = (char *)malloc(len + 1);  
      strcpy(this->remark, remark);  
    }  
};
```

Q: 参数名与数据
成员名相同;
如何区分两者?

类中数据成员的
三种限定方法

this->
(*this).

Student::

```
Student xu = { "xuxy", 21, 90, "very good student" };
```





4.1 类的声明及定义

显示 xu.remark 指向的串, 显示zhang.remark指向的串

```
void display_remark(struct Student* s)
{
    printf("remark : %s\n", s->remark);
}
display_remark(&xu);
display_remark(&zhang);
```

C
函数的定义

函数的调用

```
void display_remark()    // 写在类内
{
    printf("remark : %s\n", remark);
}
xu.display_remark();
zhang.display_remark();
```

C++
函数成员的定义

函数成员的调用

两种表示有什么差别? 对象的地址会自动作为第一参数





4.1 类的声明及定义

函数成员可以写在类中定义，如 `Student(...)`；也可以在类定义中声明，在类外定义，如 `display_remark`

```
struct Student {  
    char name[10];  
    short age;  
    float score;  
    char* remark;  
    Student(const char* name, short age, float score, const char*  
    remark)  
    { ..... }  
    void display_remark();  
};  
  
void Student::display_remark()  
{  
    printf("remark : %s\n", remark);  
}
```

体外定义函数成员，
在函数名前要加上类型的名称





4.1 类的声明及定义

Q: 如何将 zhang 中的内容复制到 xu 中?

xu = zhang; 隐含什么缺陷?

C 语言的赋值函数

```
struct Student * Assign(struct Student* t, const struct Student* s)
{
    strcpy(t->name, s->name);
    t->age = s->age;
    t->score = s->score;
    int len = strlen(s->remark) + 1;
    if (t->remark) free(t->remark);
    t->remark = (char *)malloc(len);
    strcpy(t->remark, s->remark);
    return t;
}
```

Assign(&xu, &zhang);





4.1 类的声明及定义

Q: 如何将 zhang 中的内容复制到 xu 中?

C++ 的赋值函数

```
Student & Student::Assign(const Student &s)
{
    strcpy(name, s.name);
    age = s.age;
    score = s.score;
    int len = strlen(s.remark) + 1;
    if (remark) free(remark);
    remark = (char *)malloc(len);
    strcpy(remark, s.remark);
    return *this;
}
```

```
xu.Assign(zhang);
```





4.1 类的声明及定义

Q: 如何将 zhang 中的内容复制到 xu 中?

```
C++ 的赋值函数  operator =  
Student& Student::operator =(const Student& s)  
{  
    strcpy(name, s.name);  
    age = s.age;  
    score = s.score;  
    int len = strlen(s.remark) + 1;  
    if (remark) free(remark);  
    remark = (char*)malloc(len);  
    strcpy(remark, s.remark);  
    return *this;  
}
```

xu =zhang; xu =zhang =yang;

用 operator = 代替了函数 Assign.

赋值运算符重载





4.1 类的声明及定义

在 程序运行结束前，如何释放 remark 指向的空间？

```
void _Student(struct Student* s)
{
    if (s->remark) {
        free(s->remark);
        s->remark = NULL;
    }
}

_Student(&xu);      _Student(&zhang);
```

C 的表示法

```
~Student()
{
    if (remark) {
        free(remark);
        remark = NULL;
    }
}
```

C++ 的表示法

在 xu、zhang 的生命周期结束时，会自动调用析构函数。

析构函数并不释放 对象本身的空间





4.1 类的声明及定义

析构造函数

只有一个析构造函数；
函数名与类型相同，前面加 ~；
无返回；
无参数；
可以写析构造函数的调用语句；
析构造函数并不释放 对象本身的空间；
只是做释放体外空间等工作。

构造函数

有多个构造函数；
函数名与类型相同；
无返回；
多种参数，以支持重载；
只能自动调用；
构造函数并不申请 对象本身的空间；
只是做空间初始化等工作。





4.1 类的声明及定义

一个完整的例子

Student.h	定义类型
Student.cpp	函数成员的定义
Main.cpp	主程序





4.1 类的声明及定义

一个完整的例子 Student.h

```
struct Student {  
    char  name[10];  
    short  age;  
    float score;  
    char* remark;  
    Student(const char* name, short age, float score,  
            const char* remark);  
    Student();           // 无参的构造函数  
    void display_remark();  
    int get_age();  
    Student& operator =(const Student& s);  
    ~Student();  
};
```





4.1 类的声明及定义

一个完整的例子 Student.cpp

```
Student::Student(const char* name, short age, float score,
                 const char* remark)
{
    int len;
    if (name) {
        len = strlen(name);
        if (len <= 9) strcpy(this->name, name);
        else { cout << "name is too long; cut string" << endl;
                memcpy(this->name, name, 9);
                this->name[9] = 0; }
    }
    else this->name[0] = 0;
    (*this).age = age;
    Student::score = score;
    if (remark) { len = strlen(remark);
                 this->remark = (char*)malloc(len + 1);
                 strcpy(this->remark, remark);
    } else this->remark = NULL;
}
```





4.1 类的声明及定义

一个完整的例子 Student.cpp

```
Student::Student()  
{  
    name[0] = 0;  
    age = 0;  
    score = 0;  
    remark = NULL;  
}
```

```
Student::~~Student()  
{  
    if (remark) {  
        free(remark);  
        remark = NULL;  
    }  
    cout << "deconstruct :" << name << endl;  
}
```





4.1 类的声明及定义

一个完整的例子 Student.cpp

```
void Student::display_remark()  
{ cout<<"remark : "<< remark<<endl; }
```

```
int Student::get_age( )  
{ return age; }
```

```
Student& Student::operator =(const Student& s)  
{  
    strcpy(name, s.name);  
    age = s.age;  
    score = s.score;  
    int len = strlen(s.remark) + 1;  
    if (remark) free(remark);  
    remark = (char*)malloc(len);  
    strcpy(remark, s.remark);  
    return *this;  
}
```





4.1 类的声明及定义

一个完整的例子 main.cpp

```
# include "Student.h"
#include <iostream>
using namespace std;
int main()
{
    Student xu("xuxy123456789012", 21, 90, "very good student");
    Student zhang(0, 22, 95, 0);
    Student yang;    // 无参构造函数
    Student li("lishi", 20, 85, "good");
    Student ma(li);    // 以对象为参数的构造函数，实现浅拷贝
    yang = zhang = xu;
    cout << "xu . name " << xu.name << endl;
    cout << "zhang . score = " << zhang.score << endl;
    cout << "yang . remark =" << yang.remark << endl;
    zhang.display_remark();
    int x=zhang.get_age();
    cout << "zhang . age = " << x << endl;
    return 0;
}
```





4.1 类的声明及定义

运行结果

选定 Microsoft Visual Studio 调试控制台

```
name is too long; cut string
xu . name xuxy12345
zhang . score = 90
yang . remark =very good student
remark : very good student
zhang . age = 21
deconstruct :lishi
deconstruct :xuxy12345
deconstruct :xuxy12345
deconstruct :xuxy12345
```





4.1 类的声明及定义

简单类型参数的构造函数

无参数的构造函数

```
Student::Student(const char* name, short age, float score,  
                 const char* remark)
```

```
Student::Student()
```

默认的以对象有址引用为参数的构造函数

默认的以对象无址引用为参数的构造函数

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





4.1 类的声明及定义

简单类型参数的构造函数

无参数的构造函数

```
Student::Student(const char* name, short age, float score,  
                 const char* remark)
```

```
Student::Student()
```

默认的以对象有址引用为参数的构造函数

默认的以对象无址引用为参数的构造函数

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





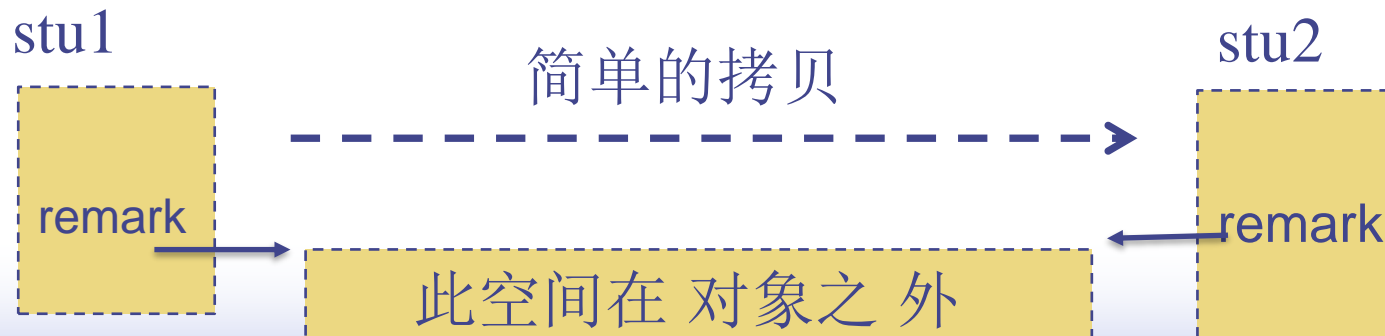
4.1 类的声明及定义

默认的以对象有址引用为参数的构造函数

Student stu2(stu1);

简单地将 stu1 中变量空间中的内容都拷贝到 stu2 的变量中。

浅拷贝 存在的问题



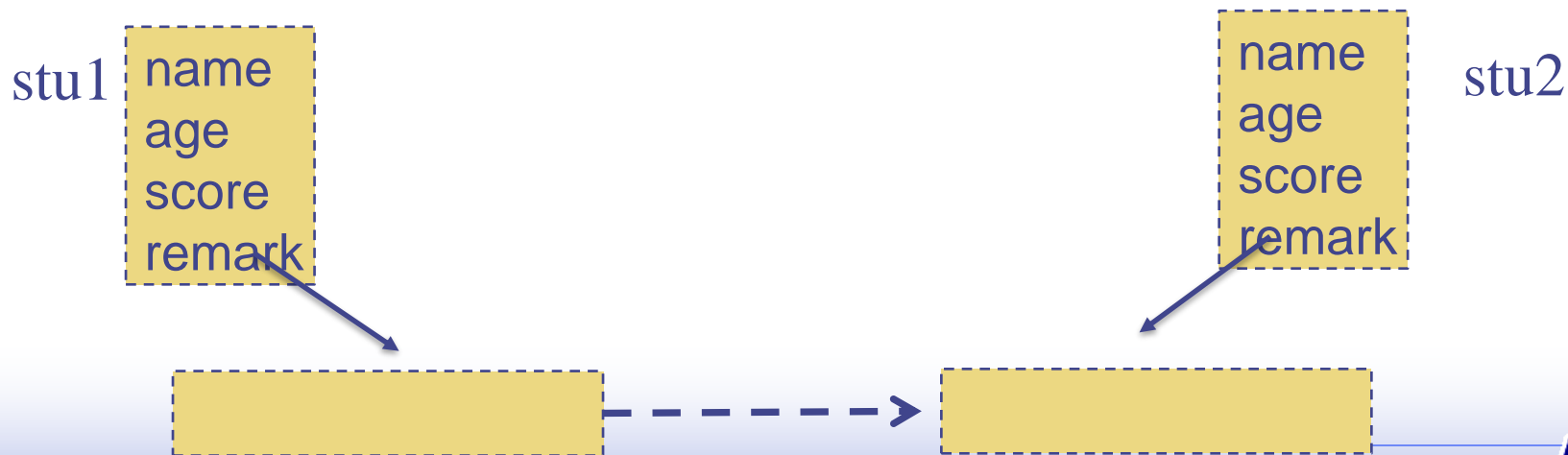


4.1 类的声明及定义

为实现深拷贝，自己编写以对象有址引用为参数的构造函数

```
Student (const Student &a)
{ ..... }
```

```
Student stu2(stu1);
```





4.1 类的声明及定义

Q: 以对象有址引用、无址引用为参数的构造函数，写法上有什么差别？

```
Student(const Student& s)
{
    .....
}
```

```
Student::Student(Student&& s)
{
    cout << "&& construct : temp object parameter" << endl;
    strcpy(name, s.name);
    age = s.age;
    score = s.score;
    remark = s.remark;
    s.remark = NULL;
}
```





4.1 类的声明及定义

构造函数及示例

```
Student(const char* name, short age, float score,  
        const char* remark);
```

```
Student();
```

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

```
Student(Student && s);
```

```
Student xu("xuxy", 21, 90, "very good");
```

```
Student yang;
```

```
Student ma(xu);
```

```
Student ppqq = CreateStudent();
```

```
Student CreateStudent()    // 普通函数，非Student的函数成员  
{  
    Student temp("temp", 0, 0, NULL);  
    return temp;  
}
```





4.1 类的声明及定义

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





4.1 类的声明及定义

构造函数

无参构造函数

类名();

以对象有址引用为参数的构造函数

类名(const 类名 &);

以对象无址引用为参数的构造函数

类名(类名 &&);

默认的三个构造函数的功能（以及取代）

析构函数

析构函数

~类名();

默认的析构函数的功能

赋值函数

以对象有址引用为参数的赋值函数

类名 & operator =(const 类名 &);

以对象无址引用为参数的赋值函数

类名 & operator =(类名 &&);

默认的二个赋值函数的功能

隐含的参数

this





4.1 类的声明及定义

构造函数

- 函数名与类名相同
- 没有返回类型 (void 也不行)
- 可以有多个构造函数，但参数应有所不同

➤ 自动调用

执行到定义对象的代码时，被自动调用

对于全局对象，在执行main()之前被自动调用

- 对象的初始化，给成员变量赋值





4.1 类的声明及定义

构造函数

- 当类中未定义构造函数时，编译器提供三个默认的构造函数：
 - 无参构造函数、以对象引用为参数的构造函数
 - 以对象无址引用为参数的构造函数
- 当类中定义了构造函数时，不再提供默认的非参构造函数；此时，若只定义一个对象，则必须定义一个非参的构造函数；但此时依然提供默认的以对象引用为参数的构造函数；
- 默认的以对象为参数的构造函数，实现的方式是简单拷贝；（浅拷贝），对于指针成员，则会指向相同的空间。





4.1 类的声明及定义

向 安全 迈进

类的函数/数据成员的使用受到限制

增强保密特性 隐藏

简化外部接口 封装





4.1 类的声明及定义

类保留字： class、struct 或 union 可用来声明和定义类。

类的定义：

```
class 类型名{  
    [private:]  
        私有成员声明或定义;  
    protected:  
        保护成员声明或定义;  
    public:  
        公有成员声明或定义;  
};
```

类的实现： 类的函数成员的实现，即定义类的函数成员。

类的申明： class 类型名;





4.2 成员访问权限及突破方法

- ◆ private、protected和public 标识每一区间的访问权限
- ◆ private、protected和public 可以多次出现；
- ◆ 同一区间内可以有数据成员、函数成员和类型成员；
- ◆ 习惯上按类型成员、数据成员和函数成员分开；
- ◆ 成员可以任意顺序出现
- ◆ 函数成员的实现既可放在类体外，也可在类体中；
- ◆ 若函数成员在类的定义体外实现，
使用“**类名::**”指明该函数成员所属的类；
- ◆ 类的定义体花括号后要有**分号作为结束标志**。





4.2 成员访问权限及突破方法

private: 私有成员

仅可被本类的函数成员访问

不能被派生类、其它类和普通函数访问

protected: 受保护成员

可被本类和派生类的函数成员访问

不能被其它类函数成员和普通函数访问

public: 公有成员

可被任何函数成员和普通函数访问

class定义的类缺省访问权限为private

struct和union定义的类缺省访问权限为public。





4.2 成员访问权限及突破方法

```
class Student
{
private:
    int number;
    float score;
public:
    char name[15];
public:
    Student(int number, char *name, float score); //声明
    Student( );
    void Modify(float score) //声明与定义
    {
        Student::score=score;
    }
    void Print(); // 在体外定义init 与 Print
};
```





4.2 成员访问权限及突破方法

```
int main(int argc, char* argv[])
{
    Student stu1, stu2;

    strcpy_s(stu1.name, "xu" );

    stu1.number = 123;

    // error C2248: 'number' : cannot access private member declared in
    // class 'Student'

    return 0;
}
```





4.2 成员访问权限及突破方法

用强制类型转换
方法修改常变量

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;
*((float *) &stu1 + 1) = 99.5;
```

```
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 number
// 未直接私有成员 number
// 访问 score
```





4.2 成员访问权限及突破方法

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





4.3 内联、匿名类及位段

4.3.1 函数成员的内联

- 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？





4.3 内联、匿名类及位段

为何内联？

- 引入内联函数目的是为了优化性能；
- 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。





4.3 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
```





4.3 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。





4.3 内联、匿名类及位段

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





4.3 内联、匿名类及位段

4.3.2 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
```



4.3 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

➤ 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
```





4.3 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。





4.3 内联、匿名类及位段

4.3.3 局部类及位段成员

- 类体中定义类;
- 函数体中定义类;





4.3 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;
```

1 0 0 0	0 1 1 1 1	1 1 0
---------	-----------	-------

7E

有何优点？

VS 三个独立变量 VS 一个变量





4.4 new和delete运算符

`new` : 申请空间
 自动调用构造函数初始化

`delete` : 自动调用析构函数
 释放空间

`malloc`: 申请空间

`free` : 释放空间





4.4 new和delete运算符

- 4.4.1 简单类型及单个对象的内存管理
- 4.4.2 复杂类型及对象数组内存管理





4.4 new和delete运算符

```
class Student {  
private:  
    int number;  
    char *name;    // 原来是 char name[15];  
    float score;   // 在构造函数中要为name分配空间  
public:  
    Student(int number, char *name, float score);  
    ~Student();  
    void Modify(float score)  
    {    this->score=score;    }  
    void Print();  
};
```





4.4 new和delete运算符

- 使用 malloc

```
char *name;
```

```
#include <malloc.h>
```

```
name = (char *)malloc(15);
```

- 使用 new

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

生成一个类对象，怎么办？

```
class ARRAY { .... };
```

```
ARRAY *p;
```

如何为 **p** 分配指向的空间？

p 指向的**ARRAY**又如何初始化？





4.4 new和delete运算符

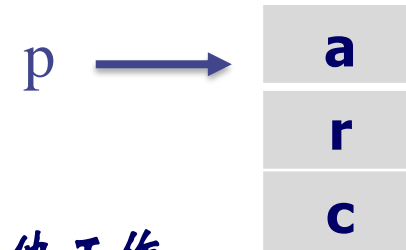
```
class ARRAY{                                //class体的缺省访问权限为private
private:
    int  r, c;                             // 行数, 列数
    int *a,                                // 数组元素存放区
public:
    ARRAY(int x, int y)  {
        r=x; c=y;
        a=new int[x*y];                    // int型可用malloc
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



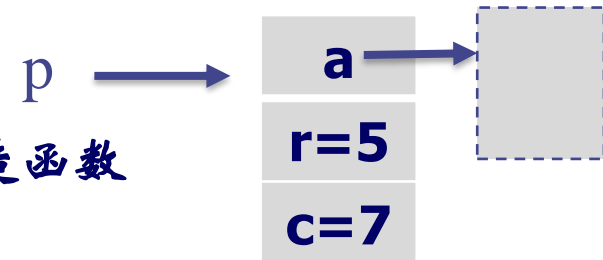


4.4 new和delete运算符

```
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	0xfefefeee
- r	-17891602
- c	-17891602

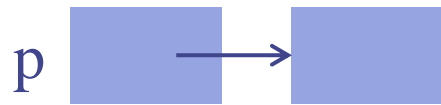
= p	0x00380830
⊕ a	0x00380ac0
- r	5
- c	7



4.4 new和delete运算符

◆ new <类型表达式>

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



数组指针

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

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

```
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], 都是ARRAY对象





4.4 new和delete运算符

- 对象数组初始化

```
int    x[5];    // 整数数组  x[0], x[1], x[2]...  
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}, .....};
```

```
int    x[5]={0, 1};
```

```
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





4.4 new和delete运算符

- 对象数组初始化

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

```
ARRAY  q[5]={ {对象q[0]的构造参数}, .....};
```

```
ARRAY  q[5]={ARRAY(对象q[0]的构造参数), .....};
```





4.4 new和delete运算符

```
int    ia[10];    // 整数数组    ia[0], ia[1], ia[2]...  
ARRAY  oa[10];    // 对象数组    oa[0], oa[1], oa[2]...
```

```
int     *ap = new int[10];    // 数组指针  
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。



4.4 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
```





4.4 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





4.4 new和delete运算符

对于简单类型(没有构造、析构造函数)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。





4.5 隐含参数this

```
Student(int number, char *name, float score);  
void Modify(float score)  
{    this->score=score;    }
```

```
Student wang(123, "wangji",97);  
Student xu(125,"xuxiangyang",90);
```

```
    xu.Modify(95);
```

P1:

```
    wang.Modify(100);
```

P2:

P1 断点地址

xu的地址

95

有两个对象，是修改哪个对象中的变量？

VS2019 用 ECX来传递对象的地址





4.5 隐含参数this

```
void Modify(float score) {  
    score=score;  
}
```

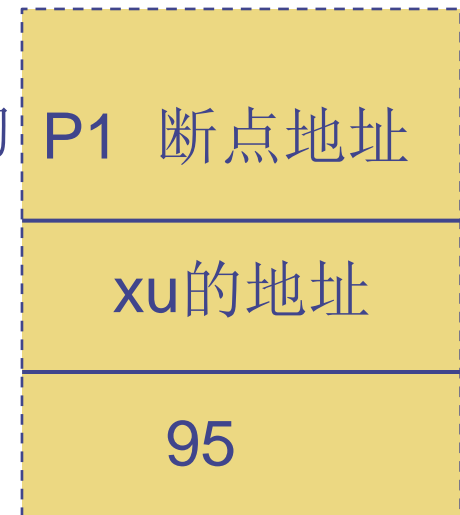
```
void Modify(float score) {  
    this->score=score;  
    // (*this).score=score; 等价语句  
    // Student::score=score;  
}
```

```
Student xu(125,"xuxiangyang",90);
```

```
    xu.Modify(95);
```

P1:

使用**this**访问数
据成员，区分函
数参数的访问





4.5 隐含参数this

- ◆ **this**是一个隐含的const指针，不能移动或对该指针赋值。

this = p; // =左操作数必须是左值

// 等同： 类名 * const this;

- ◆ 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。





4.5 隐含参数this

```
class TREE{
    int value;
    TREE *left, *right;
public:
    TREE (int);
    ~TREE( );
    const TREE *find(int) const;
};

TREE::TREE(int value){
    this->value=value;
    left=right=0;
}

const TREE* TREE::find(int v) const {
    if(v==value) return this;
    if(v<value) return left!=0?left->find(v):0;
    return right!=0?right->find(v):0;
}
```

在树中找节点值为v 的节点
this 相当于树的根指针

//this 类型: TREE * const this
//this 类型: const TREE * const this

//隐含参数this指向要构造的对象
//等价于TREE::value=value
//C++提倡空指针NULL用0表示

//this指向调用对象
//this指向找到的节点
//查左子树
//查右子树, 调用时新this=left





4.6 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 常成员（只读成员）、引用成员、静态成员、对象成员，或者是一个该类的基类，如何初始化？按什么顺序初始化？





4.6 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？





4.6 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，用{ } 代替()  
        A *p = new A(...);  
        A &q=a;  
};
```





4.6 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;  
        int  &y {x};    // y 中的内容为 x的地址  
        const int z{20};  
        int  u=30;  
        A    a{.....};  
};
```





4.6 对象的构造与析构

类成员不能像一般的变量那样用 () 来初始化

```
class A {  
    public:  
        int x1 ;           // x1 未初始  
        int x2 (10);         // Error; 正确: x2=10; x2{10};  
        int &y1;             // y1未初始化  
        int &y2 (x2);         // error 正确: &y2=x2; &y2{x2};  
        const int u1 ;      // u1 未初始  
        const int u2 {20};  // u2 初始化为 20  
};
```

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





4.6 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(...)  
        { .....}
```

```
};
```

- 对象指针可在构造函数中初始化
- 静态成员在类外初始化

类中的各种数据成员如何初始化？

在构造函数体前初始化：

只读成员、引用成员、对象成员





4.6 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。

Why?

类比：普通的常变量、引用变量（即不是一个类的数据成员），是应该在申明时就初始化的。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。





4.6 对象的构造与析构

```
Class A { .....};
```

类中的各种数据成员如何初始化？

```
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
- 用构造函数的形式给各变量赋初值，如 y(t)
- 可以用列表的形式{} 赋初值，如 y{t}
- 不能采用 = 来初始化 : y=t // error





4.6 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;
```





4.6 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;
    }
};
```





4.6 对象的构造与析构

```
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();
```

```
};
```

类中含有其他类的对象
组合类对象的初始化





4.6 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(name,name1);  
    score=score1;  
}
```

构造函数体之前初始化，冒号分隔





4.6 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；
- 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，此时必然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)；





4.6 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；
- 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





4.6 对象的构造与析构

析构函数

对象撤销时，释放空间或其他处理

- 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





4.6 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。





4.7 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!

`#pragma pack(n)`

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

`#pragma pack(1)` // 紧凑方式

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。



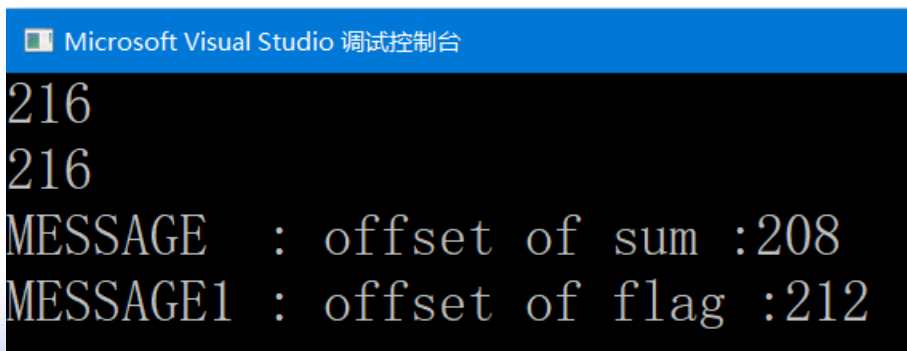


4.7 类及对象的内存布局

```
struct MESSAGE {  
    char  flag;  
    int   size;  
    char  buff[200];  
    double sum;  
};
```

```
struct MESSAGE1 {  
    double  sum;  
    int     size;  
    char    buff[200];  
    char    flag;  
};
```

```
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





4.7 类及对象的内存布局

```
struct alignas(16) MESSAGE2 {  
    double sum;  
    int size;  
    char buff[200];  
    char flag;  
};  
  
struct MESSAGE1 {  
    double sum;  
    int size;  
    char buff[200];  
    char flag;  
};
```

```
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

216

224

MESSAGE1 : offset of flag :212

MESSAGE2 : offset of flag :212





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 默认访问权限是 private；
struct 默认访问权限是 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。



总结



华中科技大学

- 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

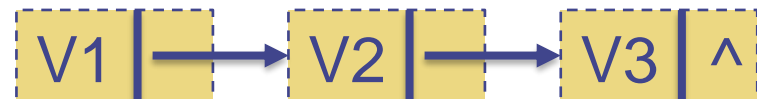
栈（整型数的栈）的设计：

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);  
};
```

NODE



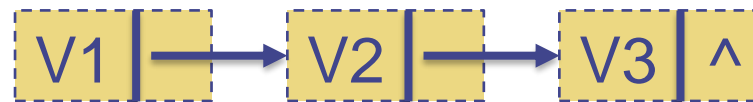


类的设计思考

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

NODE



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 未声明的标识符  
          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



华中科技大学

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

