

The background image is a scenic view of a traditional Chinese pagoda, likely the Yellow Crane Tower in Wuhan, China. The pagoda is illuminated with warm lights, and its multiple tiers and ornate roof are clearly visible. In the background, a city skyline is visible across a body of water, with a long bridge spanning the water. The sky is a mix of orange and blue, suggesting a sunset or sunrise. The overall atmosphere is serene and majestic.

C++程序设计精要教程

华中科技大学

第4章 C++的类

◆4.1 类的声明及定义

类保留字：class、struct或union可用来声明和定义类。

类的声明：由保留字class、struct或union加上类的名称构成。

类的定义：包括类名的声明部分和由{ }括起来的主体两部分构成。

类的实现：通常指类的函数成员的实现：即定义类的函数成员的函数体。

```
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;
};
```

示例

- 下面是一个圆类：

类的定义

数据
成员

构造函数
声明

函数成员
声明

```
class Circle {
```

Circle.h

```
private:
```

```
double radius ;
```

```
public:
```

```
Circle() ;
```

```
Circle(double r) ;
```

```
double findArea() ;
```

```
};
```

Circle:c1
radius=1.0

Circle:c2
radius=10.0

Circle:c3
radius=15.0

类的实现

构造函数
定义

函数成员
定义

```
#include "Circle.h"
```

```
Circle::Circle() { radius = 1.0;}
```

```
Circle::Circle(double r) { radius = r;}
```

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

Circle.cpp

```
Circle c1(10.0);  
int i(0;) //C++ Style  
int i = 0; //C Style
```

第4章 C++的类

◆4.1 类的声明及定义

定义类时应注意的问题：

- 使用private、protected和public保留字标识主体中每一区间的访问权限，同一保留字可以多次出现；
- 同一区间内可以有数据成员、函数成员和类型成员，习惯上按类型成员、数据成员和函数成员分开；
- 成员在类定义体中出现的顺序可以任意，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**）。
- 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，应使用类名和作用域运算符“::”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。

第4章 C++的类

◆4.1 类的声明及定义

定义类时应注意的问题：

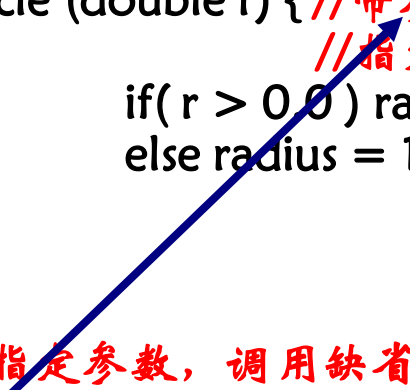
- 在类定义体中允许对数据成员定义默认值，若在构造函数的参数列表后面的“:”和函数体的“{”之间对其进行了初始化（在成员初始化列表进行初始化），则默认值无效，否则用默认值初始化；
- 构造函数和析构函数都不能定义返回类型。
- 如果类没有自定义的构造函数和析构函数，则C++为类生成默认的构造函数（不用给实参的构造函数）和析构函数。
- 构造函数的参数表可以出现参数，因此可以重载。

构造函数

构造函数：用来产生对象，为对象申请资源，初始化数据成员。

```
class Circle{
public:
    double radius;
public:
    Circle() { radius = 1.0;} // 缺省构造函数，将半径设为1.0
    Circle (double r) { // 带参数构造函数，用户创建Circle对象时可以
                        // 指定圆的半径r
                        if( r > 0.0 ) radius = r;
                        else radius = 1.0;
    }
};
```

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.radius; // 1.0
cout << c2.radius; // 5.0



第4章 C++的类

◆4.1 类的声明及定义

定义类时应注意的问题:

- 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。
- 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。

第4章 C++的类

◆4.1 类的声明及定义

定义类时应注意的问题:

- 构造函数用来为对象申请各种资源，并初始化对象的数据成员。构造函数有隐含参数this,可以在参数表定义若干参数，用于初始化数据成员。
- 析构函数是用来毁灭对象的，析构过程是构造过程的逆过程。析构函数释放对象申请的所有资源。
- 析构函数既能被显式调用，也能被隐式（自动）调用。由于只有一个固定类型的this，故不可能重载，只能有一个析构函数。
- 若实例数据成员有指针指向malloc/new的内存，应当防止反复析构（用指针是否为空做标志）。
- 联合也是类，可定义构造、析构以及其它函数成员。

第4章 C++的类

例4.1】定义字符串类型和字符串对象。

```
#include <alloc.h>

#include <string.h> //如果函数同名,可用单目::访问string.h里的全局函数

//在windows下头文件名是不分大小写的,因此最好改成MYSTRING(相应的头文件为MYSTRING.h,这样就不会和运行库
//的include <string.h>混淆)

struct MYSTRING { //struct, union里面成员如果没有访问权限限定,默认为public, class正好相反,默认权限为
private
    typedef char * CHARPTR;           //定义类型成员
    CHARPTR s;                       //定义数据成员,等价于char *s;
    int strlen();                     //声明函数成员,求字符串对象的长读(有隐含this)
    MYSTRING(const char *);           //声明构造函数,无返回类型,有隐含this,注意必须是const char *,后面有解释
    ~MYSTRING();                      //声明析构函数,有隐含this
};

//必须用MYSTRING ::限制strlen,

int MYSTRING ::strlen() {             //用运算符::在类体外定义
    int k;
    for(k=0; s[k]!=0; k++);
    return k;
}
```

第4章 C++的类

MYSTRING :: MYSTRING(const char *t){//在类体外定义构造函数, 必须加类名::限定, 无返回类型

int k;

for(k =0; t[k]!=0; k++);

s=(char *)malloc(k+1); //s等价于this->s

for(k=0; (s[k]=t[k])!=0; k++);

}

MYSTRING ::~ MYSTRING() {//在类体外定义析构函数, 必须加类名::限定, 无返回类型

free(s);

}

struct MYSTRING x("simple"); //struct可以省略, 程序结束时自动调用x的析构函数

void main(){

MYSTRING y("complex"), *z=&y;

int m=y.strlen(); //当前对象包含的字符串的长度

m=z->strlen();

} //返回时自动调用y的析构函数

第4章 C++的类

例4.1】 定义字符串类型和字符串对象。

```
#include <stdlib.h>

#include <string.h> //如果函数同名,可用单目::访问string.h里的全局函数

struct MYSTRING {
    typedef char * CHARPTR;           //定义类型成员
    CHARPTR s;                        //定义数据成员, 等价于char *s;
    int strlen();                     //声明函数成员, 求字符串对象的的长读(有隐含this)
    MYSTRING(const char *);           //声明构造函数, 无返回类型, 有隐含this
    ~ MYSTRING();                     //声明析构函数, 有this
};

//另外一种实现, 必须用MYSTRING ::限制strlen, 否则递归

int MYSTRING ::strlen() {             //用运算符::在类体外定义
    return ::strlen (s);
}
```

不同对象的构造和析构

构造函数在自动变量(对象)被实例化或通过new产生对象时被自动调用一次,是**唯一不能被显式调用的函数成员**。

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)。由于**析构函数可被显式反复调用**,而有些资源是不能反复析构的,例如不能反复关闭文件,因此,必要时要防止对象反复释放资源。

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

不同对象的构造和析构

//全局对象,mian函数之前被开工函数构造，由收工函数自动析构，全局对象在数据段里

MYSTRING gs(“Global String”);

void f(){

//局部自动变量,函数返回后自动析构，局部对象在堆栈里

MYSTRING x(“Auto local variable”);

//静态局部变量，由收工函数自动析构,静态局部对象在数据段里

static **MYSTRING** y(“Static local variable”);

//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

}

C++11新标准中出现了智能指针的概念，就是要减轻程序员的必须承担的手动delete动态分配的内存（new出来的东西）的负担，后面会介绍。

析构函数要防止反复释放资源

```
#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 () {
    cout<<"Deconstruct:"<<s;
    free (s) ;
}
```

```
void main (void) {
    MYSTRING s1 ("String 1\n") ;
    s1.~ MYSTRING (); //显式析构s1, 会free(s)
} //自动析构s1,会导致s1的s指针指向内存又释放, 会出现运行时错误
```

构造s1时, 传进去的是字符串字面量“String 1\n”, 类型被编译器解释为const char *, const char *是不能传给构造函数的参数char *的 (课堂上讲过, int * = const int *是不成立的, 换成char类型是一样的) 因此构造函数参数必须是const char *

最先定义的自动对象最后**自动析构**。可**随时**手动调用析构函数, 但要防止反复释放资源。

析构函数要防止反复释放资源

```
#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
} //自动析构s2, s1, 不会出错
```

显示内容:

```
Construct:String1
Construct:String2
Construct:Constant
Deconstruct:Constant
RETURN
Deconstruct:String2
Deconstruct:String1 //是显示调用s1.~ MYSTRING ()的输出
//s1的自动析构没有输出, 因为s==0
```

最先定义的自动对象最后自动析构。可随时手动调用析构函数，但要防止反复释放资源。

第4章 C++的类

◆4.1 类的声明及定义

接受与删除编译自动生成的函数：default接受, delete：删除。

例4.4使用delete禁止构造函数以及default接受构造函数。

```
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();”
```

第4章 C++的类

◆4.2 成员访问权限及其访问

- 封装机制规定了数据成员、函数成员和类型成员的访问权限。包括三类：
 - 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。
- 类的友元不受这些限制，可以访问类的所有成员。可以在private、protected和public等任意位置说明，友元可以像类自己的函数成员一样访问类的所有成员。另外，通过强制类型转换可突破访问权限的限制。
- 构造函数和析构函数可以定义为任何访问权限。不能访问构造函数则无法用其初始化对象。
- 进入class定义的类时，缺省访问权限为private；进入struct和union定义的类时，缺省访问权限为public

第4章 C++的类

【例4.5】为女性定义FEMALE类。

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

第4章 C++的类

FEMALE::NAME FEMALE::getnickname() { // 类型成员 **FEMALE::NAME** 是公有的，可以到处访问（使用）

return nickname; // 自己的函数成员访问自己的成员

}

void main(void) { // main 没有定义为类 FEMALE 的友员

FEMALE w;

FEMALE::NAME(FEMALE::*f)(); // 实例成员函数指针 f

FEMALE::NAME n;

n=w.name; // 任何函数都能访问公有 name

n=w.nickname; // 错误，main 不得访问保护成员

n=w.getnickname(); // 错误，main 不得调用保护成员

int d=w.age; // 错误，main 不得访问私有 age

f=&FEMALE::getnickname; // 错误，不得取保护成员地址

}

第4章 C++的类

◆4.3 内联、匿名类及位段

●函数成员的内联说明：

- 在类体内定义的任何函数成员都会自动内联。

- 如果在类外实现函数，在类内或类外使用inline保留字说明函数成员。

- 内联失败：有分支类语句、定义在使用后，取函数地址，定义(纯)虚函数。

- 内联函数成员的作用域局限于当前代码文件。

- 匿名类函数成员只能在类体内定义(内联)。

- 函数局部类的函数成员只能在类体内定义(内联)，某些编译器不支持局部类。

内联

```
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
```

第4章 C++的类

◆4.4 new和delete

●内存管理的区别：

- C：函数malloc、free； C++：运算符new、delete。
- 内存分配：malloc为函数，参数为值表达式，分配后内存初始化为0； new为运算符，操作数为类型表达式，**先底层调用malloc，然后调用构造函数**；
- 用“new 类型表达式 {}”可使分配的内存清零，若“{}”中有数值可用于初始化。
- 内存释放：free为函数，参数为指针类型值表达式，直接释放内存； delete为运算符，操作数为指针类型值表达式，**先调用析构函数，然后底层调用free**。
- 如为简单类型(没有构造、析构函数)分配和释放内存，则new和malloc、 delete和free没有区别，可混合使用：比如new分配的内存用free释放。
- 无法用malloc代替new初始化
- 注意delete的形参类型应为const void*，因为它可接受任意指针实参。

第4章 C++的类

◆4.4 new和delete

●new <类型表达式> //后接()或{}用于初始化或构造。{}可用于数组元素

- 类型表达式: `int *p=new int;` //等价`int *p=new int(0);`
- 数组形式仅第一维下标可为任意表达式, 其它维为常量表达式: `int (*q)[6][8]=new int[x+20][6][8];`
- 为对象数组分配内存时, 必须调用默认构造函数, 如`Circle *p =new Circle[10]`, 如`Circle`无默认构造函数如何? 编译器报错。因此保证一个类有默认构造函数非常重要
- 注意`int *p=new int (10);`和`int *p=new int [10];`的区别

●delete <指针>

- 指针指向非数组的单个实体: `delete p;`可能调析构造函数。

●delete []<数组指针>

- 指针指向任意维的数组时: `delete []q;`
- 如为对象数组, 对所有对象(元素)调用析构造函数, 然后释放对象数组占有的内存。
- 若数组元素为简单类型, 则可用`delete <指针>`代替。

第4章 C++的类

【例4.13】 定义二维整型动态数组的类。

```
#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
}
```

第4章 C++的类

```
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

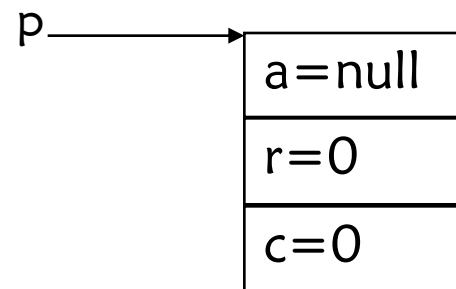
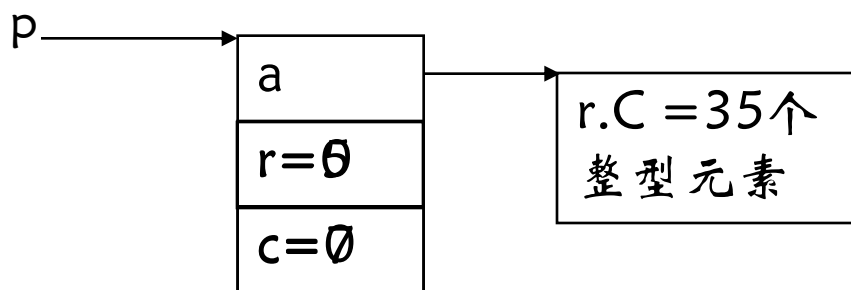
void main(void){
    ARRAY y(3, 5), *p; //退出main时析构y
    p=new ARRAY(5, 7); //不能用malloc, ARRAY有构造函数
    delete p; //不能用free, 否则未调用析构函数
} //退出main时, y被自动析构

//程序结束时, 收工函数析构全局对象x
```

两种内存管理方式的区别

(a) `p = new Array(5,7)`

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



Step 1:

为对象分配内存

`p = (Array *)`

`malloc(sizeof(Array))`

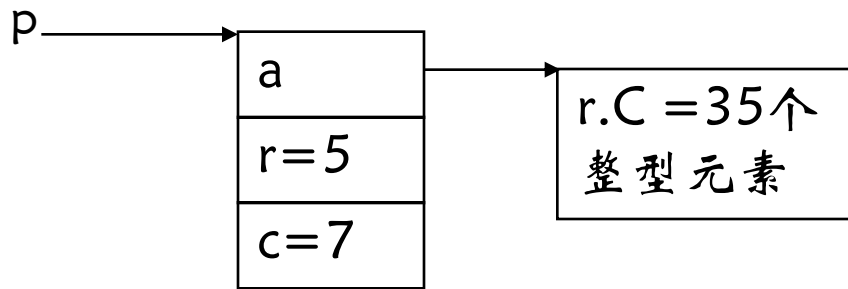
Step 2:

调用构造函数

`a = new int[r * c]`

两种内存管理方式的区别

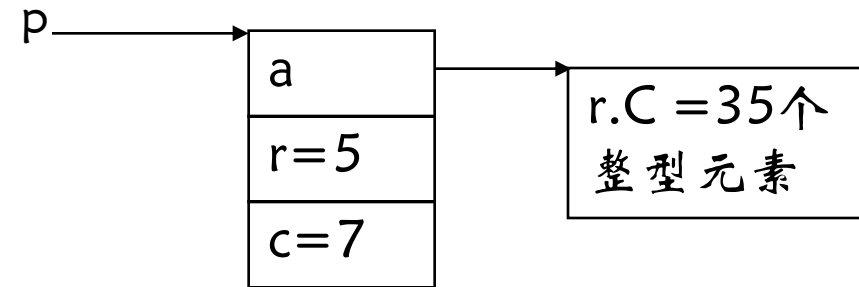
(d) delete p



Step 2:
释放Array对象内存
`free(p)`

Step 1:
先调析构函数
`delete[] a`, 释放
`a`
指向的内存

(b) free(p)



直接释放Array对象内存,
没有调析构函数, 导致`a`指
向的35个整型元素所占内存
没释放

第4章 C++的类

◆4.4 new和delete

- new还可以对已经析构的变量重新构造。可以减少对象的说明个数，提高内存的使用效率。(不是所有C++编译器都支持)

```
STRING x ("Hello!"), *p=&x;
```

```
x. ~STRING ();
```

```
new (&x) STRING ("The World");
```

```
new (p) STRING ("The World");
```

- 这种用法可以节省内存或栈的空间。

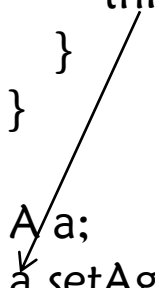
第4章 C++的类

◆4.5 隐含参数this

- this指针是一个特殊的指针，它是非静态函数成员（包括构造和析构）隐含的第一个参数，其类型是指向要调用该函数成员的对象const指针。（A* const this）
- 当通过对象调用函数成员时，对象的地址作为函数的第一个实参首先压栈，通过这种方式将对象地址传递给隐含参数this。
- 构造函数和析构函数的this参数类型固定。例如A::~~A()的this参数类型为A*const this; //析构函数的this指向可写对象，但this本身是只读的
- 注意：可用*this来引用或访问调用该函数成员的普通、const或volatile对象；类的静态函数成员没有隐含的this指针；this指针不允许移动。

隐含参数this指针

```
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

第4章 C++的类

【例4.16】在二叉树中查找节点。

```
#include <iostream.h>
```

```
class TREE{
```

```
    int value;
```

```
    TREE *left, *right;
```

```
public:
```

```
    TREE (int); //this类型: TREE * const this
```

```
    ~TREE(); //this类型: TREE * const this, 析构函数不能重载
```

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

```
};
```

```
TREE::TREE(int value){
```

```
    this->value=value;
```

```
    left=right=0;
```

```
}
```

```
//隐含参数this指向要构造的对象
```

```
//等价于TREE::value=value
```

```
//C++提倡空指针NULL用0表示
```

第4章 C++的类

```
TREE::~~TREE(){ //this指向要析构的对象
    if(left) { delete left; left=0; }
    if(right) { delete right; right=0; }
}
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 *
TREE root(5); //收工函数将析构对象root
void main(void){
    if(root.find(4)) cout<< "Found\n" ;
}
//find函数最后加了const有什么区别？在find函数里，*this是不能出现在=左边的
```

第4章 C++的类

◆4.6 对象的构造与析构

●类的非静态数据成员可以在如下位置初始化：

- ✓ 类体中进行就地初始化（称为就地初始化）（=或{}都可以），但成员初始化列表的效果优于就地初始化。
- ✓ 成员初始化列表初始化
- ✓ 可以同时就地初始化和在成员初始化列表里初始化，当二者同时使用时，并不冲突，初始化列表发生在就地初始化之后，即最终的初始化结果以初始化列表为准。
- ✓ 在构造函数体内严格来说不叫初始化，叫赋值

非静态数据成员初始化

类内就地初始化

```
class Person {  
private:  
    int id;  
public:  
    Person(int id){ this->id = id; }  
};
```

```
class A {  
public:  
    int i = 10;           //可以用=号初始化  
    double d{ 3.14 };     //可以用{}初始化  
    int j = { 0 };        //可以用 = {}来初始化  
    Person p{ 100 }; //没有默认构造函数的类成员也可以就地初始化而不用出现在成员初始化列表里  
    //int k(0);           //不能用()来初始化  
    //Person p(100);       //不能用()初始化  
} a;
```

非静态数据成员初始化

//可以同时就地初始化和在成员初始化列表里初始化，当二者同时使用时，即最终的初始化结果以初始化列表为准（即使是引用和const成员）。

```
class B {  
public:  
    int i = 0;           //就地初始化  
    int &r = i;           //引用也可以就地初始化而不用出现在成员初始化列表里  
    int j;  
    const double PI = 3.14; //const成员也可以就地初始化而不用出现在成员初始化列表里  
public:  
    B() :i(10),j(100),r(j),PI(6.28) {} // 成员初始化列表初始化,这时  
} b;  
  
void testB() {  
    cout << b.i << " " << b.r << " " << b.PI << endl;  
    //输出: 10 100 6.28 , 说明r绑定到j, const常量PI的值为6.28  
}
```

非静态数据成员初始化

为什么不能用 () 进行类内初始化

//不同类型 (指变量类型、类类型、函数类型) 的标识符可以同名

```
int C = 7;           //定义整型变量C
```

```
struct C {           //定义classC
```

```
    C(int i) {}
```

```
};
```

//但是同类型 (指变量类型、类类型、函数类型) 的标识符不能同名

//int C = 9; //不能定义二个变量都叫C

//当要使用类型C时, 必须加class来修饰, 让编译器知道是用class C

```
class C o{C};         //定义一个C类型的对象o, 构造函数的实参是变量C
```

```
struct D {
```

//定义一个C类型的数据成员x, 并用变量C作为构造函数实参。若可用()来就地初始化, 如下

//class C x(C); //这时编译器很难区分这不是不是一个成员函数声明: 函数名x, 形参和返回为C类型

```
};
```

//顺便说明, 如果要声明一个函数成员: 函数名x, 形参类型和返回为类C类型, 应该写成

```
class C x(class C);
```

类内就地初始化

非静态数据成员初始化

成员初始化列表初始化

```
class A {  
    int i;  
    int j;  
public:  
    A():j(0),i(0) { //还是先初始化i, 后初始化j。初始化按定义顺序  
        i = 1; j = 1; //在函数体内不应看做初始化, 而是赋值  
    }  
};
```

成员按照在类体内定义的先后次序初始化, 与出现在初始化位置列表的次序无关;

第4章 C++的类

◆4.6 对象的构造与析构

- 类可能会定义只读(const)和引用类型的非静态(static)数据成员，在使用它们之前必须初始化；若这二种成员没有类内就地初始化，该类必须定义构造函数，在成员初始化列表初始化这类成员。在构造函数里是赋值，因此这二种成员没法在构造函数里初始化
- 类A还可能定义类B类型的非静态对象成员，若B类型对象成员必须用带参数的构造函数构造，且该成员没有类内就地初始化，则A必须自定义初始化构造函数，在成员初始化列表初始化这类成员(自定义的类A的构造函数，传递实参初始化类B的非静态对象成员：缺省的无参的A()自动只调用无参的B())。

非静态数据成员初始化

const和引用成员初始化

```
class WithReferenceAndConstMembers {  
public:  
  
    int i = 0;  
    int &r;  
    const double PI;  
  
} ws;
```

//就地初始化

//引用成员，没有就地初始化

//const成员，没有就地初始化

编译报错：error C2280: “WithReferenceAndConstMembers(void)”：尝试引用已删除的函数（意思是编译器无法提供默认构造函数。为什么？包含了const或引用成员，且没有就地初始化）

```
class WithReferenceAndConstMembers {  
public:  
  
    int i = 0;  
    int &r;  
    const double PI;  
    WithReferenceAndConstMembers() :r(i), PI(3.14) {}  
  
} ws;
```

//就地初始化

//引用成员，没有就地初始化

//没有就地初始化

WithReferenceAndConstMembers() :r(i), PI(3.14) {} //这时r和PI必须在成员初始化列表初始化

这时必须自己定义构造函数，没有自定义构造函数何来成员初始化列表。构造函数是否带参数取决于程序员的选择。关键是需要一个成员初始化列表

非静态数据成员初始化

对象成员初始化

```
class Person {  
private:  
    int id;  
  
public:  
    Person(int id){ this->id = id; } // 注意Person没有默认构造函数，实例化对象必须带参数  
};  
class WithPersonObject {  
public:  
    Person p; //没有就地初始化，如Person p{1};  
} wpo;
```

编译报错error C2280: “non_static_member_initialize::WithPersonObject::WithPersonObject(void)”：尝试引用已删除的函数。
因为数据成员 “WithPersonObject::p” 没有适合的默认构造函数或重载决策不明确，所以已隐式删除函数

```
class WithPersonObject {  
public:  
    Person p;  
    //这时p必须在成员初始化列表初始化  
    WithPersonObject() :p(10) {}  
} wpo;
```

这时必须自己定义构造函数，没有自定义构造函数何来成员初始化列表。构造函数是否带参数取决于程序员的选择。关键是需要一个成员初始化列表

第4章 C++的类

◆4.6 对象的构造与析构

- 构造函数的成员初始化列表在参数表的“:”后, {}之前. 所有数据成员建议在此初始化, 未在成员初始化列表初始化的成员也可类内就地初始化, 既未在成员初始化列表初始化, 也未类内就地初始化的非只读、非引用、非对象成员的值根据被构造对象的存储位置可取随机值 (对象是局部变量, 栈段) 或0及nullptr值 (对象是全局变量或静态变量, 数据段)。
- 按定义顺序初始化或构造数据成员 (大部分编译支持)。

第4章 C++的类

◆4.6 对象的构造与析构

- 如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）。联合仅需初始化第一个成员。
- 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。
- 单个参数的构造函数能自动转换单个实参值成为对象
- 若类未自定义构造函数，编译器会尝试自动生成构造函数。
- 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。
- 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。

合成的默认构造函数

合成的默认构造函数：当没有自定义类的构造函数时，编译器会提供一个默认构造函数，称为**合成的默认构造函数**。合成的默认构造函数会按如下规则工作：

- ◆如果为数据成员提供了类内初始值，则按类内初始值进行初始化

- ◆否则，执行默认初始化

但是有些场合下默认初始化失败，导致合成的默认构造函数工作失败。

合成的默认构造函数

合成的默认构造函数会工作是否成功，分以下情况

1) 如果为数据成员提供了类内初始值，则工作成功；

2) 如果没有为数据成员提供类内初始值，则：

- ◆ A: 如果只包含非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，则这些成员被默认初始化为0，如果类的对象是局部的，如果程序要访问这些成员，则编译器报错；合成的默认构造函数工作失败；如果对象是new出来的（在堆里），访问这些成员编译器不报错，但值为随机值；
- ◆ B: 如果包含了const、引用数据成员，一旦实例化对象，编译器会报错；合成的默认构造函数工作失败；
- ◆ C: 如果包含了其他class类型成员且这个类型没有默认构造函数，那么编译器无法默认初始化该成员；编译器会报错；合成的默认构造函数工作失败；

任何情况下，只要合成的默认构造函数工作失败，编译器会报错。

合成的默认构造函数

//如果非引用，非const的内置数据成员没有进行任何形式的初始化 情况2) A

```
class A {  
public:
```

```
    int i;
```

```
} e;      //全局对象，在数据段
```

```
void testA() {
```

```
    static A se;      //局部静态对象，在数据段
```

```
    A o;      //局部对象，在堆栈段，只要不访问对象成员，编译器不报错
```

```
    cout << e.i << endl;      //对e.i执行了默认初始化，为0
```

```
    cout << se.i << endl;      //对se.i执行了默认初始化，为0
```

```
    //cout << o.i << endl;      //一旦访问局部对象的未初始化数据成员，编译器就报错：使用了未初始化的局部变量“o”。因为o.i初始化失败
```

```
    A *p = new A; //p指向堆里new出来的对象
```

```
    cout << p->i << endl; //编译器不报错，但i的值是一个随机值-842150451
```

```
    delete p;
```

```
}
```

合成的默认构造函数

情况2) B

```
class B {  
public:  
    int i = 0;  
    int &ri;  
    const int c;  
}; //只要不实例化对象，编译器不报错
```

B ob; //只要实例化对象，编译器立刻报错

编译器报错信息 (VS2017) :

error C2280: “synthesized_default_constructor::B::B(void)” : 尝试引用已删除的函数

note: 编译器已在此处生成 “synthesized_default_constructor::B::B”

note: “synthesized_default_constructor::B::B(void)” : 因为

“synthesized_default_constructor::B” 有一个未初始化的常量限定的数据成员

“synthesized_default_constructor::B::c” , 所以已隐式删除函数

note: 参见 “synthesized_default_constructor::B::c” 的声明

合成的默认构造函数

情况2) C

//Person自定义构造函数，因此编译器不再提供合成的构造函数

//更糟糕的是Person自定义的构造函数是带参数的，这就意味着Person类无法默认初始化，即该类对象没有默认值

```
class Person {  
public:  
    int i;  
public:  
    Person(int i) { this->i = i; }  
};
```

```
class C {  
public:  
    Person p; //包含一个Person类型成员，且这个类没有默认构造函数  
};
```

C oc; //只要实例化对象，编译器立刻报错，报错信息如下 (VS2017)

error C2280: “synthesized_default_constructor::C::C(void)”：尝试引用已删除的函数

note: 编译器已在此处生成 “synthesized_default_constructor::C::C”

note: “synthesized_default_constructor::C::C(void)”：因为 数据成员 “synthesized_default_constructor::C::p” 没有适合的 默认构造函数 或重载决策不明确，所以已隐式删除函数

note: 参见 “synthesized_default_constructor::C::p” 的声明

合成的默认构造函数

任何情况下，只要合成的默认构造函数工作失败，编译器会报错。工作失败的原因就是有成员没有初始值，而且也不能默认初始化该成员。因此，我们一定要保证类的所有数据成员或者有初始值，或者能默认初始化。

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。

=default

当没有自定义类的构造函数时，编译器会提供一个合成的默认构造函数。但自定义了构造函数后，能否要求编译器仍然提供合成的默认构造函数呢？可以，使用= default

=default

```
class PersonWithoutDefaultConstructor {
public:
    int i = 100;        //就地初始化
public:
    //自定义带参数构造函数，因此没有默认构造函数
    PersonWithoutDefaultConstructor(int i) { this->i = i; }
};
//PersonWithoutDefaultConstructor p; //编译报错：没有合适的默认构造函数可用

class PersonWithDefaultConstructor {
public:
    int i = 100; //就地初始化
public:
    //自定义带参数构造函数
    PersonWithDefaultConstructor(int i) { this->i = i; }

    //要求编译器提供合成的默认构造函数
    PersonWithDefaultConstructor() = default; // =default 出现在类内部，这时
    默认构造函数是内联的
};
PersonWithDefaultConstructor p; //这时p可以执行默认构造了
```

=default

```
class PersonWithDefaultConstructor {  
public:  
    int i = 100;        //就地初始化  
public:  
    //自定义带参数构造函数  
    PersonWithDefaultConstructor(int i) { this->i = i; }  
  
    PersonWithDefaultConstructor();  
};  
PersonWithDefaultConstructor::PersonWithDefaultConstructor() = default;  
//=default也可以作为实现出现在类外部，这时默认构造函数不是内联的
```

隐式的类型转换和显式构造函数

如果构造函数只接受一个实参时，实际上它定义了转换为此类类型的转换机制，这种构造函数称为**转换构造函数**。如果想抑制这种隐式转换，必须将转换构造函数声明为explicit。

explicit只对**接受一个实参**的构造函数有效，需要多个实参的构造函数不能用于执行隐式转换。

只能在类内声明构造函数时使用explicit关键字

隐式的类型转换和显式构造函数

```
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) {}  
};
```

```
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  
}
```

隐式的类型转换和显式构造函数

```
//有时想抑制这种隐式转换，这时必须把构造函数  
//声明为explicit  
class ExplicitInteger {  
public:  
    int value;  
public:  
    explicit ExplicitInteger(int value) : value(value) {}  
};
```

```
class ExplicitAge {  
public:  
    Integer age{ 0 };  
public:  
    Explicit Age(Integer i) :age(i) {}  
    Explicit Age() = default;  
};
```

```
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
```

隐式的类型转换和显式构造函数

```
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
```

第4章 C++的类 (课堂略)

【例4.17】 包含只读、引用及对象成员的类。

```
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
```

第4章 C++的类 (课堂略)

```
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
}                       //故(7,8)为C的扩号表达式, (7,8)=8
```

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);

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

拷贝构造函数

```
class A{  
public:  
    A() = default;  
};
```

```
const A o1;
```

```
A o2(o1);
```

//编译通过，说明A o2(o1)调用的合成的默认拷贝构造函数A(const A &)。因为：若合成的默认拷贝构造函数是A(A &)，实参o1是无法传给A &形参的，因A &是无法引用const A的

```
class A{
```

```
public:
```

```
    A() = default;
```

```
    A(A &o) { cout << "A copied" << endl;}    //自己定义的拷贝构造函数
```

```
};
```

```
const A o1;
```

A o2(o1); // 编译报错，A &o = o1;错误。因为自定义了拷贝构造函数，编译器不再提供合成的默认构造函数

拷贝构造函数

什么时候会调用拷贝构造函数？

◆ 用一个已经存在的对象去构造另外一个对象，包括如下形式：

A o1;

A o2(o1);

A o3 = o1;

A o4{o1};

A o5 = {o1};

◆ 拷贝构造函数更多的用在函数传递值参和返回值参时，包括：

◆ 把对象作为实参传递给非引用形参

◆ 返回类型为非引用类型的函数返回一个对象

拷贝构造函数

```
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;
}
```

拷贝构造函数

```
guxiwu@guxiwu-virtual-machine:~/cpp/project/CopyConstructorDemo/build$ ../bin/CopyConstructorDemo  
pass by value:  
ACopyable is copied  
ACopyable is copied  
pass by reference:  
ACopyable is copied
```

由于编译默认开启了RVO/NRVO，所以必须关闭该选项才能看到拷贝构造函数被调用。

例如，在CMakeLists.txt加上：

#关闭编译器优化

add_compile_options(-fno-elide-constructors)

对象构造的编译器优化

RVO (Return Value Optimization) /NRVO (Named Return Value Optimization) 优化

```
class A{
public:
    A() { cout<<"Construct"<<endl; }
    A(const A &a) { cout<<"Copy Construct"<<endl; }
};
A getA() { return A(); }

int main(){
    A a = getA();
    return 0;
}
```

A a = getA(); 会调用调用三次构造

- 1) A(); 产生临时对象, 记为tmp1
- 2) 函数return, 调用拷贝构造由tmp1构造另外一个tmp2返回
- 3) 用tmp2拷贝构造a

但是如果关闭优化, 则只会调用一次构造函数, 编译器会将A a = getA() 优化为: A a;

拷贝构造函数

编译器提供的合成的默认拷贝构造函数其行为是：**按成员依次拷贝**。如果用类型A的对象o1拷贝构造对象o2，则依次将对象o1的每个非静态数据成员拷贝给对象o2的对应的非静态数据成员，其中

- ◆ 如果数据成员是内置类型、指针、引用，则直接拷贝
- ◆ 如果数据成员是类类型，执行该数据成员类型的拷贝构造函数
- ◆ 如果是数组（不是指针指向的），则逐元素拷贝；如果元素类型是类类型，逐元素拷贝时会调用元素类型的拷贝构造函数

拷贝构造函数

按成员依次拷贝也被称为**浅拷贝**。当函数的参数为值参（非引用）时，实参传递给值参（非引用）会调用拷贝构造函数，**如果拷贝构造函数的实现为浅拷贝（如编译器提供的合成的默认拷贝构造函数），就存在如下问题：**

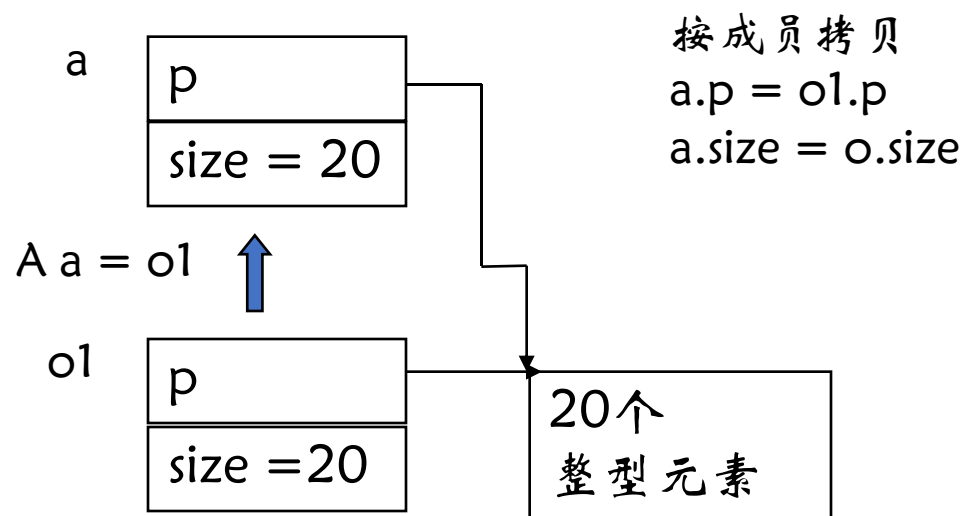
若实参对象包含**指针类型（或引用）的实例数据成员**，则只复制指针值而未复制指针所指的单元内容，实参和形参两变量的指针成员指向同一块内存。

- 当被调函数返回，形参对象就要析构，释放其指针成员所指的存储单元。若释放的内存被操作系统分配给其他程序，返回后若实参继续访问该存储单元，就会造成当前程序非法访问其他程序页面，导致操作系统报告一般性保护错误。
- 若释放的内存分配给当前程序，则变量之间共享内存将产生副作用。

浅拷贝（按成员拷贝）

```
struct A {  
    int *p;  
    int size;  
    A (int s):size(s), p(new int[size]){ }  
    A():size(0),p(0){}  
    ~A() { if(p) {delete p; p=0;}}  
};  
void f(A a) {}; //函数参数为值参  
A o1(20);  
f(o1); //等价于A a = o1; 调用编译器提供的缺省浅拷贝构造函数  
//用o1构造a
```

浅拷贝（按成员拷贝）



函数调用f(o1)发生时
实参对象o1赋值给形参
对象a，等价于
A a = o1;

当函数返回时，形参a
被析构（生命结束），
析构函数执行delete p，
导致p指向的内存被释放

函数返回后实参o1的生命周期没结束。若通过
o1继续访问该存储单元，如 o1.p[0] = 1;
就会造成当前程序非法访问其他程序页面

若实参变量包含指针类型的数据成员，则浅拷贝只复制指针的
值而未复制指针所指的单元内容，实参和形参两个变量(对象)的
指针成员指向同一块内存。造成无法估量的副作用

拷贝构造函数

深拷贝：在传递参数时先为形参对象的指针成员分配新的存储单元，而后将实参对象的指针成员所指向的单元内容复制到新分配的存储单元中。

必须进行深拷贝才能避免出现内存保护错误或副作用

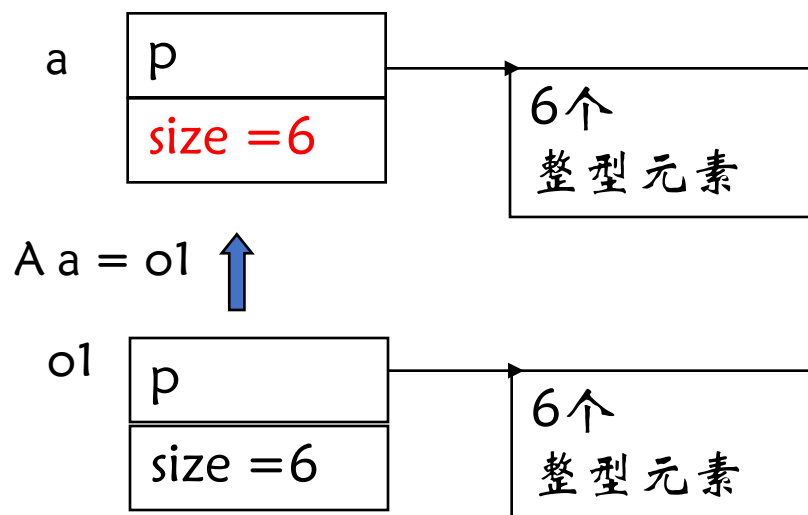
为了在传递参数时能进行深拷贝，**必须自定义参数类型为类的引用的拷贝构造函数**，即定义 `A (A &)`、`A (const A&)` 等形式的构造函数。建议使用 `A (const A&)`。

拷贝构造函数参数必须为引用（为什么）。

深拷贝

```
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
```

深拷贝



函数调用`f(o1)`发生时
用实参对象`o1`构造形参
对象`a`，等价于
`A a = o1;`
但这次是调用自定义的
拷贝构造函数，进行深拷贝

当函数返回时，形参`a`
被析构（生命结束），
析构函数执行`delete p`，
导致`p`指向的内存被释放
但不会影响`a.p`

深拷贝构造函数和深拷贝赋值运算符重载

为了避免类似的问题，当类包含指针成员时，安全的做法是

- ◆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。
- ◆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
- ◆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。
- ◆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

赋值运算符重载函数

与拷贝构造函数一样，如果类未定义自己的赋值运算符函数，编译器会提供合成的赋值运算符函数。

编译器提供的合成的赋值运算符函数是浅拷贝赋值。因此和拷贝构造函数一样，在必要的时候需要自己定义赋值运算符函数，且实现为深拷贝赋值。

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

赋值运算符重载函数

```
class MyString {
private:
    char *s;
public:
    MyString(const char *t = "" ){ ...}
    ~MyString() { ... }
    MyString & operator=(const MyString &rhs); //重载=
};

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
};

MyString s1( "s1" ), s2( "s2" ); s1 = s2;
```

移动构造和移动赋值

当函数的参数为值参以及函数返回类型为值时，都会出现对象的频繁拷贝。在很多情况下，对象被拷贝后就立即销毁（特别是函数返回值时），在这些情况下，采用移动对象会大幅提高性能。

C++11开始，可以定义类的移动构造函数和移动赋值运算符函数，从而实现对象的移动。对应类A，其移动构造函数和移动赋值运算符函数的原型为：

```
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)

移动构造和移动赋值

```
class MyString {  
private:  
    int len; //字符个数 (包括\0)  
    char *s;  
public:  
    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=  
    int length() { return strlen(s); } //返回字符串长度, 不舍\0  
    void print() { cout << this->s << endl; }  
};
```

移动构造和移动赋值

```
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;  
}
```

构造函数

```
MyString::~~MyString() {
```

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

析构函数

移动构造和移动赋值

```
MyString::MyString(const MyString &old):
```

```
    len(old.len), s(static_cast<char *>(malloc(len))) {
```

```
    strcpy(s,old.s);
```

```
    cout << "Copy Constructor:MyString: " << s << endl;
```

```
}
```

拷贝构造函数
实现为深拷贝

```
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
```

```
}
```

赋值函数=重载
实现为深拷贝

移动构造和移动赋值

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

移动构造函数实现时也必须加noexcept声明

//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;
```

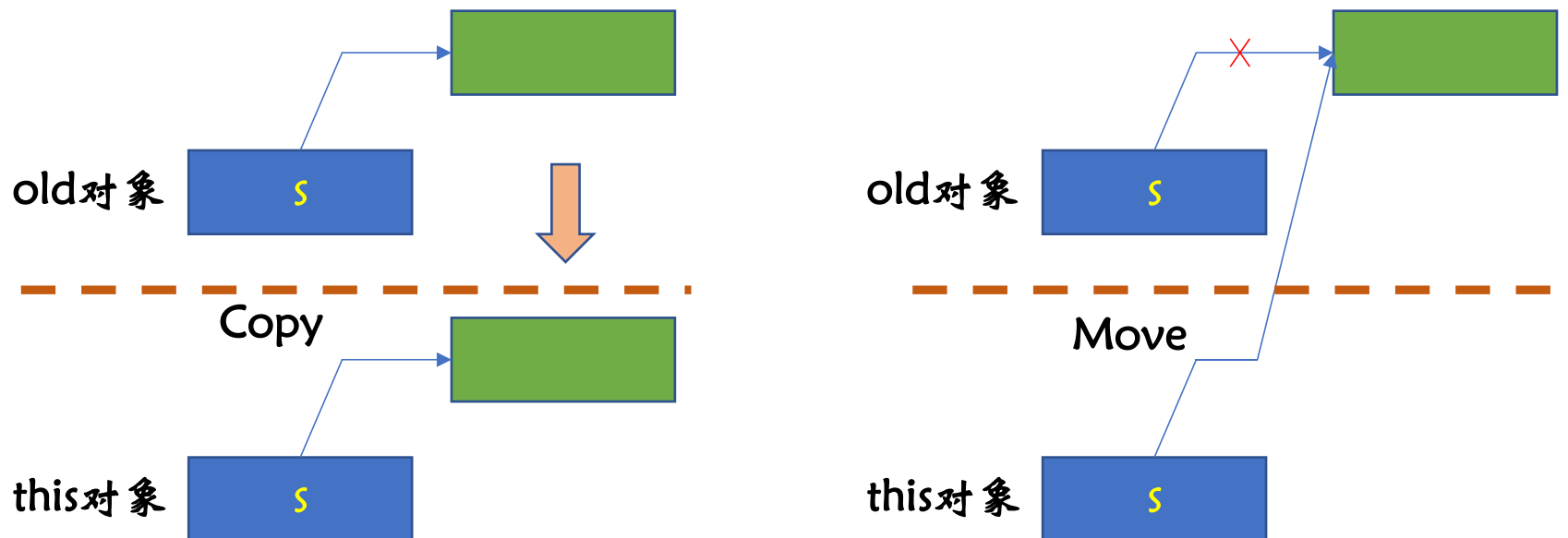
```
old.len = 0;
```

```
cout << "Move Constructor:MyString: " << s << endl;
```

```
}
```

- 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

移动构造和移动赋值



拷贝对象和移动对象的区别

移动构造的old对象是右值，是临时对象，其生命周期本来就是很短，资源不好好利用也是浪费。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。

移动构造和移动赋值

```
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;
```

```
}
```

移动赋值函数实现时也必须加noexcept声明

移动赋值函数实现时首先要检测是否是自赋值

移动构造和移动赋值

测试拷贝构造和移动构造(c++11, c++14, 关闭优化)

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

移动构造和移动赋值

测试拷贝构造和移动构造(c++11, c++14, 关闭优化)

运行环境配置，以cmake为例。MyString这个例子cmake工程的CMakeLists.txt配置为：

```
cmake_minimum_required(VERSION 3.13)
project(CopyAndMoveObject)

SET(CMAKE_C_COMPILER g++)

#用于添加编译器命令行标志（选项）
ADD_DEFINITIONS(-std=c++14)

#关闭编译器优化
add_compile_options(-fno-elide-constructors)

#支持UNICODE
ADD_DEFINITIONS(-DUNICODE -D_UNICODE)
INCLUDE_DIRECTORIES(${PROJECT_SOURCE_DIR}/include)
#查找在某个路径下的所有源文件，将输出结果列表储存在指定的变量中
AUX_SOURCE_DIRECTORY(${PROJECT_SOURCE_DIR}/src SRC_LIST)
#可执行文件输出目录
SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
#将${SRC_LIST}下所有源文件编译进CopyAndMoveObject
ADD_EXECUTABLE(CopyAndMoveObject ${SRC_LIST})
```

移动构造和移动赋值

测试拷贝构造和移动构造(c++11, c++14)

如果不关闭优化, 即: 去掉`add compile options(-fno-elide-constructors)`, 输出为

```
Constructor:MyString: Hello           //构造s12
Copy Constructor:MyString: Hello       //拷贝构造s13
Constructor:MyString: TempHello        //直接构造s14 (没有构造临时对象和调用移动构造)
Destructor:MyString: TempHello         //析构s14
Destructor:MyString: Hello              //析构s13
Destructor:MyString: Hello              //析构s12
```

如何优化? 记住这是编译器优化, 也就是发生在编译时

请回忆constexper: 在编译时可以求值

`MyString s14 = MyString(“TempHello”);`这一条语句在编译时, 编译器可以知道要构造一个对象s14, 其内容为“TempHello”, 既然这样, 干嘛还需要构造这个临时对象, 然后调用移动构造呢? 直接构造s14不就行了? 这就是优化! `MyString s14 = MyString(“TempHello”);`被优化成: `MyString s14(“TempHello”)`

注意: 在c++11和c++14, 是否进行优化是可选的, 即我们可以

`add_compile_options(-fno-elide-constructors)`来关闭优化。但是到c++17, 这种优化是强制性的。C++17这种新特性叫(强制省略拷贝copy elision mandatory)。

移动构造和移动赋值

测试拷贝构造和移动构造(c++17)

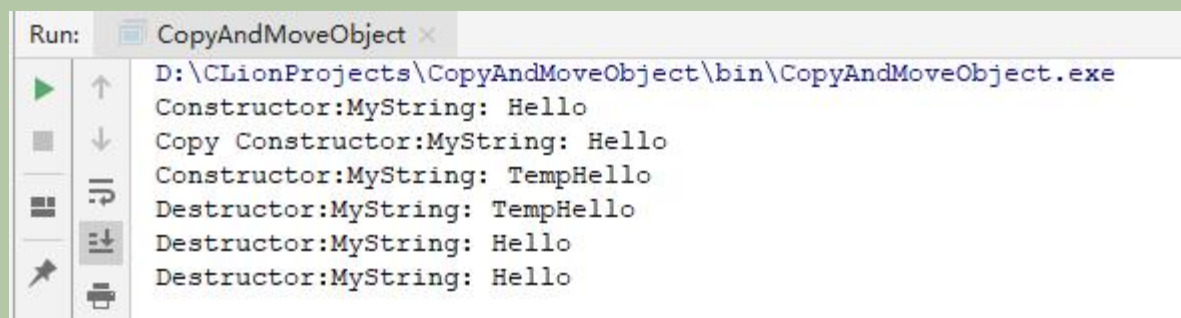
```
cmake_minimum_required(VERSION 3.13)
project(CopyAndMoveObject)

SET(CMAKE_C_COMPILER g++)

#用于添加编译器命令行标志（选项）
ADD_DEFINITIONS(-std=c++17)

#关闭编译器优化
add_compile_options(-fno-elide-constructors)

#支持UNICODE
ADD_DEFINITIONS(-DUNICODE -D_UNICODE)
INCLUDE_DIRECTORIES(${PROJECT_SOURCE_DIR}/include)
#查找在某个路径下的所有源文件，将输出结果列表储存在指定的变量中
AUX_SOURCE_DIRECTORY(${PROJECT_SOURCE_DIR}/src SRC_LIST)
#可执行文件输出目录
SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
#将${SRC_LIST}下所有源文件编译进CopyAndMoveObject
ADD_EXECUTABLE(CopyAndMoveObject ${SRC_LIST})
```



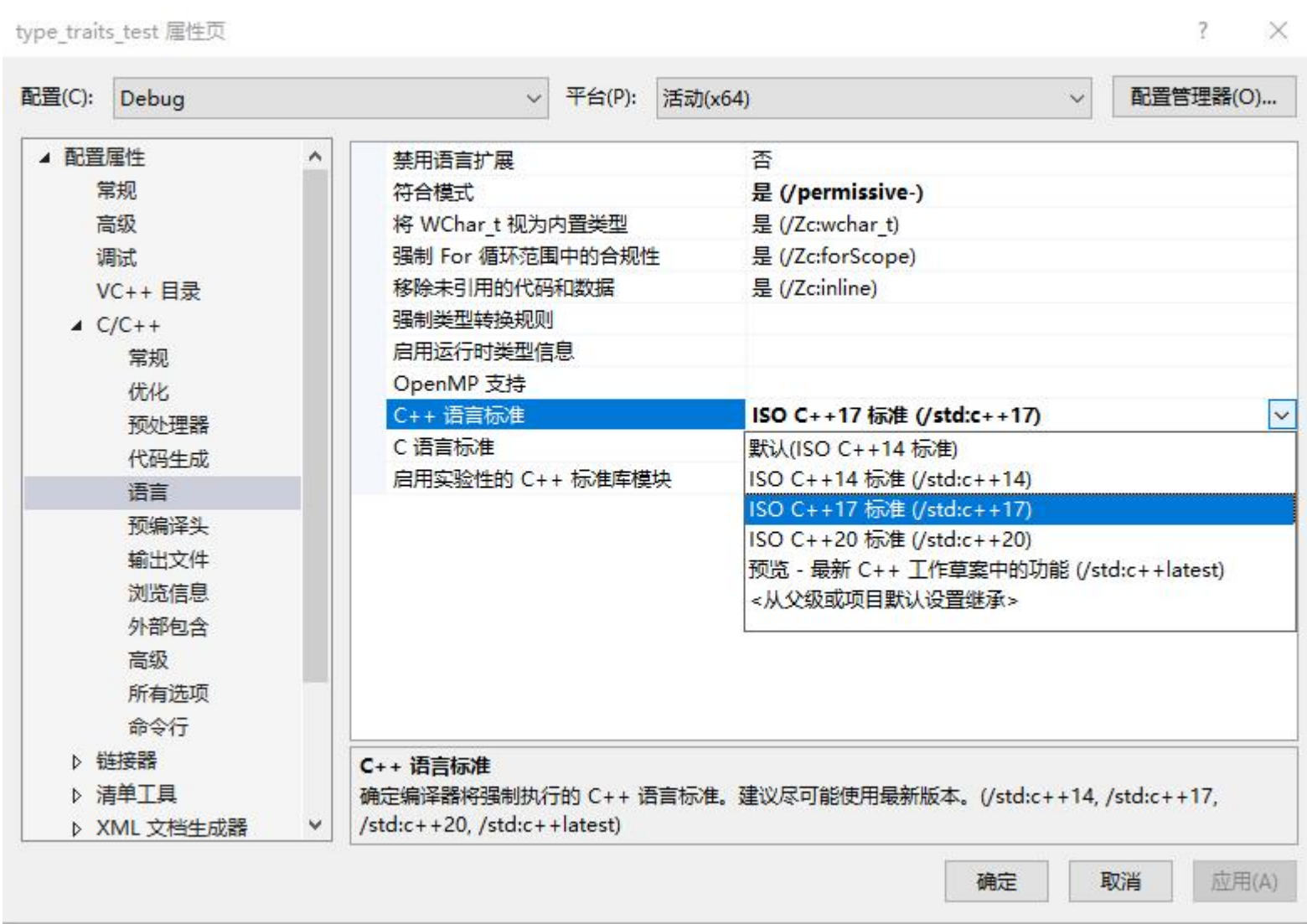
```
Run: CopyAndMoveObject x
D:\CLionProjects\CopyAndMoveObject\bin\CopyAndMoveObject.exe
Constructor:MyString: Hello
Copy Constructor:MyString: Hello
Constructor:MyString: TempHello
Destructor:MyString: TempHello
Destructor:MyString: Hello
Destructor:MyString: Hello
```

可以看到：如果采用c++17, 即使加上`add_compile_options(-fno-elide-constructors)`, 对象s14的创建还是被优化了（不再构造临时对象，不再调用移动构造），即这种优化是强制性的。

The main changes compared to the previous edition are as follows:

- expression evaluation order is specified in more cases
- removal of trigraphs
- adjustments to value categories resulting in copy elision being mandatory

Visual Studio如何选择c++标准以及打开关闭优化



打开项目属性

在Visual Studio中关闭RVO（[Return Value Optimization](#)，返回值优化）和NRVO（[Named Return Value Optimization](#)，命名返回值优化）的选项并不直接可用，因为Visual Studio无论是Debug还是Release模式下，都默认启用这些优化。至少在VS2019下是这样。

这也是为什么推荐用cmake及Vscode、CLion的原因。

移动构造和移动赋值

◆ 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

移动构造和移动赋值

测试拷贝构造和移动构造 (c++17, 是否关闭优化输出结果一样)

//但有时需要对一个左值对象执行移动构造怎么办

//可以调用std::move函数, 将一个左值对象转换成右值引用类型

MyString s15("Hello"); //s15是左值

MyString s16 = s15; //拷贝构造

MyString s17 = std::move(s15); //这时执行的是移动构造

```
D:\CLionProjects\CopyAndMoveObject\bin\CopyAndMoveObject.exe
Constructor:MyString: Hello           //构造s15
Copy Constructor:MyString: Hello      //拷贝构造s16
Move Constructor:MyString: Hello      //移动构造s17
Destructor:MyString: Hello            //析构s17
Destructor:MyString: Hello            //析构s16

Process finished with exit code 0
```

为什么只打印出二次Destructor:MyString: Hello? 课后思考

另外要注意调用move后, 不能对移动源对象是s15做任何假设。因为其生命周期还在, 使用时要非常小心。注意s17的构造不能被优化(?)

移动构造和移动赋值

测试函数返回值 (c++14, 关闭优化)

```
MyString f1(){  
    return MyString("f returned MyString object");  
}
```

//lamda表达式本质上就是一个匿名函数，函数自动内联

//注意该函数返回MyString值类型

```
auto f2 = []()->MyString{return MyString("f returned MyString object");};
```

```
MyString s = f2(); //等价于 MyString s = f1();
```

```
D:\CLionProjects\CopyAndMoveObject\bin\CopyAndMoveObject.exe  
Constructor:MyString: f returned MyString object  
Move Constructor:MyString: f returned MyString object  
Move Constructor:MyString: f returned MyString object  
Destructor:MyString: f returned MyString object  
Process finished with exit code 0
```

可以看到发生了二次对象移动：第一次是函数返回临时对象时（临时对象是右值，调用移动构造函数），第二次发生在用函数返回对象构造对象s。可以看到由于采用了移动构造，省掉了二次对象拷贝构造的开销。

移动构造和移动赋值

测试函数返回值 (c++14, 不关闭优化。在c++17下是强制性的)

```
D:\CLionProjects\CopyAndMoveObject\bin\CopyAndMoveObject.exe  
Constructor:MyString: f returned MyString object  
Destructor:MyString: f returned MyString object  
Process finished with exit code 0
```

可以被优化了, 根本没有创建临时对象和调用移动构造。直接构造了对象s

MyString s = f2();被优化成

MyString s ("f returned MyString object");

4.8 移动构造和移动赋值

测试拷贝构造和移动构造的性能 (c++14 关闭优化)

```
clock_t start, end;  
start = clock();  
const int counts = 2000;  
for(int i = 0; i < counts; i++){ //2000次循环  
    //很多个Hello, 模拟一个具有复杂结构的对象  
    MyString s15 = MyString( "HelloHello.....Hello" ); //调用拷贝构造  
}  
end = clock();  
cout<<"Run time: "<<(double)(end - start) * 1000.0 / CLOCKS_PER_SEC <<  
"ms"<<endl;
```

去掉移动构造函数, 同时去掉所有的cout语句

```
D:\CLionProjects\CopyAndMoveObject\bin\CopyAndMoveObject.exe  
Run time: 292ms  
Process finished with exit code 0
```

移动构造和移动赋值

测试拷贝构造和移动构造的性能 (c++14 关闭优化)

```
clock_t start, end;  
start = clock();  
const int counts = 2000;  
for(int i = 0; i < counts; i++){ //20万次循环  
    //很多个Hello, 模拟一个具有复杂结构的对象  
    MyString s15 = MyString( "HelloHello.....Hello" ); //调用移动构造  
}  
end = clock();  
cout<<"Run time: "<<(double)(end - start) * 1000.0 / CLOCKS_PER_SEC <<  
"ms"<<endl;
```

加上移动构造函数, 同时去掉所有的cout语句

```
D:\CLionProjects\CopyAndMoveObject\bin\CopyAndMoveObject.exe
```

```
Run time: 178ms
```

```
Process finished with exit code 0
```

时间减少了39%

移动构造和移动赋值

移动构造和移动赋值函数声明为noexcept的作用：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::push_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。

移动构造和移动赋值

C++11标准库容器类同时支持拷贝和移动 (c++14 关闭优化)

```
MyString s18("hello_18");  
MyString s19("hello_19");  
vector<MyString> vs;  
vs.push_back(s18); //执行拷贝构造  
vs.push_back(std::move(s19)); //执行移动构造
```

```
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 //移动构造  
Move Constructor:MyString: hello_18 //扩容移动s18  
Destructor:MyString: hello_18 //析构容器里的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
```

移动构造和移动赋值

测试移动构造的noexcept声明对容器的影响 (c++14 关闭优化)

```
MyString s18("hello_18");  
MyString s19("hello_19");  
vector<MyString> vs;  
vs.push_back(s18); //执行拷贝构造  
vs.push_back(std::move(s19)); //执行移动构造
```

去掉移动构造函数的noexcept声明

移动构造函数的noexcept声明只对容器扩容时产生影响

vs.push_back(std::move(s19))还是执行移动构造, 尽管去掉了noexcept声明

```
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
```

移动构造和移动赋值

测试拷贝赋值和移动赋值的区别 (c++14 关闭优化)

```
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s20  
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
```

移动构造和移动赋值

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。

```
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
```

右值引用和成员函数

除了拷贝构造和赋值函数。其他成员函数也可以提供拷贝和移动版本。例如std::vector就提供了push_back函数的拷贝版本和移动版本。

类型T的移动和拷贝的重载函数，拷贝版本接受参数类型是const T&, 移动版本接受的参数类型是T &&。例如

```
class Foo{  
public:  
    //其它定义  
    void f(const Foo &); //拷贝版本，函数f里拷贝对象  
    void f(Foo &&); //移动版本，函数f里移动对象  
};
```

=default和=delete

一个类的以下六个函数（称为Big Six）可以用=default显式要求编译器提供默认版本。除了Big Six之外的其他成员函数不能=default

```
class A{
    public:
        //以下6个函数都可以要求编译器提供默认版本
        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
        // void f() = default; //编译错：其他成员函数定义成default没有意义
};
```

=default和=delete

一个类的任何函数都可以用=delete定义为删除的。例如有时为了阻止对象的赋值和拷贝，需要将拷贝构造函数和赋值运算函数定义为删除的。被定义为删除的函数相当于给出了函数定义。

```
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

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
}
```

=default和=delete

```
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
}
```


=default和=delete

```
class Foo{
private:
    int _i;
public:
    Foo(int i):_i(i) {}
    Foo() = default; //与Foo(int i)重载
    Foo(const Foo &x):_i(x._i){} //自定义拷贝构造
    // Foo(const Foo &x) = default; //编译器报错：相同名称相同参数的函数不能重载
    // Foo(const Foo &x) = delete; //编译器报错：相同名称相同参数的函数不能重载
    Foo &operator=(const Foo &x){ _i = x._i; return *this;} //自定义拷贝赋值
    // Foo &operator=(const Foo &x) = default; //编译器报错：原因同上
    // Foo &operator=(const Foo &x) = delete; //编译器报错：原因同上
    // ~Foo() = delete; //允许，但是Foo object;语句会报错
    ~Foo() = default; //要求编译器提供默认版本
};
```

```
Foo f1(5);
Foo f2; //如果没有写出Foo() = default; , 会编译报错
Foo f3(f1); //如果拷贝构造=delete。会编译报错
f3 = f2; //如果拷贝赋值=delete, 会编译报错
```