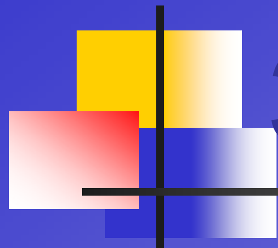
华中科技大学计算机学院

李瑞轩

第3章 C++的类

- 3.1 类的声明及定义
- 3.2 访问权限
- 3.3 内联及位段
- 3.4 new和delete
- 3.5 隐含参数this
- 3.6 对象初始化
- 3.7 类的存储空间



3.1 类的声明及定义

3.1 类的声明及定义

- 类保留字：class、struct或union可用来声明和定义类。struct和union是C原有的保留字，class是C++新增加的。
- 构造函数：是函数名和类名相同的函数成员。用于产生对象后初始化对象，如为对象申请内存等。无返回类型，可以重载。
- 析构函数：是函数名和类名相同且带波浪线的无参函数成员。用于销毁对象，销毁前先释放对象申请的内存。无返回类型，不得重载。
- 如果类没有自定义的构造函数和析构函数，则C++为类提供缺省的无参构造函数和析构函数。

- 【例3.1】定义字符串类型和字符串对象。

```
#include <alloc.h>
struct STRING
{   typedef  char *CHARPTR;              //定义类型成员
    CHARPTR  s;                          //定义数据成员
    int  strlen( );                      //定义函数成员
    STRING(CHARPTR);                     //定义构造函数
    ~STRING( );                          //定义析构函数
};
int STRING::strlen( )                    //用运算符::在类体外定义
{   for(int  k =0; s[k]!=0; k++);
     return k;
}
```

```
STRING::STRING(char *t)          //用运算符::在类体外定义
{   for(int  k =0; t[k]!=0; k++);
     s=(char *)malloc(k+1);
     for(k=0; (s[k]=t[k])!=0; k++);
}
STRING::~STRING( ) {  free(s); }//用运算符::在类体外定义
struct  STRING          x("simple");
void main( ){
     STRING  y("complex"), *z=&y;
     int  m=y.strlen( );
     m=z->strlen( );
}
```

3.1 类的声明及定义

- **成员访问：**.，->，.*，->*，::。其中::主要用于访问基类或静态成员。
- **构造函数：**唯一不能被显式调用成员函数。定义该类的变量或常量时隐式调用。
- **析构函数：**既能被显式调用，也能被隐式调用。如果程序非正常退出，析构函数隐式调用可能没机会执行，资源可能没有释放。隐式调用在函数返回前调用。

3.1 类的声明及定义

- **exit退出**：隐式调用的析构函数不能执行。局部对象的资源不能被释放，全局对象的资源可以被释放，即exit退出main后执行收工函数。
- **abort退出**：隐式调用的析构函数不能执行。局部和全局对象的资源都不能被释放，即abort退出main后不执行收工函数。
- **return返回**：隐式调用的析构函数得以执行。局部和全局对象的资源被释放。提倡使用return。
  - **void main( ){ …; if (error) return; …;}**
  - 如果用abort和exit，则要显式调用析构函数。
  - 使用异常处理，隐式调用的析构函数会执行。

3.1 类的声明及定义

- 反复析构：同一对象的析构函数被反复调用。Unix, Windows的内存释放可以反复进行，但某些资源如设备、文件不能反复析构。例如，既显式调用又隐式调用析构函数导致反复析构。
- 解决方案：对象内部成员设置标志，如可利用指针是否为空判断内存是否已经释放。释放完毕立即置空标志。

【例3.3】定义能防止反复析构的字符串类。

```
#include <string.h>
#include <alloc.h>
#include <iostream.h>
struct STRING{
    char *s;      STRING(char *);      ~STRING( );
};
STRING::STRING(char *t){
    s=(char *)malloc(strlen(t)+1);
    strcpy(s,t);
    cout<<"CONSTRUCT: "<<s;
}
```

```
STRING::~STRING( ){
    if(s==0)  return;    //去掉本行，s1析构两次
    cout<<"DECONSTRUCT: "<<s;
    free(s);   s=0;          //在析构后置析构标志
}
void main(void){
    STRING s1("String varible 1\n");
    STRING s2("String varible 2\n");
    STRING("Constant\n");
    cout<< "RETURN\n";
}
```

3.1 类的声明及定义

- 构造顺序：按定义的顺序构造。
- 析构顺序：在退出对象的作用域的时析构，析构时按定义的逆序析构。
  - 常量对象的作用域局限于当前表达式语句，语句结束时析构
  - 局部非静态对象的作用域局限于当前函数，由当前函数析构
  - 静态对象和全局对象的作用域为整个程序，由收工函数析构。

3.1 类的声明及定义

例3.3的输出：

```
CONSTRUCT: String varible 1
CONSTRUCT: String varible 2
CONSTRUCT: Constant
DECONSTRUCT: Constant
RETURN
DECONSTRUCT: String varible 2
DECONSTRUCT: String varible 1
```

逆序析构

语句结束，析构常量

3.2 访问权限

- 封装机制规定了数据成员、函数成员和类型成员的访问权限。三类权限：
  - private:私有成员，仅限于由本类成员访问，派生类的成员不能访问。其他类的成员函数和C的函数不能访问。
  - protected: 保护成员，由本类和其派生类的成员访问。其他类的成员函数和C的函数不能访问。
  - public: 公有成员，任何实体均可访问。
- 注意：成员继承到派生类后，访问权限可能发生变化。如果一个(成员或C的)函数为该类的友员，则该函数可访问该类的任何成员。通过强制类型转换可突破访问权限。

3.2 访问权限

- 缺省权限：进入由class定义的类体时，缺省的访问权限为private；进入由struct和union定义的类体时，缺省的访问权限为public(同C兼容)。
- 访问形式：取值，赋值，引用、调用、取地址，取内容等。
- 构造函数和析构函数可为任何访问权限，访问时要遵守相应权限。注意：构造函数在定义对象时隐式调用，不能显式调用，也不能取地址，否则通过函数指针便可显式调用。

3.2 访问权限

- 【例3.4】为女性定义FEMALE类。

```
class  FEMALE{ //缺省访问权限为private
    int  age;          //私有的，自己的成员和友员可访问
public:                //访问权限改为public
    typedef char *NAME;//公有的，都能访问
protected:                         //访问权限改为protected
    NAME nickname;//自己的和派生类成员、友员可访问
    NAME getnickname( );
public:                      //访问权限改为public
    NAME name;         //公有的，都能访问
};
```

3.2 访问权限

```
FEMALE::NAME FEMALE::getnickname( ){
      return  nickname; //自己的函数成员访问自己的成员
}
void  main(void){ //main没有定义为类FEMALE的友员
      FEMALE  w; FEMALE::NAME(*f)( ); FEMALE::NAME n;
      n=w.name;      //任何函数都能访问公有name
      n=w.nickname; //错误，main不得访问保护成员
      n=w.getnickname( );//错误，main不得调用保护成员
      int  d=w.age;     //错误，main不得访问私有age
      f=&w.getnickname;//错误，main不得取保护成员地址
}
```

3.3 内联及位段

- 函数成员的内联说明：
  - 在类体内定义的任何函数成员都会自动内联。
  - 使用inline保留字说明
  - 内联失败见第2章：有分支类语句、定义在使用后，取函数地址，定义(纯)虚函数。
- 内联函数成员的作用域局限于当前程序文件。匿名类的函数成员只能在类体内定义(从而内联)。局部于函数的类的函数成员只能在类体内定义(从而内联)，某些编译器不支持局部类。

【例3.7】定义一个复数类型。

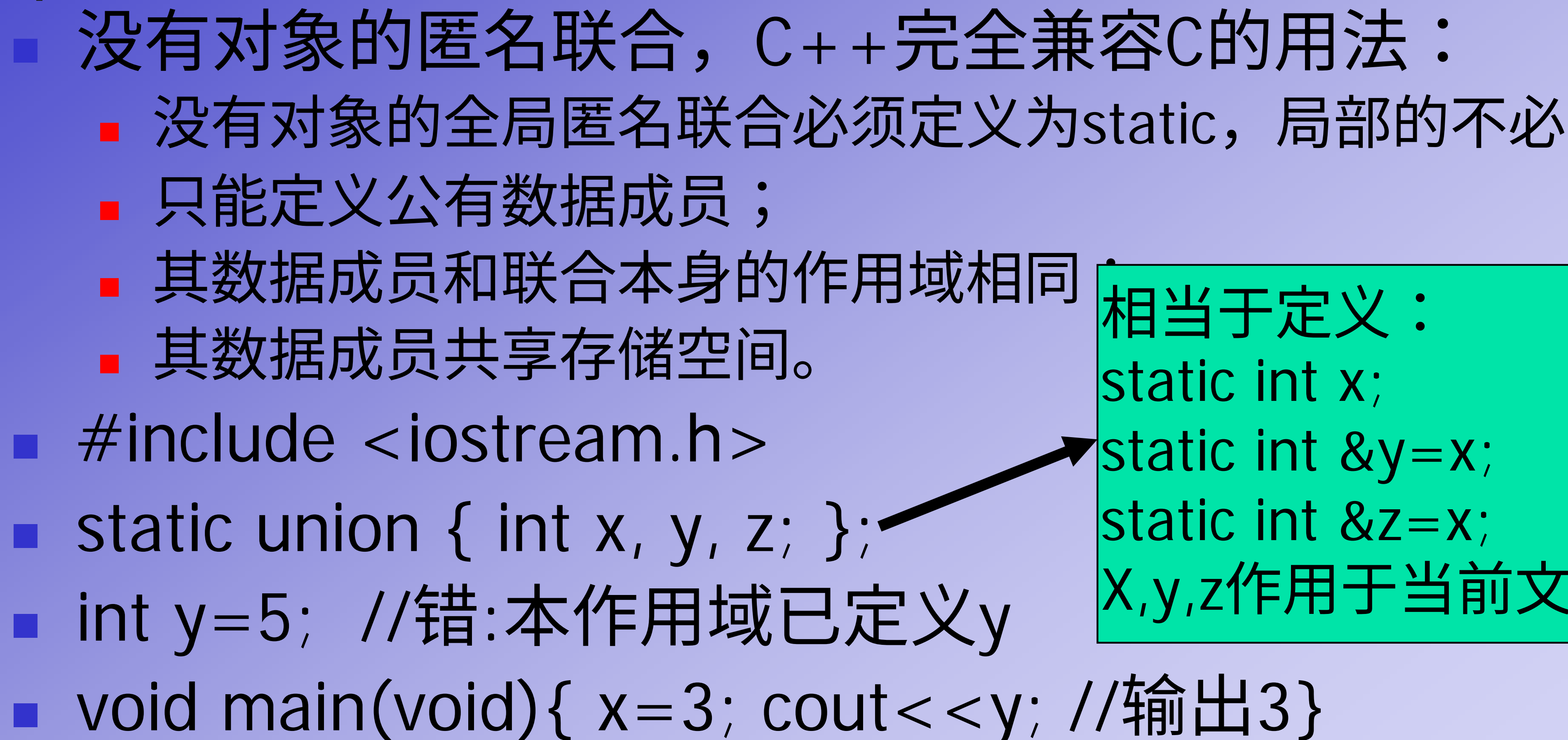
```
class COMPLEX{
    double r, v;
public:
    COMPLEX(double rp, double vp){ r=rp; v=vp; } //自动内联
    inline double getr( ); //inline保留字可以省略
    double getv( );
};
inline double COMPLEX::getv( ){return v; }     //定义内联
void main(void){
    COMPLEX c(3, 4);
    double  r=c.getr( );          //getr的函数体未定义，内联失败
    double  v=c.getv( );          //内联成功
}
inline double COMPLEX::getr( ){ return r; }     //定义内联失败
```

3.3 内联及位段

- 【例3.8】定义产生随机数的匿名类。

```
#include <iostream.h>
struct{//匿名类只能在体内定义(内联)函数成员random
    int x;
    int random( ) { return x=(23*x+19)%101; }
}r={1};
void main(void){
    for(int i=0; i<8; i++)cout<<r.random( )<<"\n";
}
```

3.3 内联及位段



- 【例3.10】定义一个产生随机数的局部类。

```
#include <iostream.h>
void main(void){
    int  x=0;
    class  RANDOM{
    int x;     //函数成员只能在类体内定义(从而内联),
    public:    //因为C语言不能在main中嵌套定义函数
        int random( ){ return x=(23*x+19)%101;}
        RANDOM(int s){ x=s; }
    } r(x+1);  //任意表达式初始化对象r
    for(x=0; x<10; x++) cout<<r.random( )<<"\n";
}
```

3.3 内联及位段

- 位段成员：按位分配内存的数据成员。
  - class、struct和union都能定义位段成员
  - 位段类型必须<long: char, short, int, enum
  - 相临位段成员分配内存时可能出现若干成员共同处于一个字节的情况
- 位段用法：
  - 生产控制的各种开关，指示灯等
  - 布尔运算，图象处理
  - 位段成员不能取地址，现代计算机按字节编址。

- 【例3.11】定义生产控制的开关和指示灯类。

```
#include <iostream.h>
enum ALPHA{a, b, c, d};
class  SWITCH{//总位数28位，紧凑存储4字节
    int  power:3;
    int  water:5;
    int  gas:5;
    int  oil:6;
    int  start:1;
    int  alarm:3;
    ALPHA  stop:1;
    int  manual:4;
};
```

3.3 内联及位段

```
union  STATE{//总位数9位，采用紧凑方式时其字节数为2
    int  speed:9;//最大成员speed的单元为所有成员共享
    unsigned  run:2;
};
void main(void){
    cout<<"The size of int is "<<sizeof(int)<<" bytes\n";
    cout<<"The size of SWITCH is "<<sizeof(SWITCH)<<" bytes\n";
    cout<<"The size of STATE is "<<sizeof(STATE)<<" bytes\n";
}
```

3.4 new和delete

- 内存管理：分配malloc、new；释放free、delete。
- 内存分配： malloc为函数，参数为值表达式，分配后内存初始化为0；new为运算符，操作数为类型表达式，先底层调用malloc，然后调用构造函数。
- 内存释放： free为函数，操作数为值表达式，直接释放内存；delete为运算符，参数为值表达式，先调用析构函数，然后底层调用free。
- 如果为简单类型分配释放内存，new和malloc、 delete和free则没有区别，可以混合使用。

3.4 new和delete

- new <类型表达式>
  - 类型表达式一般不用括号：**int *p=new int**;
  - 类型表达式可为数组：**int *q=new int[20]**;
  - 为对象数组对象分配内存时，调用无参构造函数
- delete <指针>
  - 指针指向非数组的单个实体：**delete p;**
- delete [ ]<数组指针>
  - 指针指向数组：**delete [ ]q**;
  - 数组可以是任意维的
  - 如果数组元素为简单类型，则可用delete <指针>代替。

3.4 new和delete

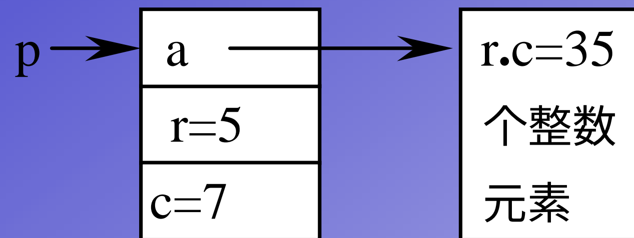
- 【例3.12】定义二维整型动态数组的类。

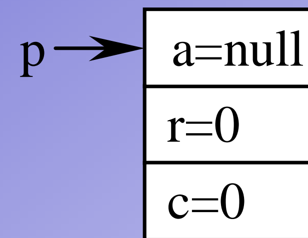
```
#include <alloc.h>
#include <process.h>
class  ARRAY{                    //class体的缺省访问权限为private
    int          *a, r, c;       //成员a、r、c的访问权限为private
public:                          //访问权限改为public
    ARRAY(int x, int y);         //构造及析构函数的访问权限为public
    ~ARRAY( );
};
ARRAY::ARRAY(int x, int y){
    a=new int[(r=x)*(c=y)];//可用malloc：int为简单类型
}
```

3.4 new和delete

```
ARRAY::~ARRAY( ){        //a指向的简单类型int数组无析构函数
    if(a){ delete [ ]a; a=0;}//可用free, 也可用delete a
}
ARRAY  x(3, 5);
void main(void)
{  int error=0;
    ARRAY  y(3, 5), *p;
    p=new ARRAY(5, 7); //不能用malloc，ARRAY有构造函数
    delete  p;                  //不能用free，ARRAY有析构函数。
}
```

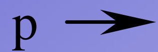
3.4 new和delete

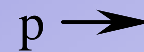




(a) p=new ARRAY(5,7)

(b) p=(ARRAY*)malloc(sizeof(ARRAY))





r.c=35
个整数
元素

(c) delete p

(d) free (p)

3.4 new和delete

- 【例3.13】用new为对象数组申请内存。
- 说明：为什么int (*)[10][20]=int[x][10][20] ？

```
#include <iostream.h>
class A{  //A没有指针类型的成员，故没定义析构函数
    int i;
public:
    A(int x) { i=x; }      //如未定义A(int x)和A( ),
    A( ) { i=0; }          //就可以使用缺省的构造函数
};
```

3.4 new和delete

```
void main(void){
    int  x(3);                      //等价于int x=3
    int *m=new int(5);              //等价于m=new int; *m=5
    int *n=new int[x];
    int (*p)[10]=new int[x][10];
    int (*q)[10][20]=new int[x][10][20];
    A *r=new A(5);                  //使用构造函数A(int)
    A *s=new A[x];                  //使用构造函数A( )
    A (*t)[10]=new A[x][10];                    //用A( )构造
    A (*u)[10][20]=new A[x][10][20];            //用A( )构造
} //delete m,n,p,q,r,s,t,u
```

3.4 new和delete

- new还可以对已经析构的变量重新构造。利用例3.14定义的STRING类，可以编写如下程序：
- STRING x ("Hello!");
- x. ~STRING ( );
- new (&x) STRING ("The World");

3.5 隐含参数this

- 函数成员：普通函数成员，虚函数，纯虚函数，静态函数成员。静态函数成员无隐含参数this。
- 隐含参数：调用对象的地址作为首个实参先压栈。this是非静态函数成员的首个参数，为指向此类对象的const指针。
  - this++或this=...是不允许的
  - *this表示的当前对象可能为const或volatile

- 【例3.15】在二叉树中查找节点。

```
#include <iostream.h>
class  TREE{
    int   value;      TREE  *left, *right;
public:
    TREE (int);     ~TREE( );     TREE *find(int);
};
TREE::TREE(int value){       //隐含参数this指向要构造的对象
    this->value=value;          //等价TREE::value=value
    left=right=0;               //C++提倡NULL用0表示
}
```

3.5 隐含参数this

```
TREE::~TREE( ){//this指向要析构的对象
    if(left) { delete left; left=0; }
    if(right) { delete right; right=0; }
}
TREE* TREE::find(int v){   //this指向调用对象
    if(v==value) return this; //this指向找到的节点
    if(v<value) //满足时查左子树,即下次进入this=left
        return  left!=0?left->find(v):0;
    return  right!=0?right->find(v):0; //否则查右子树
}
```

3.6 对象初始化

- 若类自定义了构造函数，则对象必须用自定义的构造函数初始化
- 当类含有只读、引用或自定义有参构造函数类的非静态数据成员时，必须为初始化这些成员自定义构造函数。
- 所有数据成员都可在构造函数函数体前初始化，而上述三类成员必须在构造函数体前初始化
- 用new分配的对象数组必须用无参构造函数初始化
- 用{}初始化时，必须所有类成员为公有且类没自定义构造函数，这样便同C兼容。

- 例3.16】包含只读、引用及对象成员的类。
- class A{ int a; public: A(int x) { a=x;} A( ){ a=0; } };
- class B{
-     const int b;   //数据成员不能在定义的同时初始化
-     int c, &d, e, f;   //b,d,g,h只能在构造函数体前初始化
-     A g, h; //数据成员按定义顺序b, c, d, e, f, g, h初始化
- public: //类B的构造函数体前未出现h，因此h用A( )初始化
-     B(int y): d(c), c(y), g(y) ,b(y), e(y){ c+=y; f=y; }//f被赋值为y
- };
- void main(void){
-     int x(5);   //int x=5等价于int x(5)
-     A a(x), y=5;   //A y=5等价于A y(5)，请和上一行比较
-     B b(7), z=(7,8);   //B z=(7,8)等价于B z(8)
- }

3.6 对象初始化

- 【例3.18】本例说明了对象数组初始化的方法。

```
class A{//因无指针成员(申请内存)，故没定义析构函数
    int  a;
public:
    A( );                //自定义无参构造函数
    A(int x);            //重载构造函数
    A(int x, int y);     //重载构造函数
};
class B{ int b; }    //未定义构造函数
```

```
void main(void){
    A a[6]={3,(4,5),A(6),A(7,8),A( )};//用自定义构造函数
    A *b=new A[4];        //用自定义无参构造函数构造
    B c, *d=new B[4];   //用C++缺省无参构造函数构造
    delete [ ]b;
    delete [ ]d;
}
```

3.7 类的存储空间

- 存储空间：主要同编译有关，也同机器字长有关。如有继承、虚基类和虚函数，则更复杂。
- 对齐方式：编译时分紧凑方式和松散方式。
  - 紧凑方式，数据成员之间没有空白字节，程序占用的内存较少，跨边界访问指令的访问时间较长。若成员X的地址不能被sizeof(X)整除，则称X为跨边界的。边界同寄存器有关：如MOV EAX, X中X的地址不能被4整除，则指令执行略慢。
  - 松散方式，数据成员之间因边界对齐填补空白字节，程序占用的空间较多，但不跨边界访问时，执行时间较短。

- 【例3.19】定义一个消息结构。

```
#include <iostream.h>
struct  MESSAGE{
    char        flag;       //消息类别标志
    int         size;       //消息长度
    char        buff[255];//消息缓冲区
    long        sum;        //消息累加和
};
void main(void) {
    cout<<"Size of single word is "<<sizeof(int);
    cout<<"Size of double word is "<<sizeof(long);
    cout<<"\nSize of Message is: "<<sizeof(Message);
}
```

- 若sizeof(int)=2, sizeof(long)=4:
  - 紧凑方式：一个成员可以紧接前一个成员存放，则sizeof(MESSAGE)=1+2+255+4=302
  - 松散方式：成员不跨边界存放，即成员开始地址必须能被size(成员类型)除尽，对数组仅考虑其元素类型。 假定MESSAGE的开始地址为0，则：
    - 存放flag后，填补1字节，使size地址能被sizeof(int)除尽
    - 存放size后，不填字节，因buff地址能被sizeof(char)除尽
    - 存放buff后地址用到259，填1字节使sum地址能被4除尽
    - 故sizeof(MESSAGE)=1+1+2+255+1+4=304