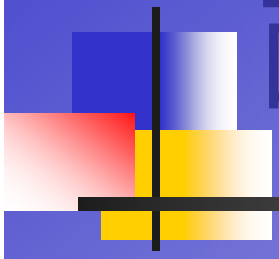
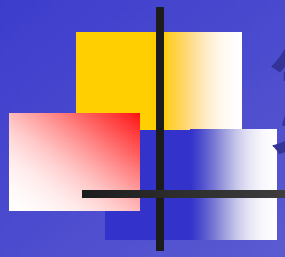


面向对象的程序设计

华中科技大学计算机学院
李瑞轩



第4章 作用域及成员指针

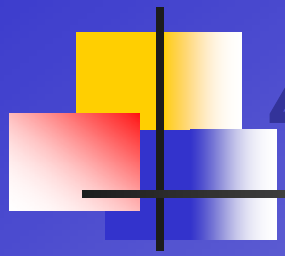
4.1 作用域

4.2 名字空间

4.3 成员指针

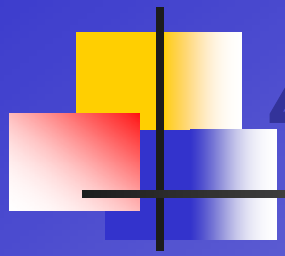
4.4 const、volatile和mutable

4.5 引用对象



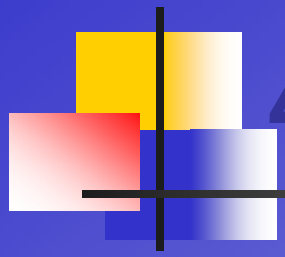
4.1 作用域

- 作用域：标识符起作用的范围。作用域运算符::既是单目运算符，又是双目运算符。优先级和结合性同括号。
  - 单目::用于限定全局标识符(类型名、变量名以及常量名等)
  - 双目::用于限定类的枚举元素、数据成员、函数成员以及类型成员等。双目运算符::还用于限定名字空间成员，以及恢复自基类继承的成员的访问权限。
- 在类体外定义数据和函数成员时，必须用双目::限定类的数据和函数成员，以便区分不同类之间的同名成员。



4.1 作用域

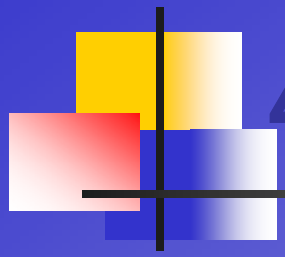
- 作用域分为面向过程的作用域(C传统的作用域)和面向对象的作用域。
  - 面向过程的：词法单位的作用范围从小到大可以分为四级：①作用于表达式内，②作用于函数内，③作用于程序文件内，④作用于整个程序。
  - 面向对象的：词法单位的作用范围从小到大可以分为五级：①作用于表达式内，②作用于函数成员内，③作用于类或派生类内，④作用于基类内，⑤作用于虚基类内。
- 标识符的作用域越小，访问的优先级别就越高。当函数成员参数和数据成员名称相同时，优先访问的是函数成员的参数。



4.1 作用域

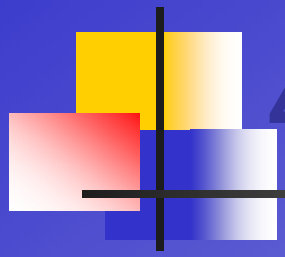
【例4.1】定义二维及三维坐标上的点的类型。

```
class  POINT2D{                  //定义二维坐标点
    int  x, y;
public:
    int  getx( );                //获得点的x轴坐标
    POINT2D (int x, int y){ // int x访问优先于数据成员x
        POINT2D::x=x;            //POINT2D::x为类的数据成员x
        POINT2D::y=y;
    }
};
```



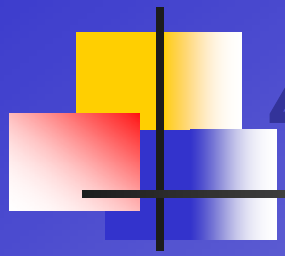
4.1 作用域

```
class  POINT3D{                    //定义三维坐标点
    int  x, y, z;
public:
    int  getx( );                  //获得点的x轴坐标
    POINT3D (int x, int y, int z){
        POINT3D::x=x;              //POINT3D::x为类的数据成员x
        POINT3D::y=y;
        POINT3D::z=z;
    }
};
```



4.1 作用域

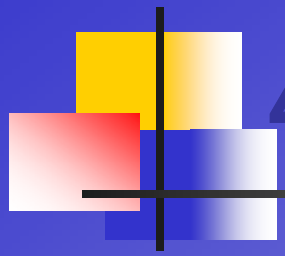
```
//以下代码在类的体外定义getx( )，用::限定getx所属的类
int  POINT2D::getx( ) {return x;}
int  POINT3D::getx( ) {return x;}
static int x;              //C作用域：当前程序文件
void main(int argc, char *argv[ ]) {
    POINT2D  p(3,5);
    x=p.POINT2D::getx( );
    x=p.getx( );       //等价于x=p.POINT2D::getx( );
    x=POINT2D(4,7).getx( );
    //以上常量POINT2D(4,7)的作用域局限于表达式
}
```



4.1 作用域

例4.2】用链表定义容量无限的栈。

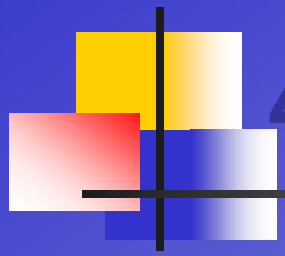
```
#include <iostream.h>
class STACK{
    struct NODE{ int  val;  NODE *next;  NODE(int v); }*head;
public:
    STACK( ){ head=0; }
    ~STACK( );
    int push(int v),  int pop(int &v);
};
STACK::NODE::NODE(int v){  //::自左向右结合
    val=v;
    next=0;
}
```



4.1 作用域

【例4.3】定义用于进程管理的类。

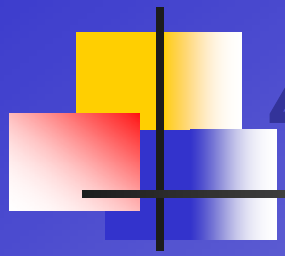
```
static int processes=1; //总的进程数
extern int fork( );          //声明类库提供的fork函数
class Process{
    int processes;           //本进程fork的进程数
public:
    int fork( );
};
int Process::fork( ){
    processes++;             //访问数据成员processes
    ::processes++;           //访问全局变量processes
    return ::fork( );        //调用类库中的fork函数
}
```



4.1 作用域

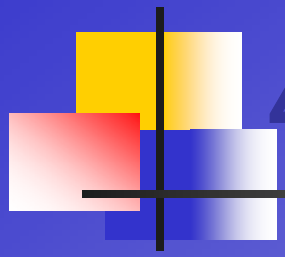
- 注意：
  - 单目运算符::可以限定存储类为static和extern的全局变量、函数、类型以及枚举元素等。
  - 当同一作用域的标识符和类名同名时，可以用class、struct和union限定标识符为类名。
- 【例4.4】定义一个部门的职员。

```
#include<string.h>
class  CLERK{
    char *CLERK;          //数据成员CLERK记录职员姓名
    ::CLERK  *next;       //等价于class CLERK *next
public:
```



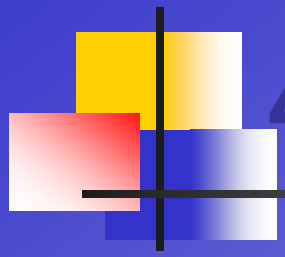
4.1 作用域

```
    CLERK(char *s, ::CLERK*n);//等价于 class CLERK *n
};
//以下class CLERK避免访问数据成员char *CLERK
CLERK::CLERK(char *name, class CLERK *next=0) {
    CLERK=new  char[strlen(name)+1];
    strcpy(CLERK ,name);
    CLERK::next=next;       //CLERK:: next限定访问数据成员
}
int CLERK;                  //定义整型变量CLERK
class CLERK w("Wan", &n);//省略class时类名和变量名混淆
```



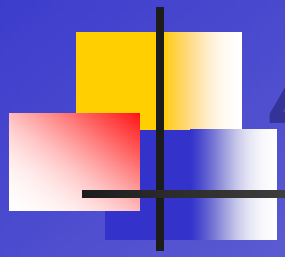
4.2 名字空间

- 名字空间必须在全局作用域内用namespace定义，不能在函数及函数成员内定义，最外层名字空间名称必须在全局作用域唯一。
- 名字空间是C++引入的一种新作用域，类似于面向对象的包、类簇、主题等概念。不同是C++的名字空间既面向对象又面向过程。
- 同一名字空间中的标识符名必须唯一，不同名字空间中的标识符名可以相同。当一个程序引用多个名字空间的同名成员时，可以用名字空间加作用域运算符限定。



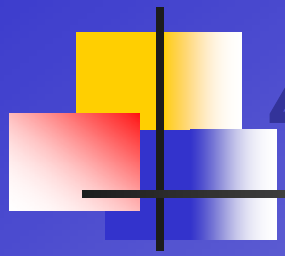
4.2 名字空间

- 名字空间可以分多次定义，即可以先在初始定义中定义一部分成员，然后在扩展定义中再定义另一部分成员，或者定义初始定义中声明的函数原型。初始定义和扩展定义的语法格式相同。
- 保留字using用于指示程序要引用的名字空间，或者用于声明程序要引用的名字空间成员。在引用名字空间的某个成员之前，该成员必须已经在名字空间中声明了原型或进行了定义。



4.2 名字空间

- 名字空间成员三种访问方式：①直接访问成员，②引用名字空间成员，③引用名字空间。
  - 直接访问成员的形式为：<名字空间名称>::<成员名称>。直接访问总能唯一访问名字空间成员。
  - 引用成员的声明形式为：using <名字空间名称>::<成员名称>。如果引用时只声明或定义了一部分重载函数原型，则只引用这些函数。引用这些函数时只能给出函数名，不能带函数参数。
  - 引用名字空间的形式为： using namespace <名字空间名称>。其中所有的成员可用，多个名字空间成员同名。
  - 名字空间的成员同名时用作用域运算符限定。



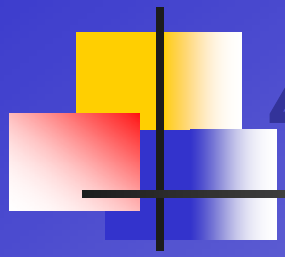
4.2 名字空间

- 【例4.5】名字空间ALPHA定义的变量x及函数g。

```
#include <iostream.h>
namespace ALPHA {          //初始定义ALPHA
    extern int x;          //声明整型变量x
    void g(int);           //声明函数原型void g(int)
    void g(long) {
        cout << "Processing a long argument.\n";
    }
}
using ALPHA::x;            //声明引用变量x
using ALPHA::g;            //声明引用void g(int)和g(long)
```

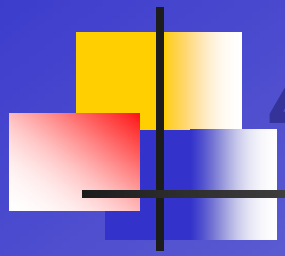
4.2 名字空间

```
namespace ALPHA {          //扩展定义ALPHA
    int x=5;               //定义整型变量x
    void g(int a)          //定义函数原型void g(int)
    { cout << "Processing a int argument.\n";  }
    void g(void)           //定义新的函数void g(void)
    { cout << "Processing a void argument.\n";  }
}
void main(void) {
    g(4);                  //调用函数void g(int)
    g(4L);                 //调用函数void g(long)
    cout<<"X="<<x;         //访问整型变量x
    g(void);               //using之前无该 原型，失败
}
```



4.2 名字空间

- 嵌套名字空间：名字空间内定义名字空间，形成多个层次的作用域，引用时多个作用域运算符自左向右结合。
- 引用名字空间后，其内部定义的成员、其内部引用的名字空间成员、其内部引用的名字空间(所有成员)都能被访问。
- 引用名字空间后，可在当前作用域定义同名标识符。但访问时必须用作用域运算符限定。



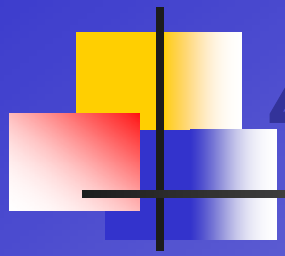
4.2 名字空间

```
#include <iostream.h>
namespace ALPHA {
    void e( ) { cout << "ALPHA\n"; }
    void f( ) { cout << "ALPHA\n"; }
}
namespace DELTA{
    using namespace ALPHA;
    void g( ) { cout << "DELTA\n"; }
}
```

4.2 名字空间

```
using namespace DELTA;
void f( ) { cout << "Global\n"; }
void main(void) {
    ::f( );          //全局函数f
    ALPHA::f( ); //ALPHA::f
    DELTA::f( ); //ALPHA::f
    e( );            //ALPHA::e
    g( );            //DELTA::g
}
```

必须限定，同名



4.2 名字空间

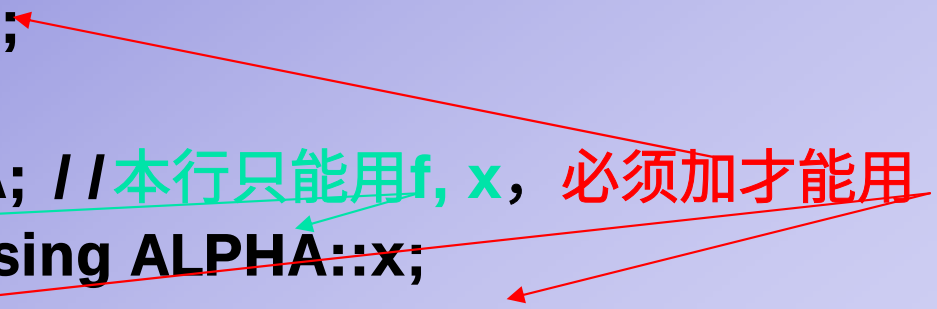
【例4.6】调用ALPHA和DELTA中同名函数成员g。

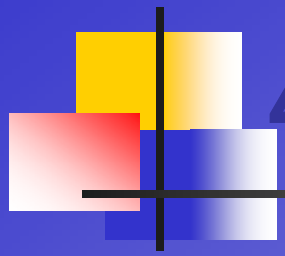
```
#include <iostream.h>
namespace ALPHA { void g( ) { cout<< "ALPHA\n"; }}
namespace DELTA { void g( ) { cout<< "DELTA\n"; }}
int main(void) {
    ALPHA::g( );  //成员同名时直接访问ALPHA的g( )
    DELTA::g( );  //成员同名时直接访问DELTA的g( )
    return 0;
}
```

4.2 名字空间

【例4.7】定义并访问嵌套的名字空间的成员。

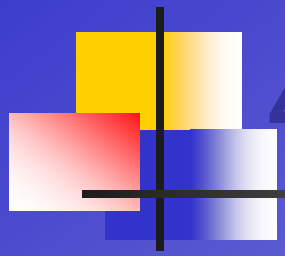
```
namespace ALPHA {
    int x=7;
    void f( ) {cout<<"ALPHA";}
    namespace DELTA{ int x=9;  void g( ) {cout<<"DELTA";} }
//using namespace DELTA;
}
//using namespace ALPHA; //本行只能用f, x，必须加才能用
using ALPHA::f;                 using ALPHA::x;
using ALPHA::DELTA::g;  using ALPHA::DELTA::x;
void main(void) {  f( );   g( );
//cout<<x;//错误：无法确定x属于ALPHA还是DELTA
}
```





4.2 名字空间

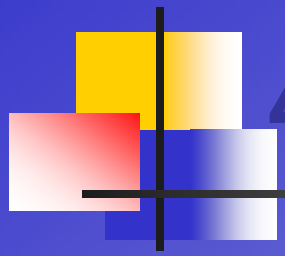
- 引用名字空间与引用名字空间成员不同：
  - 引用名字空间不将其成员加入当前作用域
  - 引用名字空间成员将成员加入当前作用域
- 可以为名字空间定义别名，以代替过长和难懂的名字空间名称。对于嵌套定义的名字空间，使用别名可以大大提高程序的可读性。
- 匿名名字空间的作用域为当前程序文件，其成员被自动引用，但其成员不加入当前作用，即可以在当前作用域定义同名成员。一旦同名冲突，匿名名字空间的成员将是不可引用的。



4.2 名字空间

【例4.9】不能再定义和using引用成员同名的标识符

```
namespace A {  float a=0, b=0;  float d(float y) { return y; } }
namespace B { void g( ) { cout << "B\n"; } }
using A::a;
//long a=1;   //错误，a已被using加入main的作用域
void main(void) {
    using A::b;
//long b=1;  //错误，b已被using加入main的作用域
    using A::d;
    using B::g;
    a = d(2.1);//正确，调用float A::d(float)
    g( );          //正确，调用void B::g( )
}
```



4.2 名字空间

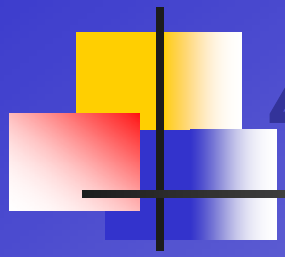
【例4.10】名字空间别名和匿名名字空间。

程序文件A.CPP如下:

- **#include <iostream.h>**
- **namespace { //独立的，局限于A.CPP, 不和B.CPP的合并**
- **void f( ) {cout<<"A.CPP\n";} //不能不定义函数体**
- **}**
- **namespace A{int g( ){ return 0;}} //将和B.CPP的合并**
- **int m( ){ f( ); return A::g( ); }**
- 程序文件B.CPP如下：

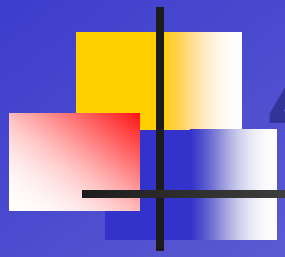
4.2 名字空间

```
#include <iostream.h>
namespace A{int  g( ); namespace B{ namespace C{ int  k=4;}}}
namespace ABCD=A::B::C; //定义别名ABCD
using ABCD::k;                    //引用成员A::B::C::k
namespace {  //独立的，局限于B.CPP, 不和A.CPP的合并
   int  x=3;    //相当于在本文件定义static  int x=3
   void  f( ) { cout<<“B.CPP\n”; } //不能不定义函数体
   class ANT{ char c; };
}
int  x=5;
int  z=::x+k;  //冲突，必须使用::，匿名名字空间的x永远不能引用
extern int m();
int main(void){   ANT  a;   m( );    f( );   return A::g( );        }
```



4.3 成员指针

- **成员指针**：指向类的普通成员的普通成员指针，指向类的静态成员的静态成员指针。变量、成员、函数参数、返回类型都可定义为成员指针类型。访问方式(例4.11)：
  - 普通成员指针：访问成员时必须和对象关联。.*和->*
  - 静态成员指针：访问成员时不须和对象关联。*
  - .*和->*的优先级为14级，结合性自左向右
  - .*左操作数为类的对象，右操作数为成员指针
  - ->*左操作数为对象指针，右操作数为成员指针



4.3 成员指针

- 成员指针不能移动：数据成员的大小及类型不一定相同:
  - 移动后指向的内存可能是某个成员的一部分，或者跨越两个(或以上)成员的内存；
  - 即使移动前后指向的成员的类型正好相同，这两个成员的访问权限也有可能不同，此时便有可能出现越权访问问题。
- 成员指针不能转换类型：否则便可以通过类型转换间接实现指针移动。

4.3 成员指针

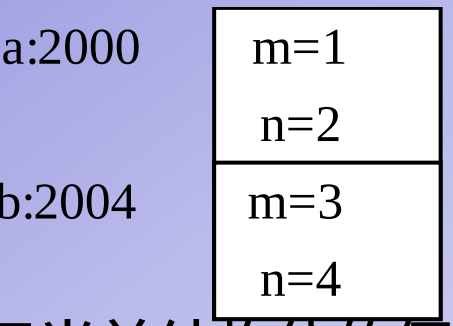
【例4.12】本例说明成员指针不能移动。

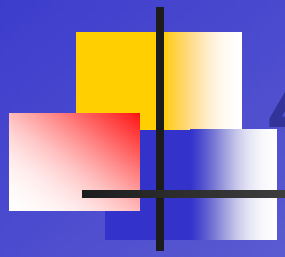
```
#include <iostream.h>
struct A{ int i, f(){cout<<"F\n"; return 1; };
private:   long j; void  g( ) { cout<<"Function g\n";  };
} a;
void main(void){
    int  A::*pi=&A::i;      //数据成员指针pi指向public成员A::i
    int(A::*pf)( )=&A::f;//函数成员指针pf指向函数成员A::f
    long  x=a.*pi;          //等价于x=a.*(&A::i)=a.A::i=a.i
    x=(a.*pf)( );           //.*的优先级低，故用(a.*pf)
    pi++; pf+=1;            //错误， pi不能移动，否则将指向私有成员
    x=(long) pi;            //错误，pi不能转换为长整型
    x+=sizeof(int);         //正确，通过x移动pi
    pi=(int A::*)x;         //错误，x不能转换为成员指针
}
```

4.3 成员指针

普通成员指针实际是相对某个地址的偏移量。

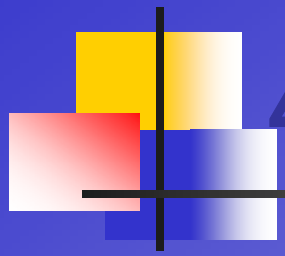
```
struct A{
    int m, n;
} a={1,2}, b={3,4};
void main(void){
    int x, A::*p=&A::m; //p=0: m相对于当前结构体的偏移
    x=a.*p;  //x=*(a的地址+p)=*(2000+0)=1
    x=b.*p;  //x=*(b的地址+p)=*(2004+0)=3
    p=&A::n;//p=2: n相对于当前结构体的偏移
    x=a.*p;  //x=*(a的地址+p)=*(2000+2)=2
    x=b.*p;  //x=*(b的地址+p)=*(2004+2)=4
}
```





4.4 const,volatile和mutable

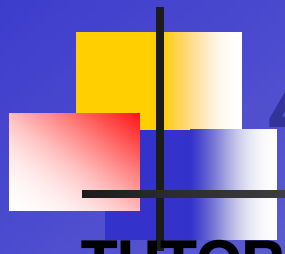
- const和volatile可定义变量、(数据和函数)成员。任何函数的参数和返回类型。
- mutable只能用来定义数据成员。
- 含const数据成员的类必须定义构造函数，且数据成员必须在构造函数参数表的函数体前初始化。
- 含volatile、 mutable数据成员的类则不一定需要定义构造函数
- const只读，volatile挥发，mutable机动



4.4 const,volatile和mutable

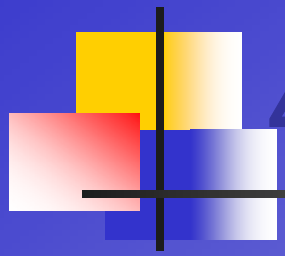
- 【例4.13】定义导师类,允许改名但不允许改性别

```
#include <string.h>
#include <iostream.h>
class TUTOR{
    char        name[20];
    const       char sex;
    int         salary;
public:
    TUTOR(const char *name, const TUTOR *t);
    TUTOR(const char *name, char gender, int salary);
    const char *getname( ) { return name; }
    char *setname(const char *name);
};
```



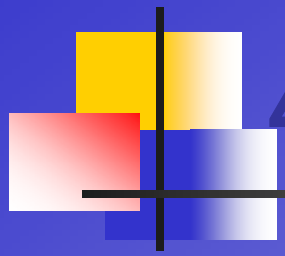
4.4 const,volatile和mutable

```
TUTOR::TUTOR(const char *n, const TUTOR *t): sex(t->sex){
    strcpy(name,n);
    salary=t->salary;
}
TUTOR::TUTOR(const char *n, char g, int s): sex(g),sarlary(s){
    strcpy(name,n);
}
char *TUTOR::setname(const char*n){ return strcpy(name, n);}
void main(void){
    TUTOR  wang("wang", 'F', 2000);
    TUTOR  yang("yang", &wang);
    *wang.getname( )='W';//错误:不能改wang.getname( )指的字符
    *yang.setname("Zang")='Y';
}
```



4.4 const,volatile和mutable

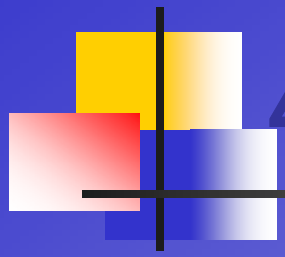
- 普通函数成员参数表后出现const或volatile修饰隐含参数this指向的对象。出现const表示this指向的对象(非静态数据成员)不能被函数修改，但可以修改this指向对象的非只读类型的静态数据成员。
- const或volatile修饰this，但构造或析构函数的this不能被修饰(应状态稳定)，参数修饰会影响函数成员的重载。
- 普通对象应调用参数表后不带const和volatile的函数成员；const和volatile对象应分别调用参数表后出现const和volatile的函数成员，否则编译程序会对函数调用发出警告。



4.4 const,volatile和mutable

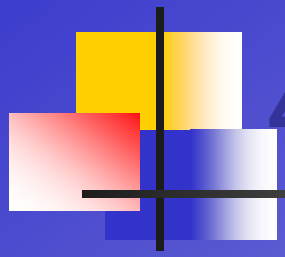
【例4.14】参数表后出现const和volatile。

```
#include <iostream.h>
class A{
    int a;
public:
    int f( ){          // this类型为A * const
        a++;           //正确，可以修改普通数据成员
        return a;
    }
    int f( )volatile{// this类型为volatile A * const
        a++;           //正确，可以修改普通数据成员
        return a;
    }
```



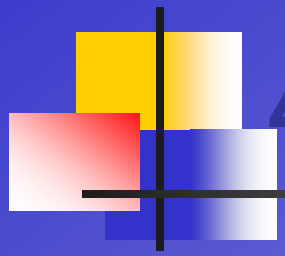
4.4 const,volatile和mutable

```
int f( )const volatile{ // this类型为const volatile * const
    //a++;      //错误，不能修改普通数据成员
    return a;
  };
  A(int x) { a=x; };
} x(3);              //等价于A  x(3);
const A y(6);
const volatile A z(8);
void main(void) {
  x.f( );            //普通对象x调用void f( )
  y.f( );            //只读对象y调用void f( )const
  z.f( );            //只读挥发对象z调用void f( )const volatile
}
```



4.4 const,volatile和mutable

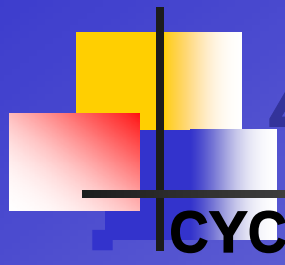
- 参数表后出现volatile，常表示调用该函数成员的对象是挥发对象，这通常意味着存在并发执行的进程。
- C++编译程序几乎都支持编写并发进程，编译时不对挥发对象作任何访问优化，即不利用寄存器存放中间计算结果，而是直接访问对象以便获得对象的最新值。



4.4 const,volatile和mutable

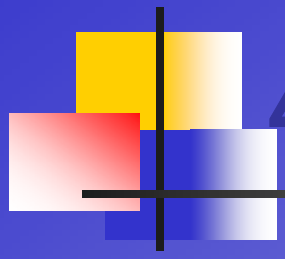
- 【例4.15】定义并发插入和删除的循环队列。

```
class CYCQUE{
    int *queue, size, front, rear;
public:
    int enter(int elem)volatile;
    int leave(int &elem)volatile;
    CYCQUE(int size);
    ~CYCQUE(void);
};
CYCQUE::CYCQUE(int sz){
    queue=new int[size=sz];
    front=rear=0;
}
```



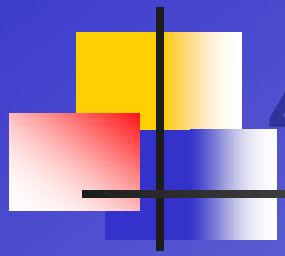
4.4 const,volatile和mutable

```
CYCQUE::~CYCQUE(void){while(rear!=front); delete queue; }
int CYCQUE::enter(int elem)volatile{
    if((rear+1)%size==front) return  0;
    queue[rear=(rear+1)%size]=elem;
    return 1;
}
int CYCQUE::leave(int &elem)volatile{
    if(rear==front) return  0;
    elem=queue[front=(front+1)%size];
    return 1;
}
void main(void){ CYCQUE queue(20); }
```



4.4 const,volatile和mutable

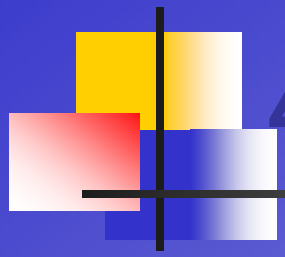
- 函数成员参数表后出现const时，不能修改调用对象的非静态数据成员，但如果数据成员的存储类为mutable，则该数据成员就可以被修改。
- mutable说明数据成员为机动数据成员，该类成员不能用const、volatile以及static修饰。



4.4 const,volatile和mutable

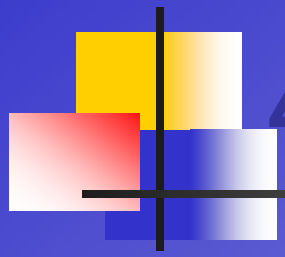
【例4.16】定义产品类，提供产品查询和销售服务。

```
#include <string.h>
#include <iostream.h>
class PRODUCT{
    char *name;                          //产品名称
    int  price, quantity ;               //产品价格,数量
   mutable int count;                    //产品查询次数
public:
    PRODUCT(char *n, int m, int p);
    void get(int &p, int &q)const;
    ~PRODUCT(void);
};
```



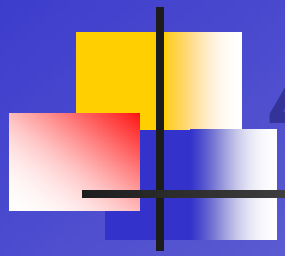
4.4 const,volatile和mutable

```
PRODUCT::PRODUCT(char *n, int p, int q){
    name=new char[strlen(n)+1];
    strcpy(name, n);
    price=p;
    quantity=q;
    count=0;
}
PRODUCT::~PRODUCT( ){
    if(name) {
        delete [ ] name;
        name=0;
    }
}
```



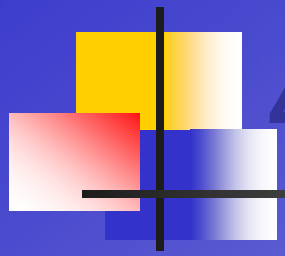
4.4 const,volatile和mutable

```
void PRODUCT::get(int &p, int &q)const{
    p=price;      q=quantity;
    count++;     //count为mutable成员，可以修改
}
void main(int argc, char **argv){
    int  p, q;
    if(argc!=4) return 1;
    PRODUCT m(argv[1], atoi(argv[2], atoi(argv[3]));
    m.get(p, q);
    cout<<"Price="<<p<<"  Quantity="<<q;
}
```



4.5 引用对象

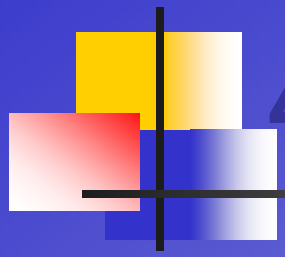
- 被引用对象自己负责构造和析构，引用变量只是被引用实体的别名，故引用变量不必构造和析构。
- 如果引用变量r引用了new生成的对象x，且r在退出作用域前没将引用传到r的作用域之外，则应使用delete &r析构x(同时释放其所占内存)。注意，r.~A( )仅析构x而不释放其所占内存(内存泄漏)。
- 引用变量在定义的同时初始化，引用参数则在调用函数时初始化。左值引用变量和参数必须用左值表达式初始化，如果用(右)值表达式初始化，就会生成一个匿名临时变量。



4.5 引用对象

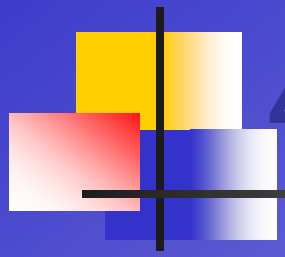
【例4.17】本例用于说明引用变量的用法。

```
#include <iostream.h>
class A{
    int  i;
public:
    A(int i) { A::i=i; cout<<"A: i="<<i<<"\n"; };
    ~A( ) { if(i) cout<<"~A: i="<<i<<"\n"; i=0; };
};
void g(A &a) {//引用参数，无须构造和析构
    cout<< "g is running\n";
}
```



4.5 引用对象

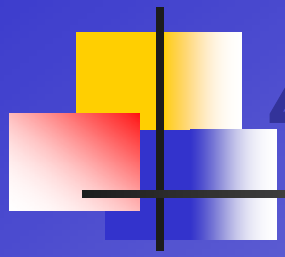
```
void main(void){
    A a(1), b(2);                  //构造a，b
    A &p=a;                        //p本身不用构造和析构a
    A &q=*new A(3);                //q引用new生成的无名对象
    A &r=p;                        //r引用p所引用的对象a
    cout<<"CALL g(b)\n";
    g(b);
    cout<<"main return\n";
    delete &q;                     //q析构并释放new分配的对象
}                                  //退出main时自动析构a,b
```



4.5 引用对象

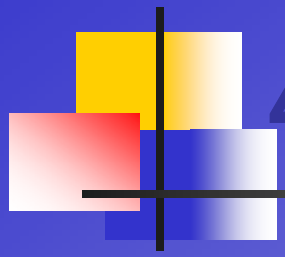
【例4.18】引用产生的匿名变量的析构

```
#include <iostream.h>
class A{
    int  i;
public:
    A(int i) { A::i=i; cout<<"A: i="<<i<<"\n"; };
    ~A( ) { if(i) cout<<"~A: i="<<i<<"\n"; i=0; };
};
void g(A &a) { cout<<"g is running\n"; }
```



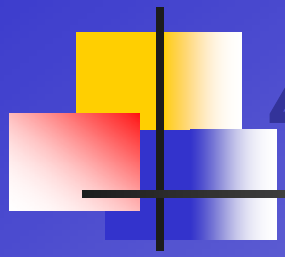
4.5 引用对象

```
void main(void){
  A &p=A(3);//说明时产生匿名变量以A(3)构造
  A(4), cout<<“F\n”;//执行时构造常量A(4),
                    //F后析构
  cout<<"Call g(A(5))\n";
  g(A(5)), cout<<“R\n”;//执行时构造匿名A(5) ,
                       //返回后析构
  cout<<"M\n";        //返回前析构匿名变量A(3)
}
```



4.5 引用对象

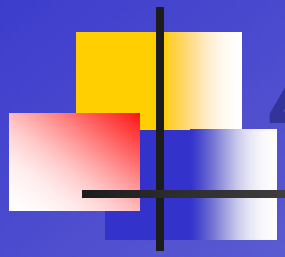
- 非引用对象的形参等价于作用域局限于函数的局部变量，故其析构在函数调用返回前完成，其构造则在调用时由值参传递完成。
- 值参传递将实参数据成员的值相应地赋给形参的数据成员，对于指针类型的数据成员则只复制指针的值，而没有复制指针所指的存储单元内容(浅拷贝赋值)。
- 浅拷贝赋值导致形参和实参(对象)的指针成员指向共同存储单元。一旦被调函数返回，形参对象析构会释放其指针成员所指的存储单元(可能被操作系统立即分配给其他程序)，返回后若实参继续访问该存储单元，就会造成一程序非法访问另一程序的页面，这就是操作系统经常报告的页面保护错误。



4.5 引用对象

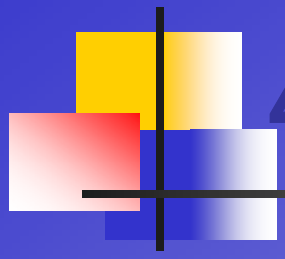
【例4.19】定义一个一维动态整型数组。

```
#include <iostream.h>
class ARRAY {
    int  size, *p;
public:
    int  get(int x) { return p[x]; };
    ARRAY(int s);
    ~ARRAY( ){
        if(p) { delete p;   p=0;  }
        cout<< "Deconstruct ARRAY("<< size <<")\n";
    };
};
```



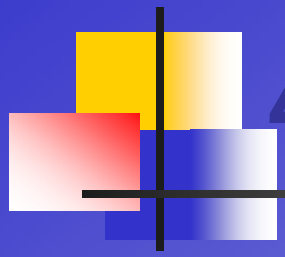
4.5 引用对象

```
ARRAY::ARRAY(int s){
    p=new int[size=s];
    for(int i=0; i<s; i++) p[i]=1;
    cout<<"Construct ARRAY("<< s <<")\n";
}
void func(ARRAY y) { cout << "func: "; }
void main(void){
    ARRAY a(6);
    cout<<"main: a[0]="<<a.get(0)<<"\n";
    func(a);   //浅拷贝赋值
    int *q=new int[6];  q[0]=8;
    cout<<"main: a[0]="<<a.get(0)<<"\nmain: ";
}
```



4.5 引用对象

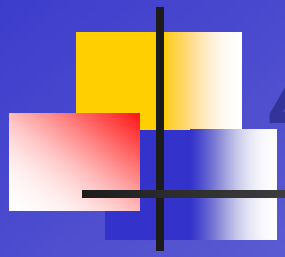
- 深拷贝赋值：在传递参数时先为形参对象的指针成员分配新的存储单元，而后将实参对象的指针成员所指向的存储单元内容复制到新分配的存储单元中。
- 必须进行深拷贝赋值才能避免出现内存保护错误，为了在传递参数时能进行深拷贝赋值，必须将构造函数的参数类型定义为类或类的引用。



4.5 引用对象

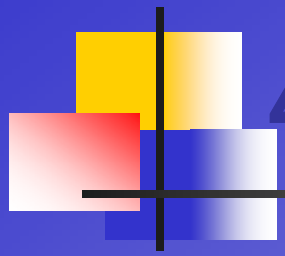
【例4.20】重新定义一维动态整型数组。

```
#include <iostream.h>
class ARRAY{
    int  size, *p;
public:
    int  get(int x) { return p[x]; };
    ARRAY(int s);
    ARRAY(ARRAY &r); // ARRAY(ARRAY r)也可深拷贝构造
    ~ARRAY( ){
        if(p) { delete p; p=0; }
        cout<<"Deconstruct ARRAY("<<size<<")\n";
    };
};
```



4.5 引用对象

```
ARRAY::ARRAY(int s){
    p=new int[size=s];
    for(int i =0; i<s; i++) p[i]=1;
    cout<<"Construct ARRAY("<<s<<")\n";
}
ARRAY::ARRAY(ARRAY &r){
    p=new int[size=r.size];
    for(int i=0; i<size; i++) p[i]=r.p[i];
    cout<<"Construct ARRAY("<<size<<")\n";
}
```



4.5 引用对象

- void func(ARRAY y) { cout<<"func: "; }
- void main(void){
- ARRAY a(6);
- cout<<"main: a[0]="<<a.get(0)<<"\n";
- cout<<"func: ";
- func(a); //相当于ARRAY y(a),故调ARRAY(ARRAY&)
- int *q=new int[6]; q[0]=8;
- cout<<"main: a[0]="<<a.get(0)<<"\nmain: ";
- }