2024秋季 c++ 辜希武

第二章

char16_t表示双字节字符类型，支持UTF-16

char32_t表示四字节字符类型，支持UTF-32。

wchar_t表示char16_t ，或char32_t

可对%f或%lf设定宽度和精度及对齐方式。“%-8.2f”表示左对齐、总宽度8(包括符号位和小数部分)，其中精度为2位小数

第一个出题点

模块静态变量：使用static在函数外部定义的变量，只在当前文件（模块）可用。可通过单目::访问。局部静态变量：使用static在函数内部定义的变量。

```
static int x, y; //模块静态变量x、y定义，默认初始值均为0
int main( ){
    static int y;        //局部静态变量y定义，  初始值y=0
    return  ::y+x+y;//分别访问模块静态变量y,模块静态变量x,局部静态变量
}
```

如果想要声明一个变量而非定义它，就在前面加关键字extern，而且不要显示地初始化变量： 任何包含显式地初始化的声明成为定义。如果对extern的变量显式初始化，则extern的作用被抵消。

经典错误：

由于头文件Test.h分别在二个cpp文件里二次被包含，因此头文件里定义 的变量在二个cpp文件里被定义二次，会报链接错误

列表初始化{} 和括号初始化

使用{}初始化 最初是在结构体 列表初始化会报类型不匹配的错

```
int b = { 0 };
int c(0);
int d{ 0 };
```

const constexpr和volatile

只读变量：使用const或constexpr说明或定义的变量，定义时必须同时初始化。当前程序只能读不能修改其值。constexpr变量必须用编译时可计算表达式初始化。易变变量：使用volatile说明或定义的变量，可以后初始化。当前程序没有修改其值，但是变量的值变了。不排出其它程序修改。

定义“const volatile int z=0;”是由意义的，不排除其它程序修改z使其值易变。

常量表达式

是指值不会改变而且在编译时可以求值的表达式，如字面量就是常量表达式，用常量表达式初始化的const变量也是常量表达式

就是不用运行就可以知道的值 且这个值在运行过程中不会改变

```
const int max = 20;  // max是常量表达式
    const int limit = max + 1;  // limit是常量表达式
    int i = 0;            //i不是常量表达式，非const
    const int  min = get_min(); //min的值只有运行是才能确定，因此不是
由于人有时很难判断一个表达式是否为常量表达式，因此costexper粉墨登场
```

inline 内联符号 只局限于当前代码

保留字inline用于定义函数外部变量或函数外部静态变量、类内部的静态数据成员。inline函数外部变量的作用域和inline函数外部静态变量一样,都是局限于当前代码文件的，相当于默认加了static

常数的指针 和 指针是常数

```
const int x=3;  const int *y=&x;  //x是只读单元，y是x的地址
               int *z=y;     //错：y是指向只读单元的指针
               z=&x;         //错：&x是是只读单元的地址
```

前例的const换成volatile或者const volatile，结论一样。int *可以赋值给const int *，const int *不能赋值给int *

假如常数的指针可以复制个指针 我们可以通过普通指针修改常数

泛型指针

void用于表示函数无参或者无返回值。void *p所指向的实体单元字节数不定（泛型指针）。**因此，p可以指向任何实体，即可以将任何指针和任意类型变量的地址赋给p，或者调用时通过值参传递给参数p (如果p是函数参数)**

```
void *p; int i = 0; int * q = &i; p = q;
    void f(void *p); f(q);  //实参传递给形参等价于void *p = q;
```

对void *p指向的实体单元**赋值**时，类型或字节数必须确定, 所以必须进行强制类型转

```
    int x = 0;
    void *p = &x;//可以将任意类型的指针或变量地址赋给p
    *p = 345; //错误，p指向的单元字节数不确定
    *(int *)p = 345;  //OK
```

引用

引用(reference)为变量起了一个别名，引用类型变量的声明符用&来修饰变量名：

引用的本质还是指针 引用变量在功能上等于一个指针常量

```
int   &r = i; //定义一个引用变量r，引用了变量i，r是i的别名
    定义引用变量时必须马上初始化，即马上指明被引用的变量。
        int   &r2;  //编译错误，引用必须初始化，即指定被引用变量
```

左右值

可以寻址的对象（变量）是左值，不可以寻址的对象（变量）是右值。这里的可以寻址就是指是否可以用&运算符取对象（变量）的地址。 左值可以在右边和左边 右值只能右边

常量左值只能在右边 右值都可以用左值替换

区分左值和右值的另一个原则就是：左值持久、右值短暂。左值具有持久的状态（取决于对象的生命周期），而右值要么是字面量，要么是在表达式求值过程中创建的临时对象。

//i++等价于用i作为实参调用下列函数//第一个参数为引用x，引用实参，因此x = x + 1就是将实参+1；第二个int参数只是告诉编译器是后置++int

```
int operator++(int &x, int)   {     int tmp= i; //先取i的值赋给temp    x = x + 1;
return tmp; }
```

因此i++ = 1不成立，因为1是要赋值给函数的返回值，而函数返回后，tmp生命周期已经结束，不能赋值给tmpi++等价于operator++(i, int),实参i传递给形参x等价于int &x = i;

```
int& operator++(int &x)
{
    x = x + 1;
    return x;
}
```

引用在逻辑上是“幽灵”，是不分配物理内存的，因此无法取得引用的地址，也不能定义引用的引用，也不能定义引用类型的数组。引用定义时必须初始化，一旦绑定到一个变量，绑定关系再也不变

```
/int &*p = &ri;      //错误：不能声明指向引用的指针(没有必要)
    //int & &rri = ri;  //错误：不能定义引用的引用
    //int  &s[4];   //错误：数组元素不能为引用类型，否则数组空间逻辑为0
但是可以：int * & p 一个指向指针的引用
```

定义了引用后，对引用进行的所有操作实际上都是作用在与之绑定的对象之上。

被引用的实体必须是分配内存的实体（能按字节寻址）寄存器变量可被引用，

因其可被编译为分配内存的自动变量。

位段成员不能被引用，计算机没有按位编址，而是按字节编址。

注意有址引用被编译为指针，存放被引用实体内存地址。引用变量不能被引用。对于int x; int &y=x; int &z=y; 并非表示z引用y， int &z=y表示z引用了y所引用的变量i。

```
struct A {
    int j : 4;      //j为位段成员 声明这玩意占4个bit 但是我们是按字节寻址的
    int k;
} a;
void f() {
    int i = 10;
```

```
    int &ri = i;     //引用定义必须初始化，绑定被引用的变量
    ri = 20;         //实际是对i赋值20
    int *p = &ri;    //实际是取i的地址，p指向i，注意这不是取引用ri的地址
    //int &*p = &ri;     //错误：不能声明指向引用的指针
    //int & &rri = ri;  //错误：不能定义引用的引用
    //int  &s[4];    //错误：数组元素不能为引用类型，否则数组空间逻辑为0

    register int i = 0, &j = i; //正确：i、j都编译为(基于栈的)自动变量
    int  t[6], (&u)[6] = t; //正确：有址引用u可引用分配内存的数组t
    int  &v = t[0]; //正确：有址引用变量v可引用分配内存的数组元素
    //int  &w = a.j;     //错误：位段不能被有址引用，按字节编址才算有内存
    int  &x = a.k;  //正确：a.k不是位段有内存
```

怎么读&这个符号呢？&+变量：变量是一个引用，这个引用指向一个xxx后面和运算顺序一样

```
int (&u)[6] = t;
```

u是一个引用，这个引用指向一个6个元素的数组，每个类型为int

引用初始化

引用初始化时，除了二种例外情况，引用类型都要与绑定的对象严格匹配：即必须是用求值结果类型相同的左值表达式来初始化。二种例外是：const引用 父类引用绑定到子类对象

```
int j = 0;
const int c = 100;
double d = 3.14;
int &rj1 = j;        //用求值结果类型相同的左值表达式来初始化
//int &rj2 = j + 10;     //错误：j + 10是右值表达式
//int &rj3 = c;      //错误：c是左值，但类型是const int，类型不一致
//int &rj4 = j++;        //错误：j++是右值表达式
//int &rd = d;       //错误：d是double类型左值，类型不一致
```

j+10是一个整体 这个整体无法寻址 假设是左值 你放在左边看看?

而const引用则是万金油，可以用类型相同（如类型不同，看编译器）的左值表达式和右值表达式来初始化。

```
int j = 0;
const int c = 100;
double d = 3.14;
const int &cr1 = j;      //常量引用可以绑定非const左值
const int &cr2 = c;      //常量引用可以绑定const左值
const int &cr3 = j + 10;     //常量引用可以绑定右值
const int &cr4 = d; //类型不一致，报警告错误     （VS2017）

int &&rr = 1;        //rr为右值引用
const int &cr5 = rr;     //常量引用可以绑定同类型右值引用
```

c++ 里面一个变量的地址是右值 没有持久的储存地址 你就考虑整体可不可以放在左边！！！

自学的引用指针 const翻译成只读的 接后面那个词

```
如何引用指针
        int ival = 1024;
            int * &pi_ref = &ival;
        //错误， &ival为右值,不能初始化非const引用
    因为只有const类型引用才能用右值初始化，因此必须要把pi_ref声明为const引用
            const int * &pi_ref = &ival; // 错误，为什么？
        //pi_ref还是非const引用，引用了一个const int *指针，该指针指向了const对象
            int *  const &pi_ref = &ival; //OK，这时pi_ref才是const 引用
```

引用初始化

```
    为什么非const引用不能用右值或不同类型的对象初始化？
    对不可寻址的右值或不同类型对象，编译器为了实现引用，必须生成一个临时（如常量）或不同类型的值，对象，引用实际上指向该临时对象，但用户不能通过引用访问。如当我们写
            double dval = 3.14;
                int &ri = dval;
编译器将其转换成
                 int temp = dval;  //注意将dval转换成int类型
                    int &ri = temp;
        如果我们给ri赋给新值，改变的是temp而不是dval。对用户来说，感觉赋值没有生效（这不是好事）。
    const引用不会暴露这个问题，因为它本来就是只读的。
    干脆禁止用右值或不同类型的变量来初始化非const引用比“允许这样做，但实际上不会生效”的方案好得多。
```

上面这个例子是类型不同

nt &可以赋值给const int &，const int &不能赋值给int &

记忆：const很伟大 可以引用吃掉所有的类型 可以被赋值 不允许自己去占领别人

右值引用

1. 右值引用：就是必须绑定到右值的引用。
2. 右值引用的重要性质：只能绑定到即将销毁的对象，包括字面量，表达式求值过程中创建的临时对象。 返回非引用类型的函数、算术运算、布尔运算、位运算、后置++，后置--都生成右值，
3. 右值引用和const左值引用可以绑定到这些运算的结果上。
4. c++ 11中的右值引用使用的修饰符是&&，如： int &&aa = 1; //实质上就是将不具名(匿名)变量取了个别名 aa = 2; //可以。
5. 匿名变量1的生命周期本来应该在语句结束后马上结束，但是由于被右值引用变量引用，其生命期将与右值引用类型变量aa的生命期一样。这里aa的类型是右值引用类型(int &&)，但是如果从左值和右值的角度区分它，它实际上是个**左值 有地址了**

```
但：const int &&y=2；//不可赋值：y=3;
同理：右值引用共享被引用对象的“缓存”，本身不分配内存。
int &&  *p; //错：p不能指向没有内存的无址引用
        int &&  &q; //错：int &&没有内存，不能被q引用
        int &  &&r; //错：int &没有内存，不能被r引用。
        int &&  &&s;    //错：int &&没有内存，不能被s引用
        int &&t[4];     //错：数组的元素不能为int &&：数组内存空间为0。
    const int a[3]={1,2,3};   int(&& t)[3]=a; //错：a是有址的，有名的均是有址的。&&不能引用有址的
            int(&& u)[3]= {1,2,3};    //正确，{1，2，3}是无址右值
```

这段话的意思是说，右值引用的变量的地址和右值的缓存地址一样 只是延长了生命 所以第一句话是p是一个指针 指向一个右值引用的int 右值引用的空间是缓存是右值 无法取地址（右值的特点） r是一个左值引用，引用的对象是一个右值引用 但是左值本身也是地址（地址常量） 所以不行 右值引用 引用对象是一个右值引用 但是右值引用本身是一个左值 所以不行； t是一个数组 每一个数组的元素是一个右值int引用 但是元素本身没有内容 所以报错

【】数组优先级最高

```
若函数不返回（左值）引用类型，则该函数调用的返回值是无址（右值）的；
    int &&x=printf(“abcdefg”);  //对：printf( )返回无址右值
        int &&a=2;  //对：引用无址右值
        int &&b=a;  //错：a是有名有址的，a是左值
        int&& f( ) { return 2; }
    int &&c=f( );       //对：f返回的是无址引用，是无址的
位段成员是无址的。
    struct A {   int a; /*普通成员：有址*/     int b : 3; /*位段成员：无址*/ }p = { 1,2 };
        int &&q=p.a;    //错：不能引用有址的变量，p.a是左值
        int &&r=p.b;    //对：引用无址左值
```

```
int&& f( ) { return 2; }
int &&c=f( );       //对：f返回的是无址引用，是无址的
```

这两句如何理解 f返回的不是一个右值引用 返回的是一个临时对象 只是因为它满足了右值引用的特性

其实，C++ 是允许将一个 **右值引用** 返回的对象绑定到另一个右值引用变量的。

枚举

```
enum  WEEKDAY {Sun, Mon, Tue, Wed, Thu, Fri, Sat};  //Sun=0, mon=1
    WEEKDAY  w1=Sun, w2(Mon);   //可用限定名WEEKDAY::Sun，
```

```
也可以为枚举元素指定值，哪怕是重复的整数值。
enum  E{e=1,  s,  w= -1,  n,  p};   //正确，s=2，p= 1和e相等
```

```
如果使用“enum class”或者“enum struct”定义枚举类型，则其元素必须使用类型名限定元素名
    enum struct RND{e=2, f=0, g, h};    //正确：e=2，f=0，g=1，h= 2
    RND m= RND::h;          //必须用限定名RND::h
    int  n=sizeof(RND::h);      //n=4，枚举元素实现为整数
```

枚举类型而已 和int一样

数组

```
char d[ ]=“abc”;        /自动计算
```

```
const char*p=“abc”;//sieof(p)=4, p[0]=‘a’,“abc”看作const char指针，注意必须加const
可以用abc[1]或者p【1】输出
```

constexpr

上面又说：constexpr变量必须用编译时可计算表达式初始化。

```
    字面值类型：对声明constexpr用到的类型必须有限制，这样的类型称为字面值类型（literal type）。
算术类型（字符、布尔值、整型数、浮点数）、引用、指针都是字面值类型
自定义类型（类）都不是字面值类型，因此不能被定义成constexpr
其他的字面值类型包括字面值常量类、枚举


    constexpr类型的指针的初始值必须是nullptr

指向具有固定地址的对象（全局、局部静态）。注意局部变量在堆栈里，地址不固定，因此不能被constexpr类型的指针指向
当用constexpr声明或定义一个指针时， constexpr仅对指针有效，即指针是const的，
```

```
constexpr函数：是指能用于常量表达式的函数，规定：
函数返回类型和形参类型都必须是字面值类型
函数体有且只有一条return语句
constexpr函数被隐式地指定为内联函数，因此函数定义可放头文件
constexpr函数体内也可以包括其他非可执行语句，包括空语句，类型别名，using声明
在编译时，编译器会将函数调用替换成其结果值，即这种函数在编译时就求值
constexpr函数返回的不一定是常量表达式(毕竟可以自动求值)，返回的结果会和传入的东西有关，决定返回的是否为常量表达式（虽然会在编译的时候自动求解 但是在编译过程中 这个值还是不知道的）
```

第三章

swtich

括号中的expression只能是 小于等于int的类型包括枚举。

inline

内联函数可在程序文件内或类内说明或定义，只能被当前程序文件的程序调用。它是局部文件作用域的，可被编译优化(掉)

编译会对内联inline函数调用进行优化，即直接将其函数体插入到调用处，而不是编译为call指令，这样可以减少调用开销，提高程序执行效率

若函数为虚函数、或包含分支(if, switch,?:,循环,调用)，或取过函数地址，或调用时未见函数体（函数体定义在调用处后面），则内联失败。失败不代表程序有错，只是被编译为函数调用指令。

这句话的意思是说 内敛函数需要提前知道函数的地址和预测未来的情况，所以跳转无法达到，因为跳转要是跳转的行为在汇编层次的地址难以预测，虚函数有动态绑定地址不确定，取过函数的地址的例子看下面，原因是是通过函数指针间接调用函数，编译器无法在编译时确定调用哪个函数（函数指针可以指向其他函数，不像名字定位那么确定），因此无法进行内联优化。未见函数体都不知道函数什么内容

内联失败是优化失败 不是运行失败

函数体可以先声明再定义

静态变量和函数也只有当前程序文件可以使用

main函数的参数

注意第1个参数为可执行程序.EXE的绝对路径名,第2个参数是传入的要求字符串长度的串

省略参数...

省略参数...表示可以接受0至任意个任意类型的参数。通常须提供一个参数表示省略了多少个实参。long sum(int n, ...) {

```
long sum(int n, ...) {
          long s = 0; int* p = &n + 1;    //p指向第1个省略参数
          for (int k = 0; k < n; k++)  s += p[k];
      return s;
    }

void main( ) {
          int a = 4;  long s = sum(3, a, 2, 3); //执行完后s=9
    }
```

函数默认参数

声明或定义函数时也可定义参数默认值，调用时若未传实参则用默认值。

函数说明或者函数定义只能定义一次默认值。

默认值所用的表达式不能出现同参数表的参数。意思是说在参数表里面的内容不可以用参数表来默认

所有默认值必须出现在参数表的右边 意思是说：如果从一个变量开始默认化 他的右边所有的值都要默认化

```
int u=3;
   int f(int x, int y=u+2, int z=3) { return x+y+z; }
   int w=f(3)+f(2,6)+f(1,4,7); //等价于w=f(3,5,3)+f(2,6,3)+f(1,4,7);
若同时定义有函数int f(int m, ...); 则调用f(3)可解释为调用int f(int m, ...); 或调用int
f(int x, int y=u+2, int z=3)均可，故编译会报二义性错误。
```

一般而言，由于计算机底层的逻辑 参数传递是从右传到左 这解释了为什么不能包含参数表里面的参数进行默认化 有一些机器是从左到右的

不能同时在声明和定义中定义缺省参数的(即使表达式相同)值

```
int  w=3,  b(int x=w);  //声明正确，指定缺省值x=w
int  u=++w;
int b(int x=w){return x;}
```

重载函数

重载函数：通过参数差异识别重载函数，即若参数的个数或者类型有所不同（至少一个参数类型不一致），则同名的函数被自动视为重载函数。 重载只与参数类型有关，与返回类型无关，若参数个数和类型完全相同、仅仅返回类型不同是不允许的。

在编译时编译器根据调用函数的实参决定使用哪个版本的重载函数（静态绑定，这也是为什么重载也被称为静态多态）

但这个例子编译器无法找到合适的函数入口地址，因此静态绑定失败。编译器会报错。

```
当使用缺省参数和省略号时，函数重载容易引起歧义
        void show (char * message){ … }
        void show (char * message ,int code = 0){ …}
        void show (char * message , …){ …}
        show (“Hello”); //到底调用哪个?
```

多义性：如果有多个函数满足了该定义的最优解函数，那么多义性报错

生命期

```
函数参数或自动变量的生命期当退出其作用域时结束。
静态变量的生命期从其被运行到的位置开始，直到整个程序结束。
全局变量的生命期从其初始化位置开始，直到整个程序结束。
通过new产生的对象如果不delete，则永远生存（内存泄漏）
```

第四章 类

class、struct或union可用来声明和定义类

可以先声明类

```
class 类型名;//前向声明
class 类型名{//类的定义
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;
};
```

```
class Circle {
   private:
         double radius ;

   public:
        Circle() ; \\构造函数
```

```
        Circle(double r)；\\构造函数的重载

        double findArea() ；}

#include “Circle.h”
Circle::Circle() { radius = 1.0;}\\在外面定义自己的类的函数 radius用在类的函数里面
Circle::Circle(double r) { radius = r;}\\在外面定义自己的类的函数 radius用在类的函数里面

double  Circle::findArea(){
    return radius * radius * 3.14;
}
```

函数成员的实现既可以放在类的外面，也可以内嵌在类定义体中（此时会自动成为内联函数）

但是数据成员的声明/定义顺序与初始化顺序有关（先声明/定义的先初始化）

```
class MyClass {
private:
    int a;
    int b;

public:
    MyClass(int x, int y) : b(y), a(x) {}  // 初始化列表中的顺序与声明顺序不同
};
```

在上面的代码中，`a` 和 `b` 是 `MyClass` 的数据成员，`a` 在 `b` 之前声明，但在构造函数的初始化列表中，`b` 被先初始化，然后是 `a`。虽然你在构造函数中指定了初始化的顺序，但它们实际上会按照声明顺序初始化：

1. **初始化顺序**
    - `a` 先初始化（因为它在类中声明在 `b` 之前）。
    - `b` 后初始化。

因此，即使在构造函数初始化列表中将 `b` 放在 `a` 之前，实际初始化的顺序仍然是先初始化 `a`，再初始化 `b`。

若函数成员在类定义体外实现，则在函数返回类型和函数名之间，应使用运算符“::”来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定义体结束标志**

在类定义体中允许对数据成员定义默认值，若在构造函数的参数列表后面的“：”和函数体的“{”之间对其进行了初始化（在成员初始化列表进行初始化），则默认值无效，否则用默认值初始化

```
public:
    MyClass(int x, int y) : b(y), a(x) {}  // 初始化列表中的顺序与声明顺序不同
};
```

构造函数和析构函数都不能定义返回类型。构造函数的参数表可以出现参数，因此可以重载。

构造函数的定义和数据成员

```
    Circle c1; //用户没指定参数，调用缺省构造函数
    Circle c2( 5.0 ); //调用带参数构造函数，实参5.0传给形参r，将对象c2的半径设为5.0
    cout  << c1.radiuse; //1.0
    cout  << c2.radiuse; //5.0
```

构造函数和析构函数

是类封装的两个特殊函数成员，都有固定类型的隐含**参数**this（对类A，this指针为A* const this）

构造函数：函数名和类名相同的函数成员。可在参数表显式定义参数，通过参数变化实现重载。析构函数：函数名和类名相同且带波浪线的参数表无参函数成员。故无法通过参数变化重载析构函数

类和重定义类型一模一样 在 A* const this里面，你可以把A看成是int，这样就理解了为什么这是一个A类型的指针

构造函数和析构函数的使用

定义变量或其生命期开始时自动调用构造函数，生命期结束时自动调用析构函数。同一个对象仅自动构造一次。构造函数是唯一不能被显式（人工，非自动）调用的函数成员。

其中：**对象创建**：当你声明一个对象时，构造函数会自动被调用，进行初始化。

对象销毁：当对象超出作用域时（例如函数返回或**显式删除对象**时），析构函数会自动被调用，进行清理操作。

同一个对象仅自动构造一次。构造函数是唯一不能被显式（人工，非自动）调用的函数成员。

```
class MyClass {
public:
    MyClass() {
        std::cout << "构造函数被调用" << std::endl;
    }
    void show() {
        std::cout << "普通函数 show 被调用" << std::endl;
    }
};

int main() {
    MyClass obj;  // 正常创建对象，构造函数自动调用
    obj.show();   // 正常调用成员函数
    // obj.MyClass();  // 错误：不能显式调用构造函数
    Myclass obj2;//正常创建一个叫做obj2的对象
    return 0;
}
```

那如何使用构造函数呢?

```
 Circle c1; //用户没指定参数，调用缺省构造函数
 Circle c2( 5.0 ); //调用带参数构造函数，实参5.0传给形参r，将对象c2的半径设为5.0
```

析构函数的使用细节

析构函数在变量 (对象)的生命期结束时被自动调用一次，通过new产生的对象需要用delete手动释放(自动调用析构)在heap里面

```
//new出来的对象在堆里（Heap）
     MYSTRING *p = new MYSTRING(“ABC”);
    delete(p); //delete new出来的对象的析构是程序员的责任，delete会自动调用析构函数
```

析构函数 是倒着析构的

```
#include <string.h>
#include <stdlib.h>
#include <iostream.h>
struct MYSTRING{
    char *s;
    MYSTRING (const char *) ;
    ~ MYSTRING ( ) ;
};
MYSTRING :: MYSTRING (const char *t)   {
   s= (char *) malloc  (strlen (t) +1)   ;
    strcpy (s, t)   ;
    cout<<"Construct: "<<s;
}
MYSTRING ::~ MYSTRING ( )   {
//防止反复释放内存
    if (s==0)    return;
    cout<<"Deconstruct:"<<s;
    free (s)   ;
    s=0;  //提倡0代替NULL指针
}

void main (void)   {
   MYSTRING s1 ("String 1\n")   ;
   MYSTRING s2 ("String 2\n")   ;
   MYSTRING ("Constant\n")   ;
    cout<< "RETURN\n";
   s1.~ MYSTRING ( ) ; //显式析构s1
}//自动析够s2,  s1，不会出错
```

上面这个代码有几个地方需要注意！首先是析构函数里面的那个s 那个s是在类里面的变量 所以使用的时候直接就可以用了的 还有一个MYSTRING ("Constant\n")这个没有定义一个变量 所以其实是一个常量 记住类是重定义数据类型

```
Construct:String1
Construct:String2
Construct:Constant
Deconstruct:Constant
RETURN
Deconstruct:String1
Deconstruct:String2
```

变量？对象?

全局变量 (对象)： main执行之前由开工函数调用构造函数，main执行之后由收工函数调用析构函数。

局部自动对象 (非static变量)： 在当前函数内对象实例化时自动调用构造函数, 在当前函数正常返回时自动调用析构函数。

局部静态对象 (static变量)： 定义时自动调用构造函数, main执行之后由收工函数调用析构函数。 常量对象：在当前表达式语句定义时自动调用构造函数，语句结束时自动调用析构函数

如何使用定义的类的成员呢?

和**结构体使用一模一样**，名字加.或者指针加->

this

在 C++ 中，`this` 是一个隐式的指针，**它指向当前对象的地址。**每当你在类的成员函数中使用 `this` 时，它代表的是调用该成员函数的对象实例。`this` 指针只有在成员函数内有效。

复习课堂上讲过，int * = const int *是不成立的，换成char类型是一样的

退出（不重要 ppt都隐藏了）

abort 是不会执行所有对象的析构函数

exit不能自动执行自动局部自动对象的析构函数，static和全局倒是可以

return执行所有析构函数

```
#include <process.h>
#include “string.cpp”   //不提倡这样include：因为string.cpp内有函数定义
STRING x("global"); //自动调用构造函数初始化x
void main(void){
      short error=0;
      STRING y("local");    //自动调用构造函数初始化y
      switch(error) {
      case  0: return;      //正常返回时自动析构x、y
      case  1: y.~STRING( );    //为防内存泄漏，exit退出前必须显式析构y
                  exit(1);
```

```
      default: x.~STRING( );     //为防内存泄漏，abort退出前须显式析构x、y
        y.~STRING( );
        abort( );
       }
}
```

接受和删除自动生成的函数：delete和default

```
struct A {
    int x=0;
    A( ) = delete;       //删除由编译器产生构造函数A( )
    A(int m): x(m) { }
    A(const A&a) = default; //接受编译生成的拷贝构造函数A(const A&)
};
void main(void) {
    A x(2);          //调用程序员自定义的单参构造函数A(int)
    A y(x);          //调用编译生成的拷贝构造函数A(const A&)
    //A u;           //错误：u要调用构造函数A( )，但A( )被删除
    A v( );          //正确：声明外部无参非成员函数v，且返回类型为A
}//“A v( );”等价于“extern  A v( );”
```

=default就是内联函数 在类外等于default就不是内联函数（A::A(const A&a) = default;）

delete表示删除函数 不可以定义该类型的函数 但是可以修改他的参数重定义 只是不能一模一样

成员类型

private：私有成员，本类函数成员可以访问；派生类函数成员、其他类函数成员和普通函数都不能访问。protected：保护成员，本类和派生类的函数成员可以访问，其他类函数成员和普通函数都不能访问

public：公有成员，任何函数均可访问。

类的友元不受限制 可以访问所有成员 构造函数和析构函数也可以定义声明域

强制类型转换可以突破访问权限的限制

怎么理解这个“访问”？其实很简单，就是有没有出现

类的内联

类内部定义函数会自动定义为内联函数 在类外面定义的函数要加inline才能内联

匿名类只能在类体内定义函数不然在类外面定义：：符号前面怎么说明所属类

函数局部类只能在类里面定义函数

什么是函数局部类？就是在函数里面定义的类

```
class COMP {
    double  r,  v;
public:
    COMP (double rp, double vp)   { r=rp; v=vp; } //自动内联
    inline double getr ( );  //inline保留字可以省略，后面又定义
    double getv ( )   ;
};
```

```
inline double COMP::getv ( )   { return v;  }    //定义内联
void main (void)   {
    COMP  c(3,  4)   ;
    double  r=c.getr ( )   ;    //此时getr的函数体未定义，内联失败
    double  v=c.getv ( )   ;    //函数定义在调用之前，内联成功
}
inline double COMP::getr ( )   { return r;  }     //定义内联
```

new和delete

new为运算符，操作数是类型表达式，先用malloc再用构造函数

malloc为函数参数是值表达式，内存分配初始化为0

free和delete参数为指针类型表达式，前者直接释放内存，后者先析构再释放内存，delete可以接受任何指针实参，无论类型

怎么用?

new+类型表达式可以+ {} 可以使分配的内存为0 {}里面的数值用于初始化数组元素， () 为初始化一个值 但是运算符返回的是一个地址

初始化一个值为10的变量的地址：int* p=new int (10) 初始化一个有10个int的数组的指针：int *q=new int [10];这里的int【10】和int（10）直接理解为类型，注意多维数组只有最高位可以写为指针，其他维要照搬，多维数组第一维可以使任意表达式

delete +指针 或者delete+[]+数组指针 这种模式可以删除所有维度的数独 对简单类型 可以用delete+指针代替 其他的对象类型只能用这个

free，malloc和delete，new的区别

其实很直接，对象有析构函数就用new，delete反之就用free和malloc。

具体的解释如下：

```
    #include <alloc.h>
    class  ARRAY{           //class体的缺省访问权限为private
        int *a, r, c;
    public:                 //访问权限改为public
        ARRAY(int x, int y);
        ~ARRAY( );
    };
    ARRAY::ARRAY(int x, int y){
        a=new int[(r=x)*(c=y)]; //可用malloc：int为简单类型
    }
ARRAY::~ARRAY( ){     //在析构函数里释放指针a指向的内存
    if(a){ delete [ ]a; a=0;}   //可用free(a), 也可用delete a
}
ARRAY  x(3, 5);               //开工函数构造，收工函数析构x
void main(void){
    ARRAY  y(3, 5), *p;       //退出main时析构y
    p=new ARRAY(5, 7);   //不能用malloc，ARRAY有构造函数
    delete  p;           //不能用free，否则未调用析构函数
}
```

你看上面的代码，p指向一个对象，这个对象里面有一个指针a，用new给他构造数组 如果我们对p用malloc，他只能创建这个对象，此时里面的a对应一个nnull；拥有这个对象的内存无法使用构造函数，也就无法实现数组的创建；同理，如果我们free了p，它无法释放数组的内存，free它只是释放了除开数组之外的其他变量的空间 就是说 new在malloc之后（分配类的内存），根据类的定义要占的内容构造函数构造内容；delete相反，都是两步走

new还可以重新构造已经析构的对象 节约栈的空间?

```
STRING  x ("Hello!"), *p=&x;
    x. ~STRING ( );
    new (&x) STRING ("The World");
    new (p) STRING ("The World");
```

this指针

最重要的this指针是非静态函数成员，包括析构和构造的隐含的第一个参数，类型是函数成员所属对象的指针常量：A*const this

this存的是函数的成员 除了静态局部 其他都没有this 可以用*this调用普通 const和volatile成员

```
class A{
    int age;
public:
    void setAge( int age){
        this->age = age;    //this类型：A*const this
    }
}

A a;
a.setAge(30);
//函数setAge通过对象a被调用时，setAge函数的第一个参数是
//A*const this指针，指向调用对象a。this->age = a.age = 30
//*this就是对象a
```

小tip函数签名const

```
const TREE* TREE::find(int v) const {   //this指向调用对象
    if(v==value) return this;   //this指向找到的节点
    if(v<value) //小于时查左子树，即下次递归进入时新this=left
        return  left!=0?left->find(v):0;
    return  right!=0?right->find(v):0; //否则查右子树
} //注意find函数返回类型必须是const TREE *
```

后面那个const表示函数不会进行任何值的修改 一旦我们在这里用了const 对应的类定义也要const

```
这个const是修饰this指针，this类型: const TREE * const this
const TREE *find(int) const;
```

初始化

就地初始化

```
class A {
public:
    int x = 10;  // 就地初始化
    double y = 5.5;  // 就地初始化
};
```

列表初始化

```
class A {
public:
    int x = 10;  // 就地初始化
    double y = 5.5;  // 就地初始化

    A(int a, double b) : x(a), y(b) {  // 成员初始化列表
        // 构造函数体
    }
};
```

可以用=，{} 这样子初始化：

```
int i = 10;      //可以用=号初始化
    double d{ 3.14 };        //可以用{}初始化
    int j = { 0 };
```

用括号初始化列表初始化

```
B() :i(10),j(100),r(j),PI(6.28) { }// 成员初始化列表初始化
```

可以定义名字相同但是类型不同的变量 不能定义名字相同类型相同的变量 不能用括号是因为函数参数表也是括号

类里面要是有const和&，还有其他类的非静态数据成员 必须在使用他们初始化（如果没有就地初始化必须有构造函数）

1. 如果未定义或生成构造函数，且类的成员都是公有的，则可以用 `{}` 的形式初始化（和 C 的结构初始化一样）

这是指如果类没有显式定义构造函数，且类的成员变量是 **公有的**，则可以直接用花括号 `{}` 对象进行初始化。这种初始化方式类似于 C 语言中结构体的初始化方式。此时，编译器会自动为你生成一个 **默认构造函数。**

示例：

```
cpp复制代码class A {
public:
    int x;
    double y;
};

int main() {
    A obj = {10, 20.5};  // 使用花括号初始化成员变量
    cout << obj.x << " " << obj.y << endl;  // 输出 10 20.5
    return 0;
}
```

在这个例子中，`obj` 被初始化为 `{10, 20.5}`，相当于 `obj.x = 10; obj.y = 20.5;`。这种初始化方式适用于没有显式构造函数的类，且成员是公有的。

2. 联合仅需初始化第一个成员

在联合（`union`）中，**所有成员共享同一内存位置**，因此一次只能存储一个成员的值。因此，初始化联合时，只需要初始化第一个成员，后面的成员不需要初始化。

示例：

```
cpp复制代码union U {
    int x;
    double y;
};

int main() {
    U u = {10};  // 只初始化第一个成员 x
    cout << u.x << endl;  // 输出 10
    return 0;
}
```

这里的 `union` 只初始化了 `x`，因为 `union` 的特性是所有成员共享内存，因此只需要初始化一个成员。注意，只有最后一个被赋值的成员的值才会保持有效。

3. 对象数组的每个元素都必须初始化，默认采用无参构造函数初始化

对于一个对象数组，C++ 要求数组的每个元素都必须进行初始化。如果没有显式初始化，编译器会使用**默认构造函数**（如果有）来初始化每个对象。

示例：

```
cpp复制代码class A {
public:
    int x;
    A() : x(0) {}  // 默认构造函数
};

int main() {
    A arr[3];  // 使用默认构造函数初始化每个元素
    for (int i = 0; i < 3; ++i) {
        cout << arr[i].x << " ";  // 输出 0 0 0
    }
    return 0;
}
```

在这个例子中，`arr` 是一个 `A` 类型的对象数组，每个元素都通过 `A()` 的默认构造函数初始化。

4. 单个参数的构造函数能自动转换单个实参值成为对象

这意味着，如果你的类有一个 **单参构造函数**，并且你提供了一个合适类型的实参，编译器会自动将该实参转换为对象。这是 **C++ 中的隐式类型转换**（或称为**单参数构造函数的隐式调用**）。

示例：

```
cpp复制代码class A {
public:
    int x;
    A(int val) : x(val) {}
};

int main() {
    A obj = 10;  // 隐式调用单参构造函数 A(10)
    cout << obj.x << endl;  // 输出 10
    return 0;
}
```

在这个例子中，`obj = 10` 实际上是调用了 `A(int)` 构造函数，并将 `10` 传递给它。

5. 如果类未自定义构造函数，编译器会尝试自动生成构造函数

当你没有为类显式定义构造函数时，编译器会自动为你生成一个 **默认构造函数**，这个构造函数会将类的成员变量初始化为默认值：

- 对于内置类型成员，通常会没有默认值，可能是未定义的。
- 对于类类型成员，会调用其默认构造函数。

示例：

```
cpp复制代码class A {
public:
    int x;
};

int main() {
    A obj;  // 编译器生成默认构造函数
    cout << obj.x << endl;  // 未定义行为，因为 x 没有被初始化
    return 0;
}
```

6. 一旦自定义构造函数，将不能接受编译生成的构造函数，除非用 `=default` 接受

当你自定义了构造函数时，编译器将不会自动生成默认构造函数（除非你显式地指定 `= default`）。如果你希望同时拥有自定义构造函数和编译器自动生成的默认构造函数，可以显式声明 `= default` 来告诉编译器生成默认构造函数。

示例：

```
cpp复制代码class A {
public:
    int x;
    A(int val) : x(val) {}  // 自定义构造函数
    A() = default;  // 显式声明编译器生成默认构造函数
};

int main() {
    A obj1;  // 调用默认构造函数
    A obj2(10);  // 调用自定义构造函数
    return 0;
}
```

在这个例子中，`A()` 使用 `= default` 显式声明了一个默认构造函数。否则，编译器不会自动生成默认构造函数。

7. 用常量对象做实参，总是优先调用参数为 `&&` 类型的构造函数；用变量等做实参，总是优先调用参数为 `&` 类型的构造函数（结合法则4来看）

这是关于 C++ 中 **右值引用** 和 **左值引用** 的一个重要概念。当你将常量对象传递给函数时，C++ 会优先调用右值引用（`&&`）的构造函数，而当传递的是一个变量时，C++ 会优先调用左值引用（`&`）的构造函数。

示例：

```
cpp复制代码class A {
public:
    int x;
    A(int val) : x(val) {}
    A(A&& other) : x(other.x) {  // 右值引用构造函数
        cout << "Move constructor\n";
    }
    A(const A& other) : x(other.x) {  // 左值引用构造函数
        cout << "Copy constructor\n";
    }
};

int main() {
    A obj1(10);  // 调用 A(int) 构造函数
    A obj2 = obj1;  // 调用拷贝构造函数
    A obj3 = A(20);  // 调用移动构造函数（右值引用）
    return 0;
}
```

在这个例子中：

- `obj1` 是一个变量，因此会调用 **拷贝构造函数**（`const A&`）。
- `obj3` 是一个临时对象，因此会调用 **移动构造函数**（`A&&`）。

默认构造函数

1) 如果为数据成员提供了类内初始值，则工作成功

这是指在类定义时，如果你为类的成员变量提供了初始值（即 **类内初始化**），那么合成的默认构造函数会成功工作。例如：

```
cpp复制代码class A {
public:
    int x = 10;  // 类内初始化
    double y = 20.5;  // 类内初始化
};
```

在这种情况下，即使没有定义构造函数，编译器会自动生成一个默认构造函数，并且 `x` **和** `y` **会被初始化为 10 和 20.5**，因为它们有类内初始值。

2) 如果没有为数据成员提供类内初始值：和int之类的一样 类的对象在全局会为0 在内部会随机值报错

这种情况下，是否能成功生成合成的默认构造函数取决于数据成员的类型和对象的存储方式。具体来说，分为以下几种情形：

A) 非 const、非引用内置类型数据成员：

- **如果类的对象是全局的或局部静态的**，这些成员会被 **默认初始化为0。**在这种情况下，合成的默认构造函数会正常工作。

```
cpp复制代码class A {
public:
    int x;  // 没有类内初始化
};

A obj;  // obj 是全局的或局部静态的，x 会默认初始化为 0
```

- **如果类的对象是局部的**，那么 **这些非 const、非引用内置类型的数据成员不会自动初始化**，会导致编译器报错，提示未初始化。这意味着编译器无法保证数据成员的初始值，因此合成的默认构造函数会失败。实际上 只要不访问 就没有错 和int类型一样

```
cpp复制代码class A {
public:
    int x;  // 没有类内初始化
};

void func() {
    A obj;  // 局部对象，x 不会自动初始化，编译器会报错
}
```

- **如果对象是通过 `new` 在堆上分配的**，则合成的默认构造函数不会报错，但成员会包含**随机值。**这是因为堆内存是未初始化的，C++ 不会自动为堆分配的内存成员初始化值。

```
cpp复制代码class A {
public:
    int x;  // 没有类内初始化
};

void func() {
    A* p = new A;  // x 会有随机值，编译器不会报错
    cout << p->x << endl;  // 输出随机值
}
```

B) 包含 `const` 或引用成员：引用和const都是常量类型 必须定义及时初始化 这里也和int一样写了const不初始化直接报错

- 如果类有 `const` 类型或引用类型的数据成员，则 **合成的默认构造函数无法正常工作。**因为 `const` 类型和引用类型的数据成员在对象创建时必须被初始化，**编译器无法为它们提供默认值**，因此会报错。

```
cpp复制代码class A {
public:
    const int x;  // const 类型成员
    int& y;  // 引用类型成员
};

A obj;  // 编译器会报错，无法合成默认构造函数
```

对于 `const` 和引用成员，编译器需要你在构造函数中显式地提供它们的初始值，因为它们无法通过合成的默认构造函数来初始化。

C) 包含其他类类型成员且该类型没有默认构造函数：

- 如果类中包含一个其他类类型的成员，而这个成员 **没有默认构造函数**，那么编译器就无法为这个成员生成一个默认值。因此，合成的默认构造函数会失败并报错。

```
cpp复制代码class B {
public:
    B(int v) {}  // B 类型的构造函数没有默认构造函数
};

class A {
public:
    B b;  // B 类型没有默认构造函数
};

A obj;  // 编译器报错，因为无法合成 A 的默认构造函数
```

在这种情况下，编译器无法为 `B` 类型的成员 `b` 调用默认构造函数，因此合成的默认构造函数无法正常工作。

对于const、引用成员，没有默认构造函数的对象成员，只能就地初始化和在成员初始化列表里初始化，不能在构造函数体内被赋值。

类的构造实际上和int的一模一样，只不过有隐形转化，把一些常量从int变成了类的类型，你看起来就像是一个int给类里面的一个int赋值，实际上不是的，实际上是一个int变成了类的类型，再赋值给整个类 构造函数的意思 实际上描述的就是如何构造这个int变成一个类的方法 当我们用等号去初始化的时候 就是隐形的构造 当我们不用等号而是用花括号和中括号的时候 实际上也是像上面的理解 把数字转化为对应的类的类型去赋值 只不过等号有一个明显的类型转化罢了 需要注意的是 直接用等号的构造函数只能用于单参数构造函数有效 原因很简单 多参数怎么使用等号?

```
class Integer {
public:
    int value;
public:
    //隐式地提供了从int到Integer的转换功能
    Integer(int value): value(value){}
};
```

```
class Age {
public:
    Integer age{ 0 };
public:
    //隐式地提供了从Integer到Age的转换功能
    Age(Integer i) :age(i) {}
};
```

类型转化只有一次 他的这个一次就是构造函数里面那个括号里面的类型生成一个对应的类，这个类的元素内容是那个括号的内容 对应类的数据成员 没有跨阶级传递

```
void test() {
// 隐式地将int转换成Integer
Integer integer = 25; // 对于转换构造函数（单参数），可以用=初始化；也可以Integer b(35)
Age age = integer;  //隐式地将Integer转换成Age

//但编译器只能自动进行一次转换
//Age a = 100; //编译报错：不存在int到Age的转换构造函数。如果成立，就有二次隐式转换：int->Integer->Age
//只能这样
Age a = Integer(100);   //先显式地int->Integer,再隐式地Integer->Age  Integer(100)是一个Integer类型的立即数，类似int里面的100
}
```

explicit 定义构造函数类型 抑制类型转化的

通知C++标准库（例如容器 std::vector），MyString类的移动函数不会抛出异常。 以std::vector为例，C++标准库的容器大小是可以自动增长的。当调用std::vector::puch_back方法往里面不断放对象时，如果容器内部的buffer到达上限，容器会自动将buffer按一定的策略扩容。这时需要把以前buffer里的对象拷贝到扩容后的新buffer里，如果容器里对象有移动构造函数且声明为noexcept，这时容器会优先调用对象的移动构造函数来移动对象；如果容器里对象的移动构造函数没有声明为noexcept，则容器会调用对象的拷贝构造函数来拷贝对象，这样会影响性能。

移动构造函数的noexcept声明只对容器扩容时产生影响

```
void test_explicit() {
//编译报错：不存在从int到ExplicitInteger的构造函数，这种用=来初始化的形式不再支持
//ExplicitInteger i = 10;

//编译也报错：标记为explicit的构造函数也不支持={}形式
//ExplicitInteger j = { 10 };

ExplicitInteger k(0);   //只能以这种形式

ExplicitInteger l{ 0 }; //或以这种形式
}
```

注意 虽然等号模式的隐形构造方法只能用于一个参数的构造 但是要是只有一个参数没有默认化 也是可以的

```
class A {
public:
    int i;
    int j;
public:
    A(int i, int j = 0) {}  // 同样是转换构造函数
};
A a = 0;     //这样初始化也成立
class A {
public:
    int i;
    int j;
public:
    explicit A(int i, int j = 0) {}  // 不是转换构造函数
};
//A a = 0;       //这样就不成立
```

坑坑的

```
class A{
        int  a;
    public:
        A(int x) { a=x;} //重载构造函数，自动内联
        A( ){ a=0; }         //重载构造函数，自动内联
    };
    class B{
        const  int  b;  //b没有就地初始化
        int  c, &d, e, f;   //b,d,g,h都没有就地初始化，故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构造函数体前未出现h，故h用A( )初始化
    B(int y): d(c), c(y), g(y) , b(y) , e(y){//自动内联
        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
    A *p=new A[3]{ 1, A(2)}; //初始化的元素为A(1), A(2), A(0)
    B b(7), z=(7,8);     //B z=(7,8)等价于B z(8),等号右边必单值
    delete [ ]p;         //防止内存泄漏：new产生的所有对象必须用delete
}
```

解释一下 其实我们可以把那个列表初始化A *p=new A[3]{ 1, A(2)}; **看成是A【0】=1，A【1】=A（2），A【2】取默认的值 就说的通了**

逗号运算符，取右边那个值

拷贝构造函数

如果class A的构造函数的第一个参数是自身类型引用(const A &或A &), 且其它参数都有默认值（或没有其它参数），则此构造函数是拷贝构造函数。

```
class ACopyable{
    public:
        ACopyable() = default;
        ACopyable(const ACopyable &o);  //拷贝构造函数
    };
```

如果没有为类定义拷贝构造函数，编译器会为我们定义一个合成的默认拷贝构造函数，编译器提供的合成的默认拷贝构造函数原型是ACopyable(const ACopyable &o); 默认带const

用一个已经构造好对象去构造另外一个对象时会调用拷贝构造函数

```
A o1;
A o2(o1);
A o3 = o1;
A o4{o1};
A o5 = {o1};
```

这里注意的细节是由于拷贝构造函数参数可以是const和不带const 我们要注意用的时候一定只能把非const给const不可以反过来

拷贝构造函数一般在传递值参和返回值参各自一次 分别解释为：1把对象作为实参传递给非引用形参 2返回类型为非引用类型的函数返回一个对象 拷贝给返回处 引用类型的时候则都是一个绑定的关系 就是他本身 所以不用拷贝构造 看下面的例子：

```
class ACopyable{
public:
    ACopyable() = default;
    ACopyable(const ACopyable &o){//拷贝构造函数
         cout << "ACopyable is copied" << endl;
    }
};

ACopyable FuncReturnRvalue(){  return ACopyable(); }   //函数返回非引用类型
void FuncAcceptValue(ACopyable o){  }            //函数接受值参
void FuncAcceptReference(const ACopyable &o){ }     //函数接受引用

int main(){
    cout << "pass by value: " << endl;
    FuncAcceptValue(FuncReturnRvalue());    // 应该调用两次拷贝构造函数
    cout << "pass by reference: " << endl;
    FuncAcceptReference(FuncReturnRvalue()); //应该只调用一次拷贝构造函数
    return 0;
}
```

一次发生在FuncReturnRvalue()返回临时对象到声明处，一次是把声明处的对象传给形参o

拷贝构造函数的内容

1. **首先是非静态成员的拷贝，按成员依次拷贝，内置类型，指针，引用，普通数组，直接拷贝**

2. **类 类型，用该类的拷贝构造函数；同理还有类 类型的数组，逐个使用拷贝构造**

浅拷贝：按成员依次拷贝 函数参数为值参而把实参传给值参 不是引用

问题：拷贝的内容要是有指针，我们只拷贝了一个指针的内容，它对应的地址的内容没有拷贝，两个对象指向同一片内容。当函数返回的时候，浅拷贝导致的那个形参要析构，释放指针成员存储单元（delete），实参无法再次访问

深拷贝：在传递实参给形参的时候，把形参对象的指针成员分配新空间，将实参成员的指针所指向的单元拷贝过去 参数必须是类的引用，推荐用const A& 而且要自己写这个函数

```
struct A {
     int *p;
    int size;
    A (int s):size(s), p(new int[size]){ }
    A( ):size(0),p(0){ }
    A (const  A &r) ;   //A  (const A&r)    自定义拷贝构造函数
    ~A( ) { if(p) {delete p; p=0;}}
};
A::A (const A  &r)   {       //实现时不是浅拷贝，而是深拷贝
    p=new int[size=r.size];          //构造时p指向新分配内存
    for (int i =0;  i<size;  i++)     p[i]=r.p[i] ;
}
void f(A a) {};  //函数参数为值参
 A o1(20);
 f(o1);      //调用自定义的拷贝构造函数，不用浅拷贝使o.p=a.p，
                 //而是o.p!=a.p 但是二块内存的内容一样
```

与等号的重载联动：

1. 为每个类自定义形为A(const & A) 的拷贝构造函数，而且要实现为深拷贝。
2. 同时要自己重载=运算符，重载赋值运算符是也要实现为**深拷贝赋值**
3. 如果自己没定义拷贝构造函数，编译器会自动添加一个拷贝构造函数，其实现为浅拷贝。
4. 如果没有为类A重载=号运算符，编译器会自动为A添加一个=号运算符的重载函数，其实现为浅拷贝赋值

```
class A{
    public:
        //赋值运算符返回非const &，参数是const 引用
        A & operator=(const A &){ …}
    }
```

上面为普通形式

下面是深拷贝：复习一下：this指向当前对象 也就是谁是老大 谁被赋值 谁在成长

```
MyString& MyString::operator=(const MyString &rhs) { //实现为深拷贝赋值
    char *t = static_cast<char *>(malloc(rhs.len));
    strcpy(t,rhs.s);
    //把右边对象的内容复制到新的内存后，再释放this->s,这样做最安全
    if(this->s) {   delete[] this->s; }  //这里为什么需要判断？
    this->s = t;
    return *this;  //最后必须返回*this
};
```

这里的逻辑是rhs把内容传给t空间，s1先看看自己的s空不空，然后指向t。不判断可能导致二次delete的多次对delete过的再delete 不好

为什么拷贝构造函数一定类型是引用?

如果传的是形参，一个值，比如说：Myclass obj2 = obj1 ，那么，首先要调用一个拷贝构造传入obj1的副本作为参数给obj2，那谁来创建obj1的副本呢？还需要一个拷贝构造函数，那谁来创建传入obj1副本的参数呢？还得来一个

移动构造和移动赋值

参数为值参以及返回类型为值，会出现频繁拷贝

长得像：

```
class A{
    public:
        A(A && o) noexcept; //移动构造
        A & operator=(A && rhs) noexcept; //移动赋值
    }
```

移动构造和移动赋值函数参数都是右值引用，都必须被声明为noexcept(不会抛出异常)

先来个四种函数的模板

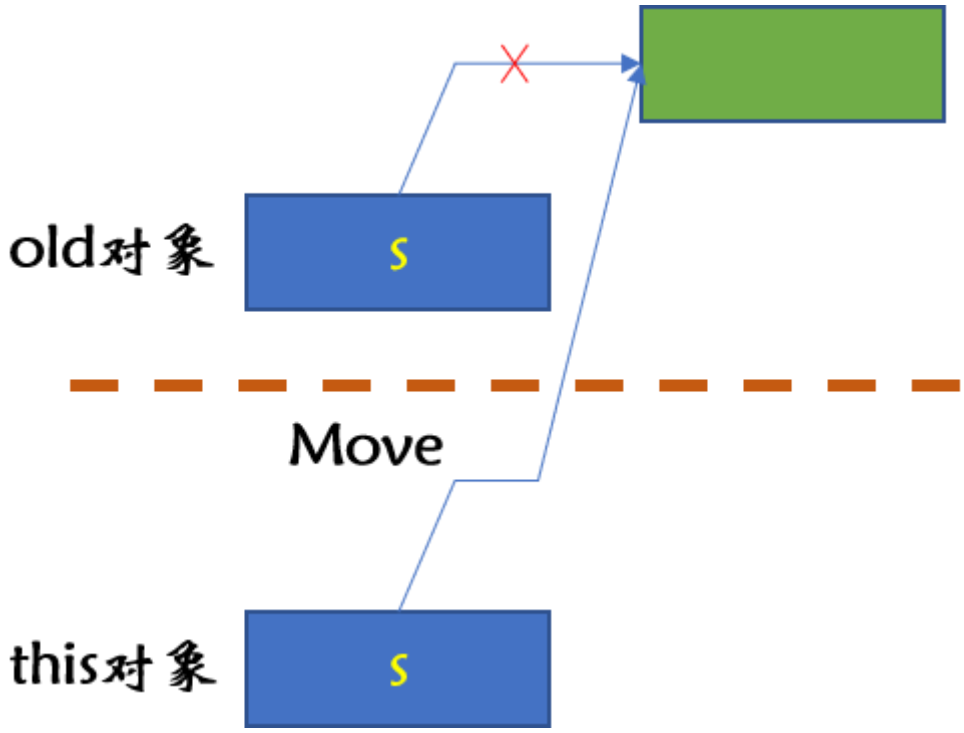
```
MyString(const char *t = "");
    ~MyString();
    MyString(const MyString &old); //拷贝构造函数
    MyString & operator=(const MyString &rhs); //重载=
    MyString(MyString &&old) noexcept ; //移动构造
    MyString &operator=(MyString &&rhs) noexcept ; //移动=
MyString::MyString(const char *t):
    len(strlen(t)+1), s(static_cast<char *>(malloc(len))) {
    strcpy(s,t); //拷贝时包含\0
    cout << "Constructor:MyString: " << s << endl;
}

MyString::~MyString() {
    if(s != nullptr){
        cout << "Destructor:MyString: " << s << endl;
        delete[] s;
        s = nullptr;
        len = 0;
    }
}
MyString::MyString(const MyString &old):
```

```
        len(old.len), s(static_cast<char *>(malloc(len))){
    strcpy(s,old.s);
    cout << "Copy Constructor:MyString: " << s << endl;
}深拷贝
MyString& MyString::operator=(const MyString &rhs) {
    char *t = static_cast<char *>(malloc(rhs.len));
    strcpy(t,rhs.s);
    //把右边对象的内容复制到新的内存后，再释放this->s,这样做最安全
    if(this->s){ delete[] this->s; }
    this->s = t;
    this->len = rhs.len;
    cout << "Copy =:MyString: " << s << endl;
    return *this;  //最后必须返回*this
}深拷贝重载
```

```
MyString::MyString(MyString &&old) noexcept
        :s(old.s), len(old.len) {

   //令old进入安全的可析构状态
   //一定要加这一句，否则old生命周期马上结束，
   //会调用析构函数释放old.s指向的内存，导致this.s指向的内存无效
    old.s = nullptr ;
    old.len = 0;
    cout << "Move Constructor:MyString: " << s << endl;
}

MyString & MyString::operator=(MyString &&rhs) noexcept{

    //检测是否自赋值
    if(this != &rhs){ //rhs是右值引用引用了右值，所以rhs是左值，可以取地址
        char *t = this->s;  //先保存this->s
        //接管rhs的内存
        this->s = rhs.s;  this->len = rhs.len;
        //将rhs置于可安全析构的状态
        rhs.s = nullptr;   rhs.len = 0;
        //最后释放旧的this->s
        if(t){ delete[] t; }
        cout << "Move =:MyString: " << s << endl;
    }
    return *this;
}
```

1：在成员初始化列表里，s(old.s)使得this->s=old.s，this对象接管（窃取）了old.s指向的内存，从这个意义上讲，移动构造本质上就是浅拷贝2：但是马上将old.s指针设为空指针，使得old对象进入安全的可析构状态（意思是old的析构函数不会释放old.s指向的内存）3：移动构造函数的参数限制了只能是右值对象被移动到this对象。因为右值对象是临时对象，在其生命周期结束前安全接管其内存没有问题因此，从第2、第3点讲，移动构造和浅拷贝又有区别

在其生命周期结束前，接管其内存，这样充分利用资源，才能很高效。



这个例子也说明了移动构造函数存在的意义若没有移动构造，

MyString s14=MyString("Hello");的构造过程如下：1：首先构造匿名对象MyString("Hello")2：调用拷贝构造，去构造对象S14。拷贝构造函数里要分配新的内存，并把匿名对象里的字符串拷贝到S14.s指向的内存3:匿名对象MyString("Hello")的生命周期马上结束，被析构可以看到这种场景下匿名对象被拷贝后立即就销毁了，因此第2步的开销是没有必要的，如果第2步改成接管匿名对象的内存，可以提升性能。另外也可以看到为什么移动构造函数参数必须是右值引用：因为右值生命周期短暂，因此可以在其生命周期前接管其内存，之后右值对象就被销毁，因此节省了内存拷贝操作。但如果是左值对象，则不能接管其内存，例如用对象s12构造s13时，如果也采用移动构造，则s13接管s12的内存(s13.s=s12.s)。由于s12的生命周期没结束，会造成不可预料的后果

可以调用std:move函数，将一个左值对象转换成右值引用类型

根据上面的代码看看下面的题：

```
MyString s12("Hello");  //S12是左值
MyString s13 = s12; //用左值对象去构造一个新的对象，会调用拷贝构造
MyString s14 = MyString("TempHello"); //MyString("TempHello")是匿名临时对象，声明周期就是当前表达式，是右值，会调用移动构造函数
D:\CLionProjects\CopyAndMoveObject\bin\CopyAndMoveObject.exe
Constructor:MyString: Hello     //构造s12
Copy Constructor:MyString: Hello        //拷贝构造s13
Constructor:MyString: TempHello     //构造临时对象
Move Constructor:MyString: TempHello    //移动构造s14
Destructor:MyString: TempHello      //析构s14
Destructor:MyString: Hello          //析构s13
Destructor:MyString: Hello          //析构s12
Process finished with exit code 0
```

为什么只打印出三次Destructor:MyString: Hello？课后思考如果没有移动构造函数，输出结果是什么？课后思考

因为在我们的那个移动构造函数里面，我们已经把那个临时变量的s已经nullptr了，所以在这句话结束打算调出析构函数的时候，已经不会打印了

移动构造移动赋值和vector的关系

以std::vector为例，C++标准库的容器大小是可以自动增长的。当调用std::vector::puch_back方法往里面不断放对象时，如果容器内部的buffer到达上限，容器会自动将buffer按一定的策略扩容。这时需要把以前buffer里的对象拷贝到扩容后的新buffer里，如果容器里对象有移动构造函数且声明为noexcept，这时容器会优先调用对象的移动构造函数来移动对象；如果容器里对象的移动构造函数没有声明为noexcept，则容器会调用对象的拷贝构造函数来拷贝对象，这样会影响性能。

```
D:\CLionProjects\CopyAndMoveObject\bin\CopyAndMoveObject.exe
Constructor:MyString: hello_18
Constructor:MyString: hello_19
Copy Constructor:MyString: hello_18 //拷贝构造
Move Constructor:MyString: hello_19 //这时放入s19还是移动构造
Copy Constructor:MyString: hello_18 //这时扩容时拷贝构造s18
Destructor:MyString: hello_18       //析构容器里扩容前的hello_18
Destructor:MyString: hello_18       //析构容器里扩容后hello_18
Destructor:MyString: hello_19       //析构容器里的hello_19
Destructor:MyString: hello_18       //析构s18
Process finished with exit code 0
//为什么s19析构没有输出？S19被移动到容器里，s19.s=0
```

```
MyString s20("Hello_s20");
MyString s21;
MyString s22;
s21 = s20; //copy =
s21.print();
s22 = MyString("Hello_s22"); //move =
s22.print();
D:\CLionProjects\CopyAndMoveObject\bin\CopyAndMoveObject.exe
Constructor:MyString: Hello_s20 //构造20
Constructor:MyString:           //默认构造s21
Constructor:MyString:           //默认构造s22
Copy =:MyString: Hello_s20      //copy =
Hello_s20               //打印s21
Constructor:MyString: Hello_s22 //构造临时对象MyString("Hello_s22")
Move =:MyString: Hello_s22      //move =
Hello_s22               //打印s22
Destructor:MyString: Hello_s22  //析构时s22
Destructor:MyString: Hello_s20  //析构s21
Destructor:MyString: Hello_s20  //析构s20
//注意临时对象的析构没有输出，因为临时对象的s已经=0
```

上面这几个题目都在说一个事情：移动构造函数的时候临时变量已经把内部的s变成了0 不会再析构了

如果右值没有移动构造?

如果一个类既有拷贝构造函数，也有移动构造函数，编译器根据参数类型进行匹配来确定使用哪个构造函数，赋值操作类似。因此这时遵循一个重要原则：移动右值，拷贝左值。前面的示例都说明了这个原则。 **若没有移动构造/移动赋值函数，右值也被拷贝构造/拷贝赋值。 默认的拷贝构造可以是移动构造**

```
class Foo{
public:
    Foo() = default;
    Foo(const Foo &) {…}
    // 未定义移动构造函数
};
Foo x;
Foo y(x) ; //拷贝构造
Foo z(std::move(x)); //还是拷贝构造，因为没有定义移动构造函数
```

类里面有默认的函数：

```
      A() = default;                //默认构造
        A(const A &) = default;          //拷贝构造
        A &operator=(const A &rhs) = default;   //拷贝赋值
        A(A &&) = default;               //移动构造
        A &operator=(A &&) = default;        //移动赋值
        ~A() = default;              //析构
```

```
class NoCopy{
public:
    NoCopy() = default;      //要求编译器合成默认构造函数
    NoCopy(const NoCopy &) = delete; //拷贝构造函数定义为删除的，禁止拷贝构造
    NoCopy &operator=(const NoCopy &) = delete; //禁止NoCopy对象的相互赋值
    ~NoCopy() = default; //要求编译器合成默认析构函数
};
void f(NoCopy o){ }  //值参要求拷贝构造，注意这时编译不报错(原因：没有调用拷贝构造)
void t1(){
//    f(NoCopy());        //如果调用f则编译错，拷贝构造函数是删除的，无法传递值参
    NoCopy o1, o2;
//    o1 = o2;                //编译错，operator=是删除的，无法赋值
}
```

```
class NoDestroy{
public:
    ~NoDestroy() = delete;  //析构函数也可以是删除的
    void f() = delete;  //可以将任何成员函数指定为删除的,这时相当于定义了函数f
};
void t2(){
//    NoDestroy o; //编译报错，o无法被自动析构
    NoDestroy *p = new NoDestroy();  //可new一个对象，因为生命周期由程序控制

//    p->f(); //编译错，调用删除的函数
//    delete p; //只要delete p则编译错，无法delete对象，因为析构函数是删除的
}
```

sum up：对于类型为A且内部有指针的类，应自定义A()、A(A&&) noexcept 、A(const A&)、A& operator=(const A&)、 A& operator=(A&&) noexcept以及~A( )函数。

```
class A {
    int*  p;
    int m;
public:
    A(): p(nullptr), m(0) {}
    A(int m): p(new int[m]), m(p?m:0){ }
    A(const A&a): p(new int[a.m]), m(p ? a.m : 0) {//深拷贝构造必须为p重新分配内存
        for (int x = 0; x < m; x++) p[x] = a.p[x];
    }
    A(A&& a) noexcept: p(a.p), m(a.m) {//深拷贝构造必须为p重新分配内存
        a.p = nullptr; a.m = 0;
    }
 ~A() {
        if (p){ delete p; p = nullptr; m = 0; }
    };
   A& operator=(const A&a) {//深拷贝赋值
        if (&a == this) return *this;
        if (p) delete p;
        p = new int[a.m];
        m = p ? a.m: 0;
        for (int x = 0; x < m; x++) p[x] = a.p[x];
        return *this;
    }
    A& operator=(A&& a) noexcept{//浅拷贝移动构造不为e重新分配内存
        if (&a == this) return *this;
        if (p) delete p;
        p = a.p;    m = p ? a.m : 0; //移动语义：资源a.p转移
        a.p = nullptr;  a.m = 0;        //移动语义：资源a.p已经转移，故资源数量设为 0
        return *this;
    }
};
```

当A&&作为参数的时候 他是左值 放在寄存器里面，有自己的地址

第六章 继承和构造 访问继承的类取决于继承类的访问权限 和基类无关

继承：派生新类，属性有所变异 基类，派生类 对于**数据成员 一个数据被继承后 他的这个数据实际上是同一块数据！！**

单继承类：只能获取一个基类的属性和行为

```
class <派生类名>:<继承方式><基类名>
          {
       <派生类新定义成员>
       <派生类重定义基类同名的数据和函数成员>
       <派生类声明修改基类成员访问权限>
          };
<继承方式>指明派生类采用什么继承方式从基类获得成员，分为三种：private表示私有继承基类；
protected表示保护继承基类；public表示公有继承基类
```

`public` **继承**：
基类的 `public` 和 `protected` 成员在派生类中保持原有的访问权限（即 `public` 成员仍然是 `public`，`protected` 成员仍然是 `protected`）。基类的 `private` 成员对派生类不可访问。

`protected` **继承**：
基类的 `public` 和 `protected` 成员在派生类中变为 `protected`，即派生类内可以访问，但外部无法直接访问。基类的 `private` 成员依然不可访问。

`private` **继承**：
基类的 `public` 和 `protected` 成员在派生类中变为 `private`，即它们仅在派生类内部可访问，外部无法直接访问。基类的 `private` 成员同样不可访问。

基类的私有成员同样也被继承到派生类中，构成派生类的一部分（sizeof会计算基类私有实例数据成员），但对派生类函数成员不可见，不能被派生类函数成员访问。若派生类函数成员要访问基类的私有成员，则必须将其声明为基类的成员友元

区分

注意区别继承方式（派生控制）和访问权限，类的成员访问控制符的。**一个是哪个"派生类名"，一个是类里面的public**

派生控制和类的成员访问控制符的区别：派生控制作用于基类成员，类的成员访问控制符作用于当前类自定义的成员。

例子

```
#include <graphics.h>
class LOCATION{     //定义定位坐标类
    int x, y;
public:
    int getx( ); int gety( );    //gety( )获得当前坐标y
    void moveto(int x,int y);    //定义移动坐标函数成员
    LOCATION(int x,int y);
    ~LOCATION( );
};
void LOCATION::moveto(int x,int y){
    LOCATION::x=x;
    LOCATION::y=y;
}
int LOCATION::getx( ){ return x; }
int LOCATION::gety( ){ return y; }
LOCATION::LOCATION(int x,int y){
    LOCATION::x=x;    LOCATION::y=y;
```

```
}
LOCATION::~LOCATION( ){ }

class POINT:public LOCATION{
//定义点类，从LOCATION类继承，继承方式为public
    int visible;             //新增可见属性
public:
    int isvisible( ){ return visible; }     //新增函数成员
    void show( ),hide( );
    void moveto(int x,int y);        //重新定义与基类同名函数
    POINT(int x,int y):LOCATION(x,y) { visible=0; } //在构造派生类对象前先构造基类对象
~POINT( ){ hide( ); }
};
void POINT::show( ){
    visible=1;
    putpixel(getx( ),gety( ));
}
void POINT::hide( ){
    visible=0;
    putpixel(getx( ),gety( ));
}
void POINT::moveto(int x,int y){
    int v=isvisible( );
    if(v) hide( );
    LOCATION::moveto(x,y);     //不能去掉LOCATION::，会自递归
    if(!v) show( );
}
void main(void){
    POINT p(3,6);
    p.LOCATION::moveto(7,8);                        //调用基类moveto函数
    p
```

访问基类的函数带上基类的名字 访问派生类用派生类

注意： LOCATION::moveto(x,y); //不能去掉LOCATION::，会自递归 如果去掉了location：：那就是point函数递归的无限递归

main函数调用基类moveto函数的逻辑问题： 调用基类的 `moveto` 函数时，隐藏了派生类的 `moveto` 函数： `POINT` 类中， `moveto` 函数被重新定义为覆盖了 `LOCATION` 类中的同名函数。如果你通过 `p.LOCATION::moveto(7, 8);` 调用基类的 `moveto` 函数，派生类的 `moveto` 函数不会被调用，基类的 `moveto` 会直接被执行。此时， `x` 和 `y` 的值会被改变，但并不会触发派生类中关于 `visible` 的逻辑**

多继承

多继承的派生类有多于一个的基类，派生类将是所有基类行为的组合

在控制派生类的时候：

添加新的数据成员和函数成员；改变继承来的基类成员的访问权限；重新定义同名的数据和函数成员（特别是实例函数成员，称为Override）

什么是实例函数成员？就是控制和修改类的非静态数据，然后只能用类的实例访问

实例函数成员（Instance Member Functions）是指在类中定义的成员函数，它们操作类的实例（对象）的数据成员。换句话说，实例函数成员是与特定对象的状态和行为相关联的函数。每个对象都可以调用这些函数，并且每个对象调用时，函数会作用于该对象的成员数据。

特点：

1. **操作实例数据：** 实例函数成员通常用来操作类的实例数据成员（即对象的属性）。这些成员函数通过对象来调用，作用于对象的具体数据。
2. **访问对象的成员：** 实例函数成员可以访问和修改类中定义的非静态数据成员（即普通成员变量）。
3. **需要对象来调用：** 实例函数成员必须通过类的实例来调用，不能直接通过类名来调用（除非在类中声明了静态成员）。

示例：

```
class Rectangle {
private:
    int width, height;

public:
    // 实例函数成员
    void setDimensions(int w, int h) {
        width = w;
        height = h;
    }

    int area() {
        return width * height; // 计算矩形的面积
    }
};
```

在上面的例子中，`setDimensions` 和 `area` 都是实例函数成员：

- `setDimensions` 设置了矩形的宽度和高度。
- `area` 返回了矩形的面积。

这些函数都与对象的具体实例（即 `Rectangle` 类型的对象）相关联。

默认类型类继承

union不能作为基类和派生类，struct默认public，class默认private

什么是合理的类继承？

1.如果我们需要基类的元素，我们可以继承基类的元素

2.但是，如果我们需要重定义一个和基类同名的函数，就不能把这个同名的函数放在public

如何强制修改访问权限

继承方式为private时，基类公有成员在派生类中的访问权限变为private。不合理时可以使用“**基类名::成员**”或“**using 基类名::成员**”修改某些成员的访问权限，派生类不能再定义同名的成员。

```
    class POINT:private LOCATION{ //private可省略
        int visible;
    public:
        LOCATION::getx; //修改权限成public，或者using LOCATION::getx;
        LOCATION::gety; //修改权限成public
        int isvisible( ){ return visible; }
        void show( ), hide( );
        void moveto(int x,int y);
        POINT(int x,int y):LOCATION(x,y){ visible=0; }
        ~POINT( ){ hide( ); }
    };
```

在public里面用出：：就是public

恢复访问权限

恢复访问权限是将派生类继承的基类成员的访问权限复原成和该成员在基类定义时的访问权限一样。**派生类不仅可以恢复基类成员的访问权限，还可以改变访问权限（但不可改变基类私有成员访问权限）。**

其实修改就是恢复----

```
class B:protected A{
     int a;
     A::cc; // using A::cc;降低为private
protected:
    int b,f;
public:
    int e,g;
    using A::d; //恢复
    A::bb; //提升
}b;
```

继承其他注意的点：

基类成员经过继承方式被继承到派生类后，要注意访问权限的变化。按面向对象的作用域，**和基类同名的派生类成员被优先访问。派生类中改写基类同名函数时要注意区分这些同名函数，否则可能造成自递归调用。**

标识符的作用范围可分为从小到大四种级别：①作用于函数成员内；②作用于类或者派生类内；③作用于基类内；④作用于虚基类内。标识符的作用范围越小，被访问到的优先级越高。如果希望访问作用范围更大的标识符，则可以用类名和作用域运算符进行限定。

“和基类同名的派生类成员被优先访问”指的是，在派生类中定义了与基类同名的成员时，派生类的成员会优先被访问，覆盖或隐藏了基类中的同名成员。如果需要访问基类的成员，可以通过作用域解析符来明确指定访问基类的成员。

写派生类的构造函数的时候 一句话实际上代表了两个构造函数：

```
  SET( ){ };                  //等价于SET( ):LIST( ){ };
```

在 C++ 中，类的构造函数（如 `SET()`）用于初始化对象时，通常会隐式或显式调用基类的构造函数。对于构造函数的调用规则，尤其是在继承关系中，如果派生类的构造函数没有明确调用基类的构造函数，编译器会自动调用基类的默认构造函数。

为了访问基类的友元化：

派生类友元化：

```
class B;         //前向声明类B
class A{
    int a, b;
public:
    A(int x){a=x;}
    friend B;        //声明B为A的友元类，B类成员可以访问A任何成员
};
class B:A{        //缺省为private继承，等价于class B: private A{
    int b;
public:
    B(int x):A(x){ b=x; A::b=x; a+=3; } //可访问私有成员A::a,A::b
};
void main(void){ B x(7); }
```

派生函数友元化：

```
#include <iostream>

// 前向声明
class Derived;

class Base {
private:
    int x;

public:
    Base() : x(10) {}

    // 将Derived的静态成员函数声明为友元
    friend void Derived::show(Base& b);
};

class Derived : public Base {
public:
    static void show(Base& b) {
        // 静态函数可以直接通过类名调用
        std::cout << "Base private member x = " << b.x << std::endl;
    }
};

int main() {
    Base b;
    Derived::show(b);  // 输出: Base private member x = 10
    return 0;
}
```

构造和析构

构造：

1.虚基类先构造

2.基类构造

3.按照派生类中的数据成员的声明循序，调用数据成员的构造函数或者初始化数据成员

4.最后是派生类的构造

必须有派生类自己定义的构造函数：

虚基类和基类只有带参数的构造函数（无法通过派生类的一句不带参数的构造函数构造基类）**也就是说所有的派生类都会先定义基类！！！**

派生类定义了引用和只读成员（他们都是常量 无法再次赋值），且没有就地初始化,**而且只能括号和中括号初始化**

派生类定义了用带参数构造函数初始化的其他类对象，且没有就地初始化（意思是说 某一个类里面有另外一个类 然后这个类的初始化需要带参数的构造函数）

如果虚基类和基类的构造函数是无参的，则构造派生类对象时，派生类构造函数可以不用显式调用基类/虚基类的构造函数，编译程序会自动调用虚基类或基类的无参构造函数

new出来的引用对象和指针对象都要delete处理

如果被引用的对象是用new生成的，则引用变量r必须用delete &r析构对象，否则被引用的对象将因无法完全释放空间（为对象申请的空间）而产生内存泄漏。若被p（指针）指向的对象是用new生成的，则指针变量p必须用delete p析构对象，不能使用不调用析构函数的free(p)，否则将产生内存泄漏。

/c=v不是初始化，是重新赋值

```
#include <iostream.h>
class A{
    int a;
public:
    A(int x):a(x){cout<<a;}//非const成员a也可在构造函数体内再次对a赋值
    ~A( ){cout<<a;}
};
class B:A{     //私有继承，等价于class B: private A{
    int b,c;
    const int d;//B中定义有只读成员且没有就地初始化，故必须定义构造函数初始化
    A x,y; //x、y的构造必须带参数，且没有就地初始化，故必须定义构造函数初始化
public:
    B(int v):b(v),y(b+2),x(b+1),d(b),A(v){//注意构造次序与成员初始化列表的出现顺序无关
        c=v;         cout<<b<<c<<d;        cout<<“C”;  //c=v不是初始化，是重新赋值
    }
    ~B( ){cout<<“D”;}        //派生类数据成员实际构造顺序为b,c, d,x,y
};
void main(void){ B z(1); }  //输出结果：123111CD321
```

析构是构造倒着来

```
void sub1(void) {
    A &p=*new A(1);
}//内存泄露
void sub2(void){
    A *q=new A(2);
} //内存泄露
void sub3(void){
    A &p=*new A(3);
    delete &p;
}
void sub4(void) {
    A *q=new A(4);
    delete q;
}
void main(void){
    sub1( );    sub2( );
    sub3( );    sub4( );
}
```

父类和子类

派生类为public继承 父类指针可以直接指向子类对象，父类引用直接引用子类对象，按照父类说明的成员权限访问 如果不是public继承就要用强制类型转化使得引用和指针可用

```
 A &p=*new C(3);    //直接引用C类对象：A和C父子
    A &q=*(A *)new B(5);
```

父类指针的使用必须遵守他自己函数的访问权限，也不能用子类定义的新函数 父类指针指的是子类的东西还是父类的东西要在运行时确定

```
#include <iostream>
using namespace std;

class Base {
public:
    int publicVar;
    void publicFunc() {
        cout << "Base public function" << endl;
    }

protected:
    int protectedVar;
    void protectedFunc() {
        cout << "Base protected function" << endl;
    }

private:
    int privateVar;
    void privateFunc() {
        cout << "Base private function" << endl;
    }
```

```
};

class Derived : public Base {
public:
    int derivedVar;
    void derivedFunc() {
        cout << "Derived function" << endl;
    }

    void accessBaseMembers() {
        cout << publicVar << endl;    // 可访问
        cout << protectedVar << endl; // 可访问
        // cout << privateVar << endl; // 编译错误，父类的 private 成员不能访问
    }
};

int main() {
    Derived d;
    Base* basePtr = &d;

    // 可以访问父类的 public 成员
    basePtr->publicFunc();  // 可以访问父类的公有函数
    cout << basePtr->publicVar << endl; // 可以访问父类的公有成员变量

    // 以下两行代码会报错，因为父类指针不能直接访问 protected 和 private 成员
    // basePtr->protectedFunc(); // 编译错误
    // basePtr->protectedVar; // 编译错误
    // basePtr->privateFunc(); // 编译错误
    // basePtr->privateVar; // 编译错误

    // 尝试访问派生类新增的成员变量
    // basePtr->derivedVar = 10; // 编译错误，父类指针不能访问派生类的成员
    // basePtr->derivedFunc(); // 编译错误，父类指针不能访问派生类的函数

    return 0;
}
```

```
    class Person { public: void f() { cout << “P\n”;}};
    class Teacher: public Person()
    { public: void f(){ cout << “T\n”;}};
    Person *p = new Teacher();//左边父类，右边子类
    p->f(); //输出的是P，为什么不输出T
    1.声明一个类型为Person的指针p，Person为声明类型
    2.创建一个Teacher类型的对象
  3.Person类型的指针指向一个Teacher对象
  但第2步、第三步发生在运行时（Run time）
因为编译时程序还没运行，编译器无法知道p会指向什么类型对象，编译器在编译时只能根据变量p的声明类型（Person *）来类型检查
因此编译器在编译到p->f()语句时，认为调用的是Person的f()，绑定到Person的f()
```

代码编译和运行的差异了----

子类不能指向父类直接 子类指向父类要强制转化而且父类的要是实例的

父类要是实例的 如果指向是student实例临时对象 但是我们用了teacher 报错

子类指针不能指向父类对象（子类=父类）必须强制转换Person *p = new Teacher(); Teacher *t = (Teacher *)p; // Teacher *t = p会出错或者 Teahcer *t = (Teacher *)new Person(); //必须强制类型转换首先编译时编译器认为p的类型是Person*，t的类型是Teacher *（按声明类型检查）因为父类型不一定是子类型（Person不一定是Teacher）所以当编译器检查到Teacher *t = p时，认为一个Person类型的指针要赋值给Teacher类型的指针，类型是不匹配的强制转换Teacher *t = (Teacher *)p的意思是告诉编译器，请不要再做类型检查，风险我自己承担。强制类型转换的风险是：运行时如果p指向的对象不是Teacher的实例时程序会出错（Run Time Error）因此在赋值前必须利用RTTI（运行时类型标识）来检查p是不是指向Teacher的实例

```
if(! strcmp(typeid(*p).name(),"Teacher"))
    Teacher *t = (Teacher *)p;
```

派生类对象内部基类指针

定义的基类指针可以直接指向该派生类对象，**即对派生类函数成员而言，基类被等同地当作父类。**

如果函数声明为派生类的友元，则该友元定义的基类指针也可以直接指向该基类的派生类对象，也不必通过强制类型转换。

第八章 虚函数 多态（这一章最容易错的：派生类构造先要基类构造）

虚函数：即用virtual定义的实例成员函数 基类对象指针(引用)指向(引用)不同类型派生类对象时，通过虚函数到基类或派生类中同名函数的映射实现(动态)多态。虚函数内联是失败的

虚函数作为实例对象 一定有指针this 重载函数是静态多态函数（编译） 虚函数是动态多态函数（运行）

多态：每个子类对象都是父类的实例，如　　Person *p = new Teacher( )；这是基础

class A{　void info() { cout << "A\n"); }　};class B: public A{　　void info() { cout << "B\n"); }};class C: public B{　　void info() { cout << "C\n"); }};　如果需要编写一个全局函数PrintInfo，能够打印A、B、C类对象的信息，怎么实现?

我们可以用重载printinfo函数 但是新定义一个类的话又要写多一个重载函数

多态的含义就是实现一个printfinfo函数 可以输出无论多少个父类的子类

我们可以:

```
class A{
    virtual void info() { cout << "A\n"); }
    //首先将info定义为虚函数
};
```

```
class B: public A{
     virtual void info() { cout << “B\n”); }
};

class C: public B{
     virtual void info() { cout << “C\n”); }
};
void PrintInfo (A *p) { p ->info(); } //形参定义为顶级父类指针
A *a = new A( ); B *b = new B( ); C *c = new C( );
PrintInfo (a);       //调用A的info，显示A。A * p = a;
PrintInfo (b);       //调用B的info，显示B。 A *p = b;
PrintInfo (c);       //调用C的info，显示C。 A *p = c;
程序编译时，形参p的类型是A *， p->info()调用的绑定的是A::Info( )
     但程序运行时，当顶级父类指针指向p继承链中不同子类对象时，会自动地调用相应子类的info函数
     同一条语句p->info( )在运行时表现出动态的行为。运行时行为取决于new后面的类型
     面向对象的程序设计语言的这种特性称为多态
```

如果方法参数是父类指针（引用），那么这个参数可以接受任何子类对象指针（引用）作为实参。当调用这对象的方法时，将动态绑定方法的实现。

下面这个例子：说明了和原型的虚函数定义一样的时候会自动变虚函数

```
#include<bits/stdc++.h>
using namespace  std;
class POINT2D{
    int  x, y;
public:
    int getx( ) { return x; }
    int gety( ) { return y; }
    POINT2D(int x, int y) { POINT2D::x=x; POINT2D::y=y; }
    virtual POINT2D* show( ){ cout<<"Show a point\n"; return this;} //定义虚函数
};
class CIRCLE: public POINT2D{   //POINT2D和CIRCE满足父子关系
    int r;
    CIRCLE* show( ) { cout<<"Show a circle\n"; return this;}//原型“一样”，自动成为虚函数
public:
    int getr( ) { return r; }
    CIRCLE(int x, int y, int r):POINT2D(x, y) { CIRCLE::r=r; }
};
int main(void)
{
    CIRCLE c(3, 7, 8);
    POINT2D *p=&c;           //父类指针p可以直接指向子类对象c
    cout<<"The circle with radius "<<c.getr( );
    cout<<" is at ("<<p->getx( )<<", "<<p->gety( )<<")\n";
    p->show( );  // Show a circle，如果把Circle里的show函数定义为私有的会如何？请思考
}
```

原型“一样”，自动成为虚函数

回答一下结尾的问题：父类指针本来就可以指向子类 但是在没有定义虚函数的时候 它会在编译就绑定 定义虚函数的时候变成动态绑定 但是 它始终遵循父类的权限 也就是说 即使我们把circle的虚函数放在private里面 2dpoint里面是public就还是public权限 所以 2dpoint里面放在private里面就会报错 所以 这里和父类和子类的区别唯一区别就是虚函数的绑定

注意点

虚函数必须是实例成员函数，非类的成员函数不能是虚函数，包括普通函数

虚函数一般在public和protect，为了派生类重新定义或者默认定义虚函数，函数原型必须完全相同（返回值不用一样）

虚函数有隐含的this，参数表可以有const和volatile，静态函数没有this，不能是虚函数，不能有virtual static

构造函数构造的类型固定 不能是虚函数 析构函数可以通过父类调用直接删除 可以是虚函数

一旦父类(基类)定义了虚函数，所有派生类中原型相同的非静态成员函数自动成为虚函数（即使没有“virtual”声明）

union没有基类没有派生类 不可能是虚函数

虚函数同普通函数成员一样，可声明为或自动成为inline函数（但内联肯定失败），也可重载、缺省和省略参数(只是这个时候不能自动虚函数了)

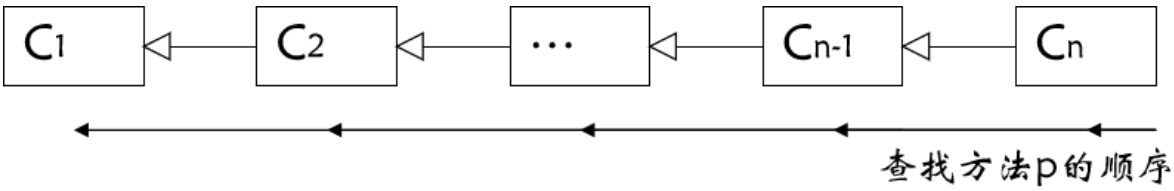
必须用最基类指针(或引用)访问虚函数才有多态性。根据父类指针所指对象类型的不同，动态绑定相应对象的虚函数；(虚函数的动态多态性)

ppt的巨大错误！虚函数的默认是继承的结果，不是父类子类的结果，一个类能用虚函数访问，一定是用最基类的指针，不是父类！！！！但是，只有父类不用类型转化

```
#include <iostream>          //【例8.2】虚函数的使用方法
using namespace std;
struct A{
    virtual void f1( ){ cout<<"A::f1\n"; }; //定义虚函数f1()
    virtual void f2( ){ cout<<"A::f2\n"; }; //this指向基类对象，定义虚函数f2()
    virtual void f3( ){ cout<<"A::f3\n"; }; //定义虚函数f3()
    virtual void f4( ){ cout<<"A::f4\n"; }; //定义虚函数f4()
};
class B: public A{   //A和B满足父子关系
    virtual void f1( ){//virtual可省略，f1()自动成为虚函数
        cout<<"B::f1\n";
    };
    void f2( ) {          //除this指向派生类对象外，f2()和基类函数原型相同，自动成为虚函数
        cout<<"B::f2\n";
    };
};
class C: B{       //B和C不满足父子关系，故A和C也不满足父子关系
    void f4( ) {          //f4()自动成为虚函数，即使不是父子关系，也有多态性
        cout<<"C::f4\n";
    };
};
void main(void)
{
    C  c;
```

```
    A *p=(A *)&c;    //A和C不满足父子关系，需要进行强制类型转换
    p->f1( );        //调用B::f1( )
    p->f2( );        //调用B::f2( )
    p->f3( );        //调用A::f3( )
    p->f4( );        //调用C::f4( )
    p->A::f2( );         //明确调用实函数A::f2( )，没有多态性
}
```

有一条继承链基类要申明(或者原型一样导致的默认)virtual类型的成员函数 用基类型的指针（或引用）指向（引用）派生类（不一定必须是父子关系）



这时会沿着继承链从子类到父类Cn,Cn-1,... C1查找f的实现，一旦找到一个实现，将停止查找，并执行找到的第一个实现

对于父类A中声明的虚函数f( )，若在子类B中重定义f( )，必须确保子类B::f( )与父类A::f( )具有完全相同的函数原型，才能覆盖原虚函数f( )而产生虚特性，执行动态联编机制。否则，只要有一个参数不同，编译系统就认为它是一个全新的（函数名相同时重载）函数，而不实现动态联编。 比如下面这个：

```
#include <iostream>
using namespace std;

class A {
public:
    virtual void f(int x) { cout << "A::f with " << x << endl; }  // 基类的虚函数
};

class B : public A {
public:
    void f(int x) { cout << "B::f without argument\n"<<x; }  // 错误的重写，参数不同
};
class C : public A {
public:
    void f() { cout << "c::f without argument\n"<<5; }  // 错误的重写，参数不同
};
int main() {
    A* a = new C();  // A类指针指向B类对象
    a->f(10);          // 错误调用，应该是B::f(int)，但B::f()没有参数
    delete a;
    return 0;
}
输出：A::f with 10 在编译的时候绑定了A*a
```

请看“子类不能指向父类。。。那里”

关于子类B::f( )与父类A::f( )具有完全相同的函数原型，有个地方需要补充说明：就是函数返回类型不一定完全一样，只要相容就行（例如A::f( )函数返回A类型（或A），*B::f( )函数返回B类型（或B*）就是相容)

特殊：如果基类的析构函数是虚析构函数 则派生类的析构函数就会自动成为虚析构函数（即使原型不同，即函数名不一样） 动态内存一定要虚析构函数

```
好处：
for(k=0; k<max; k++)
        delete s[k];  //多态调用析构函数
        //若s[k]指向STRING的对象，则用s[k]->~STRING( )析构
        //若s[k]指向CLERK的对象，则用s[k]->~CLERK( )析构
}
```

类的引用问题：

用父类引用实现动态多态性时需要注意， new产生的被引用对象必须用delete &析构：

```
STRING  &z=*new CLERK("zang","982021",23);
            delete &z;  //析构对象z并释放对象z占用的内存
```

详细分析下面的代码：

```
#include <iostream>
using namespace std;
#include <stdio.h>
#include <string.h>

class STRING {
    char *str;
public:
    STRING(char *s);
    virtual ~STRING() {
        cout<<this->str<<endl<<'2'<<endl;
        if (str) {
            delete[] str;
            str = 0;
        }
    }
};

STRING::STRING(char *s) {
    str = new char[strlen(s) + 1];
    strcpy(str, s);
}

class CLERK : public STRING {
    STRING clkid; // 注意这个成员是一个STRING类型的对象
    int age;
public:
    CLERK(char *n, char *i, int a);
    ~CLERK() {
        cout<<'1'<<endl;
    } // 自动成为虚函数
};
```

```
CLERK::CLERK(char *n, char *i, int a) : STRING(n), clkid(i) {
    age = a;
}

int main() {
    STRING &z = *new CLERK("zang", "982021", 23);
    // 不能手动调用析构函数 z.~STRING(); 这样是非法的
    delete &z; // 正确的做法是通过delete来销毁对象
    return 0;
}
输出：1
982021
2
zang
2
```

这就是上面说的 如果子类或者说派生类构造函数没有参数的时候会自动把基类的没有构造函数的 一起调用从而完成构造 要是基类有参数构造函数 那么在子类的构造函数里面一定要把基类的构造给构造了 当子类完全析构 基类才析构

小tips：传入形参的临时变量在函数结束后才析构

抽象类：含有纯虚函数的类，没有对象或者类实例

纯虚函数：没有函数体的虚函数，可以重载、缺省参数、省略参数、内联等，格式：定义格式：virtual 函数原型=0。 (0即函数体为空)纯虚函数有this，不能同时用static定义(表示无this)。他的函数体应该要在派生类里面实现，成为非纯虚函数

一旦派生类继承了抽象类的纯虚函数但是没有重定义虚函数的函数内容，或者定义了新的纯虚函数，那么这就是新的抽象类 请看下面代码的注释：

```
struct A { virtual void f1() = 0, f2() = 0; };  //A为抽象类，不能定义A a;  A f ( )
//但还是可以给出纯虚函数的函数体,尽管如此，f1,f2还是纯虚函数，A还是抽象类
void A::f2() { cout << "A2"; }
void A::f1() { cout << "A1"; }
class B:public A {  //取代型定义f2，未定义f1，B为抽象类
    void f2() { A::f2();  cout << "B2"; }    //自动成虚函数，导致内联失败
};  //B为抽象类，不能定义f (B b)，不能定义B *s=new B
class C:public B { // f1和f2均取代型定义，具体类C可定义变量、常量等
    void f1() { cout << "C1"; }  //自动成虚函数，虚函数导致内联失败
}c;
void test(void) {
    A *p = &c;   //A、C满足父子关系
//虽然抽象类没有可供引用或指向的对象,但是抽象类仍然可以作为父类定义指针或引用,指向或引用子对象
    p->f1();//调用C::f1 ( )
    p->f2 (  );  //调用B::f2 ( )
}   //C1A2B2
```

抽象类不能定义或者产生任何对象，所以不能作为函数参数值类型或者返回来想

虽然抽象类没有可供引用或指向的对象,但是抽象类仍然可以作为父类定义指针或引用,指向或引用子对象

抽象类指针或引用可以作为函数参数或返回类型,通过抽象类指针或引用可调用抽象类的纯虚函数，根据多态性，实际调用的应是该类的非抽象派生类的虚函数。

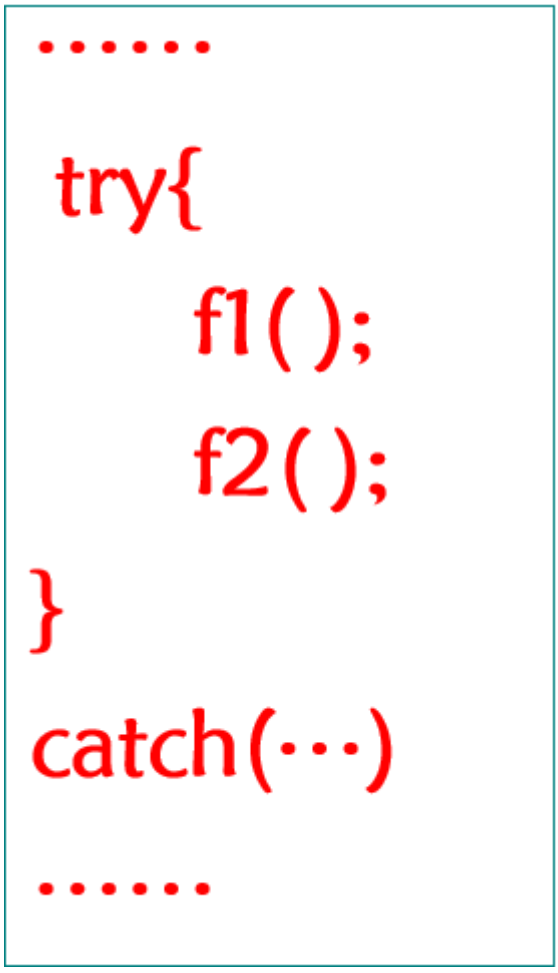
友元没有传递性

定义的友元只能用定义的 没有抽象类的传递过程 纯虚函数和虚函数都可以定义为另外一个类的友元

第九章 异常断言（不重要）

异常发生后直接析构调用链里面所有对象

由软件引发的异常用throw语句抛出，会在抛出点建立一个描述异常的对象，由catch捕获相应类型的异常。



try语句块里的语句或函数调用都可能产生异常。异常应在某个try语句块中抛出每个try语句块后必须至少有一个catchtry-catch语句块可以嵌套任何一条语句都可以用try-catch语句替换(包括catch中的语句)

throw语句:

抛出异常的throw语句一定要间接或直接在try语句块中

如果抛出的异常不在任何一个try语句块中，这种没有异常处理过程捕获的异常将引起程序终止执行

在try中抛出的异常如果没有合适的catch捕获，也将引起程序终止执行

一个try可以抛出多个异常（就是说呢 抛出的异常不在测试语句 终止 抛出的异常没有处理 终止）

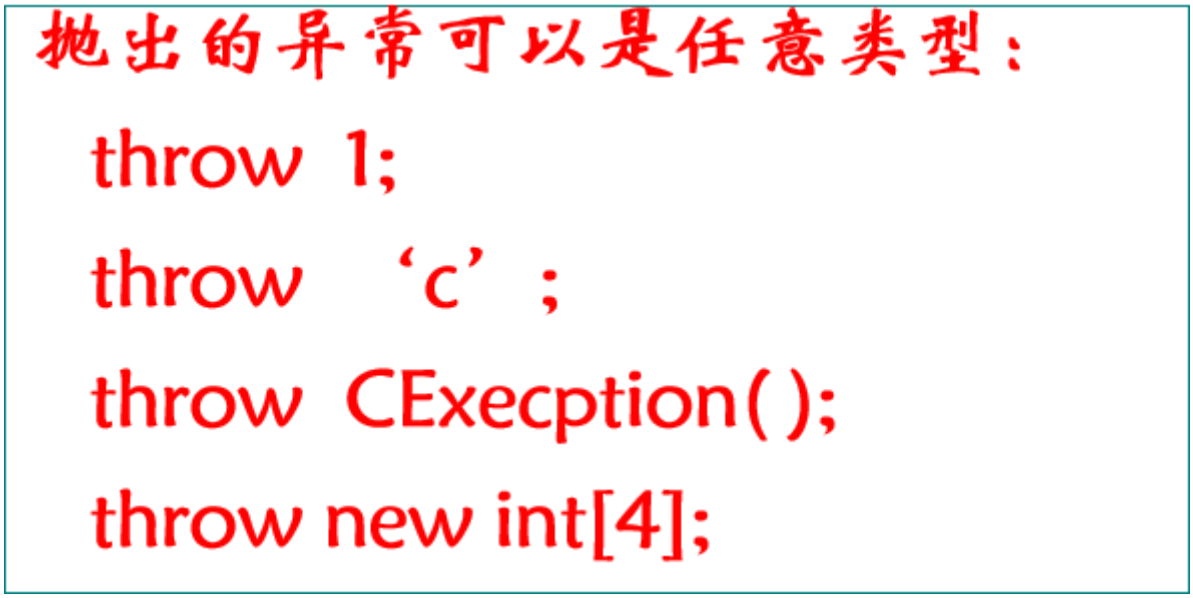
```
try {
   // statements throw E1
   // statements throw E2
   ...
 }
catch(E1){
  //handle the exception E1
}
catch(E2){
  //handle the exception E2
}
statements;
```

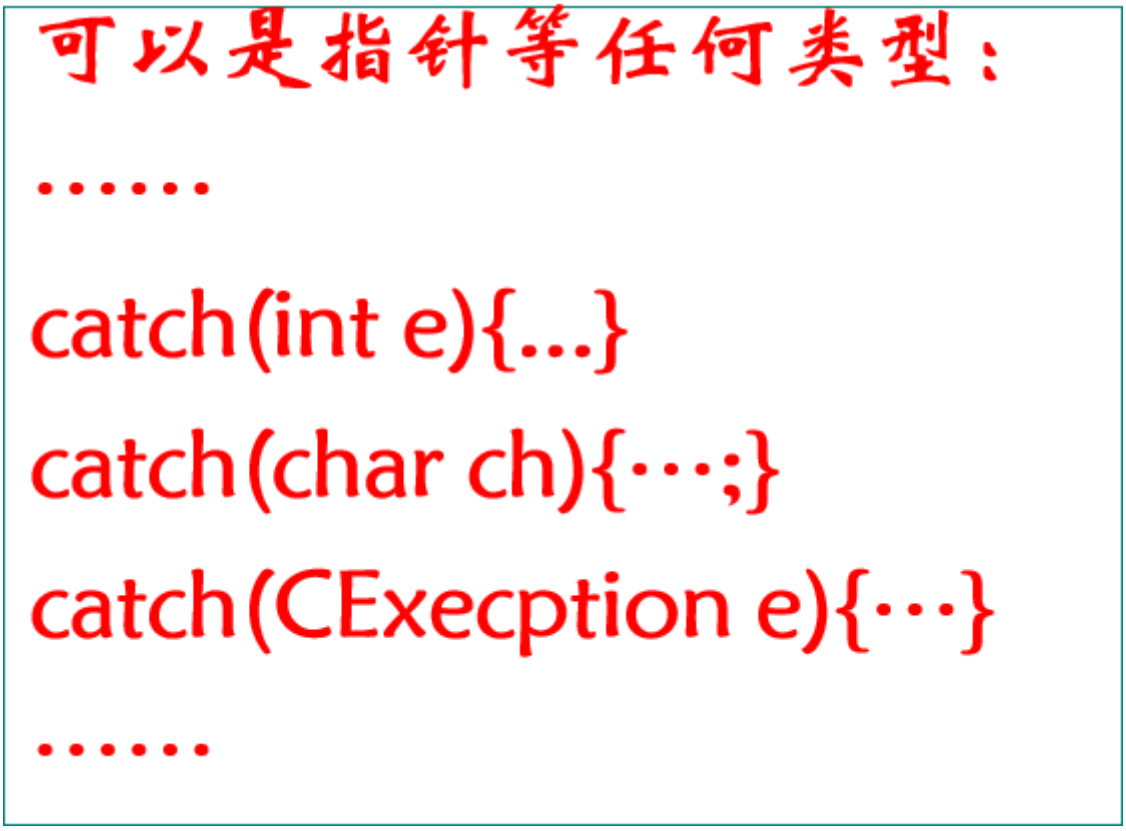
具体处理

try语句块中的语句可能抛出各种异常。但只要抛出第一个异常，try语句块马上结束，转到catch子句。如果try语句块执行成功，则跳过catch，执行try{}catch{}的后续statements;

可定义多个catch子句截获可能抛出的各种异常。但最多只会执行其中一个异常处理过程。在相应异常被处理后，其他异常处理过程都将被忽略。如果异常处理成功，则执行后续statements。如果异常处理过程中又抛出新的异常，则后续statements不会被执行

抛出的异常可以是任何类型





样子是抛出的错误类型+变量，要且只能定义一个参数，不能是void和右值引用

一个try抛出了一个异常后，找关联catch块，没找到catch找外层try的catch块，还找不到在当前函数块外面找catch，再不行到main，再不行交给os

有时catch子句在捕获异常并作了一些处理后，需要由嵌套的外层try/catch或调用链更上一层的函数继续处理异常，这时可以用不带表达式的throw语句重新抛出异常，

```
catch(MyException &ex){
        //do something
        throw; //重新抛出异常
    }
```

有时希望能捕获所有异常，或者不知道可能抛出的异常对象类型，这时可以用catch(...)来捕获所有异常。但要注意如果有其他catch子句，那么catch(...)应该在最后位置 因为他会按照上下顺序依次寻找匹配

如果是通过new产生的指针类型的异常，在catch处理过程捕获后，通常应使用合适的delete释放内存，否则可能造成内存泄漏。

如果继续传播指针类型的异常，则可以不使用delete。

异常处理

在一个catch块里面：没有任何实参的throw用于传播已经捕获的异常。 任何throw后面的语句都会被忽略，直接进入异常捕获处理过程catch。如果是通过new产生的指针类型的异常，并且该异常不再传播，则一定要使用delete释放，以免造成内存泄漏。

处理顺序

先声明的异常处理过程将先得到执行机会，因此，可将需要先执行的异常处理过程放在前面。

异常的类型只要和catch参数的类型相同或相容即可匹配成功。即派生异常对象(及对象指针或引用)可以被带基类型对象、指针及引用参数的catch子句捕获。

如果A的子类为B，B类异常也能被catch(A)、catch(const A)、 catch(volatile A)、 catch(const volatile A) 等捕获，以及将上述A改为A&等的捕获如果A的子类为B，指向可写B类对象的指针异常也能被catch(A)、 *catch(const A)*、 catch(volatile A)、 *catch(const volatile A)*等捕获。

如果产生了B类异常，且catch(const A &)在catch(const B &)之前，则catch(const A &)会捕获异常，从而使catch(const B &)没有捕获的机会。

因此，在排列catch时通常将catch父类参数的catch放在后面，否则子类(还包含指针或引用)参数的catch永远没有机会捕获异常。

注意catch(const volatile void *)能捕获任意指针类型的异常，catch(...)能捕获任意类型的异常。

可用throw( )表示函数不引发任何异常

调用链里面的局部变量会被全部依次析构 但是new对象要delete

断言

断言（assert）是一个带有整型参数的用于调试程序的函数，如果实参的值为真则程序继续执行。 否则，将输出断言表达式、断言所在代码文件名称以及断言所在程序的行号，然后调用abort()终止程序的执行。

第五章 成员和成员指针

成员指针：指向类的成员（普通和静态成员）的指针，分为实例成员指针和静态成员指针。变量、数据成员、函数参数和返回类型都可定义为成员指针类型，即普通指针能用的地方成员指针都能用

实例成员指针：

```
int  A::*p；//A类的成员指针，指向A类的int类型实例数据成员
```

```
实例成员指针声明：
     int  A::*p；//A类的成员指针，指向A类的int类型实例数据成员
实例成员指针使用：
    struct A{
            int   m,  n;
    }a={1, 2},  b={3, 4};
    p =  &A::m; //实例成员指针p指向A类的实例数据成员m
    int x = a.*p    //x = a.m;    利用实例成员指针访问成员   对象名.*指针名或对象指针->*指针
```

利用实例成员指针访问成员 对象名.*指针名或对象指针*->指针

运算符

.* 和 ->* 是双目运算符，从左到右结合

.*的左操作数为类的实例(对象)，右操作数为指向实例成员的指针

->*的左操作数为对象指针，右操作数为指向该对象实例成员的指针

```
int  A::*pi=&A::i;          //实例数据成员指针pi指向public成员A::i
   int (A::*pf)( )=&A::f;  //实例函数成员指针pf指向函数成员A::f
```

实例成员指针是指向实例成员的指针，可分为实例数据成员指针和实例函数成员指针。

实例成员指针必须直接或间接同.*或*->左边的实例(对象)结合，以便访问该对象的实例数据成员或函数成员。

构造函数不能被显式调用，故不能有指向构造函数的实例成员指针。

```
数据类型  类名::*指针名 = &类名::实例成员名 &符号表示取地址
```

实例函数指针是类头的偏移地址 不能移动 不能强制类型转化 因为移动的本质是让一个新对象指向一个旧对象的内存 然后新对象的内存和指针对应的内存可能不一样 不能强制类型转换也是因为这个原因

实例成员指针不能和其它类型互相转换：否则便可以通过类型转换间接实现指针移动。

```
pi++;               //错误，  pi不能移动，否则指向私有成员j
   pf+=1;           //错误，  pf不能移动
   x=(long) pi;             //错误，pi不能转换为长整型
   x=x+ sizeof(int)                 //间接移动指针
   pi=(int A::*)x;          //错误，x不能转换为成员指针
```

const只读、volatile易变和mutable机动（只需要在声明的时候带上）

const和volatile可以定义变量、类的数据成员、函数成员及普通函数的参数和返回类型。mutable只能用来定义类的数据成员。

含const数据成员的类必须定义构造函数（如果没有类内就地初始），且数据成员必须在构造函数参数表之后，函数体之前初始化。也就是：和{}之间 变成xxx（对应类型）A const

```
} x(3);                     //等价于A  x(3)，x可修改，
const A y(6);       // y、z不可改
const volatile A z(8);
```

```
void query (char* &n,  char &s,  int &w)   const
   //函数体不能修改当前对象，但修改机动成员。&name[0]类型为const char *
   {  n= (char*) &name[0];  s=gender;  w=wage;
      wage ++;           //错误，不能修改const对象的成员
     querytimes++;          //ok，querytimes是mutable成员
   }
```

含volatile、mutable数据成员的类则不一定需要定义构造函数。

```
   #include <string.h>
   #include <iostream.h>
   class TUTOR{
```

```
        char    name[20];
        const   char  sex;  //性别为只读成员
        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);
    };
TUTOR::TUTOR(const char *n, const TUTOR *t): sex(t->sex){
    strcpy(name,n);    salary=t->salary;
}//只读成员sex必须在构造函数体之前初始化
TUTOR::TUTOR(const char *n, char g, int s): sex(g),sarlary(s){
    strcpy(name,n);
}//非只读成员sarlary可在函数体前初始化，也可在构造函数体内再次赋值
char *TUTOR::setname(const char*n){
    return strcpy(name, n); //注意：strcpy的返回值为name
}
void main(void){
    TUTOR  wang("wang", 'F', 2000);
    TUTOR  yang("yang", &wang);
    *wang.getname( )='W';   //错误:不能改wang.getname( )返回的指针指向的字符(const
char *)
    *yang.setname("Zang")='Y';// 可改wang.setname( )返回的指针指向的字符(char *)
}
```

复习；this指针只能指向局部类的非静态数据成员 指的是当前类的成员this类型为A * const this

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(其非静态数据成员)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。**构造或析构函数的this不能被说明为const或volatile的(即要构造或析构的对象应该能被修改，且状态要稳定不易变) 同时 对一个类 不能用类的this指向析构和构造 普通函数没必要声明特殊的。**

反正就类型尽可能匹配：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。

强制类型改变

由于this指针会加前缀修饰，指的是当前的数据成员，在易变类型的时候会改变数据成员类型为易变

```
/由于this指针被volatile修饰，导致name类型为 voliatile char [20]
    {return strcpy ( (char *) name,  n)   ;  } //必须强制将volatile char * 转换为
char *
```

mutable 等级高于const 只要有mutable就可以修改

mutable说明数据成员为机动数据成员，该成员不能用const、volatile或static修饰。 该数据成员就可以被修改。

保留字mutable还可用于定义Lambda表达式的参数列表是否允许在Lambda的表达式内修改捕获的外部的参数列表的值。

全军复诵！！！

有址（左值）引用变量(&)只是被引用对象的别名，被引用对象自己负责构造和析构，该引用变量(逻辑上不分配内存的实体)不必构造和析构。

无址（右值）引用变量(&&)常用来引用常量对象或者生命周期即将结束的对象，该引用变量(逻辑上不分配缓存的实体)不必构造和析构。

无址引用变量为左值，但若同时用const定义则不能出现在=左边。（常量左值看成右值）

如果A类型的有址（左值）引用变量r引用了new生成的(一定有址的）对象x，则应使用delete &r析构x，同时释放其所占内存。

引用变量必须在定义的同时初始化，函数的引用参数则在调用函数时初始化。非const左值引用变量和参数必须用同类型的左值表达式初始化。

```
int a = 10;
int &ref = a;  // 正确：ref 是 a 的引用，且在定义时就初始化

int &ref2;  // 错误：引用变量必须在定义时初始化
```

2. 函数的引用参数在调用时初始化

对于函数的引用参数，它在函数调用时被初始化为传递给函数的实参。注意，传递给函数的实参必须是左值，且类型必须匹配。

```
cpp复制代码void modifyValue(int &x) {
    x = 100;
}

int main() {
    int a = 10;
    modifyValue(a);  // 正确：a 是一个左值，传递给引用参数 x
    cout << "a = " << a << endl;  // 输出 100
    return 0;
}
```

3. 非const左值引用变量和参数必须用同类型的左值表达式初始化

当使用**非const左值引用**时，它必须绑定到一个**左值**（lvalue），并且类型必须匹配。如果你传递的是右值（rvalue）或类型不匹配的对象，编译会报错。

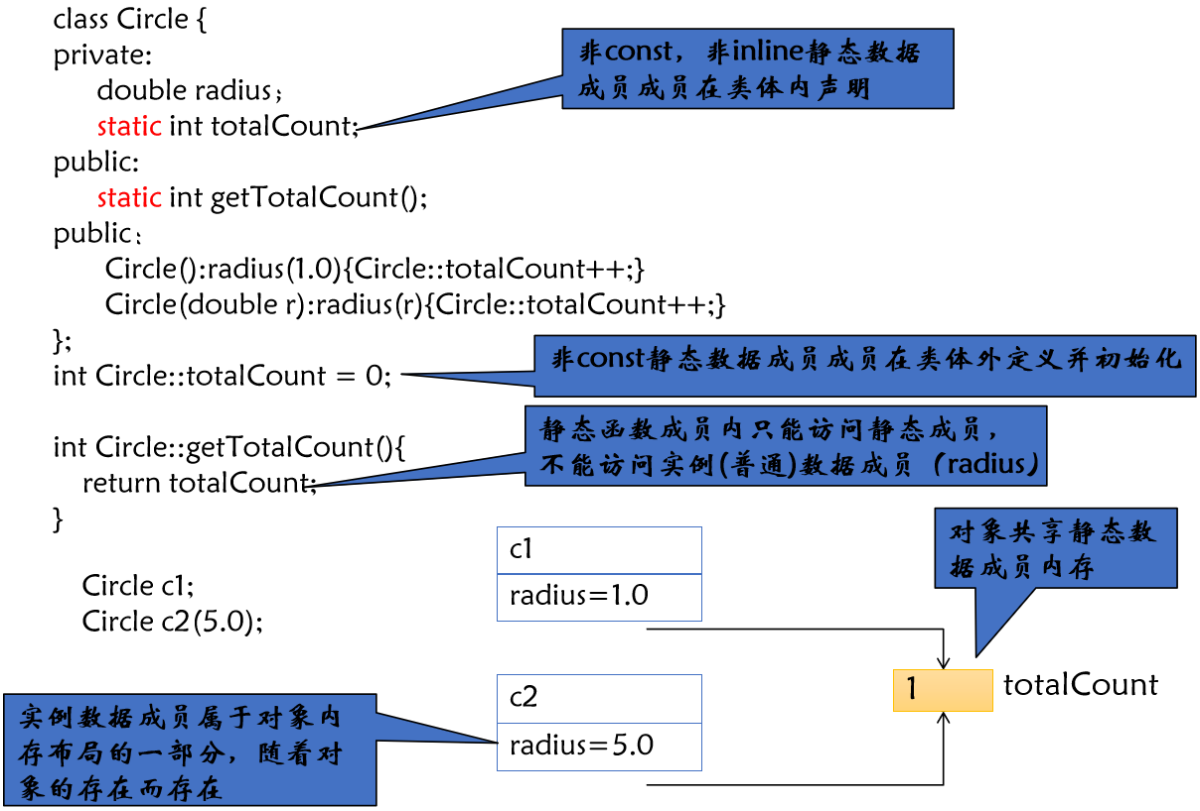
例子：函数参数的左值引用与传递的实参类型必须匹配

```
cpp复制代码void modifyValue(int &x) {  // 参数是左值引用
    x = 100;
}

int main() {
    int a = 10;
    modifyValue(a);  // 正确：a 是一个左值，可以传递给左值引用参数

    // modifyValue(20);  // 错误：20 是右值，不能传递给左值引用参数
    return 0;
}
```

static

static声明只能出现在类内，非const、非inline静态数据成员在类体内声明、类体外定义并初始化。

类的静态数据成员在类还没有实例化对象前就已存在，所有同一名字的静态数据用同一片空间 内存不在类内存里面



注意：定义static的时候不用static 直接用int

非常量静态成员在类内声明 类外定义和初始化 static只能在类里面

```
class A {
public:
    //非常量静态成员必须在类里声明，类外定义和初始化 ，静态成员的static声明只能出现在类里
    static int i ;  //类里声明
};

int A::i = 10;  //类体外初始化，注意不能加static
class A {
public:
    //非常量静态成员必须在类里声明，类外定义和初始化 ，静态成员的static声明只能出现在类里
    static int i = 0;  //错误
};
```

```
class B {
public:
    //const静态整型常量可以在类里初始化
    const static int i = 0;      //正确
    const static int j{ 1 };         //正确

    //错误。error C2864: B::d: 带有类内初始化表达式的静态数据成员必须具有不可变的常量整型类型，或必须被指定为“内联”
    //const static double d = 3.14 ;

    // constexpr类型的字面值类型可以在类体里初始化
    static constexpr double dd = 3.14;  //static constexpr double
};
class C {
public:
    //内联的静态数据成员可以在类里初始化
    inline static int i = 0;
    inline static double d = { 3.14 };

    //inline和const可以一起使用
    inline static const int j = 1;
};
```

函数里面的局部类不能定义静态数据成员

```
void f(void){
    class  T{       //定义函数中的局部类T
        int  c;
       //static int d;       //错误：函数中的局部类不能定义静态数据成员
    };
    T a;                //局部自动变量a
    static T s;         //局部静态变量s
}理解：假设我们用了两次f函数，局部类的生命在局部类也就是f函数里面，第一个类有静态对象，但是要到main结束静态才能结束（语言特点），但是又因为局部类的特点，第一个f结束之后就结束了，所以周期矛盾
```

静态函数成员没有this参数，而构造函数和析构函数都有this指针（在类里面，我们无法用this指向构造函数和构造函数），虚函数，纯虚函数也有this，都不能是静态函数

为什么静态函数成员没有this？因为静态函数的地址不依赖于类，单独存在。this是指向当前成员，所以没有结果

静态函数可以重载，内联和默认初始值，二次声明函数类型不可以在参数表后面加const，volatile和const volatile，但是返回类型可以是inline，const，volatile

类体内定义的静态函数和普通函数一样，相当于内联;同时，在类外定义不能加static

静态成员指针

由于静态数据成员的存储单元受到该类所有对象共享，所以这个指针静态数据指针指向的对象全类对象共享，静态的和非静态的除了访问权限和上面的几个不同，没有什么不同的，一切参数，返回值都可以是静态成员指针

静态成员指针（不管指向静态数据成员还是指向静态函数成员）就是普通指针，因此如果申明指向类的静态成员的指针，就用普通指针。

```
    int *d=&CROWD::number; //普通指针指向静态数据成员
    int (*f)( )=&CROWD::getn; //普通函数指针指向静态函数成员
 static int number;
    static int getn( ) { return number;}
```

普通函数成员指针

因为类的普通函数成员除了函数返回类型及参数列表这两个重要特性外，还有第三个重要特性：所属的类类型。
类的普通成员函数必通过一个对象来调用（this指针就是指向这个对象）。
而普通函数指针无法匹配类的普通函数成员第三个特征：类类型
所以类的普通函数成员必须用成员指针来指向。

```
        int (A::*pf_A)(int,int) = &A::sum;
```

第七章 可访问性

作用域：： 优先级最高

单目:

限定全局标识符号,还有static，和extern

```
extern int fork( );//fork    外部函数
class Process{
    int processes;
processes//类函数访问自己的元素
public:
processes
    int fork( ); //自定义fork函数
掉::会自递归
};
static int processes=1; //总进程数
```

```
int Process::fork( )
{
      processes++;    //访问数据成员
  ::processes++; //访问static变量
   return ::fork( );//调用外部fork，去
                                    }
```

当 **同一作用域中的标识符** 和 **类名（或结构体、联合体的名称）** 重名时，为了明确区分二者，可以使用 `class`、`struct` 或 `union` 来限定标识符，以指定它们是类、结构体或联合体的名称，而不是其他变量或函数的名称。这种做法是为了避免命名冲突，使代码更加明确和可读。

具体意思

在C++中，类、结构体和联合体的名称有时会与局部变量、函数或其他标识符发生冲突。为了让编译器知道某个标识符是类、结构体或联合体，而不是普通变量或函数，我们可以使用 `class`、`struct` 或 `union` 作为限定符。

示例：同名标识符和类名冲突的情况

1. 类名与局部变量重名

假设我们有一个类和一个局部变量的名称相同：

```
cpp复制代码class MyClass {
public:
    void display() {
        std::cout << "Displaying MyClass" << std::endl;
    }
};

int main() {
    int MyClass = 10;  // 局部变量 MyClass 和类名 MyClass 重名
    MyClass obj;  // 编译错误：MyClass 既是类名又是局部变量名
    obj.display();
    return 0;
}
```

在这个示例中，`MyClass` 既是类的名称，又是局部变量的名称。这会导致编译器无法区分 `MyClass` 是指类名还是局部变量，因此会出现编译错误。

2. 使用 `class` 限定符来指定类名

我们可以通过使用 `class` 来明确区分标识符是类名而不是局部变量名：

```
cpp复制代码class MyClass {
public:
    void display() {
        std::cout << "Displaying MyClass" << std::endl;
    }
};

int main() {
    int MyClass = 10;  // 局部变量 MyClass
    class MyClass obj;  // 通过 class 限定符来区分是类 MyClass
    obj.display();
    return 0;
}
```

这里，`class MyClass obj;` 表示我们希望使用的是类 `MyClass`，而不是局部变量 `MyClass`。通过 `class` 限定符，编译器明确知道 `MyClass` 是类名而非局部变量。

3. 使用 `struct` 或 `union` 限定符

同样的做法可以应用到 `struct` 和 `union`：

```
cpp复制代码struct MyStruct {
    int value;
    void display() {
        std::cout << "Displaying MyStruct" << std::endl;
    }
};

int main() {
    int MyStruct = 10;  // 局部变量 MyStruct
    struct MyStruct obj;  // 通过 struct 限定符来区分是结构体 MyStruct
    obj.display();
    return 0;
}
```

同样的道理适用于 `union`：

```
cpp复制代码union MyUnion {
    int value;
    float f;
};

int main() {
    int MyUnion = 10;  // 局部变量 MyUnion
    union MyUnion obj;  // 通过 union 限定符来区分是联合体 MyUnion
    return 0;
}
```

双目：

限定类的元素和限定空间成员，以及恢复访问权限

```
    class  POINT2D{
    //定义二维坐标点
        int  x, y;
    public:
        //获得点的二维x轴坐标
        int  getx( );
        POINT2D (int x, int y){
        //int x访问优先于数据成员x
            POINT2D::x=x;
            //POINT2D::x为数据成员x
            POINT2D::y=y;
        }
    };
class  POINT3D{
//定义三维坐标点
    int  x, y, z;
public:
    //获得点的三维x轴坐标
    int  getx( );
```

```
    POINT3D (int x, int y, int z){
        POINT3D::x=x;
        //POINT3D::x为数据成员x
        POINT3D::y=y;
        POINT3D::z=z;
    }
};
//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;
void main(int argc, char *argv[ ]) {
    POINT2D  p(3,5);
    int  x;
    x=p.POINT2D::getx( );
    x=::x+p.getx( );    //等价于x=::x+p.POINT2D::getx( );//::x是全局变量
    x=POINT2D(4,7).getx( );
    //常量POINT2D(4,7)的作用域局限于表达式
}
```

作用域

标识符作用越小越优先

```
class STACK{
    struct NODE{
        int  val; NODE *next;
        NODE(int v);
    }*head;  //head为数据成员
public:
    STACK( ){head=0;} //0为空指针
    ~STACK( );
    int push(int v);  int pop(int &v);
};
STACK::NODE::NODE(int v){
//::自左向右结合，函数的所属类
    val=v;    next=0;
}
```

名字空间

名字空间必须在全局作用域内用namespace定义，不能在类、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。

```
    namespace  A{int x, f( ){return 1;}; class B{/*…*/};};
    class B{ namespase C{ int y; }; int z; }; //错
    namespace B::C{ int z; };                        //错
    void f( ){ namespace E{ int x}; };          //错
```

同一名字空间内的标识符名必须唯一，不同名字空间内的标识符名可以相同

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。

三种访问形式

1. **直接访问成员的形式为：<名字空间名称>::<成员名称>。直接访问总能唯一的访问名字空间成员。**
2. **引用成员的形式为：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，并且引用时只能给出函数名，不能带函数参数。**
3. **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时用作用域运算符限定**

1. 直接访问成员

通过 `<命名空间名称>::<成员名称>` 的形式直接访问命名空间内的成员。

```
cpp复制代码#include <iostream>

namespace MyNamespace {
    void func() {
        std::cout << "MyNamespace::func called!" << std::endl;
    }

    int value = 42;
}

int main() {
    // 直接访问命名空间成员
    MyNamespace::func();
    std::cout << "Value: " << MyNamespace::value << std::endl;

    return 0;
}
```

输出：

```
makefile复制代码MyNamespace::func called!
Value: 42
```

---

2. 引用成员

通过 `using <命名空间名称>::<成员名称>` 引用特定成员，只引用指定的成员，其他成员仍需通过作用域访问。

```
cpp复制代码#include <iostream>

namespace MyNamespace {
    void func1() {
        std::cout << "MyNamespace::func1 called!" << std::endl;
    }

    void func2() {
        std::cout << "MyNamespace::func2 called!" << std::endl;
    }
}
```

```
int main() {
    using MyNamespace::func1;  // 引用 func1
    func1();                   // 直接调用 func1

    // MyNamespace::func2(); // 仍需要通过命名空间访问
    MyNamespace::func2();

    return 0;
}
```

输出：

```
arduino复制代码MyNamespace::func1 called!
MyNamespace::func2 called!
```

---

3. 引用整个命名空间

通过 `using namespace <命名空间名称>` 引用整个命名空间，所有成员均可直接访问。

```
cpp复制代码#include <iostream>

namespace MyNamespace {
    void func1() {
        std::cout << "MyNamespace::func1 called!" << std::endl;
    }

    void func2() {
        std::cout << "MyNamespace::func2 called!" << std::endl;
    }
}

int main() {
    using namespace MyNamespace;  // 引用整个命名空间

    func1();  // 直接调用
    func2();  // 直接调用

    return 0;
}
```

输出：

```
arduino复制代码MyNamespace::func1 called!
MyNamespace::func2 called!
```

---

4. 多个命名空间成员同名时的处理

当多个命名空间的成员同名时，需要使用作用域运算符限定访问。

```
cpp复制代码#include <iostream>
```

```
namespace NamespaceA {
    void func() {
        std::cout << "NamespaceA::func called!" << std::endl;
    }
}

namespace NamespaceB {
    void func() {
        std::cout << "NamespaceB::func called!" << std::endl;
    }
}

int main() {
    NamespaceA::func();  // 使用 NamespaceA 的 func
    NamespaceB::func();  // 使用 NamespaceB 的 func

    using NamespaceA::func;
    func();              // 默认使用 NamespaceA 的 func

    return 0;
}
```

输出:

```
arduino复制代码NamespaceA::func called!
NamespaceB::func called!
NamespaceA::func called
```

using 关键字

using + 名词空间变量名

using+名字空间成员名字（注意只有名字）using声明的效果是将包含在名字空间的一个成员引入到新的作用域（using声明所在的域），因此在using声明所在的域不能再定义同名的函数或变量

```
    namespace A { int i = 0;}
            using A::i;  //在当前域引入名字i，i的作用域从声明开始到当前域结束
            int i = 0; //错误，该域已经有名字i，不能再定义同名变量
```

using+空间

using指示符的效果是程序可以直接使用名字空间里的成员而不用加限定符，但它没有把名字空间的成员引入到当前域（即using指示符出现的域）。因此在当前域可以定义与名字空间里成员同名的变量和函数。但二义性只有当名字被使用时才被检测到

```
namespace A { int i = 0; int j= 0;}
            using A;  //并没有将成员引入到当前域，只是使程序可以直接用成员名
            int x = i; //可以直接用名字i
            int j = 1; //可以定义同名变量
            //如果程序到此结束，编译器不会报错。因为没有使用变量j
        int y = j + 1; //使用了j，编译器会报错，因为不知道是哪个j
        int z = A::j + ::j; //OK
```

避免冲突

不得已用：：声明使用的对象

嵌套名词空间

一层层解析就可以

```
namespace A {   // A的初始定义
    int x = 5;
    int f( ) { return  6; }
    namespace B { int y = 8, z = 9; }
    using namespace B;
}
using A::x;          //特定名字空间成员using声明，不能再定义变量x
using A::f;          //特定名字空间成员using声明，不能再定义函数f
using namespace A::B;   //非特定成员using，可访问A::B::y, A::B::z,还可重新定义
int y = 10;     //定义全局变量y
void main(void) {
    f( );       //调用A::f( )
    A::f( );        //调用A::f( )
    ::A::f( );  //调用A::f( )
    cout<<x+ ::y + z + A::B::y; //同一作用域有两个y,必须区分
}
```

可以定义别名：

```
可以为名字空间定义别名，以代替过长和多层的名字空间名称。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。
namespace AB=A::B
```

匿名名词空间

作用域为本文件 默认被自动引用整个名词空间，同名的时候自动被屏蔽，但是使用的时候的时候必须要用：：单目运算符去应用新定义的那个变量

```
程序文件A.CPP如下：
#include <iostream.h>
namespace {
//匿名，独立，局限于A.CPP，
//不和B.CPP的合并
    void  f( ) {cout<<“A.CPP\n”;}
    //必须在名字空间内定义函数体
}
namespace A{int g( ){return  0;}}
//名字空间A将和B.CPP合并
int m( ){
    f( );   return A::g( );
}
程序文件B.CPP如下：
#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
```

成员友元 友元关系不可以传递也不可以交换

成员友元是一种将一个类的函数成员声明为其它类友元的函数。例如，派生类访问基类的private

1.如果类A的实例函数成员被声明为类B的成员友元，则这种友元称为实例成员友元。

2.如果类A的静态函数成员被声明为类B的成员友元，则这种友元称为静态成员友元。

全部函数均为友元 友元类

如果某类A的所有函数成员都是类B的友元，则可以简单的在B的定义体内用friend A;声明，不必列出A的所有函数成员。此时称类A为类B的友元类

在声明普通友元时，也可同时定义函数体(自动内联)和类体内定义函数一样

内联的友员函数的存储类默认为static，作用域局限于当前代码文件。全局main的作用域为整个程序，故不能在类中内联并定义函数体，否则便会成为局部(即static)的main函数。（只有在类里面）

第十一章

运算符重载

对于一个符号先找重载再找朴素**前置++和带等号的是左值运算符**

> 左值运算符是运算结果为左值的运算符，其表达式可出现在等号左边，如前置++、--以及赋值运算=、+=、*=和&=等。右值运算符是运算结果为右值的运算符，如+、-、>>、 %、后置++、--等。
> 某些运算符要求第一个操作数为左值，如 ++、-- 、=、+=、&=等

C++规定运算符重载必须针对类的对象（复杂类型），即重载时至少有一个参数代表对象(类型如A、const A、A&、const A&、volatile A等)。（注意A*，A[]不是复杂类型，是简单类型：指针）

C++用operator加运算符进行运算符重载。如果用类的普通成员函数重载运算符，this隐含参数代表第一个操作数对象。

，若想实现一个整数（作为第一个操作数）和A类对象相加，这时显然只能用全局函数去重载A operator+(int, const A &);

根据能否重载及重载函数的类型，运算符分为：

- 不能重载的：sizeof　.　.*　::　?:
- 只能重载为类的普通成员函数的：=、->、()、[]
- 不能重载为类的普通成员函数的：new、delete
- 其他运算符：不能重载为类的静态成员函数，但可以重载为类的普通成员函数和全局函数。

	类的普通成员函数	类的静态成员函数	全局函数
=、->、()、[]	√	×	×
new、delete	×	√	√
其他	√	×	√

不能为全局函数和静态函数重载的原因：

为什么赋值运算符不能作为全局函数重载?

赋值运算符 `operator=` 需要修改 **左操作数**，并且左操作数通常是类的实例。为了修改左操作数，`operator=` 必须能够访问到该实例的成员，这正是 **成员函数** 的职责。成员函数会隐式地拥有一个 `this` 指针，指向调用该函数的对象（即左操作数）。这个 `this` 指针让我们可以访问和修改类对象的状态。

如果你将赋值运算符定义为全局函数，就无法通过 `this` 指针来访问左操作数了，因为全局函数没有 `this` 指针。

为什么其他运算符不能作为静态重载?

因为静态函数没有this，没法和实=实例对象关联，他的内容不依赖于对象的内存，还无法改变内容

```
class A;
int operator= (int,  A&);          //错误，=不能重载为全局函数
A& operator += (A&s, A&r)  //注意，+= 可以重载为全局函数，
{ return  s; }
//注意这时参数不能写A  s，因此时不知A的字节数。而A *r可以。
int operator+(A[6],int);     //不允许，A[6]和A *为普通类型
int operator+(A *, int);      //不允许，A[6]和A *为普通类型
class A{
        friend int operator=(int,  A&);    //错误，=不能为全局函数重载
        static int operator ( ) (A&, int);  //错误，不能为静态成员重载函数调用（）
        static int operator+ (A&, int);      //错误，不能为静态成员
        friend A& operator+=(A&, A&);    //正确，+=可以用全局函数重载
        A& operator ++( );      //隐含参数this代表一个对象
};
//注意重载左值运算符如=、+=、…时最好返回非const左值引用，凡是左值参数最好都用引用(非const)，否则改变了运算符的性质
```

其中： friend int operator=(int，A&)； //错误, =不能为全局函数重载你可能会疑惑这为啥是全局函数，他这个只是类里面的一个声明不是定义，定义在外面

重载运算符参数的位置：

1. 二元运算符（两个操作数）

对于大多数二元运算符（如 `+`, `-`, `*`, `/`, `==`, `!=`, `[]` 等），第一个参数通常是 **左操作数**，第二个参数是 **右操作数**。它们通常表示 **运算符两边的对象**。（参数位置决定第一第二操作数类型）

例子：加法运算符重载（`operator+`）

```
cpp复制代码class A {
public:
    int x;

    A(int val = 0) : x(val) {}

    // operator+ 重载为成员函数
    A operator+(const A& other) const {
        return A(x + other.x);
    }

    // operator+ 重载为全局函数
    friend A operator+(const A& lhs, const A& rhs) {
        return A(lhs.x + rhs.x);
    }
};
```

在这个例子中：

- 如果你重载运算符 `+` 为 **成员函数**，那么第一个参数隐式地是 **当前对象**（即 `this`），第二个参数是传入的对象（`other`）。
- 如果你重载运算符 `+` 为 **全局函数**，那么第一个参数通常是 **左操作数**（`lhs`），第二个参数是 **右操作数**（`rhs`）。全局函数没有 `this` 指针，所以你需要显式地传入两个对象。

区别：

- **左操作数**：通常是第一个参数。在成员函数中，这个参数是隐式的（通过 `this` 指针），在全局函数中是显式传递的。左操作数通常会被修改（如 `+=` 或 `=`），也有些运算符会根据左操作数的状态返回修改后的结果（如 `+` 运算符）。
- **右操作数**：通常是第二个参数。右操作数通常不会被修改，它仅仅作为输入来与左操作数进行计算。

2. 赋值运算符（`operator=`）左值操作符，最好返回非只读左值，不能用const去声明一个左值参数

赋值运算符是一个特殊的运算符，它涉及到对象的赋值操作。它的第一个参数是 **左操作数**（目标对象），第二个参数是 **右操作数**（源对象）。赋值运算符必须返回 **左操作数**，以便支持链式赋值。

```
cpp复制代码class A {
public:
    int x;

    A(int val = 0) : x(val) {}

    // operator= 作为成员函数
    A& operator=(const A& other) {
        if (this != &other) {  // 防止自赋值
            x = other.x;
        }
        return *this;  // 返回左操作数的引用
    }
};
```

- **第一个参数**（`this`）：是目标对象（左操作数），我们要把源对象（右操作数）的值赋给它。
- **第二个参数**（`other`）：是源对象（右操作数），它的值将被复制到左操作数。

3. 前置递增运算符（`++`）

前置递增运算符（`operator++`）也涉及到两个操作数，通常是对象本身，它的第一个参数是 **对象**，第二个参数通常是 **返回值**，不过这里通常是通过 `this` 指针隐式传递的。

```
cpp复制代码class A {
public:
    int x;

    A(int val = 0) : x(val) {}

    // 前置递增运算符重载（成员函数）
    A& operator++() {
        ++x;
        return *this;
    }
};
```

在这个例子中，`operator++` 的第一个参数隐式地是对象本身（通过 `this`），并且通过 `this` 对象的 `x` 成员进行递增。

4. 下标运算符（`[]`）

下标运算符 `[]` 的重载用于访问对象中的元素。它通常有两个参数：第一个参数是 **索引值**，第二个参数是 **要修改的值**。

```
cpp复制代码class A {
public:
    int arr[10];

    int& operator[](int index) {
        return arr[index];
    }
};
```

在这个例子中：

- **第一个参数**：是 **索引值**（如 `index`），用于访问数组中的特定元素。
- **第二个参数**：没有用作下标运算符的常规参数，但可以用于修改值。在 `const` 版本的下标运算符中，它会返回一个常量引用，因此你无法修改下标指定的元素。

重载运算符的类型

1. 重载运算符函数可以声明为类的友元（Friend Functions）

- **友元函数**是被声明为类的内部函数，但并不属于该类的成员。友元函数可以访问类的私有和保护成员，虽然它不在类的内部定义。友元函数可以是普通的非成员函数、全局函数，也可以是运算符重载函数。

  运算符重载为友元函数的示例：

  ```
  cpp复制代码class A {
  private:
      int x;

  public:
      A(int val = 0) : x(val) {}

      // 将运算符重载为友元函数
      friend A operator+(const A& lhs, const A& rhs) {
          return A(lhs.x + rhs.x);
      }
  };
  ```

  在上面的代码中，`operator+` 是 `A` 类的 **友元函数**，它可以访问 `A` 类的私有成员 `x`，尽管它并不是类的成员函数。通过将运算符重载声明为友元函数，可以让其访问类的私有数据，但它依然是类外的函数。

  总结：你可以将运算符重载函数声明为友元函数，这样它可以访问类的私有成员，但它本身不是类的成员函数。

2. 重载运算符的普通成员函数也可定义为虚函数

- **虚函数** 是指在基类中声明并希望在派生类中重写的函数。在多态机制下，虚函数允许根据对象的实际类型调用正确的函数实现。运算符重载函数也可以是虚函数。

  运算符重载为虚函数的示例：

  ```
  cpp复制代码class Base {
  public:
  ```

```
    virtual void operator()() {
        std::cout << "Base class operator() called" << std::endl;
    }
};

class Derived : public Base {
public:
    void operator()() override {
        std::cout << "Derived class operator() called" << std::endl;
    }
};

int main() {
    Base* basePtr = new Derived();
    basePtr->operator()();  // 会调用 Derived 类中的 operator()，实现多态
    delete basePtr;
    return 0;
}
```

在这个例子中，`operator()` 被声明为 **虚函数**，并且在派生类中被重写。通过多态，调用 `operator()` 时会根据指针类型的实际对象来调用相应的版本。

总结：重载运算符函数可以作为普通成员函数定义，并且可以是 **虚函数**。如果是虚函数，就能在多态情况下被派生类重写。

3. 重载运算符的非成员函数被视为普通函数

- **非成员函数** 指的是不属于类的成员函数，但它仍然能访问类的公有接口。对于运算符重载的 **非成员函数**，它们就像是普通函数，它们不依赖于对象的成员函数调用，可以通过显式传递参数来进行操作。

  非成员函数作为运算符重载的示例：

```
cpp复制代码class A {
private:
    int x;

public:
    A(int val = 0) : x(val) {}

    // 非成员函数重载 +
    friend A operator+(const A& lhs, const A& rhs) {
        return A(lhs.x + rhs.x);
    }
};

int main() {
    A a1(5), a2(10);
    A a3 = a1 + a2;  // 使用非成员函数重载 +
    return 0;
}
```

在这个例子中，`operator+` 是一个 **非成员函数**，它作为友元函数定义，但不属于类的成员函数。它通过显式传递参数来执行加法运算。

总结：重载运算符的 **非成员函数** 被视为普通的**全局函数** 或 **自由函数**，它们不能访问类的私有成员，除非是通过 `friend` 机制。

重载运算符一般不能却省参数，（）表示省略参数，转化为参数表

重载为普通函数(全局函数）：参数个数=运算符目数重载为类的普通成员函数（实例函数）：参数个数=运算符目数 - 1 (即this指针)重载为类的静态成员函数：参数个数 = 运算符目数(没有this指针)**注意:有的运算符可以双目可以单目 特殊的运算符-> ++ --除外，会改变运算符的操作数个数**

前后置++和--

他们都会改变当前对象的值，参数最好是非const左值引用类型，也就是左值。可以让实参带出结果

后置（返回值）双目

如果重载为类的普通函数成员，则该函数只需定义一个int类型的参数(已包含一个不用const修饰的this参数)；

如果重载为全局函数，则最好声明非const左值引用类型（原因同上）和int类型的两个参数(无this参数)。

前置单目

前置运算结果应为非const左值，其返回类型应该定义为非只读类型的左值引用类型；左值运算结果可继续++或--运算。

如果重载为全局函数，则最好声明非const左值引用类型一个参数 (无this参数)。

例子：

```
struct A{
    static int  x;
    int  y;
public:
    const A operator ++(int){
    return A(x++, y++);
    }
    A(int i, int j){ x=i; y=j;}
};
int A::x = 23;
int main(){
    A a(4,3);
    A b = a++; //b.x, b.y ,a.x, a.y=?
    return 0;
}
return A(x++,y++)等价于
1：先取x，y的值
temp1 = x; //4
temp2 = y; //3
2: x加1,y加1,对象a变为a(5,4)
3:构造临时对象tempo
tempo = A(temp1,temp2); //A(4,3)
x为静态成员，使得a变为a(4,4)
```

```
4:返回tempo给b，b为b(4,3)
```

语句到；就会实现++，然后再返回

重载->

重载双目->, 使其只有一个参数 (单目) , 返回指针类型

双目->,是唯一可以重载为单目运算的纯双目运算符

下面这个例子说明->重载之后做操作数不是对象指针，右操作数不是对象成员

```
struct A{ int a;  A (int x){ a=x;  } };
class  B{
    A  x;
public:
    A *operator-> ( ) { return &x; };  //只有this，故重载为单目
    B (int v): x (v) {  }


}b (5)   ;
void  main (void){
    int i=b->a;               //等价于下一条语句，i=b.x.a=5
    i=b.operator -> ( ) ->a;   //i=b.x.a=5
}
```

下面的例子 是多个重载运算符造成的重名和使用方法

```
class POINT{
    int x,y;
public:
    POINT(int x,int y):x(x),y(y){}
    POINT operator-() { return POINT(-x,-y); } //单目-
    POINT operator-(POINT p) { return POINT(x-p.x,y-p.y);}//双目-
    POINT operator+(POINT p) { return POINT(x+p.x,y+p.y); }//双目+
    friend POINT operator+(POINT p) { return POINT(p.x,p.y);}//单目+
    friend POINT operator+(POINT p1,POINT p2) { //双目+
        return POINT(p1.x+p2.x,p1.y+p2.y);
    }
};
void main(int argc, char* argv[]){
        POINT p1(1,2),p2(3,4);
        POINT p3=-p1;                       //调 POINT operator-()
        POINT p4=p1-p2;                   //调 POINT operator-(POINT p)
        POINT p5=p1.operator -(p2);     //调 POINT operator-(POINT p)
        POINT p6 = +p1;                   //调 friend POINT operator+(POINT p)
        POINT p7=operator+(p1,p2);      // friend POINT operator+(POINT p1,POINT
p2)
        POINT p8 = p1.operator +(p2);   //调 POINT operator+(POINT p)
        POINT p9 = p1+p2;                   //错误，无法确定是调用友元还是函数成员
}
```

为什么 `friend` 函数不是成员函数?

在 C++ 中，`friend` **函数** 是一个被声明为类的 **朋友** 的函数，意味着它可以访问类的私有成员和保护成员，但它 **并不是类的成员函数**。尽管它在类内部被声明，实际上它是一个 **全局函数**，只是在类内部声明，以便它能够访问类的私有成员。

（）的重载

( )只能通过类的普通成员函数重载，这意味着( )第一个操作数必须是this指针指向的类对象，这样的对象称为函数对象（也称为仿函数（Functor））

```
class AbsInt{
public:
    //语法：returnType operator( ) (参数列表)
    //调用：objectofAbsInt(实参)，类AbsInt的对象称为函数对象
    int operator()( int val) { return val > 0? val:-val; }
} absInt;
int i = absInt(-1); //函数对象可以作为函数的参数，这意味着我们可以将函数当做对象传递．在模板编程里广泛使用。在C+11里，还可以用lamda表达式
函数指针也可以作为函数的参数，但函数指针主要的缺点是：
    被函数指针指向的函数是无法内联的。而函数对象则没这个问题
```

为什么函数指针指向的函数无法内联？因为内联是在编译内联，编译的时候地址没定，指针没法确定地址

但是如果那如果我有一个内联inline声明的函数和一个指向他的函数指针不会报错

内联函数的声明只是告诉编译器尽可能将函数体插入调用位置，而并不会阻止你使用函数指针。

编译器会根据上下文决定是否内联该函数。如果内联函数通过函数指针调用，编译器 **通常不会内联**，因为它无法确定目标函数的具体位置（函数指针指向的函数是动态的）。

赋值 调用

A( )、A(A&&)、A(const A&) 、~A( ) 、A&operator=(const A&)、A& operator=(A&&)

默认赋值运算函数实现数据成员的浅拷贝赋值，如果普通数据成员为指针或引用类型且指向了一块动态分配的内存，则不复制指针所指存储单元的内容。若类不包含上述普通数据成员，浅拷贝赋值不存在问题。

如果函数参数为值参对象，当实参传值给形参时，若类A没有自定义拷贝构造函数，则值参传递也通过浅拷贝构造实现（调用编译器提供的默认拷贝构造函数）。

为了防止内存泄漏

（1）应定义“T(const T &)”形式的深拷贝构造函数；
（2）应定义“T(T &&) noexcept”形式的移动构造函数；
（3）应定义“virtual T &operator=(const T &)”形式的深拷贝赋值运算符；
（4）应定义“virtual T &operator=(T &&) noexcept”形式的移动赋值运算符；
（5）应定义“virtual ~T( )”形式的虚析构函数；
（6）在定义引用“T &p=*new T( )”后，要用“delete &p”删除对象；
（7）在定义指针“T *p=new T( )”后，要用“delete p”删除对象；
（8）对于形如“T a; T&& f( );”的定义，不要使用“T && b=f( );”之类的声明和“a=f( );”
（9）不要随便使用exit和abort退出程序。
（10）最好使用异常处理机制。
(11)注意delete p和delete [ ]p

实例

```
class A {
private:
    int size;  int *p;
public:
    A():size(1),p(new int[size]){cout << "Default Constructor, size = " << size
<< endl;}
    A(int s):size(s > 0?s:1),p(new int[size]){cout << "Constructor,size = " <<
size << endl;}
    virtual ~A() {
        cout << "Deconstructor,size = " << size << endl; if (p){delete[] p; p =
0;size = 0;}
    }
    int get_size() { return size; };
    A(const A &old):size(old.size),p(new int[size]) { //拷贝构造
        for (int i = 0; i < size; i++) {p[i] = old.p[i];} cout << "Copy
constructor,size = " << size << endl;
    }
    A(A &&old):size(old.size),p(old.p) { //移动构造
        old.p = 0; old.size = 0; cout << "Move constructor,size = " << size <<
endl;
    }
    A &operator=(A &&rhs){
        if(this == &rhs) return *this;
        int *t = this->p; this->p = rhs.p;  this->size = rhs.size; rhs.p = 0;
rhs.size = 0;
        if(t) delete t;
        cout << "Move =,size = " << size << endl;
    }
};
//返回右值引用的函数要非常小心，特别是引用了即将出栈的临时对象
A&& f() { return A(100); }
A&& g() { return A(200); }
int main() {
    A a;
//    A && rr1 = f();  //右值引用引用了一个已经出栈的对象
//    cout << rr1.get_size() << endl;  //会输出不确定值，因为局部对象出栈
//    A && rr2 = g();
```

```
//    cout << rr2.get_size() << endl;   //会输出不确定值，因为局部对象出栈

    A c = A(100);   //在C++11下，调用移动构造；但是在C++17下，被编译器优化为调用构造函数A(int)，等价于A c(100);
    A d = a;    //调用拷贝构造

    //f()返回右值引用，然后用该右值引用引用的对象去构造对象b
    // 应该调用移动构造，但是该右值引用引用的对象已经在f()返回后出栈，因此出错
//    A b = f();

    //A &r = f();  //编译错误，f()返回的右值

    //和A b = f();存在同样的问题： f()返回右值引用，然后将该右值引用引用的对象移动赋值给a
    //应该调用移动赋值，但是但是该右值引用引用的对象已经在f()返回后出栈，因此出错
    a = f(); return 0;
}
```

对于形如“T a; T&& f( );”的定义，不要使用“T && b=f( );和“a=f( );”之类的声明

运算符重载强制类型

C++是强类型的语言，运算时要求类型相容或匹配。隐含参数this匹配调用当前函数的对象，若用const、volatile说明this指向的对象，则匹配的是const、volatile对象。
如定义了合适的类型转换函数重载，就可以完成操作数的类型转换；如定义了合适的构造函数，就可以构造符合类型要求的对象，构造函数也可以起到类型转换的作用。

```
//重载类型强制转换函数：operator targetType()，不需要定义返回类型
    //( ) 只能用普通成员函数重载
    virtual operator int (  ) const{ return  i;  } //类型转换返回右值(int)
```

这个话的意思是 这个强制类型转化转化后是int类型 int表示函数返回类型 所以是右值 把这个i返回个函数就是右值（和普通函数一样）

单参数构造函数

单参数的构造函数相当于类型转换函数，单参数的T::T(const A) T::T(A&&) 、T::T(const A&)等相当于A类到T类的强制转换函数。

不应该同时定义T::operator A( )和A::A (const T&)，否则容易出现二义性错误。

按照C++约定，类型转换的结果通常为右值，故最好不要将类型转换函数的返回值定义为左值，也不应该修改当前被转换的对象 (参数表后用const说明this)。

转换的目标类型只要是能作为函数的返回类型即可。函数不能返回数组和函数。因此C++规定转换的类型表达式不能是数组和函数，但可以是指向数组和函数的指针

重载new和delete

new的参数就是要分配的内存的字节数。其函数原型为

```
      extern void * operator new(unsigned bytes);
      extern void operator delete(void *ptr);
```

在使用运算符new分配内存时，使用类型表达式而不是值表达式作为实参，编译程序会根据类型表达式计算内存大小并调用上述new函数。例如：new long[20]

模板与内存回收

变量模板使用类型形参定义变量的类型，可根据类型实参生成变量模板的实例变量

在定义变量模板、函数模板和类模板时，类型形参的名称可以使用关键字class或者typename定义，即可以使用**“template”或者“template”。**

生成模板实例变量时，将使用实际类型名代替T。实际类型名可以是内置类型名、 类名或类模板实例（类模板实例就是具体的class，是类型）

例子：

```
#include<stdio.h>
template<typename  T>
constexpr T pi = T(3.1415926535897932385L);      //定义变量模板pi，其类型形参为T

template<class T> T area(T r) {      //定义函数模板area，其类型形参为T
    printf(“%p\n”, &pi<T>);      //生成模板函数实例时附带生成变量模板pi<T>的变量模板实例
    return pi<T> * r * r;
}
实例化：
template const float pi<float>; //显式生成变量实例pi<float>，不能使用constexpr
//等价于“const float pi<float>= float (3.1415926535897932385L);”
template const double pi<double>;   //生成模板实例变量pi<double>，实参类型为double
//等价于“const double pi<double >= double (3.1415926535897932385L);”
```

实例化之后的使用格式是：

模板名字+<实例类型>（函数参数）上面的模式是显示调用 实际上直接用也可以

```
    const float &d1 =pi<float>;      //引用变量模板生成的模板实例变量pi< float>

    const double &d2=pi<double>;         //引用变量模板生成的模板实例变量pi<double>

    const long double &d3=pi<long double>;  //生成并引用模板实例变量pi<long double>

    float a1= area<float>(3);        // area<float>为函数模板area<T>的实例函数

    double a2 = area<double>(3);

    long double a3 = area<long double>(3);

    int a4 = area<int>(3);           //调用area<int>时，隐式生成变量模板实例pi<int>
}
```

所以有三种实例化方式：

```
template const float amu<float> 第一个float是·返回类型，第二个是指明实例化类型，二者要一样
const float amu<float>=float(3.133333)
直接：amu<float>
```

全军背诵！

变量模板不能在函数内部声明。显式或隐式实例化生成的变量模板实例和变量模板的作用域相同。

因此，变量模板生成的变量模板实例只能为全局变量或者模块静态变量。

模板的参数列表除了可以使用类型形参外，还可以使用非类型的形参。

在变量模板实例化时，非类型形参需要传递常量作为实参。

非类型形参也可以定义默认值，若变量模板实例化时未给出实参，则使用其默认值实例化变量模板。

```
void main(void)
{
    const double &f=pi<double>;      //引用在main()外生成的全局变量pi<double>
    double &g=girth<double,4>;  //引用在main()外生成的static girth <double>，非类型实参传递常量
    double &h=girth<double, sizeof(printf("abc"))>;//引用在main()外生成的static girth <double>
    double &k=girth<double>;         //引用在main()外用默认值生成的static girth <double>
    printf("%lf\n", girth<double, 4>);  //生成的模板实例变量不是main函数的局部自动变量
    printf("%lf\n", area<double>(4));
}
#include<stdio.h>
template<typename  T>        //定义变量模板pi，其类型形参为T
constexpr  T  pi = T(3.1415926535897932385L);

template<class T> T area(T r) {          //定义函数模板area，其类型形参为T
    return pi<T> * r * r;
}
template<class T, int x=3>       //定义变量模板girth，其类型形参为T
static  T  girth = T(3.1415926535897932385L*2*x);

template float girth<float>;     //生成变量模板实例static girth<float>，作用域与变量模板相同
```

函数模板是使用类型形参定义的函数框架，可根据类型实参生成函数模板的模板实例函数。

模板（变量模板、函数模板、类模板）的声明或定义只能在全局范围、命名空间或类范围内进行。

根据函数模板生成的模板实例函数也和函数模板的作用域相同。

在函数模板时，可以使用类型形参和非类型形参。模板实例化时非类型形参需要传递常量作为实参。可以单独定义类的函数成员为函数模板。

```
template <class T, int m=0> //class可用typename代替
void swap(T& x, T& y=m)
{
    T  temp = x;
    x = y;
    y = temp;
}

template <class D, class S>      //class可用typename来代替定义形参D、S
D convert(D& x, const S& y)      //模板形参D、S必须在函数参数表中出现
```

```
{
    return x = y;            //将y转换成类型D后赋给x
}
struct A {
    double i, j, k;
public:
    A(double x, double y, double z) :i(x), j(y), k(z) { };
};
void main(void){
    long x = 123, y = 456;
    char a = 'A', b = 'B';
    A  c(1, 2, 3), d(4, 5, 6);
    swap<long, 0>(x, y);     //显式给出类型实参，必须用常量传给非类型实参m，函数模板实例化得到实例函数
    swap(x, y); //自动生成实例函数void swap(long &x, long &y)，通过实参推断类型形参T，隐式实例化函数模板
    swap(a, b); //自动生成实例函数void swap(char &x, char &y)，隐式给出类型实参char，m=0
    swap(c, d); //自动生成实例函数void swap(A &x, A &y)
    convert(a, y);  //自动生成实例函数char convert (char &x, const long &y)
```

单独定义类的函数成员为函数模板的例子：

```
#include<typeinfo>
class ANY {          //定义一个可存储任何简单类型值的类ANY
    void* p;
    const char* t;
public:
    template <typename T> ANY(T x) {//单独定义构造函数模板
        p = new T(x);
        t = typeid(T).name();
    }
    void* P() { return p; }
    const char* T() { return t; }   //此T为函数成员的名称，不是模板的类型形参
    ~ANY() noexcept { if (p) { delete p;  p = nullptr; } }
}a(20);          //自动从构造函数模板生成构造函数ANY::ANY(int) ,通过实参类型推断
void main(void){
    double* q(nullptr); //等价于“double* q=nullptr;”
    if (a.T() == typeid(double).name()) q = (double*)a.P(); //a.P()返回void *
}
```

函数模板可省略性

函数模板的类型形参允许参数个数可变，用省略参数“...”表示任意个类型形

参用递归定义的方法可展开并处理这些类型形参。

生成实例函数时，可能因递归生成多个实例函数，这些实例函数之间递归调用

```
#include<bits/stdc++.h>
using namespace  std;
#include <iostream>
using namespace std;
template<class...Args>       //类型形参个数可变的函数模板声明
```

```
int  println(Args...args);

template<class...Args>      //递归下降展开的停止条件：println( )的参数表为空
int  println( ) {
    cout << endl;
    return 0;
}

template < class H, class...T>  //递归下降展开println()的参数表
int  println(H h, T...t) {
cout << h<<"*";
return 1+println(t...);     //递归下降调用
}
int main( ) {
    int n= println(1, "4OOP", 3.3, "EXPAND"); //根据实参，递归的推断类型实参并展开，生成实例函数（递归调用）
    return n;           //返回n=4
}
```

`println` 函数模板声明：

```
template<class...Args>  // 类型形参个数可变的函数模板声明
int  println(Args...args);
```

这里定义了一个可以接受任意数量和类型参数的模板函数 `println`。`Args...args` 是一个类型为变长参数包的参数列表。`Args...` 表示可以接受任意类型和个数的参数。

递归展开 `println` 函数的实现：

```
template<class...Args>
int println() {
    cout << endl;
    return 0;
}
```

这是递归展开的终止条件，当 `println` 被调用时没有参数时，它会输出一个换行符并返回 `0`。这是递归的基本情况。

```
template <class H, class...T>
int println(H h, T...t) {
    cout << h << "*";
    return 1 + println(t...);  // 递归调用
}
```

这个模板函数处理实际的参数列表。它接受至少一个参数 `h`，并递归地处理其余的参数。对于每个参数，`println` 会输出该参数的值并在后面添加 `*`。递归调用 `println(t...)` 会逐个处理剩余的参数。

具体来说：

- `H` 是第一个参数。
- `T...` 是剩余的参数列表。
- `cout << h << "*";` 会输出第一个参数，并加上 `*`。

- `return 1 + println(t...)` 会递归调用 `println` 处理剩余的参数，并且每次调用返回一个递增的值。直到参数列表为空（即递归结束）。

`main` **函数**:

```
cpp复制代码int main() {
    int n = println(1, "4OOP", 3.3, "EXPAND");  // 实参为多个不同类型的值
    return n;  // 返回n=4
}
```

在 `main` 函数中，调用 `println(1, "4OOP", 3.3, "EXPAND")`，这个调用会被递归展开：

- `println(1, "4OOP", 3.3, "EXPAND")` 输出 `1*`，然后递归调用 `println("4OOP", 3.3, "EXPAND")`。
- `println("4OOP", 3.3, "EXPAND")` 输出 `"4OOP*"`，然后递归调用 `println(3.3, "EXPAND")`。
- `println(3.3, "EXPAND")` 输出 `3.3*`，然后递归调用 `println("EXPAND")`。
- `println("EXPAND")` 输出 `"EXPAND*"`, 然后递归调用 `println()`。

最后，递归的终止条件会被触发，调用 `println()` 输出一个换行符，并返回 `0`。

所以，最终输出的结果是：

```
1*4OOP*3.3*EXPAND*
```

在调用函数时可隐式自动生成模板实例函数（根据实参推断）。也可使用"template 返回类型 函数名<类型实参>(形参列表)"，显式强制函数模板按类型实参显式生成模板实例函数。

特化:其实就是定义函数 可以加多一句template<>而已

```
template < >            //此行可省， template < >表示特化实例函数：特化函数将被优先调用
const char *max(const char *x, const char *y)   //特化函数：用于隐藏模板实例函数
{
    return strcmp(x, y)>0?x:y;          //进行字符串内容比较
}
```

类模板

```
template <class T, int v=20>    //类模板的模板参数列表有非类型形参v，默认值为20
class VECTOR
{
    T  *data;
    int  size;
public:
    VECTOR(int n = v+5);        //由v构成表达式v+5，将其作为构造函数成员形参的默认值
    ~VECTOR( ) noexcept;
    T &operator[ ](int);
};

template <class T, int v>    //在类体外函数实现时，加template<class T, int v>, v不能给缺省值
```

```
VECTOR <T, v>::VECTOR(int n)    //须用VECTOR <T, v>作为类名
{    data = new T[size = n];  }
template <class T, int v>  //函数实现时，加template<class T, int v>, v不能给缺省值
VECTOR <T, v>::~VECTOR( ) noexcept  //须用VECTOR <T, v>作为类名
{
    if (data) delete data;
    data = nullptr;
    size = 0;
}

template <class T, int v>
T &VECTOR <T, v>::operator[ ](int i)        //须用VECTOR <T, v>作为类名
{
    return data[i];
}
public:
    int full( ) { return top==VECTOR<T>::getsize( ); }
    int null( ) { return top==0; }
    int push(T t);
    int pop(T& t);
    STACK(int s): VECTOR<T>(s) { top = 0; };
    ~STACK( ) noexcept { };
};
template <class T>
int STACK<T>::push(T  t)
{
    if (full( )) return 0;
    (*this)[top++] = t;   //调用operator[]
    return 1;
 }
void main(void)
{
    typedef  STACK<double>  DOUBLESTACK; //对实例化的类重新命名定义
    STACK <int>     LI(20);    //类模板实例化得到实例类STACK<int>（及其父类
VECTOR<int>）
    STACK <long>  LL(30);
    DOUBLESTACK     LD(40); //等价于“STACK<double> LD(40);”
}
```

类体外实现函数不可以用缺省值 在实例化派生类时，如果基类是用类模板定义的，也会同时实例化基类。这里的实例化指模板类实例化，生成实例类

派生类函数在调用基类的函数时，最好使用“基类<类型参数>::”限定基类函数成员的名称

调用基类的函数时，最好使用“基类<类型参数>::”限定基类函数成员的名称

类模板中的多个类型形参的顺序变化对类模板的定义没有影响

```
template<class T1, class T2> struct A {     //定义类模板A
    void f1( );
    void f2( );
};
template<class T2, class T1> //正确：A<T2, T1>同类型形参一致
void A<T2, T1>::f1( ){ }
```

```
template<class T1, class T2> //正确：A<T1, T2>同类型形参一致
void A<T1, T2>::f2( ){ }

//template<class T2, class T1> //错误：A<T1, T2>与类型形参不同
void A<T1, T2>::f2( ){}
template<class...Types> struct B {
    void f3( );
    void f4( );
};
template<class ...Types> void B<Types ... >::f3( ){ }   //正确：Types表示类型形参列表
template<class ...Types> void B<Types ... >::f4( ){ }   //正确：Types表示类型形参列表
//template<class ...Types> void B<Types>::f4( ){ }  //错误：必须用“Types ...”的形式
void main(void) { }
```

反正一句话 <>里面和模板里面的<>要一幕意义！！！

//通过auto定义变量模板n，元素全部初始化为0 template auto n = new VECTOR[10]{ };

//通过auto定义变量模板ptemplate static auto p = new VECTOR[10];

//定义Lambda表达式模板template auto q = _->T& { return x; };

A<>指示模板实例的如何实例化 就是用上面去代替哪个T

```
template <typename T> auto m = new T[10];
//显式将变量模板实例化，产生变量m<int>。可以注释下面一行，直接使用m<int>(隐式实例化),特别是使用auto时
template int * m<int>;

//通过auto定义变量模板，元素全部初始化为0
template <class T> auto n = new VECTOR<T>[10]{ };

//显式将变量模板实例化，产生变量n<int>。可以注释下一行，直接使用n<int>(隐式实例化),特别是使用auto时
template VECTOR<int>*  n<int>;

//定义Lambda表达式模板，可以直接（实际上只能）隐式实例化Lambda表达式模板，如q<int>(i)
template <class T> auto q = [ ](T& x)->T& { return x; };
```

实例化类模板

```
template 类名<参数>采用“template  类名<类型实参列表>”的形式显示直接实例化类模板。参见例13.14中的：template  A <int>
```

实例化生成的实例类同类模板的作用域相同

特化类： 样式和模板实例一样

当实例化生成的类实例、函数成员实例不合适时，可以自定义（特化）类、函数成员隐藏编译自动生成的类实例或函数成员

如果类模板定义了虚函数，由编译器自动实例化产生的实例类的对应函数是虚函数。但如果自己定义特化类，可以修改为非虚函数。特化就是自己定义实例类，不是由编译器自动产生 就是自己定义了一个新的

```
template < >            //定义特化的字符指针向量类只加了这句话
class VECTOR <char*>
{
    char** data;
    int  size;
public:
    VECTOR(int);            //特化后其所属类名为VECTOR <char*>
    ~VECTOR( ) noexcept;        //特化后其所属类名为VECTOR <char*>，不是虚函数
    virtual char*& operator[ ](int i) { return data[i]; };  //特化后为虚函数
};
VECTOR <char*>::VECTOR(int n)   //使用特化后的类名VECTOR <char*>
{
    data = new char* [size = n];
    for (int i = 0; i < n; i++) data[i] = 0;
}
VECTOR <char*>::~VECTOR( ) noexcept //使用特化后的类名VECTOR <char*>
{
    if(data==nullptr) return;
    for(int i=0; i<size; i++)   delete data[i];
    delete data;    data=nullptr;
    cout << "DES C\n";
}
class A : public VECTOR<int> { //VECTOR<int>是类模板的实例类，由编译器自动产生，A继承
VECTOR<int>
public:
    A(int n): VECTOR<int>(n) { }; //调用父类VECTOR<int>的构造函数
    ~A( ){ cout << "DES A\n"; } //自动成为虚函数：因为基类析构函数是虚函数
};
class B : public VECTOR<char*> {//VECTOR<char *>是自定义的特化实例类，B继承
VECTOR<char *>
public:
    B(int n): VECTOR<char*>(n) { };
    ~B( ) noexcept { cout << "DES B\n"; }//特化的VECTOR<char *>析构函数不是虚函数，故
~B也不是
};
void main(void)
{
       VECTOR <int>     LI(10);     //自动生成的实例类VECTOR <int>
        VECTOR <char*>  LC(10);     //优先使用特化的实例类VECTOR<char*>
        VECTOR<int>* p = new VECTOR<int>(3);
        delete p;
        p = new A(3);   //VECTOR<int>是A的父类，父类指针指向子类对象
        delete p;
        VECTOR<char*> *q = new VECTOR<char*>(3);
        delete q;
        q = new B(3);    //VECTOR<char *>是B的父类，父类指针指向子类对象
        delete q;
}
```

特化函数成员

```
template < >  VECTOR <char*>::~VECTOR( ) noexcept{   }
```

实例函数可以成为类的友元函数

```
template<class T>
    void output( T &x){ cout << x.k;}  //定义函数模板


     //定义派生类B时，自动生成实例类A<int>作为B的父类
    class B:public A<int>{
        int k;
        friend void output<B>(B &); //生成模板函数实例并将其作为B的友元
    public:
        B(int x):A<int>(x) { k = x;} //必须显式调用父类构造
        operator int(){ return k + A<int>::operator int();} //自动成为虚函数
    };
```

定义派生类B时，自动生成实例类A作为B的父类 但是要在构造B的时候手动构造构造父类的构造函数 其实这里的规矩和前面上面父类子类上面时候要子类显示构造一样 如果是默认的无参构造函数不要手动 可以自动

实例成员指针

当使用类模板的实例化类，作为类模板实例化的实参时，会出现嵌套的实例化现象。原本没有问题的类型形参T，用实参int实例化new T[10]时没有问题；但在嵌套实例化时，若用A实例化new T[10]时，则会要求类模板A定义定义无参构造函数A::A()。

当使用类模板的实例化类，作为类模板实例化的实参时，会出现嵌套的实例化现象。原本没有问题的类型形参T，用实参int实例化new T[10]时没有问题；但在嵌套实例化时，若用A实例化new T[10]时，则会要求类模板A定义定义无参构造函数A::A()。

原因是 T是int的时候不会用构造函数 但是要是new一个实例类 就要有他的构造函数

若类模板中使用非类型形参，实例化时使用表达式很可能出现“>”，导致编译误认为模板参数列表已经结束，此时可用“（）”如List<int, (3>2)> L1(8)

区分

```
template <class T, int n=10>
struct A {
    static T t;
    T  u;
    T* v;
    T  A::* w;       //实例成员指针指向A的T类型成员
    T  A::* A::* x;      //实例成员指针指向A的T  A::*类型成员
    T  A::** y;      //普通指针y指向T  A::*类型成员
    T* A::* z;       //实例成员指针指向A的T* 类型成员
    A(T k=0, int h=n);           //因A( )被调用，故必须定义A( )，等价于调用A(0, n)
    ~A( ) { delete[ ]v; }
};
template <class T, int n>  T A<T, n>::t = 0;     //类模板静态成员的初始化
```

第一个w前面有指针 指针有类所以是类函数指针 第二个x有指针，有类 是一个函数指针 对象是T的一个成员指针第三个y前面有指针没有类 是普通指针 指向一个成员指针 z是一个成员指针 对像是类指针

```
template <class T, int n>
A<T, n>::A(T k, int h)           //不得再次为h指定默认值，因为其类模板已指定
{
    u = k;
    v = new T[h];                //初始化数组对象，必须调用无参构造函数T( )
    w = &A::u;
    x = &A::w;
    y = &w;
    z = &A::v;
     v = &A::t;  // &A::t：取静态成员t的地址，因此v是普通指针
}
template  struct  A<double>;
```

定义用两个栈模拟一个队列的类模板

```
#include <iostream>
using namespace std;
template <typename T>                //定义主类模板
class STACK {
    T* const  elems;                     //申请内存，用于存放栈的元素
    const int   max;                     //栈能存放的最大元素个数
    int   pos;                       //栈实际已有元素个数，栈空时pos=0;
public:
    STACK(int m=0);                  //等价于定义了STACK(),防嵌套实例化出问题
    STACK(const STACK& s);               //用栈s初始化p指向的栈
    STACK(STACK&& s)noexcept;            //用栈s初始化p指向的栈
    virtual T operator [ ] (int x)const;             //返回x指向的栈的元素
    virtual STACK& operator<<(T e);          //将e入栈，并返回p
    virtual STACK& operator>>(T& e);         //出栈到e，并返回p
    virtual ~STACK( )noexcept;           //销毁p指向的栈
    ……
};
template <typename T>
STACK<T>::STACK(STACK&& s) noexcept: elems(s.elems), max(s.max), pos(s.pos){
```

```
    const_cast<T*&>(s.elems) = nullptr;    // 等价于*(T**)&(s.elems) = nullptr;
    const_cast<int&>(s.max) = s.pos=0;
}
template <typename T>
class QUEUE: public STACK<T> {
    STACK<T>  s2;                //队列首尾指针
 public:
    QUEUE(int m=0);              //初始化队列：最多m个元素
    QUEUE(const QUEUE& s);           //用队列s复制初始化队列
    QUEUE(QUEUE&& s) noexcept;           //移动构造
    virtual QUEUE& operator=(QUEUE&& s) noexcept;   //移动赋值
    ~QUEUE( )noexcept;               //销毁队列
    ……
};
template <typename T>
//以下初始化一定要用move，否则QUEUE是移动赋值而其下层是深拷贝赋值
QUEUE<T>::QUEUE(QUEUE&& s) noexcept: STACK<T>(move(s)), s2(move(s.s2)) { }
template <typename T>
QUEUE<T>& QUEUE<T>::operator=(QUEUE<T>&& s) noexcept {
    //以下赋值一定用static_cast，否则QUEUE是移动赋值而其下层是深拷贝赋值
    *(STACK<T>*)this = static_cast<STACK<T>&&>(s);
    //等价于STACK<T>::operator=(static_cast<STACK<T>&&>(s));
    //或等价于STACK<T>::operator=(std::move(s));
    s2 = static_cast<STACK<T>&&>(s.s2);
    //等价于“s2=std::move(s.s2);”，可用“std::move”代替“static_cast<STACK<T>&&>”
    return *this;
}
```

第十章 多继承类和虚基类

多继承派生类有多个基类或虚基类。同一个类不能多次作为某个派生类的直接基类，但可多次作为一个派生类的间接基类(同编译程序相关)。

> 你提供的代码 `class QUEUE: STACK, STACK{ };` 出现错误的原因是 同一个基类不能在派生类列表中重复。在 C++ 中，类继承时，每个直接基类只能出现一次。如果一个类的继承列表中重复了基类

多继承派生类继承所有基类的数据成员和函数成员。 多继承派生类在继承基类时，各基类可采用不同的派生控制符。

基类之间的成员可能同名，基类与派生类的成员也可能同名。在出现同名时，如面向对象的作用域不能解析，可使用基类类名加作用域运算符::来指明要访问基类的成员

代理模式实现多继承

```
class A{ public:  void f( ) {}};
class B{ public:  void g( ) {}};
如果需要实现一个类，同时具有类A和类B的行为，但又不能
多继承怎么办（如JAVA）？
采用代理模式，继承一个类，将另外一个类的对象作为数据成员
```

```
class C: public A{
       B b; //B类行为的代理
public:
       void g( ) { b.g( ): } //定义一个同名的g函数，但其功能
            //委托对象b完成(通过调用b.g())，因此                //C::g的行为与B::g完全一致
       //A::f被C继承
};
//这样C就具有A的行为f和B的行为g,达到了多重继承的效果
```

委托代理在多数情况下能够满足需要，但当对象成员和基类物理上是同一个基类时（存在一个共同的基类），就可能对同一个物理对象重复初始化(可能是危险的和不必要的)。

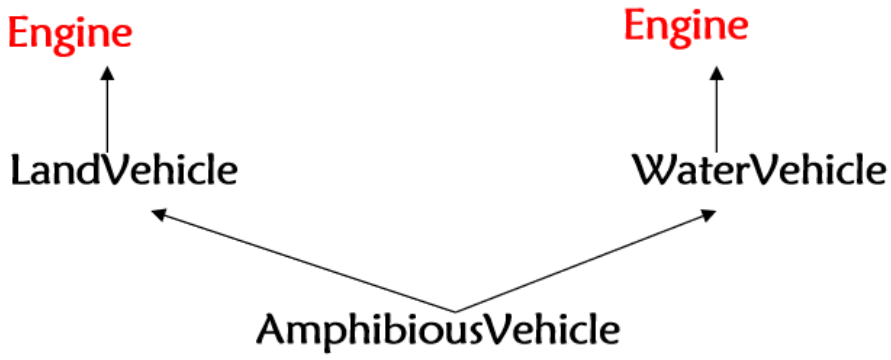
```
  class Engine{ /*...*/};
  class LandVehicle: Engine{/*...*/};
  class WaterVehicle: Engine{/*...*/};
  class AmphibiousVehicle: LandVehicle{WaterVehicle  wv; };
```

定义方式

```
class 派生类名:<派生方式> 基类1,<派生方式> 基类2,…{
    <类体>
};
```

单纯的多继承无法解决同一基类多次定义的问题

可以采用全局变量、静态数据成员做标记，解决同一个物理对象初始化两次的问题；此外，还需要解决同一物理对象两次析构问题



上述定义存在的问题：两栖机车要安装两个引擎Engine，可引入虚基类解决该问题。

虚基类

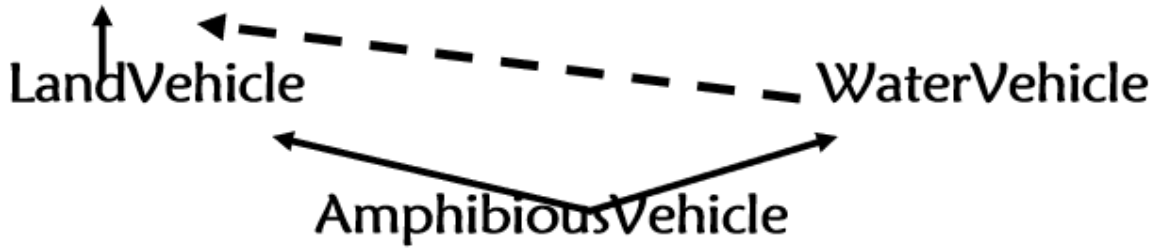
虚基类用virtual声明，可把多个逻辑基类对象映射成同一个物理虚基类对象

映射成的这个物理虚基类对象尽可能早的构造、尽可能晚的析构，且**构造和析构都只进行一次**

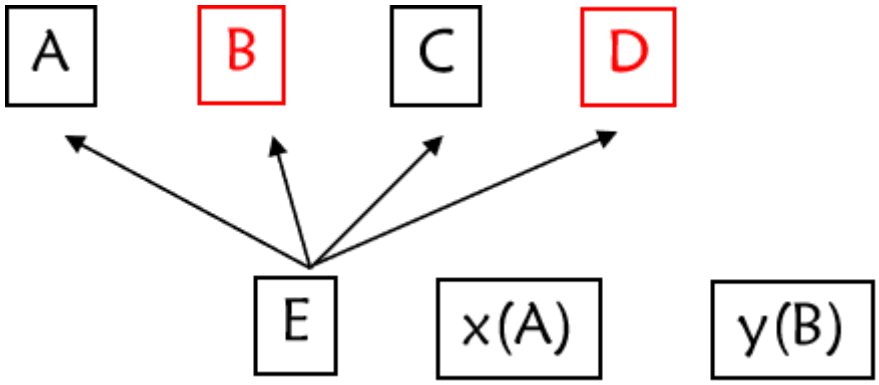
若虚基类的构造函数都有参数，必须在派生类构造函数的初始化列表中列出虚基类的构造实参的值(注意：使用虚基类继承的时候要声明基类达到virtual 在派生类的构造函数里面，不加virtual)

```
class Engine{ /*...*/ };
    class LandVehicle: virtual public Engine{ /*...*/ };
    class WaterVehicle: public virtual Engine{ /*...*/ };
    class AmphibiousVehicle: LandVehicle, WaterVehicle{  };
```

virtual和继承类型可以倒过来



同一棵派生树中的同名虚基类，共享同一个存储空间；其构造和析构仅执行1次，且构造尽可能最早执行，而析构尽可能最晚执行。由派生类（根）、基类和虚基类构成一个派生树的节点，而对象成员将成为一棵新派生树的根。红色就是虚基类




```
struct A {};
struct B{};
struct C{};
struct D{};
struct E: A, virtual B, C, virtual D{
       A x;
       B y;
};
```

虚基类的继承类型可以是private和protected也可以是public

若虚基类的构造函数都有参数，必须在派生类构造函数的初始化列表中列出虚基类的构造实参的值(注意：使用虚基类继承的时候要声明基类达到virtual 在派生类的构造函数里面，不加virtual

LandVehicle (int s, int p): Engine (p), speed (s) { } 这里可以在engine前面不加virtual

`AmphibiousVehicle` 是最终派生类，它需要确保整个继承体系中唯一的虚基类实例 `Engine` 被正确初始化。

如果 `AmphibiousVehicle` 构造函数中没有显式调用 `Engine(p)`，那么 `Engine` 的构造函数根本不会被调用，这会导致未初始化的虚基类实例，编译器会报错。

为什么 `LandVehicle` 和 `WaterVehicle` 中调用 `Engine(p)` 无效?

- `LandVehicle` 和 `WaterVehicle` 是虚继承 `Engine`，这意味着它们并没有直接初始化 `Engine` 的权利，它们的 `Engine(p)` 调用只是形式上的，实际的初始化被推迟到最终派生类 `AmphibiousVehicle` 中。

```
class Engine{ int power;   public：Engine (int  p)： power (p)  {   } };
class LandVehicle： virtual  public  Engine{
    int  speed;
public: //如从AmphibiousVehicle调LandVehicle，则此处不会调用Engine (p)
    LandVehicle (int  s,  int  p)：Engine (p),  speed (s) {  } //必须调用Engine (p)
};
class  WaterVehicle：public  virtual  Engine{
    int  speed;
public: //如从AmphibiousVehicle调WaterVehicle，则此处不会调用Engine (p)
    WaterVehicle (int s,  int p)：speed (s), Engine (p) {  } //必须调用Engine (p)
};
struct AmphibiousVehicle：LandVehicle,  WaterVehicle {
    AmphibiousVehicle (int  s1,  int  s2,  int  p) ： //先构造虚基类再基类
    WaterVehicle (s2,  p),  LandVehicle (s1,  p),  Engine (p) { } //必须调用
    //Engine (p)
     //在整个AmphibiousVehicle派生树中，Engine (p) 只执行1次
};  //初始化顺序： Engine (p) ,  LandVehicle (s1,  p) ,  WaterVehicle (s2,  p)
```

如果我们在最终派生类里面没调用 WaterVehicle 像这样子：

```
#include<bits/stdc++.h>
using namespace  std;
class Engine {
    int power;
public:
    Engine(int p) : power(p) { }
};

class LandVehicle : virtual public Engine {
    int speed;
public:
    LandVehicle(int s, int p) : Engine(p), speed(s) { }
};

class WaterVehicle : public virtual Engine {
    int speed;
public:
    WaterVehicle(int s, int p) : speed(s), Engine(p) { }
};

struct AmphibiousVehicle : LandVehicle {
    AmphibiousVehicle(int s1,  int p) : LandVehicle(s1, p), Engine(p) { }
};

AmphibiousVehicle a(10,10);
WaterVehicle b(10,5);
```

会导致虚基类重复构造

非虚类派生

派生类只能显示初始化

直接基类：是派生类直接继承的类。

间接基类：是通过中间的其他基类间接继承的类。

虚基类

虚基类的初始化则是每一级派生类的责任（在每一级的派生类里面都要初始化一次虚基类）

最终派生类：在实例化的时候截止的到的类 都是最终派生类 在一个最终派生类所在的继承链上只有他初始化虚基类 之前的类都没有初始化

```
class A{
    string msg;
public:
    A(string s):msg(s) { cout << "Constructor:" << msg << endl; }
};
class B:virtual public A{
    int b;
public:
    B(string s, int i):A(s),b(i){}
};
class C:virtual public A{
    int c;
public:
    C(string s, int j):A(s),c(j){}
};
class D:public B,public C{
    int d;
public:
    D(string s, int x, int y,int z):A(s),B(s,x),C(s,y),d(z){}
};
void main(int argc, char* argv[])
{
    B b("B", 1);     //B为最终派生类，由它调用虚基类A的构造函数
    C c("C", 2);     //C为最终派生类，由它调用虚基类A的构造函数
    D d("D",1,2,3); //D为最终派生类，由它调用虚基类A的构造函数，
            //B和C的构造函数不再调虚基类A的构造函数
}
```

同名：

当派生类有多个基类或虚基类时，不同基类或虚基类的成员之间可能出现同名；派生类和基类或虚基类的成员之间也可能出现同名。 出现上述同名问题时，必须通过面向对象的作用域解析，或者用类名加作用域运算符::指定要访问的成员，否则就会引起二义性问题。**虚基类>基类>派生类>派生类函数**

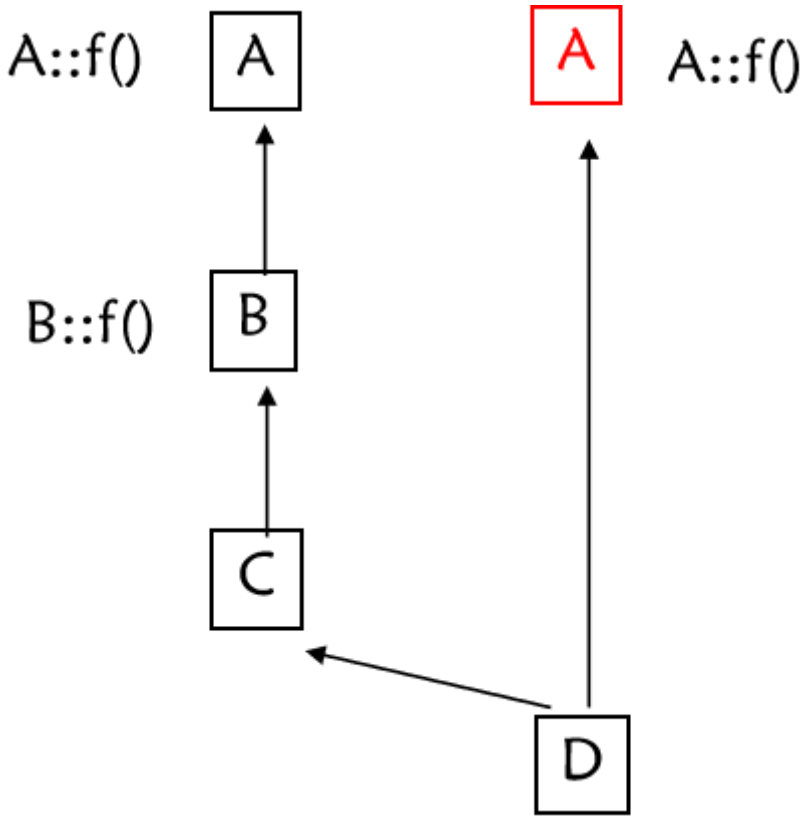
```
struct A{
    int a,  b,  c,  d;
};
struct B{
```

```
    int b,  c;
protected:
    int e;
};
class C: public A,  public B{
    int a;
public:
    int  b;    int f (int c)   ;
} x;
int C::f (int c)   {
    int i=a;            //访问C::a
    i=A::a;
    i=b+c+d;         //访问C::b和参数c
    i=A::b+B::b;  //访问基类成员
    return A::c;
}
void main (void)   {
    int  i=x.A::a;
    i=x.a;               //错,私有的
    i=x.b;               //访问C::b
    i=x.A::b+x.B::b;
    i=x.A::c;
}
```

下面这个例子中 f函数没有虚化 所以第一个pb指针没有多态性

```
struct A{
    void f() { cout << "A\n"; }
};
struct B:virtual A{
    void f() { cout << "B\n"; }
};
struct C:B{};
struct D:C,virtual A {};

void main(int argc, char* argv[])
{
    D d;
    B *pb = &d;      //B和D之间为父子关系
    D *pd = &d;

    pb->f();         //调用B::f(). B::f()被优先访问
    pd->f();     //调用B::f().pb为D类型指针,D分别从A和B继承了函      //数成员f，但优先访问基类B的f（基类比虚基类作用域小）
}
```

再看下面这个例子 D类虚基类了一个A又沿着继承链到了一个B和一个A 会产生二义性问题：




```
    void f() { cout << "A\n"; }
};
struct B:A{
    void f() { cout << "B\n"; }
};
struct C:B{};
struct D:C,virtual A {};

void main(int argc, char* argv[])
{
    D d;
    B *pb = &d;      //B和D之间为父子关系
    D *pd = &d;

    pb->f();         //OK,优先访问B::f()
    pd->f();     //有二义性，f可能是基类A的，可能是基类B的，从
                              //这里可以看出，对于D而言，A和B作用域大小一样，
            //都是基类作用域(左边分支)
            //改成pd->A::f();或pd->B::f();
}
```

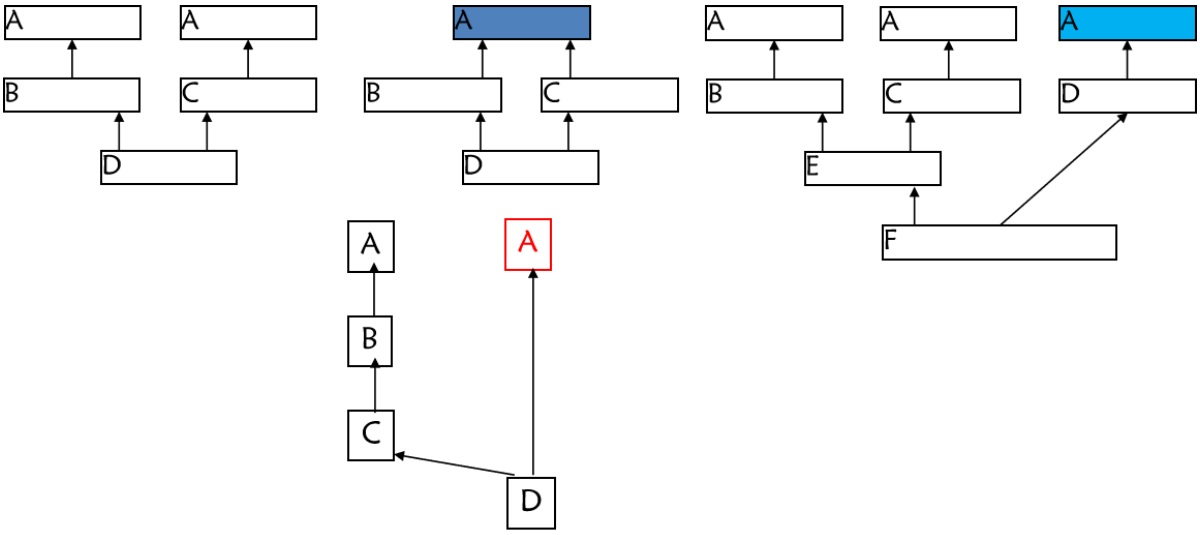
内存布局

一个类不能多次成为一个派生类的直接基类（即使是虚基类）否则会报编译错误

一个类可以多次成为一个派生类的间接基类（即使不是虚基类）

一个类（A）可以同时成为一个派生类的直接基类和间接基类，但该类（A）作为直接基类时必须是虚基类。否则会报警告错误

最后一条是为什么？因为如果直接基类不是虚基类 会出现多次定义基类



17

上面这个图 第一张图不是虚基类 第二个是虚基类 第三个是一个虚基类两个基类

构造顺序

析构和构造的顺序相反，派生类对象的构造顺序：

(1)按定义顺序构造倒派生树中所有虚基类；(2)按定义顺序构造派生类的所有直接基类；(3)按定义顺序构造 (初始化)派生类的所有数据成员，包括对象成员、const成员和引用成员；(4)执行派生类自身的构造函数体。

这个构造函数体是什么意思?

就是类似这个：

```
Derived(int x_val, int y_val, int z_val, int w_val)
      : A(x_val), B(y_val), z(z_val), w(w_val) { // 初始化列表
      cout << "Derived constructor" << endl;
  }
```

他会线构造变量成员 再去写括号里面的内容

如果构造中虚基类、基类、对象成员、const及引用成员又是派生类，则派生类对象重复上述构造过程，但同名虚基类对象在同一棵派生树中仅构造一次。

由派生类 (根)、基类和虚基类构成一个派生树的节点，而对象成员将成为一棵新派生树的根。

解释和什么是派生树?

1. 派生类、基类和虚基类构成的派生树

在 C++ 中，当类进行继承时，通常有两个概念：

- **基类**：是被继承的类。
- **派生类**：是从基类继承而来的类。

如果类之间存在多级继承或者虚继承的关系，就会形成一个继承树（继承关系的树状结构）。我们将这个树叫做 **派生树**。

- **根节点**：通常是最底层的基类，或者是最初的类。
- **内部节点**：派生类和基类之间的中间层级。

- **叶子节点**：最终的派生类。

2. 虚继承和虚基类

虚继承 是为了避免多重继承中的**菱形继承问题**（即同一基类被多次继承），通过虚继承，一个虚基类只会被初始化一次。

在虚继承的情况下，虚基类将被所有派生类共享，因此虚基类在派生树中位于**共同的根部**，并且确保其数据成员只有一个副本。

3. 对象成员成为新派生树的根

每个类的对象成员（即类中定义的成员变量）也可以是一个对象（即某个类型的实例）。这些成员变量如果是类类型的对象（而非基础类型），它们自己也会有构造函数和初始化过程，在构造派生类对象时，它们会被先行构造。

在这段话中，**对象成员成为一棵新派生树的根** 是指，当一个类的成员变量本身是另一个类的对象时，这个成员变量会被当作一个新的"子树"的根节点来处理。例如，如果一个类的成员变量是另一个类的对象，那么在构造该类对象时，会先构造这个成员变量（即该成员对象的构造函数会先被执行）。

例子：

假设我们有以下的类结构：

```
cpp复制代码#include <iostream>
using namespace std;

class Engine {
public:
    Engine() { cout << "Engine constructed\n"; }
};

class Car : virtual public Engine {
public:
    Car() { cout << "Car constructed\n"; }
};

class Boat : virtual public Engine {
public:
    Boat() { cout << "Boat constructed\n"; }
};

class AmphibiousVehicle : public Car, public Boat {
public:
    AmphibiousVehicle() { cout << "AmphibiousVehicle constructed\n"; }
};
```

分析：

1. **派生树的构成**：
    - **根节点**是 `Engine` 类（因为它没有基类）。
    - **派生树的分支**是 `Car` 和 `Boat`（它们继承了 `Engine`）。
    - **最底层的派生类**是 `AmphibiousVehicle`，它继承了 `Car` 和 `Boat`。

虚继承保证了 `Engine` 只会被构造一次。

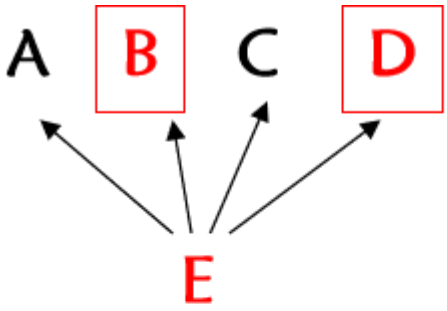
2. **对象成员成为新派生树的根**： 假设在 `AmphibiousVehicle` 中增加一个成员变量，比如 `Engine` 类型的对象成员：

```
cpp复制代码class AmphibiousVehicle : public Car, public Boat {
private:
    Engine engine;
public:
    AmphibiousVehicle() { cout << "AmphibiousVehicle constructed\n"; }
};
```

现在，`AmphibiousVehicle` 类包含一个 `Engine` 类型的成员 `engine`。当创建 `AmphibiousVehicle` 对象时，`engine` 会被首先初始化并构造（即它成为一棵新派生树的根），然后再构造 `Car` 和 `Boat` 类，以及 `AmphibiousVehicle` 自身。

ppt例子

```
#include <iostream.h>
struct A {
    A( ) { cout<<'A';}
};
struct B {
    B( ) { cout<<'B';}
};
struct C {
    int a;   int &b;
    const int c;
    C(char d): c(d), a(d) ,b(a)•    { cout<<d; }
};
struct D{
    D( ) { cout<<'D';}
};
struct E: A, virtual B, C, virtual D{
    A x, y;
    B z;
    E( ):z( ), y( ), C('C')
    {
        cout<<'E';
    }
};


void main(void)
{
    E  e;
}
```



输出：

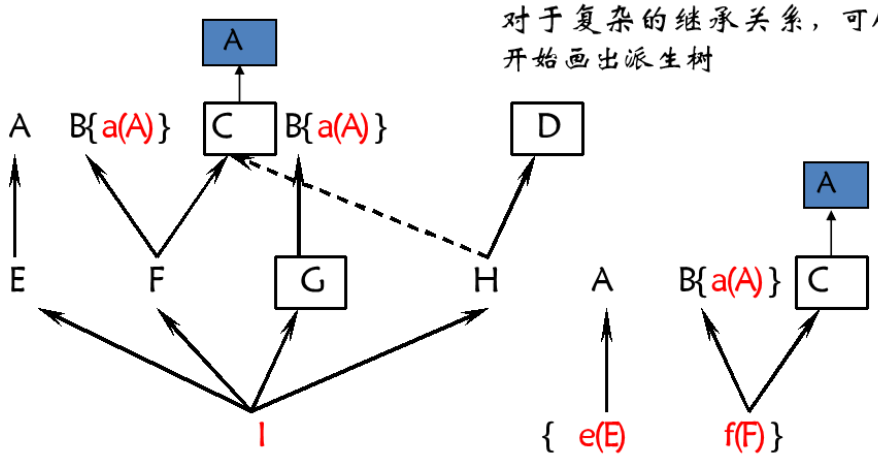
BDACAABE

B→D→A→C→x(A)→y(A)→z(B)→E

好好琢磨下面的例子！！！

```
#include <iostream.h>
struct A{ A ( )    { cout<<'A'; } };
struct B { const A a; B ( ) { cout<<‘B’; } }; //a作为新根 B( ):a( ){}
struct C: virtual A{ C ( )    { cout<<'C'; } };
struct D{ D ( )    { cout<<'D'; } };
struct E: A{ E ( )   { cout<<‘E’; } }; //等价E( ):A( ){ … }
struct F: B,  virtual C{ F ( ) { cout<<'F'; } };
struct G: B{ G ( ):B( )   { cout<<'G'; } };
struct H: virtual C,  virtual D{ H ( ) { cout<<'H'; } };
struct I: E,  F,  virtual G,  H{
    E  e;   F  f; //对象成员e、f作为新根
    I ( ) { cout<<'I'; } };
void main (void)    { I  i;  }
```

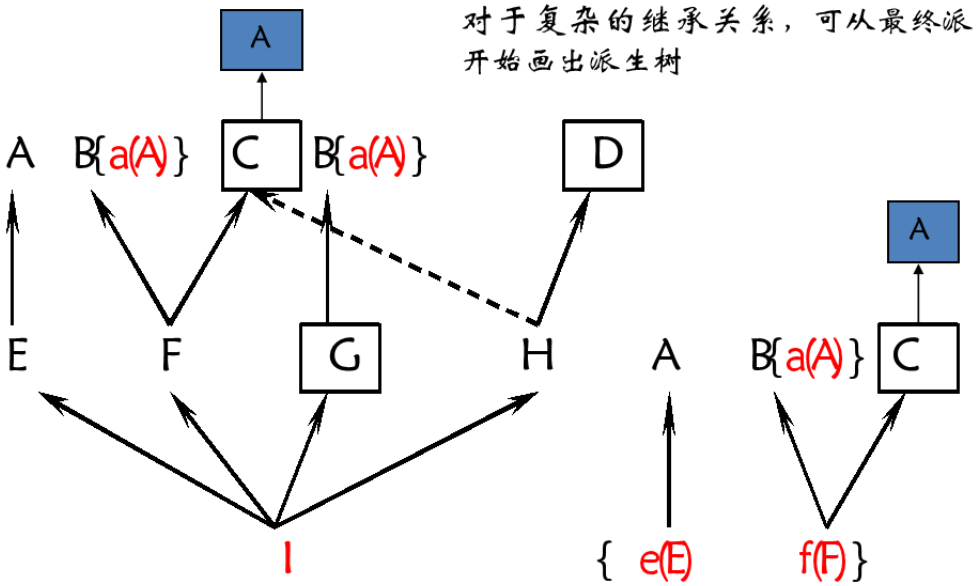
对于复杂的继承关系，可从最终派生类开始画出派生树



首先确定四个虚基类的构造顺序。从派生类I倒推：
要构造I，必须按定义顺序依次构造E、F、G、H。
1：E没有虚基类不予考虑；
2：构造F：按派生顺序依次构造虚基类A、C，对应输出为AC；
3：构造虚基类G：先构造G的基类B。构造类B必须先构造A类成员a，再调用B的构造函数体。因此对应输出为ABG；
4：构造H，必须先构造虚基类D，对应输出为D。
因此，四个虚基类构造完毕后，输出为ACABGD

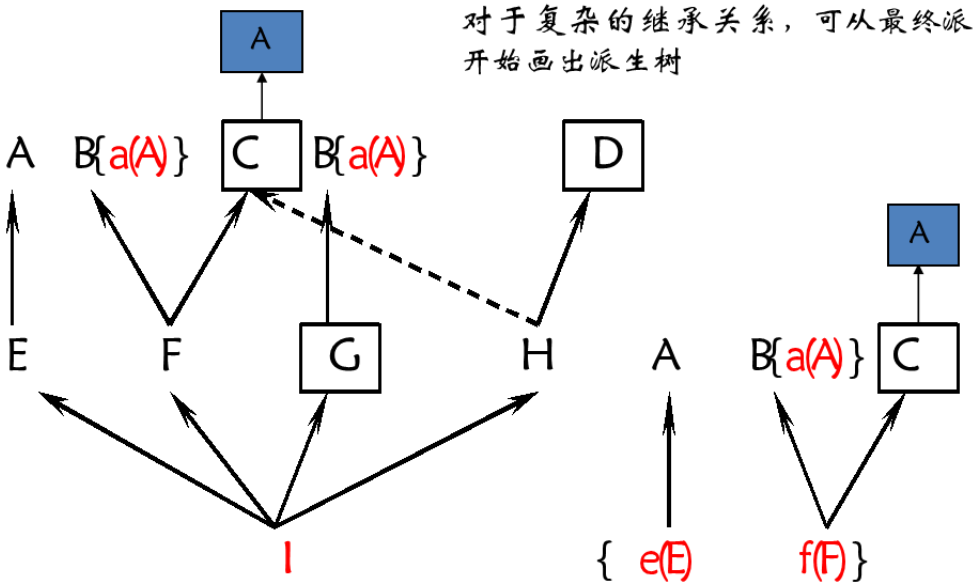
按定义顺序自左至右、自下而上地构造倒派生树中所有虚基类；

对于复杂的继承关系，可从最终派生类开始画出派生树



接着构造I的基类
1：构造E: 必须先构造A，因此对应输出为AE
2：构造F：由于虚基类A、C已构造好，只需要先构造B。前面已分析构造B输出AB，因此基类F构造好时的输出为ABF
3：构造H：输出H
因此当I的基类构造完毕，输出为AE ABF H。

对于复杂的继承关系，可从最终派生类开始画出派生树



接着构造I的对象成员

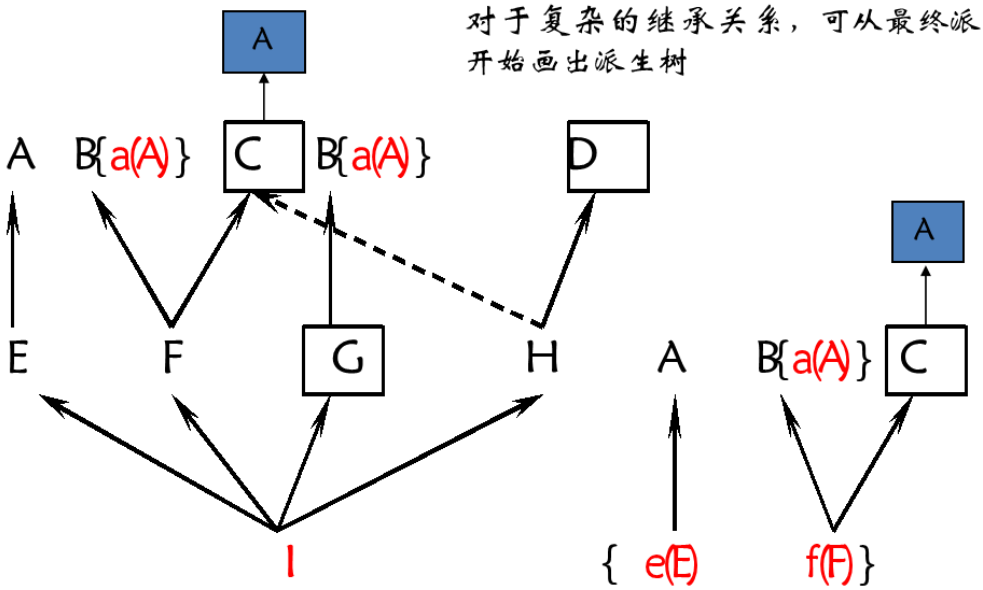
1：构造e (E)：必须先构造A，因此对应输出为AE

2：构造f (F)：首先按派生顺序依次构造虚基类A、C；再构造B（先构造B的对象成员a，再调用B的构造函数体）；最后构造F。因此对应输出为ACABF

因此当I的对象成员构造完毕，输出为AE ACABF。

最后调用I自己的构造函数，输出I

对于复杂的继承关系，可从最终派生类开始画出派生树



六棵派生树（根红色），输出：

ACABGD AE ABF H AE ACABF I