華中科技大學

类的静态成员、成员指针

许向阳

xuxy@hust.edu.cn

内容

5.1 静态数据成员　　static ……

5.2 静态函数成员　　static ……

5.3 实例成员指针　　**int A ::*p;**

int (A::*p)(参数类型…);

5.4 静态成员指针

5.5 const、volatile和mutable

5.6 联合的成员指针

静态成员

类的数据成员

- 实例数据成员
  不同对象的数据成员各有独立的空间
- **静态数据成员　　static**
  不同对象的静态数据成员 共享存储同一空间；
  对象的存储空间中，不包含静态数据成员。

类的函数成员

- 实例函数成员
  对象的地址为第一参数，即隐含参数 this
- **静态函数成员　　static**
  无对象的地址，即无隐含参数 this；
  只能访问静态成员，调用静态函数成员

回顾 函数中局部静态变量


```
ReturnType  F( Parameters )
{
    ……
    static   count=0;
    ……
}
```

Q：静态局部变量的作用域是什么？

空间分配在何处？

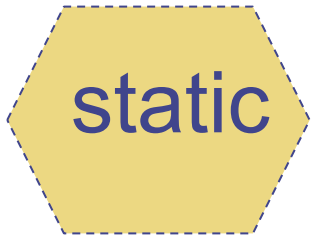
生命周期是什么？

如何赋予初值？ 修改静态变量的值？

有什么用处？

静态成员

```
class A {
    ……
    static int s;      // 静态数据成员
    static .. f(…) ;  // 静态函数成员
   ……
};
```



Q：静态数据成员 与 普通的数据成员的差别是什么？

作用域、访问权限 无差别

空间分配、生命周期 不同

5.1 静态数据成员

- 静态数据成员独立分配内存，不属于任何对象内存
- 所有对象共享静态数据成员内存
- 任何对象修改该成员的值，都会同时影响其他对象关于该成员的值
- 用于描述类的总体信息，如对象总数、连接所有对象的链表表头等
- 在类体内声明，在类体外定义并初始化

静态数据成员 VS 静态全局变量 VS 静态局部变量

作用域的差别？　生命周期：整个程序

5.1 静态数据成员

定义与初始化

```
class HUMAN{
 private:
    char name[11];
     char sex;        int  age;
public:
    static int total;   //类体内只是声明
     HUMAN(char* n,char s,int a)
     {      strncpy_s(name, n, 10);
            sex = s;      age = a;       total++;
     }
    ~HUMAN( ) {  total --; };
};
int  HUMAN::total=0;
```

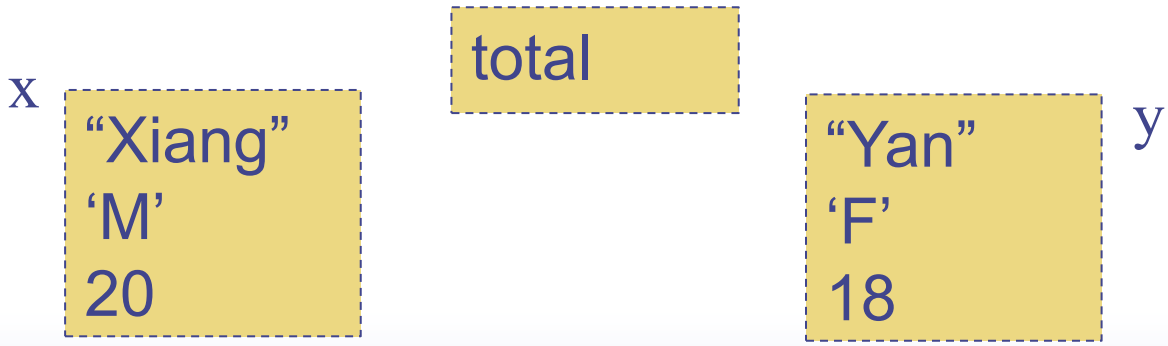
类体外定义并初始化，有访问权限的独立变量。

静态数据成员不能在函数体前初始化

5.1 静态数据成员


```
class HUMAN{
public:     static int total; ……
};
int  HUMAN::total=0;
void main(void){
   cout << HUMAN::total <<endl;        // 显示 0
    HUMAN x((char *) "Xiang", 'M', 20);//HUMAN::total=x.total=1
    HUMAN y((char *) "Yan", 'F', 18); //HUMAN::total=x.total=y.total=2
}
```

空间分配



sizeof(HUMAN)=16，不包括静态成员total

5.1 静态数据成员

- 局部类不能定义静态数据成员

```
void f(void) {
    class T {          // T 是局部类
        int  c;
        static int d;  // 错误
    };
}
```

此规定合理吗？

皮之不存，毛将焉附？

5.1 静态数据成员

- 静态且为常量数据成员的初始化（声明、定义）：

  static const int x=10;

  但不能在构造函数中初始化

- 静态数据成员必须在体外定义、初始化

  static int x;

  .....

  int 类名::x = 10;

- 在构造函数中，可以给静态数据成员赋值，但不称为初始化。

➢ 常量数据成员初始化

方法1： **const int x=10;**

方法2： **const int x;**

类名（参数）：**x(10) { ..... }** // 构造函数

5.2 静态函数成员

- 在函数的最前面加 static
- 静态函数成员的访问权限及继承规则同普通函数成员没有区别，可以缺省参数、省略参数以及重载。
- 普通函数成员的第一个参数是隐含this指针，指向调用该函数的对象；
- 静态函数成员没有this指针，因而通常需要用显式的参数来代替隐含的 this 指针
- 若无参数，只应访问类的静态数据成员和静态函数成员。

5.2 静态函数成员


```
class  A {
private:  int m;
public:   A(int m) :m(m) { }
          bool isEqual(const A& a1)
          {          if (m == a1.m) return true;    else return false;  }
          static int isEqual(A  * const t, const A & a1)
          {          if (t->m == a1.m)   return true;  else  return false; }
};
int main(void)
{         bool  flag;
          A  t1(10), t2(20);
          flag=t1.isEqual(t2);
          flag = A::isEqual(&t1, t2);
}
```

两种用法，生成的 机器语言程序 一模一样！

5.2 静态函数成员

Q：能否定义一个如下的普通函数？

```
bool  isEqual(A  * const t, const A & a1)
{
      if (t->m == a1.m)   return true;
     else  return false;
}
```

不能访问私有成员

普通函数 VS 静态成员函数

VS 友元函数

VS 成员函数 （即普通的成员函数）

5.2 静态函数成员

- 静态函数成员的调用和静态数据成员类似
- 类名::静态函数成员(...);　　推荐使用
- 对象.静态函数成员(...);
- 对象.类名::静态函数成员(...)。

5.2 静态函数成员


```
class A{
    int x; //普通成员必须在对象存在时(有this)才能访问
    static int  i;
    static int  f( ){ return  x+ i; }; //错：static int  f( )无this则无x
public:
    static int  m( ) { return  i; }; //静态函数成员无this
}a;
int A::i=0;        //私有的，体外定义并初始化
void main(void){
    int i=A::f( ); //错误，f私有的
    i=A::m( )+a.m( )+a.A::m( ); //正确，访问公有的静态函数成员
    i=a.f( );      //错误，f私有的不能访问
}
```

5.2 静态函数成员

- 不能使用**static**定义构造函数、析构函数以及虚函数。

```
class A {
   private:
        int x;
   public:
        static A( ) { x=0; };          //错误
        static virtual void f( ) { }  //错误
   };
```

构造函数、析构函数以及虚函数等必须有 **this** 参数

5.2 静态函数成员

- 常函数成员 **f( ) const** 要说明隐含**this**参数。
- 常函数成员不能定义为没有**this**参数的静态函数成员

```
class A {
    static int f( )const      // 错误
        { return 1; }
};
```

5.2 静态函数成员

- 联合union可以申明静态数据成员，静态函数成员。

```
union A { static int c;
          static long d;
          int x;
          int y;
          static void f();
        };
int A::c = 10;
long A::d = 20;

A p;
```

```
sizeof(A) == 4
```

监视 1

搜索(Ctrl+E)　搜索深度: 3

名称	值
&p.c	0x00aec000 {union_test.exe!int A::c} {10}
&p.d	0x00aec004 {union_test.exe!long A::d} {20}
&p.x	0x010ffc28 {23}
&p.y	0x010ffc28 {23}

静态数据成员只是申明，定义在类外，不在对象的空间中

静态成员---- 总结

静态数据成员：

类内申明　　static　类型　成员名;

类外定义和初始化　　类型　类名::成员名=初值

空间分配在对象之外

多对象共享

静态函数成员：

类内申明　　static 返回类型 函数名(….) ;

可在类内定义，也可在类外定义。类外定义时不加 static

无this参数　….　类名::静态函数成员(…);

静态数据/函数成员的应用

Q：为什么要引入静态成员函数？

- 兼容老的用法

  静态成员函数相当于 **一个带有命名空间的全局函数**。
  类名 :: 函数名(….);

- 可以封装某些算法。如实现一些数学函数sin, cos,…..
  这些函数本就不属于任何一个对象。定义一个 Math 类
  不含任何数据成员，无需构造和析构。

- 可以把系统API的回调函数以静态函数的形式封装到类的内部。回调函数通常都没有 this 指针。

- 实现某些特殊的设计模式，如 Singleton 。 单例模式

静态成员的应用——单例模式

单例模式：是一种创建型设计模式，确保一个类只有一个实例，并提供一个全局访问点来访问该实例。

- 配置管理
- 日志记录
- 线程池
- 连接池
- 内存池
- 对象池
- 消息队列

应用场景：
全局唯一的资源管理器

静态成员的应用——单例模式

DeepSeek：C++，配置管理一般有一些什么功能？

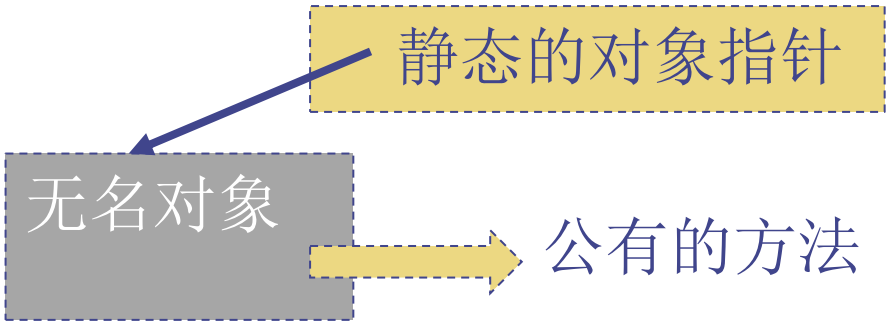
配置管理通常用于管理应用程序的配置数据，这些数据可以来自文件、环境变量、命令行参数等。

- 读取配置：从不同来源（如文件、环境变量、命令行等）加载配置。
- 解析配置：解析不同格式的配置（如JSON、XML、YAML、INI等）。
- 获取配置值：根据键获取配置值，支持不同类型（如字符串、整数、浮点数、布尔值等）。
- 设置配置值：在运行时修改配置值。
- 验证配置：检查配置项是否有效，确保必需项存在且值在允许范围内。
- 默认值：为配置项提供默认值，当配置项不存在时使用。
- 重载配置：重新加载配置（例如，当配置文件发生变化时）。
- 多级配置：支持多个配置源，并可以按优先级合并（例如，默认配置、文件配置、命令行参数等）。
- 类型安全：提供类型安全的获取方式，避免类型转换错误。
- 配置变更通知：当配置发生变化时，通知注册的观察者。

静态成员的应用——单例模式

单例模式的实现要点：

- 将**构造函数**定义为**私有函数**
- 定义一个**私有**的类的**静态实例指针**
- 提供一个公有的静态方法，
  实现创建唯一实例对象指针，获取指针的功能



- 构造函数定义为私有函数，这样就无法 定义该类的对象。
- 虽然可以定义对象指针，但对象指针如何赋初值？
- 不可能通过 new 来初始化。只能通过一个公有函数，来获得对象的指针。
- 调用该公有函数又要有对象，除非是静态的函数。在静态函数中只能访问静态成员。

静态成员的应用——单例模式


```
class singleton {
private:    string s;
    singleton(): s("very good") { }
    static singleton  *instance ;
public:
    static singleton* getInstance() {
        if (instance == nullptr) {    instance = new singleton(); }
        return instance;
    }
    void show() {          cout << s << endl;  }
    void setstring(const string &a)      {   s = a;  }
};
singleton* singleton::instance = nullptr;
int main()
{  singleton::getInstance()->setstring("hello world");
    singleton::getInstance()->show();
}
```

私有的构造函数

静态的私有对象指针

静态、公有的函数，

可以获取唯一的指向对象

静态成员的应用——单例模式

讨论：单实例对象中有指针，如何释放其指向的体外空间？
又如何释放单实例的空间？

```
class singleton {
private:
    char *p;   ……
    static singleton* instance;
public:  ……
    void setstring(const char *src)
    {  delete p;
       p = new char[strlen(src)+1];
       strcpy(p, src);
    }
};
```

```
static singleton* getInstance() {
    if (instance == nullptr)
        instance = new singleton();
    return instance;
}
```

静态成员的应用——单例模式

讨论：单实例对象中有指针，如何释放其指向的体外空间？

又如何释放单实例的空间？

- 增加公有的析构函数，来释放指针指向的体外空间；

```
~singleton() { delete p;
  p = nullptr; }
```

- 同时删除对象本身的空间

```
delete singleton::getInstance();
```

Q：有无方法，不需要人工写语句，而是自动释放空间？

静态成员的应用——单例模式


```
class Recycle {
public:
    Recycle() {};
    ~Recycle() {
        delete singleton::getInstance();
    }
};

static Recycle t;
```

t 的生命周期结束时，就会自动调用其析构函数，

在析构函数中，回收单实例对象的体外空间及其本身空间。

静态成员的应用——单例模式

DeepSeek ： C++，使用匿名类，能否实现单例模式？

可以使用匿名类实现单例模式。但在实际项目中，传统的命名类实现更加清晰和可维护。

- 类型信息丢失：匿名类没有类型名，难以在模板或类型特征中使用
- 调试困难：调试时匿名类显示为复杂类型名
- 继承限制：只能通过基类接口访问，无法直接访问派生类的特有方法
- 可读性：代码可读性可能较差

静态成员的应用——单例模式

DeepSeek ： C++，实现与单例模式同等效果，能否定义一个类，所有的公有函数成员都是静态的，这样就不需要一个静态的对象指针？

可以通过将所有成员函数定义为静态函数，并且将构造函数私有化（或删除）来达到类似单例的效果。这种方式通常被称为“静态单例”或“工具类”、“**静态工具类**”，因为它不需要实例化，直接通过类名调用方法。更加简单和安全。

静态成员的应用——单例模式

与单例模式的对比

特性	静态工具类	传统单例模式
实例化	不需要实例，直接调用静态方法	需要获取单例实例
内存管理	自动管理，无析构问题	需要处理析构顺序
线程安全	更容易保证（每个方法独立加锁）	需要整体考虑线程安全
测试难度	相对容易测试	较难测试（全局状态）
继承扩展	不支持多态	支持多态和继承
延迟初始化	不支持（静态变量立即初始化）	支持延迟初始化
依赖注入	难以实现	相对容易实现

成员指针

数据成员指针

- 实例数据成员指针　例：　int 类名A ::*p;
  - VS　普通指针　　例：　int　*q;
  - int A ::*p=&A::a;　// a 在A 中的偏移量
- 静态数据成员指针
  - 与 普通指针一样，存放指向单元的地址

函数成员指针

- 实例函数成员指针　例：　int　(类A:: *pf) (int);
  - VS　函数指针　　int　(*f)(int);
- 静态函数成员指针
  - 与 普通函数指针一样

实例数据成员　＝　非静态数据成员

5.3 实例成员指针

实例数据成员指针

- 什么是实例数据成员指针？
- 与普通的数据指针有何差别？
- 如何使用实例数据成员指针？
- 实例数据成员空指针与普通数据空指针的差别
- 使用数据成员指针有何优点？

实例函数成员指针

实例成员指针

- 实例数据成员指针 指向 类的一个实例数据成员；
- 存放数据实例成员在类中的偏移地址；

  而不是一个对象中实例成员的地址；
- 数据成员指针 不一定是 类的成员；
- 实例数据成员指针指向的成员应有相应的访问权限。

```
int    *p;      // 普通指针
int   类名A :: *q; //成员指针
                   // 指向 A 类中的一个 int 成员
q = & 类名A :: A 中的int成员 ;
```

实例成员指针


```
class Student {
public:   int  number;
     char name[15];
     float score;
public: ……
};
Student xu(123,"Xuxiangyang",100);
```

p 普通的变量指针

Q：number 在类Student 中偏移地址是多少？

```
int *p=&xu.number;        // p 指向 xu中的number，要求public权限
cout<< *p << endl;        // 输出 123

int x=offsetof(Student, number);   //取偏移地址，要求public权限
Student * s;                       // 注意与成员指针的差别
s = &xu;
*(int *)((char *)s + x)  =100 ;
```

实例成员指针


```
class Student {
public:
    int  number;
   ……
};
Student xu(123,"Xuxiangyang",100);

int *p=&xu.number;        // p 指向对象 xu中的number

int Student::*q = &Student::number;  // q 数据成员指针
                                     // 要求 number 具有public权限

cout << xu.*q<<endl;   // cout << xu.number <<endl;
int *p=&Student::number; //无法从 Student::* 转换为 int *
```

从普通指针到实例成员指针

指向的类型 类名::* 变量名;

实例成员指针

函数返回普通数据指针

```
class Student {
private:
    int  number; …
public:
    int  * getaddress1( )
    {return   &number; }

    int   getaddress2( )
    { return  int (&number); };
};
Student xu(123,"Xu",100);
```

```
void main()
{
    int  *p;
    int    q;
    p  =  xu.getaddress1( );
    q  =  xu.getaddress2( );
    cout<< *p << endl;
    cout<< *(int *)q;
}
```

结果皆为 123

p, q为xu.number的地址

不是number 在类中的偏移地址

实例成员指针

函数返回实例数据成员指针

```
class Student {
private:
    int  number; …
public:
    int *  getaddress1()
    { return  &number; }
    int Student::*getaddress2() {
        return &Student::number;  }
};

Student xu(123,"Xu",100);
```

```
void main()
{
    int  *p;
    int  Student::*q;
    p=xu.getaddress1();
    q = xu.getaddress2();
   cout<< *p;
   cout<<xu.*q;
}
```

结果皆为 123

将 q=… 带入 xu.*q

```
cout<<xu.*xu.getaddress2();
```

实例成员指针

函数返回数据成员指针

```
class Student {
public:
int Student::*getaddress3() {
        return &Student::number;  }
};

int Student::* Student::getaddress3() {
        // 类成员 Student::get…. 指向 一个成员
        return &Student::number;
}
```

类内定义函数

VS

体外定义函数

实例成员指针

数据成员空指针

```
int    *p;

int  Student::*q;
```

```
p=0;        // p中的值为  0                    *p 不可用，程序异常

q=0;        //  q 中的值为 0xffffffff      s.*q ，结果不正确
                                                程序可继续运行
```

实例成员指针

DeepSeek： C++ 中，引入数据成员指针的目的是什么？

- 泛型编程和回调机制：在某些库或框架中，我们可能需要将某个成员函数或数据成员作为参数传递，以便在后续的回调中使用。
- 访问控制：通过成员指针，我们可以访问类的私有成员，但这通常需要谨慎使用，因为它可能破坏封装。
- 序列化/反序列化：在序列化过程中，我们可能需要遍历一个类的所有成员，这时可以使用成员指针来指定要序列化的成员。
- 反射：C++ 本身不支持完整的反射，但通过成员指针，我们可以在一定程度上模拟反射，例如通过字符串名称来访问成员。
- 数据结构与算法：在某些数据结构或算法中，我们可能需要指定按照哪个成员来进行比较或操作。

实例函数成员指针

实例函数成员指针

- 指针函数与函数指针
- 普通的函数指针的用法与优点
- 实例函数成员指针的定义和使用

实例成员指针

指针函数 ： 返回结果是一个指针

```
char * fun(……);
```

函数指针：**是一个指针，指向一个函数**

```
int fadd(int x, int y)
{ return x+y; }
```

```
int fsubtract(int x, int y)
{ return x-y; }
```

```
int (*fp)(int, int);
fp=&fadd;
result=(*fp)(10,20);
```

```
fp=fadd;
result=fp(10,20);

fp=fsubtract;
result=fp(10,20);
```

实例成员指针


```
int  fadd(int x, int y);

result=fadd(22,33)

int  (*fp)(int, int);

fp=&fadd;          ⇔   fp = fadd;

result=(*fp)(22,33);   ⇔   result=fp(22,33);
```

取函数地址时，有无 & 一样；
调用函数指针时，有无 * 一样

实例成员指针

成员函数指针
的定义和使用

```
class Student {
public:
        void  SetNumber(int  x) { number=x; }
};
Student xu(123,"Xu",100);
void  (Student:: *pf) (int) ;   //成员函数指针定义
void  (*pq)(int) ;              // 函数指针
pf=&Student::SetNumber;
(xu.*pf)(200);
xu.SetNumber(200);     // 等同语句
```

实例成员指针

- **成员指针**：指向类的成员，**本身并不一定是类的一个成员**
  - 实例数据成员指针，实例函数成员指针
  - 静态数据成员指针，静态函数成员指针
- 数据成员、函数参数和返回类型都可定义为成员指针类型。

函数成员指针：

不能指向构造函数，否则通过函数指针就可实现手动调用；

可以指向析构函数，通过函数指针可以手动调用析构函数。

实例成员指针

- 使用实例成员指针访问成员时必须和对象关联
- 使用静态成员指针时不必和对象关联
- 实例成员指针通过.*和->*访问对象成员
- .*和->*优先级高， 结合性自左至右
- .*左操作数为类的对象，右操作数为成员指针

  int Student::*p;　　xu.*p
- ->*左操作数为对象指针，右操作数为成员指针

  Student *q;　　q->*p
- 若成员指针p是类的一个成员，可用 **xu.*(xu.p)**

实例成员指针

- 实例成员指针是一个偏移量，存放的不是成员地址，故不能移动，但可以再指向另一个成员

```
int  Student::*p;
p=p+1;        // 非法，不能移动指针
int  *q;
q = q+1;
```

- 实例成员指针不能进行类型转换：
  - 防止通过类型转换间接实现指针移动。

5.4 静态成员指针

- **静态成员指针**指向类的静态数据（函数）成员。
- 变量、数据成员、普通函数和函数成员的参数和返回值都可以定义成静态成员指针类型。
- 静态成员相当于具有访问权限的变量和函数，同普通变量和函数相比，除访问权限外没有区别，因此，静态成员指针和普通指针形式上也没有区别。

静态成员指针	实例成员指针
存放静态成员地址	存放普通成员偏移
可以移动	不能移动
可以转换类型	不能转换类型

静态成员指针


```
class  P{
    char   name[20];
public:
    static int n;
    static int  getn( )  { return n; }
    P(char *n) { strcpy_s(name, n);    n++; }
    ~P( ) { n--; }
}zan((char *)"zan");
int P::n=0;
void main(void){
    int  m;
    int  *d=&P::n;          // d 为静态成员指针；等价于d=&zan.n;
                            //静态成员指针 与 普通指针 没有差别
    int (*f)( )=&P::getn;  //普通函数指针指向静态函数成员
    m=*d + (*f)( );
}
```

静态成员指针


```
struct A{
    int a, *b, A::*u, A::*A::*x, A::**y, *A::*z;
    static int c, A::*d;
}z;
int A::c=0, A::*A::d=&A::a;  //静态数据成员初始化
void main(void){     //注意优先级高低：  ".">"*">".*"
    int i, A::**m;
    z.a=5;        z.u=&A::a;  i=z.*z.u; //i=z.*&A::a=z.a=5
    z.x=&A::u; i=z.*(z.*z.x);
    //i=z.*(z.*&A::u)=z.*z.u=z.a=5
    m=&A::d;  m=&z.u;  i=z.**m; //i=z.**&z.u=z.*z.u
    z.y=&z.u;    i=z.**z.y; // i=z.**&z.u=z.*z.u=z.a=5
    z.b=&z.a;    z.z=&A::b;  i=*(z.*z.z);
    //i=*(z.*&A::b)=*z.b=*&z.a
}
```

5.5 const、volatile和mutable

const 可修饰

- 普通变量
- 类的数据成员
- 函数的参数，成员函数的参数
- 函数成员
- 对象
- 函数的返回类型

5.5 const、volatile和mutable


```
class TUTOR{
    char          name[20];
    const         char  sex;       //性别为只读成员
    int           wage;
public:
    TUTOR(const char *n, char g, int s): sex(g), wage(s)
        { strcpy_s(name,n); }
    const char *getname( ) const{ return name; }
          //函数体不能修改当前对象   函数的返回类型有 const 修饰
    char *setname(const char *n)
        { strcpy_s(name, n);      return name;  }
};
TUTOR xu("xuxy",'M',2000);
*xu.getname()='X';    //  不能给常量赋值
*xu.setname("xuxiangyang") ='X';  // name 的首字母变成X
strcpy_s(xu.setname("xu123"), 6, "hello"); //name 改为hello
```

5.5 const、volatile和mutable

函数的返回类型前有const

```
TUTOR : 有私有成员  char  name[20];
const char * TUTOR::getname( ) const{
    return name;
}       // 返回值的 类型 是 const char *.

TUTOR xu("xuxy",'M',2000);
char *p;
p=xu.getname( );        // 无法从 const char * 转换为 char *
const char *q;
q = xu.getname( );
*xu.getname( ) ='X';   // 不能给常量赋值
                       //  等同 *q='X';     q[0]='X';
                       //  表达式必须是可修改的左值
```

5.5 const、volatile和mutable

函数的返回类型前有const

TUTOR : 有私有成员 char name[20];

```
const char * TUTOR::getname( ) const {return name; }

TUTOR xu("xuxy",'M',2000);
const char *q;
q = xu.getname( );
q = “hello”;
```

讨论： 能否用 xu.getname( ) 代换q, 写出语句

xu.getname( )=“hello” ?

编译器给出的错误：

表达式必须是可修改的左值； = 左操作数必须是 左值。

本质：const char * const getname;

5.5 const、volatile和mutable

讨论： 能否用 xu.getname( ) 代换q, 写出语句

xu.getname( )=“hello” ?

核心：函数是如何返回结果的！

```
int  f( ) {int x=20;  return x;}
        mov   dword ptr [x], 14h
        mov   eax, dword ptr [x]    // eax 存放返回结果
y =f( );
        mov    dword ptr [y], eax
```

eax 不是内存单元，无法获得其地址，特别是不允许修改eax中的值。 f( ) = 10; 就是企图 mov eax, 0ah

对于 const char * getname( ) const {… } 也是使用 eax来传递结果，同样，无法使用 xu.getname( ) =“hello”;

5.5 const、volatile和mutable

函数的返回引用 与 返回值的差别

```
int  & g( ) {int x=20;  return x;}          int &g=x;
                                            有警告
y =g( );                                    y=g;     // y=20;

g( ) =10;                                   g=10;
```

对于引用类型的返回，返回的是一个地址。
即被应用变量的地址；
在使用函数时，等同使用引用变量，也即被引用的变量。

5.5 const、volatile和mutable

讨论： **const char * TUTOR::getname( ) const**
{return name; }

返回结果在 eax 中，其值代表的是单元的地址，
相当于指针。虽然不能修改 eax， 即 不能 **eax = p ;**

Q: 能否使用 [eax] 的形式来修改指向单元的内容？
即 ***xu.getname( )='X' ;**

不行。函数前有 const 约束，相当于 const char *q;
不能有 ***q ='X'.**

5.5 const、volatile和mutable

讨论： char * TUTOR::getname( ) const
{return name; }
能否直接去掉 函数返回上的 const 约束，然后：
*xu.getname( )='X' ;

编译错误：返回值类型与函数类型不匹配。
return ： 无法从 const char [20] 转换为 char *.

getname 后面有一个 const, 即 代表
const TUTOR * const this. this->name 是 const的.

如何解决这一错误？

5.5 const、volatile和mutable

方法1 ： 去掉 getname后的 const

```
char * TUTOR::getname( )
 { return  name;  }

*xu.getname( )='X' ;
```

方法2 ： 返回时进行强制类型转换

```
char * TUTOR::getname( ) const
 { return  (char *)name;  }

*xu.getname( )='X' ;
```

5.5 const、volatile和mutable


```
int                 const  int
int *               const  int  *
                    const char *getname( );
                    const char *pc;      // *pc=…   错误
                                         // *getname() = …
```

const 变量是右值

5.5 const、volatile和mutable

volatile：

不稳定的、易挥发的、变化无常的

变量可能会被意想不到的改变；

优化器不对该变量的读取进行优化，用到该变量时重新读取。

正因为变化无常，而不能对涉及到该变量的语句进行优化。

每次都会根据变量的地址去访。

5.5 const、volatile和mutable

```
#include <iostream>
using namespace std;
int main()
{        int  i = 10;
         int  a = i;
         cout << "a = " << a << endl;
         __asm {        实际上改 i = 123;
            mov dword ptr [ebp-8], 123
         }
         int b = i;
         cout << "b= " << b << endl;
         return 0;
}
```

在Debug 版本下，
a =10
b = 123

在Release 版本下，
a =10
b = 10

5.5 const、volatile和mutable

```
#include <iostream>
using namespace std;
int main()
{       volatile  int  i = 10;
        int  a = i;
        cout << "a = " << a << endl;
        __asm {
            mov dword ptr [ebp-8], 123
        }
        int b = i;
        cout << "b= " << b << endl;
        return 0;
}
```

在Debug 版本下，
a =10
b = 123

在Release 版本下，
a =10
b = 123

5.5 const、volatile和mutable

编译器对程序的优化

```
int main()
{
    int   i, j;
    i=1;
    i=2;
    i=3;
        // 上面的程序等价于 i=3;
    for (i=0;i<10000;i++)  j=0;
        // 延时程序，可优化掉无用的循环
}
```

5.5 const、volatile和mutable

```
int   a;
const  int  x=100;

cout<<x<<endl;

*(int *)&x =200;

cout <<x<<endl;
```

显示 100　　100

```
int   a;
a=100;
const  int  x=a;

cout<<x<<endl;

*(int *)&x =200;

cout <<x<<endl;
```

显示 100　200

5.5 const、volatile和mutable

```
int  flag=0;
int main()
{
    while (1) {
      if (flag)
        dosomething();
    }
}

// 中断服务程序   程序2
void ISR()
{    flag=1;
}
```

编译器可能认为main 函数中未修改过 flag;

将 flag 读入到一个寄存器中；后面只用寄存器中的副本；导致flag 修改后未发现；dosomething 不被执行

5.5 const、volatile和mutable

volatile：

优化器不对该变量的读取进行优化，用到该变量时重新从它所在的内存读取数据。

修饰的变量可由操作系统、硬件、并发执行的线程在程序中进行修改。

以下情况下，应在变量前 加 **volatile**

- 多任务环境下，各任务间共享的变量；
- 中断服务程序中修改的供其他程序检测的变量；
- 存储器映射的硬件寄存器；

5.5 const、volatile和mutable

- **volatile**可以修饰变量、类的数据成员、函数成员及普通函数的参数和返回类型。
- 构造或析构函数的参数表后不能出现**const**或**volatile**

  classname(….) const; // error

  构造或析构时，**this**指向的对象应能修改且不随便变化。

- 静态函数成员参数表后不能出现**const**或**volatile(**无隐含**this)**。

  static … ff(…) const; // error

5.5 const、volatile和mutable

mutable:

可变的

- 是const 的反义词
- 为突破 const的限制而设置的
- 被mutable 修饰的变量永远处于可变得状态，即使在const函数中
- **mutable只能用来修饰数据成员**
- **不能与 const、volatile 或 static 同时出现**
- **静态成员在const 成员函数中可以修改**

5.5 const、volatile和mutable


```
class A {
        int  x;     static  int  y;   mutable int z;
public:
        void  f() const
        {  x = x + 1;  //  编译时报错，无法修改 x
           y = 20;      //   无误
           z = 30;      //   无误
        }
};
```

5.5 const、volatile和mutable

- 参数表后出现const、volatile或const volatile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。
- 参数表后出现volatile，表示调用函数成员的对象是挥发对象，意味存在并发执行的进程，正在修改当前对象。

5.5 const、volatile和mutable

```
class Date {
private:   int year, month, day;
public:
    Date(int year, int month, int day) {
        this->year = year; this->month = month; this->day = day;
    }
    void ModifyYear(int year) {
         this->year = year;
    }
    void DisplayYear() const {  cout << year << endl; }
    void DisplayMonth()  {  cout << month << endl;  }
};

const Date national_day(1949, 10, 1);
national_day.DisplayYear();
national_day.ModifyYear(1948);    // error
national_day.DisplayMonth( );     // error
```

5.5 const、volatile和mutable


```
class  A{
   int a;  const  int  b; //b为const成员
public:
   int f( ){a++; return a; } //this类型为： A * const  this
   int f( )const{return a; } //this类型为：const A * const  this。
   int f( )volatile{return a++; } //this类型为：volatile A * const this
   int f( )const volatile{ return a; }//this类型为：const volatile A* const this
   A(int x) ：b(x)  { a=x; } //不可在函数体内对b赋值修改
} x(3);                          //等价于A  x(3)
const A y(6);                    //  y 不可修改
const volatile A z(8);           //  z  不可修改
void main(void) {
    x.f( ); //普通对象x调用int f( ): this指向的对象可修改
    y.f( ); //只读对象y调用int f( )const:this指向的对象不可修改
    z.f( ); //只读挥发对象z调用int f( )const volatile:this指向的对象不可修改、挥发
}
```

5.6 联合的成员指针


联合
union
实例成员
静态成员
联合的
成员指针
实例成员
指针
实例数据成员指针
实例函数成员指针
静态成
员指针
静态数据成员指针
静态函数成员指针

5.6 联合的成员指针

- 联合可以定义实例和静态成员，故也可以定义实例和静态成员指针。
- 联合的实例数据成员共享内存，指向这些实例数据成员的指针存储的偏移量值实际上是相同的。
- 函数中局部类不能定义静态数据成员，
  函数中的局部联合也不能定义静态数据成员。
- 全局类中的联合或全局联合可以定义静态数据成员；静态数据成员指针一般指向全局类中的联合或全局联合的静态数据成员。

总结

实例数据成员、常数据成员、静态数据成员

实例函数成员、常函数成员、静态函数成员

实例成员指针、静态成员指针

const　　static　volatile　mutable

各是什么意义？　有何差别？　如何使用？

练习——p, pp,q, r的定义或用法


```
struct A {
    char* a;
    char  b;
    char  * geta();
    char A::* p;       // 成员指针，指向一个char 类型的成员
    char * A::* pp;    // 指向一个 char * 类型的成员
    char* A::* q();    // 函数返回一个指向 char *类型的成员
    char* (A::* r)(); //  成员函数指针，
                       //  指向返回类型为char *的函数
};
```

练习


```
struct A {
    char* a;
    char  b;
    char  * geta();
    char A::* p;        //  a.p = &A::b;
    char * A::* pp;    // a.pp = &A::a;
    char* A::* q();
    char* (A::* r)(); //  a.r = &A::geta;
}a;
 char* A::* A::q()  { return &A::a;  }
 char* A:: geta()   {  return 0; }
```