例一：显然

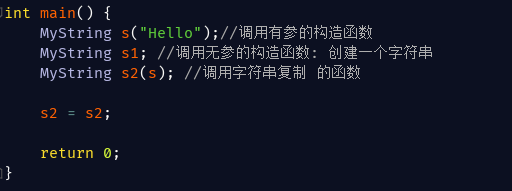
#include <iostream>  
#include <cstring>  
using namespace std;  
  
//struct 默认所有对象都是public,而class 默认private  
struct MyString{  
 char \*str;  
 size\_t length; //size\_t 当作比int 能存更大范围数的 int 型  
  
 //不带参数的构造函数  
 MyString() : str(NULL) , length(0) {  
 *cout*<<" MyString()"<<endl;  
 }  
  
 //创建一个新字符串  
 MyString(const char \*s) : str(NULL), length(0){  
 *cout*<<" MyString(const char \*s)"<<endl;  
 if(s == NULL){  
 return;  
 }  
 length = strlen(s);  
 str = new char [length + 1];  
 strcpy(str,s);//将字符串s 复制给空的 str;  
 }  
  
 //字符串复制  
 MyString(const MyString &s) :str(NULL) , length(0){  
 *cout*<<" MyString(const MyString &S)"<<endl;  
 if(s.length == 0)  
 return;  
 length = strlen(s.str);  
 str = new char [length + 1];  
 strcpy(str,s.str); //将字符串s.str 拷贝到 str;  
 }  
  
 ~MyString(){  
 *cout*<<" ~MyString()"<<endl;  
 delete [] str;  
 }  
  
   
  
};  
  
int main() {  
 MyString s("Hello");//调用有参的构造函数  
 MyString s1; //调用无参的构造函数: 创建一个字符串  
 MyString s2(s); //调用字符串复制 的函数  
  
 return 0;  
}

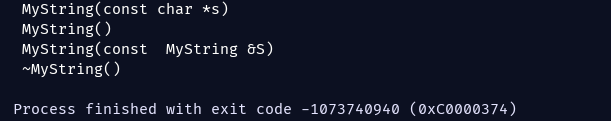


2重载 = 号，没有处理自等的情况：第一个析构MyString s2(s) 段错误，因为在处理 = 时

开始就把str 自己删了，就等于把other 删了，后再析构MyString s2(s)时，没得delete

//重载各种符号  
 MyString & operator = (const MyString &other){  
// if(this == &other){//处理other = other 自等的情况，否则段错误  
// return \*this;  
// }  
 delete [] str;  
 length = 0;  
 if(other.length == 0)//用空字符串 做 等号运算  
 return \*this;  
 str = new char[other.length + 1];  
 length = other.length + 1;  
 strcpy(str,other.str);  
 return \*this;  
 }





3. s1 = s + s2 重载操作符+ 字符串拼接,否则无法识别

注意此段程序的输出：黄1MyString(const MyString &S) 和黄2 ~MyString()

是在return t后出来的.return t 导致重新用MyString &other拷贝返回值t,返回一个临时变量 拿去给=用 黄3，和delete t黄2 ，在 = 中（黄3）然后析构临时变量（黄4）

对各种操作进行标识。

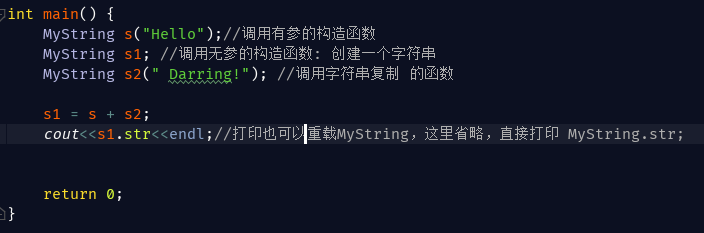
*cout*<<" MyString & operator +"<<endl;

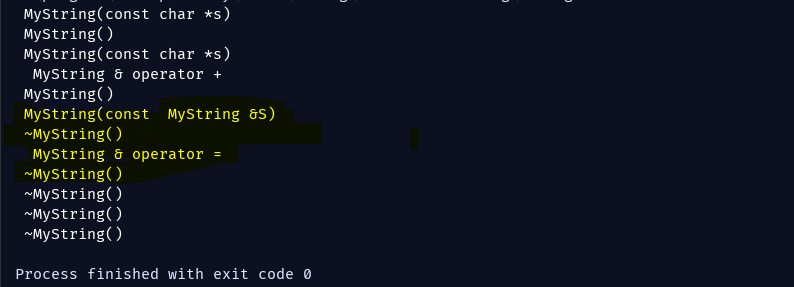
*cout*<<" MyString & operator ="<<endl;

//s1 = s + s2 重载操作符+ 字符串拼接  
MyString operator + (const MyString &other) const{

*cout*<<" MyString & operator +"<<endl;

//如果自己是空的  
 if(length == 0)  
 return other;  
 if(other.length == 0)  
 return \*this;  
 MyString t;  
 t.str = new char[length + other.length +1];  
 strcpy(t.str,str);//把自己的那部分拷贝过去t  
 strcpy(t.str + length , other.str);//将 other 的那部分拷贝过去  
 t.length = length + other.length;  
 return t;  
}





Main中看似只有四层调用，起始背后用的代价（如上图示）远不止，为了减少这种代价，c++ 不断在变化，这就是c++ 比java 复杂 但是效率高的原因。下面解决这个效率问题

4. 左值：命名定义的变量，如return t 中的t（通俗讲可以放到等号左边t = balabalaaa）

右值：不能放到等号左边，不能被赋值：比如的s+s2,

i++（int a = i , i = i+1, return a a就是临时变量 ） int &&a = i++ 合法

printf(“%p %p”, &p ,&a)

实际上左右值区别是能不能取地址（有地址的是左值）

Int i= 0;

Int &a = i ;因为a 被声明成了一个int 型的左值，而I 就是左值，可以 将 i 和a 绑定

但是Int &a = 0； 绑不了，0 是个右值，是个常量，可以用0 的值但是不能访问0的地址

但是 Int &&a = 0合法，右值引用绑定右值，Int &a = i ;左值引用绑定左值

++i++,i++++均不合法，++++i合法

//提高效率：移动构造 右值引用  
MyString(MyString &&s) : str(NULL),length(0){  
 *cout*<<" MyString(const MyString &&S)"<<endl;  
 if(s.length == 0)  
 return;  
 str = s.str;  
 length = s.length;  
 s.str = NULL;//不需要new char空间, 只需要将指针指过来就行  
}

这里就优化了上述return t 时的两步操作（开辟新空间赋值，析构原来的），直接将新指针指向原来的空间，原来的指针 = NULL

