第四课 类和对象（构造深入）

内容概述

1. 数据成员指针
2. 成员函数指针
3. 三/五法则
4. 引用计数
5. 写时拷贝
6. swap函数
7. 移动构造函数
8. 移动赋值运算符重载
9. 对象移动
10. std::vector动态增长
11. std::vector与移动
12. 移动小结

数据成员指针

定义：
数据类型 类名::*指针名 = &类名::数据成员

解引用：
对象名.*指针名
对象指针->*指针名

数据成员指针实际上是一个偏移量，区别于普通指针。

static静态成员变量不能使用数据成员指针。

```
#include <iostream>
using namespace std;
struct A {
    A(double n1 = 0.0, int n2 = 0) :num1(n1), num2(n2) {}
    double num1;
    int num2;
};
int main() {
    int i = 10;
    int *pi = &i; //指针 指向普通变量
    A a(0.1, 1), b(1.1, 2);
    int *pa = &a.num2;
    cout << a.num2 << endl; //1
    int A::*p = &A::num2; //指针 指向成员变量
    cout << a.*p << b.*p << endl; // 1 2
    //指向数据成员的指针，是针对类的，是一个偏移量
    printf("%p\n", p); // 00000008
    A* pA = new A(3.1, 3);
    cout << pA->*p << endl; //3
    auto p_double = &A::num1;
    cout << typeid(p_double).name() << endl; //double A::*
    delete pA;
    return 0;
}
```

```
1
12
00000008
3
double A::*
请按任意键继
```

成员函数指针

普通函数指针：返回值类型 (*指针名)(参数列表)
注意：void(*p_fun)(int, int); 和 void* p_fun(int, int); 的区别。
前者是定义函数指针，后者是函数声明（指针函数）。

成员函数指针的定义：
返回值类型 (类名::*指针名)(参数列表)
解引用：
(对象名.*指针名)(参数列表)
(对象指针->*指针名)(参数列表)

非静态成员函数指针与普通函数指针的区别是，隐含参数this指针。
静态成员函数由于没有this指针，所以和普通函数指针是一样的，不能 类名::*指针名

成员函数指针赋值时，不能省略 &(取地址符)

成员函数指针

```cpp
#include <iostream>
using namespace std;
struct A {
    int add(int a, int b) { cout << "add()" << endl; return a + b; }
    static void show() { cout << "show()" << endl; }
};
void f1(int a, int b) { cout << "f1()" << endl; }
void f2(int a, int b) { cout << "f2()" << endl; }
typedef void(*P_FUN)(int, int);  //typedef 定义函数指针
using P_FUN1 = void(*)(int, int);//using 定义函数指针
typedef int(A::*P_ADD)(int, int);
using P_ADD1 = int(A::*)(int, int);
int main() {
    void(*p_fun)(int, int) = &f1; //普通函数指针,可省略&
    p_fun(2, 2); // f1()
    p_fun = &f2;
    p_fun(1, 2); // f2()
    P_FUN p1 = &f1;
    p1(1, 1);  // f1()
    p1 = &f2;
    p1(2, 2);  // f2()
    P_FUN1 p2 = &f1;
    p2(1, 1);  // f1()
    p2 = &f2;
    p2(2, 2);  // f2()
```

普通函数指针

成员函数指针

```cpp
    //非静态成员函数指针，区别于普通函数，主要是由于隐含的this指针
    int(A::*p_add)(int, int) = &A::add; //非静态成员函数指针
    A a;
    cout << (a.*p_add)(0, 2) << endl; //.* 必须有对象,注意括号 运行结果add() 2
    A* pa = new A;
    cout << (pa->*p_add)(1, 2) << endl;  //->* //运行结果add() 3
    auto p_add1 = &A::add;
    cout << (a.*p_add1)(2, 2) << endl;  //运行结果add() 4
    P_ADD p11 = &A::add;
    cout << (a.*p11)(3, 2) << endl; //运行结果add() 5
    P_ADD1 p22 = &A::add;
    cout << (pa->*p22)(2, 4) << endl; //运行结果add() 6
    delete pa;
    void(*p_show)() = &A::show; //静态成员函数指针 与普通函数指针类似
    p_show();
    return 0;
}
```

成员函数指针

应用举例：Game类的方向移动函数的封装。

using Action = void(Game::*)(); //定义别名 成员函数指针
相当于 typedef void(Game::*Action)();

enum Direction { LEFT, RIGHT, UP, DOWN }; //枚举
默认从0开始，相当于LEFT=0,RIGHT=1,UP=2,DOWN=3

static Action menu[];
数组中存放的是 成员函数指针

```cpp
#include <iostream>
using namespace std;
class Game {
public:
    using Action = void(Game::*)(); //定义别名 成员函数指针
    enum Direction { LEFT, RIGHT, UP, DOWN }; //枚举
    void move(Direction d) {
        (this->*menu[d])();
    }
private:
    void left() { cout << "left" << endl; }
    void right() { cout << "right" << endl; }
    void up() { cout << "up" << endl; }
    void down() { cout << "down" << endl; }
    static Action menu[]; //函数指针 数组
};
Game::Action Game::menu[] = {   &Game::left,
                                &Game::right,
                                &Game::up,
                                &Game::down };

int main() {
    Game g;
    g.move(Game::LEFT);
    g.move(Game::RIGHT);
    g.move(Game::UP);
    g.move(Game::DOWN);
    return 0;
}
```

```
left
right
up
down
请按任
```

三/五法则

默认构造函数，拷贝构造函数，赋值运算符重载，析构函数，系统可自动合成。（自己没有定义的时候）

拷贝构造函数，赋值运算符重载，析构函数
一般情况下，要么都自己定义，要么都是系统合成。
有**资源**时，都自定义，没资源时，不必自己定义。

三个当中，只要有一个需要自定义，意味着其他两个也要自定义！

使用 =default; 显式要求编译器生成合成默认版本
使用 =delete; 定义为删除的函数。通知编译器不需要该函数。

private 也可以阻止拷贝，阻止赋值。
构造或析构函数定义为 private将无法在类外创建对象。
但是：构造public,析构private是可以用new创建对象的。

```
class A {
public:
    A() = default;
    ~A() = default;
    A(const A&) = default;
    A& operator=(const A&) = default;
};
class B {
public:
    B() = default;
    B(const B&) = delete;
    B& operator=(const B&) = delete;
};
class C {
public:
    void destroy() { delete this; }
private:
    ~C() {};
};
int main() {
    A a1;
    A a2 = a1;
    B b1;
    //B b2 = b1; //错误，拷贝构造delete
    //C c1; //错误，析构函数是 private
    C* pc = new C;
    pc->destroy();
    return 0;
}
```

引用计数1

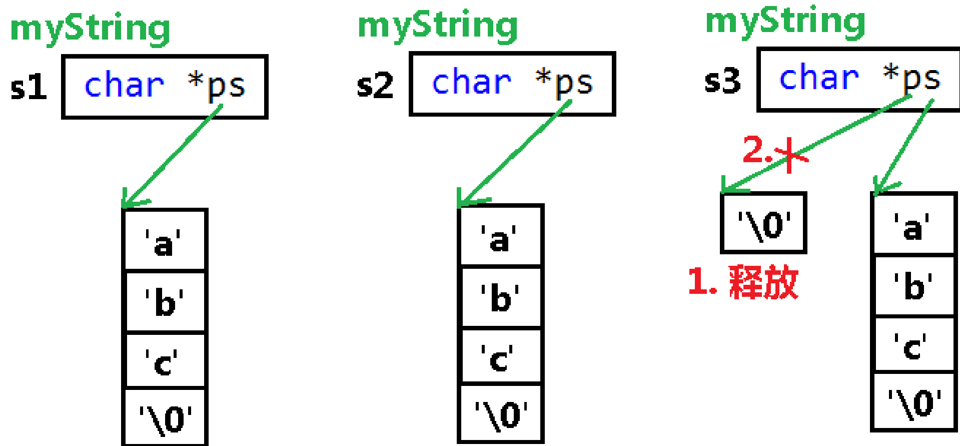
思考：深拷贝一定就好吗？浪费内存空间（数据冗余存储）

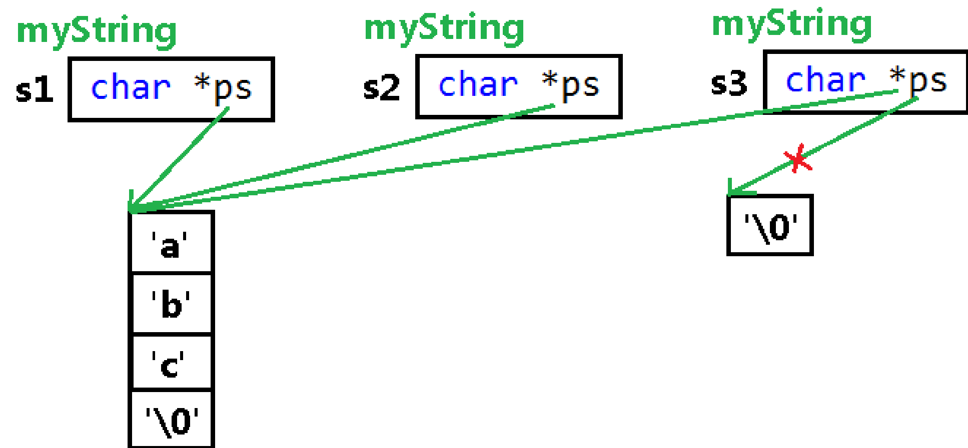
```
myString(const myString &other) {
    ps = new char[strlen(other.ps) + 1];
    strcpy(ps, other.ps);   //拷贝构造(深)
}
myString &operator=(const myString &other) {
    if (this != &other) {
        delete[] ps;
        ps = new char[strlen(other.ps) + 1];
        strcpy(ps, other.ps);
    }
    return *this;          //赋值运算符重载(深)
}
```

```
myString(const myString &other) {
    ps =  other.ps;  //拷贝构造(浅)
}
myString &operator=(const myString &other) {
    ps = other.ps;
    return *this;    //赋值运算符重载(浅)
}
```

```
class myString {
    ......
    char *ps;
};
```

```
myString s1 = "abc";
myString s2 = s1;
myString s3;
s3 = s1;
```





引用计数2

思考：为了节约内存空间，使用浅拷贝，如何解决“重析构” “内存泄漏”的问题？

引用计数：增加一个计数器，记录当前指向同一块内存的次数，拷贝构造和赋值的时候：计数+1，析构的时候：计数-1， 假如计数器==0，那么释放内存。

构造时：count = 1;

拷贝时：count++;

赋值时：count++;

析构时：

```
count--;
if(count == 0)
   delete [] ps;
```

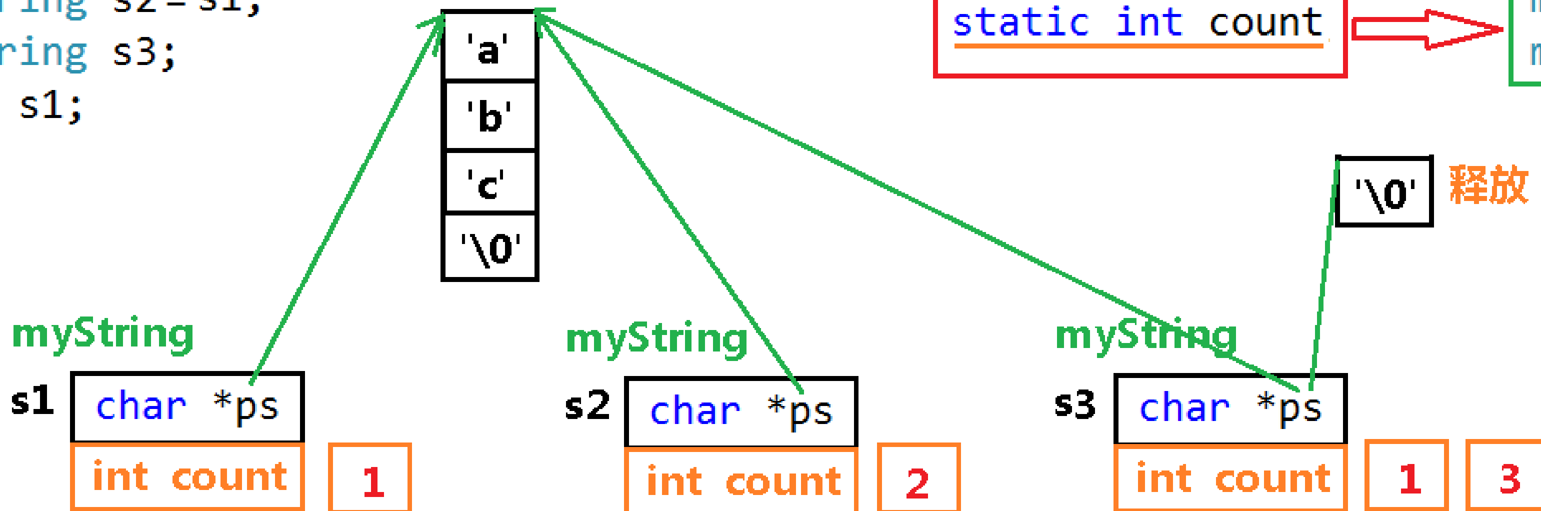
```
class myString {
    ......
    char *ps;
    int count;
};
```

这样做？

```
class myString {
    ......
    char *ps;
    static int count;
};
```

还是这样做？

```
myString s1="abc";
myString s2=s1;
myString s3;
s3 = s1;
```




```
myString s1 = "abc";
myString s2 = "123";
```

赋值：

1. 自己的count--,假如==0那么释放自己的内存；
2. 赋值的count++，并指向

引用计数3

使用成员指针，计数器既有共通性，又有独立性。

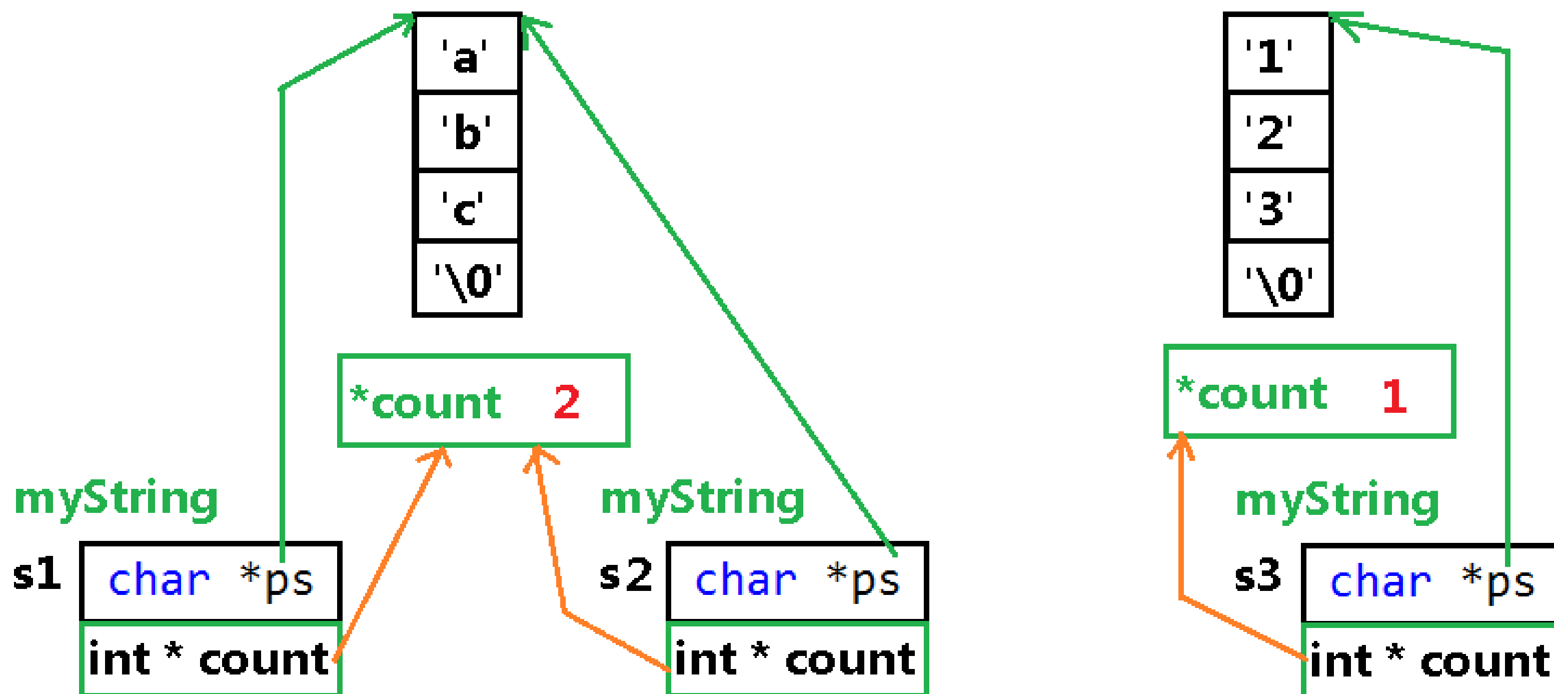
构造时，为count开辟空间，并赋值=1
在 count==0 时，和ps同时释放内存。

mySstring s1(“abc”);
myString s2=s1;
myString s3(“123”);

构造时：count = 1;
拷贝时：count++;
赋值时：count++;

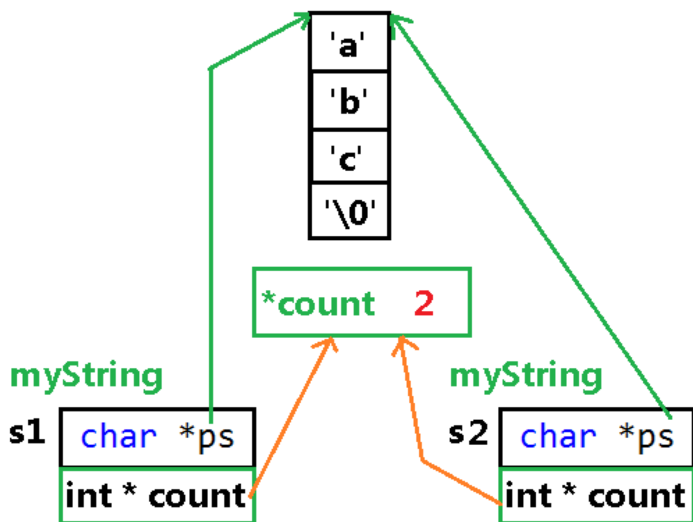
析构时：
count--;
if(count == 0)
delete [] ps;

```
class myString {
    ......
    char *ps;
    int *count;
};
```



引用计数4

```cpp
class myString {
  ......
private:
    char *ps;
    int *count;
};
```




```cpp
myString(const char * pstr = NULL) {
    if (!pstr)
        ps = new char[1]{'\0'};
    else {
        ps = new char[strlen(pstr) + 1];
        strcpy(ps, pstr);
    }
    count = new int(1); //构造时开辟空间并置为1
}
myString(const myString &other)
    :ps(other.ps), count(other.count) {
    (*count)++; //拷贝构造时 count + 1
}
~myString() {
    (*count)--; //析构时 count - 1
    if (*count == 0) { //count为 0 释放内存
        delete[] ps;
        delete count;
        cout << "delete!" << endl;
    }
}
int get_count()const { return *count; }
```

```cpp
int main() {
    myString s1 = "abc";
    cout << s1.get_count() << endl; //1
    myString s2 = s1;
    cout << s2.get_count() << endl; //2
    cout << s1.get_count() << endl; //2
    {
        myString s3 = s2;
        cout << s3.get_count() << endl; //3
    }
    cout << s2.get_count() << endl; //2
    cout << "===========" << endl;
    return 0;
}
```

```
1
2
2
3
2
========
delete!
请按任意
```

构造函数
拷贝构造
析构函数
赋值运算符重载
后面讨论。

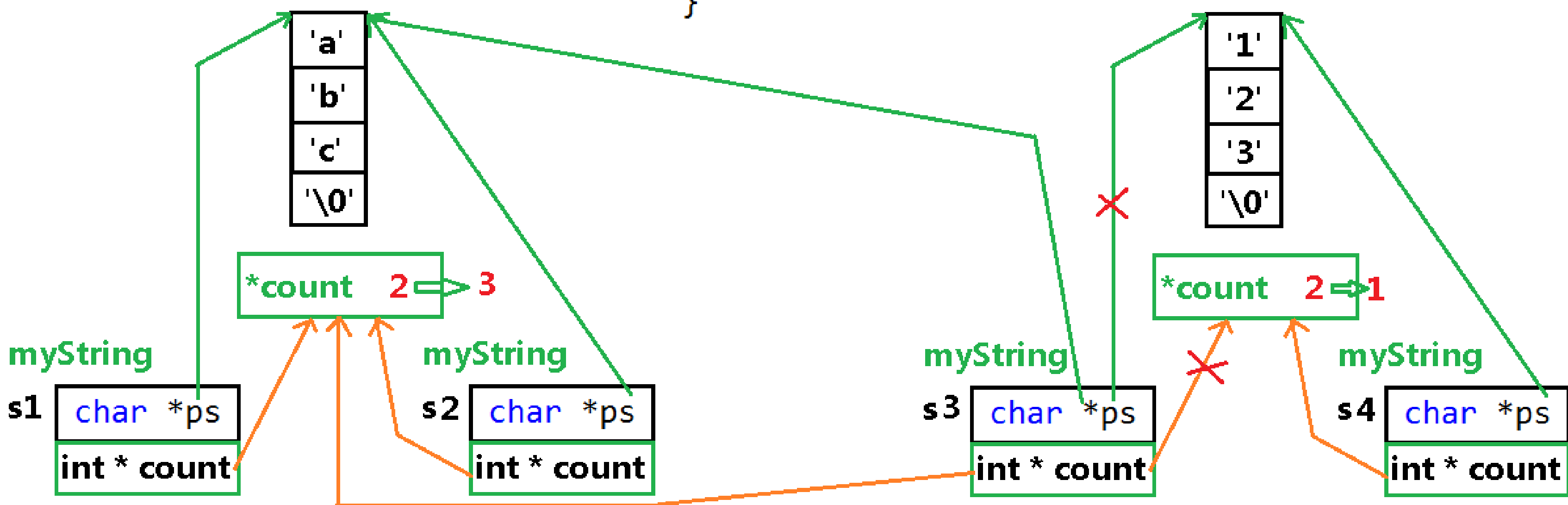
引用计数5

```
class myString {
    ......
private:
    char *ps;
    int *count;
};
```

```
myString &operator=(const myString &other) {
    if (this != &other) {
        (*count)--; //本对象原来的计数器 -1
        if (*count == 0) {
            cout << "delete in = ." << endl;
            delete[] ps;
            delete count;
        }
        ps = other.ps;
        count = other.count;
        (*count)++;
    }
    return *this;
}
```

```
int main() {
    myString s1("abc");
    myString s2 = s1; //拷贝构造
    cout << s1.get_count() << endl; //2
    myString s3("123");
    myString s4 = s3;
    cout << s3.get_count() << endl; //2
    s3 = s2;
    cout << s3.get_count() << endl; //3
    cout << s4.get_count() << endl; //1
    cout << "===========" << endl;
    return 0;
}
```

```
2
2
3
1
===========
delete!
delete!
请按任意键继续.
```



赋值运算符重载

引用计数6

```cpp
class myString {
public:
    myString(const char * pstr = NULL) {
        if (!pstr)
            ps = new char[1]{'\0'};
        else {
            ps = new char[strlen(pstr) + 1];
            strcpy(ps, pstr);
        }
        count = new int(1); //构造时开辟空间并置为1
    }
    myString(const myString &other)
        :ps(other.ps), count(other.count) {
        (*count)++; //拷贝构造时 count + 1
        cout << "拷贝构造时：*count= " << *count << endl;
    }
    ~myString() {
        cout << "析构！" << endl;
        (*count)--; //析构时 count - 1
        if (*count == 0) { //count为 0 释放内存
            delete[] ps;
            delete count;
            cout << "delete!" << endl;
        }
    }
    int get_count()const { return *count; }
```

```cpp
 myString operator+(const myString &other) {
     int len = strlen(ps) + strlen(other.ps);
     char * ps_tmp = new char[len + 1]{ 0 };
     strcpy(ps_tmp, ps);
     strcat(ps_tmp, other.ps);
     myString tmp(ps_tmp);
     return tmp;
 }
private:  char *ps;  int *count;  };
```

引用计数的测试

重载了operator+

```cpp
int main() {
    myString s1("abc"),s2("123");
    //s1.operator+(s2) 临时对象：表达式结束时销毁
    myString s3 = s1 + s2;
    cout << s3.get_count() << endl; //1
    cout << "============" << endl;
    return 0;
}
```

```
拷贝构造时: *count= 2
析构!
1
============
析构!
delete!
析构!
delete!
析构!
delete!
请按任意键继续. . .
```

```cpp
myString fun(myString s) {
    cout << "fun: " << s.get_count() << endl;
    return s;
}
int main() {
    myString s1("abc");
    myString s2 = fun(s1);
    cout << s2.get_count() << endl; //2
    cout << "----" << endl;
    return 0;
}
```

```
拷贝构造时: *count= 2
fun: 2
拷贝构造时: *count= 3
析构!
2
----
析构!
析构!
delete!
请按任意键继续. . .
```

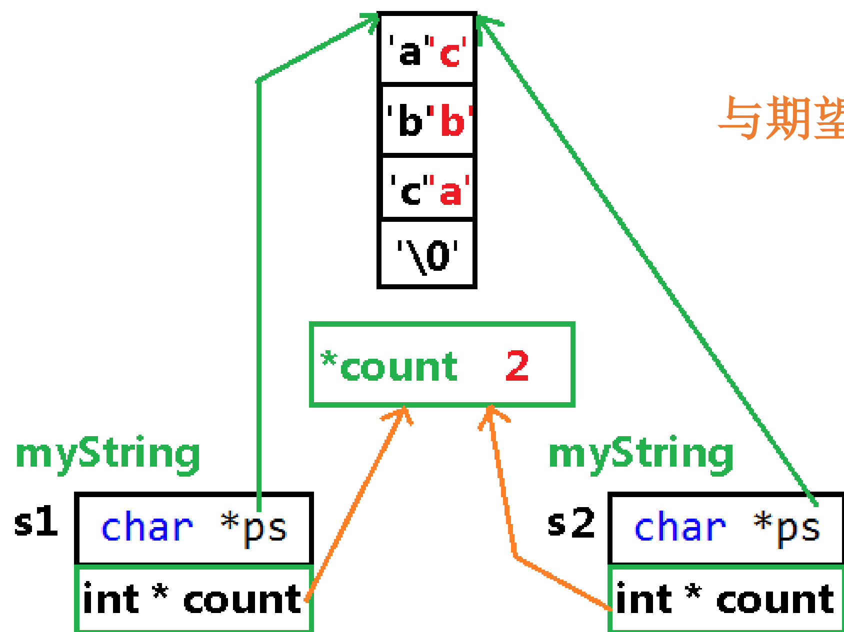
写时拷贝1

```
#include <iostream>
#include <cstring>
using namespace std;
class myString {
public:
    .........
    void reverse() {
        int len = strlen(ps) - 1;
        for (int i = 0; i < len / 2; i++) {
            char c = ps[i];
            ps[i] = ps[len - i];
            ps[len - i] = c;
        }
    }
    const char * c_str()const { return ps; }
private:
    char *ps;
    int *count;
};
```

增加函数
reverse反转

```
int main() {
    myString s1("abc");
    myString s2 = s1;
    cout << "s1=" << s1.c_str() << endl; //abc
    s1.reverse();
    cout << "s1=" << s1.c_str() << endl; //cba
    cout << "s2=" << s2.c_str() << endl; //cba
    return 0;
}
```

```
s1=abc
s1=cba
s2=cba
请按任意
```



与期望不符

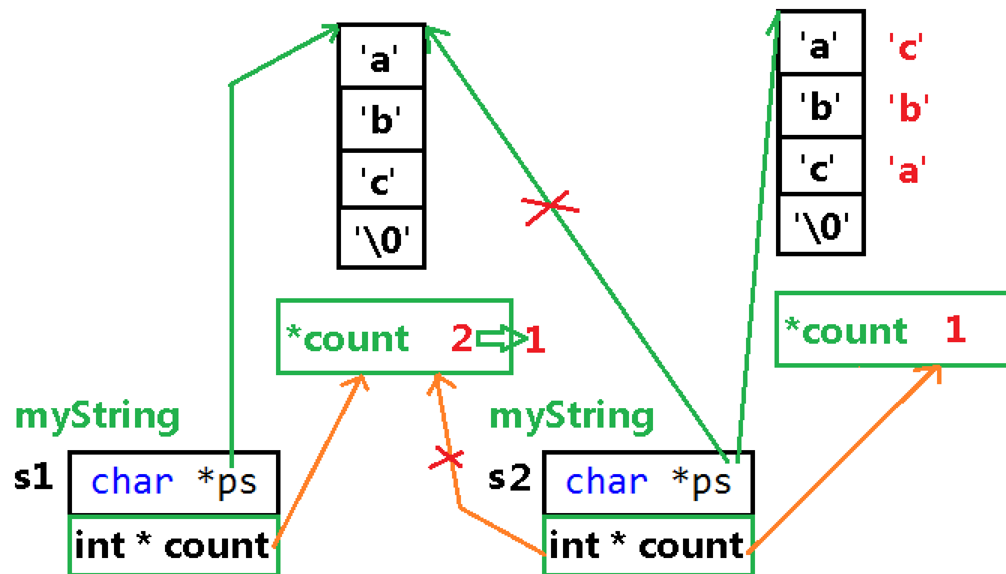
写时拷贝2

写时拷贝

```
void reverse() {
    if (*count > 1) { //除了自己外，还有人在用
        //重新复制一份以后，再处理
        (*count)--; //先把自己的计数去掉
        count = new int(1); //重新开辟新的计数空间
        char* tmp = ps; //保留原来的ps指针指向的位置
        ps = new char[strlen(tmp) + 1];
        strcpy(ps, tmp);
        //此时，本对象已经复制了一份拷贝，并新开了计数器
    }

    int len = strlen(ps) - 1;
    for (int i = 0; i < len / 2; i++) {
        char c = ps[i];
        ps[i] = ps[len - i];
        ps[len - i] = c;
    }
}
```

```
int main() {
    myString s1("abc");
    myString s2 = s1;
    cout << "s1=" << s1.c_str() << endl; //abc
    s1.reverse();
    cout << "s1=" << s1.c_str() << endl; //cba
    cout << "s2=" << s2.c_str() << endl; //abc
    return 0;
}
```



小象学院 ChinaHadoop.cn

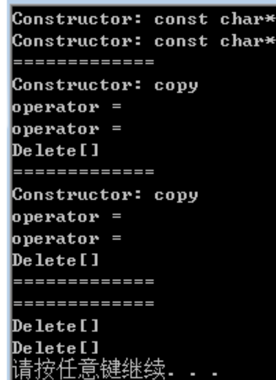
SWAP函数

交换操作：
swap函数

函数重载，
定义自己的
swap函数

```
class myString {
public:
    friend void swap(myString &a, myString &b);
    myString(const char * pstr = NULL) {
        if (!pstr)
            ps = new char[1]{ '\0' };
        else {
            ps = new char[strlen(pstr) + 1];
            strcpy(ps, pstr);
        }
        cout << "Constructor: const char*" << endl;
    }
    myString(const myString &other) {
        ps = new char[strlen(other.ps) + 1];
        strcpy(ps, other.ps);
        cout << "Constructor: copy" << endl;
    }
    myString &operator=(const myString &other) {
        if (this != &other) {
            delete[] ps;
            ps = new char[strlen(other.ps) + 1];
            strcpy(ps, other.ps);
        }
        cout << "operator = " << endl;
        return *this;
    }
    ~myString() {
        delete[] ps;
        cout << "Delete[]" << endl;
    }
private:
    char *ps;
};
```

```
void std_swap(myString &a, myString &b) {
    myString tmp = a;
    a = b;
    b = tmp;
}
void swap(myString &a, myString &b) {
    std::swap(a.ps, b.ps);
}
int main() {
    myString s1 = "abc";
    myString s2 = "123";
    cout << "============" << endl;
    std::swap(s1, s2); //std::swap函数
    cout << "============" << endl;
    std_swap(s1, s2);  //模拟std::swap
    cout << "============" << endl;
    //std::swap执行了深拷贝（拷贝构造和赋值运算符重载）
    swap(s1, s2); //自己定义的swap函数，没有深拷贝
    cout << "============" << endl;
    return 0;
}
```



对象移动1

```cpp
#include <iostream>
#include <cstring>
using namespace std;
class myString {
public:
    myString(const char *str = nullptr); //构造
    myString(const myString &other);     //拷贝构造
    myString &operator=(const myString &other);//赋值
    ~myString() {
        cout << ps << " --Destructor" << endl;
        delete[] ps;
    }
    const char * c_str()const { return ps; }
private:
    char *ps;
};
myString::myString(const char *str)
{
    if (str == nullptr) {
        ps = new char[1]{ 0 };
        cout << ps << " --Default constructor" << endl;
    }
    else {
        int length = strlen(str);
        ps = new char[length + 1];
        strcpy(ps, str);
        cout << ps << " --Str constructor" << endl;
    }
}
```

```cpp
myString::myString(const myString &other)
{
    int length = strlen(other.ps);
    ps = new char[length + 1];
    strcpy(ps, other.ps);
    cout << ps << " --Copy constructor" << endl;
}
myString &myString::operator=(const myString &other)
{
    if (this != &other) {
        delete[] ps;
        int length = strlen(other.ps);
        ps = new char[length + 1];
        strcpy(ps, other.ps);
    }
    cout << ps << " --Copy assignment" << endl;
    return *this;
}
myString operator+(const myString &a, const myString &b){
    int length = strlen(a.c_str()) + strlen(b.c_str());
    char * p = new char[length + 1];
    strcpy(p, a.c_str());
    strcat(p, b.c_str());
    myString tmp(p);
    delete[] p;
    return tmp;
}
```

```cpp
myString fun() {
    myString tmp("allok");
    return tmp;
}
int main() {
    myString s1("abc");
    myString s2("123");
    myString s3 = s1; //拷贝构造
    cout << "1====================" << endl;
    myString s4 = s1 + s2;
    // s1 + s2 是临时量,拷贝给s4后,马上析构了
    cout << "2====================" << endl;
    myString s5 = fun();
    cout << "3====================" << endl;
    return 0;
}
```

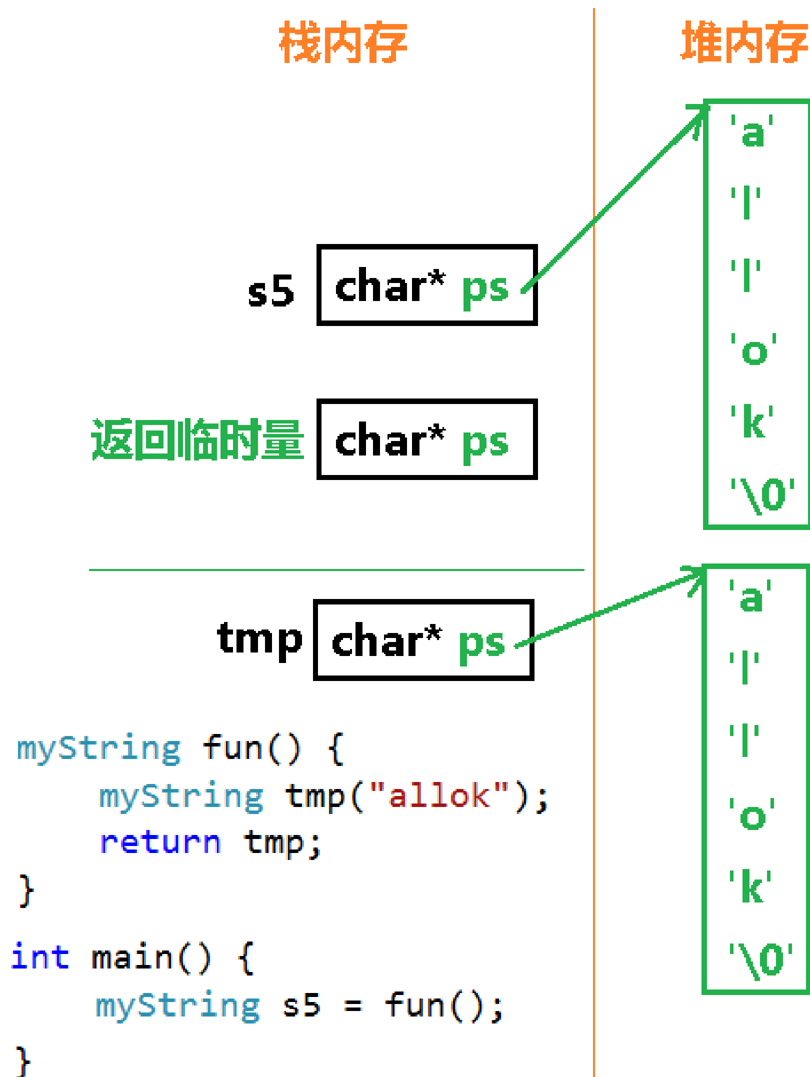
```
abc --Str constructor
123 --Str constructor
abc --Copy constructor
1====================
abc123 --Str constructor
abc123 --Copy constructor
abc123 --Destructor
2====================
allok --Str constructor
allok --Copy constructor
allok --Destructor
3====================
allok --Destructor
abc123 --Destructor
abc --Destructor
123 --Destructor
abc --Destructor
请按任意键继续. . .
```

思考：临时量、函数返回值类型，浪费在哪里？

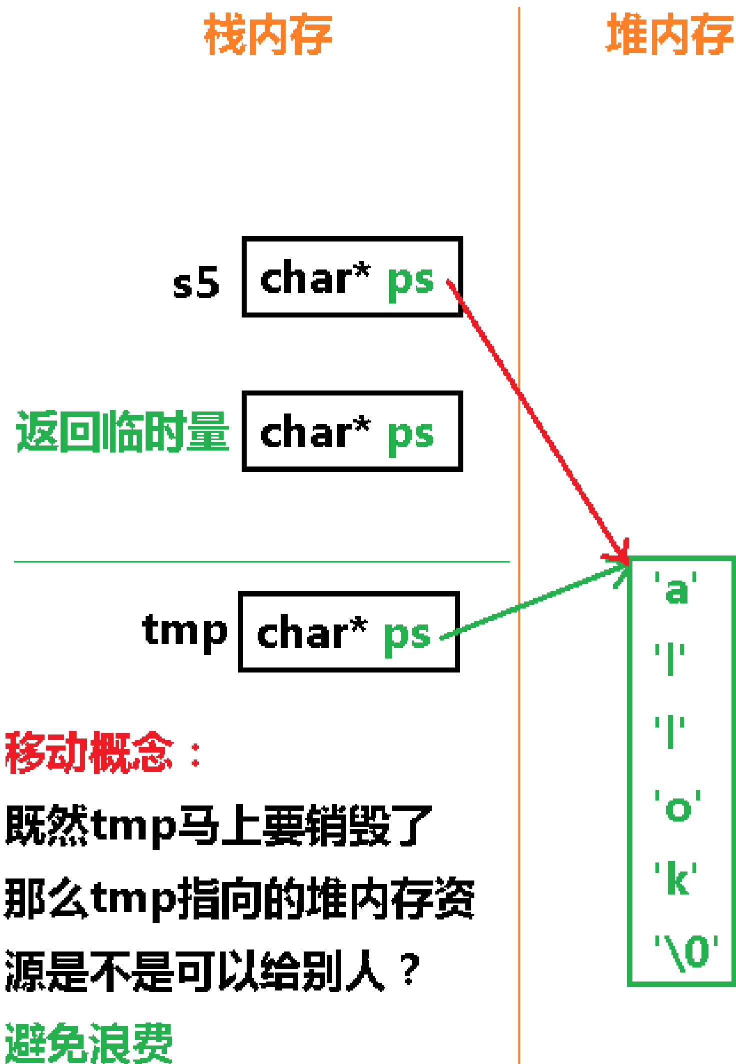
对象移动2

移动语义是C++11的特性之一，利用移动语义可以实现对象的移动而非拷贝。

在某些情况下，可以大幅度的提升性能。




```
myString fun() {
    myString tmp("allok");
    return tmp;
}
int main() {
    myString s5 = fun();
}
```



移动概念：

既然tmp马上要销毁了那么tmp指向的堆内存资源是不是可以给别人？

避免浪费

对象移动3

为了支持移动操作，引入“右值引用”。

右值引用：只能绑定到一个将要销毁的对象。因此：我们可以自由地将一个右值引用的资源“移动”到另外一个对象中。

左值持久，右值短暂。

由于右值引用只能绑定到临时对象：

1、所引用的对象将要被销毁；2、该对象没有其他用户。

上面的2个特性意味着：使用右值引用的代码可以自由地接管所引用的对象的资源。

左值、右值，左值引用、右值引用

左右值鉴别最简单的办法：左值可以用取地址操作符”&“获取地址，右值无法使用”&“。

int i=10;

int &r = i; //ok，标准的 左值引用

int &&rr = i; //错误，不能将一个右值引用绑定到一个左值上

//int &r2 = i *10; //错误，i*10是个临时量，内置类型有const属性,所以必须是 const int &r2 = i*10;

const int &r2 = i*10; //ok

int &&r3 = i *10; //ok,右值引用

对象移动4

int x = 0;   //对象实例，有名，x是左值
int* p = &++x; //可以取地址，++x是左值
++x = 10;   //前置++返回的是左值，可以赋值
//p = &x++;   //后置++操作返回一个临时对象，不能取地址或赋值，是右值，编译错误

函数返回非引用类型时，是个临时量，所以是右值。

注意：变量是左值，右值引用以后，相当于延长了临时量的生命周期，此时的临时量已经转换为左值了。
int && rr3 = i *10;
int &&rr4 = rr3; //错误, rr3已经是左值了！！

所以：**引用（包括左值引用，右值引用）习惯上用在参数传递 和 函数返回值**。

对象移动5

拷贝构造 和 移动构造

```
#include <iostream>
#include <cstring>
#include <string>
#include <utility>
using namespace std;

class myString {
public:
    myString(const char *str = nullptr); //构造
    myString(const myString &other);     //拷贝构造
    myString(myString &&other);          //移动构造
    myString &operator=(const myString &other);//赋值运算符重载
    myString &operator=(myString &&other); //移动赋值运算符重载
    ~myString() {
        if (ps)
            cout << ps << " --Destructor" << endl;
        else
            cout << "ps is NULL" << " --Destructor" << endl;
        delete[] ps;
    }
    const char * c_str()const { return ps; }
private:
    char *ps;
};
```

```
myString::myString(const char *str) {
    if (str == nullptr) {
        ps = new char[1]{ 0 };
        cout << ps << " --Default constructor" << endl;
    }
    else {
        int length = strlen(str);
        ps = new char[length + 1];
        strcpy(ps, str);
        cout << ps << " --Str constructor" << endl;
    }
}
```

```
myString::myString(const myString &other) {//拷贝构造
    int length = strlen(other.ps);
    ps = new char[length + 1];
    strcpy(ps, other.ps);
    cout << ps << " --Copy constructor" << endl;
}
```

```
myString::myString(myString &&other) :ps(other.ps) { //移动构造
    other.ps = nullptr; //要保证：移后源对象能正常析构
    cout << ps << " --Move constructor" << endl;
}
```

对象移动6

```
#include <iostream>
#include <cstring>
#include <string>
#include <utility>
using namespace std;

class myString {
public:
    myString(const char *str = nullptr); //构造
    myString(const myString &other);     //拷贝构造
    myString(myString &&other);          //移动构造
    myString &operator=(const myString &other);//赋值运算符重载
    myString &operator=(myString &&other); //移动赋值运算符重载
    ~myString() {
        if (ps)
            cout << ps << " --Destructor" << endl;
        else
            cout << "ps is NULL" << " --Destructor" << endl;
        delete[] ps;
    }
    const char * c_str()const { return ps; }
private:
    char *ps;
};
```

赋值运算符重载

移动赋值运算符重载

```
myString &myString::operator=(const myString &other) { //赋值运算符重载
    if (this != &other) {
        delete[] ps;
        int length = strlen(other.ps);
        ps = new char[length + 1];
        strcpy(ps, other.ps);
    }
    cout << ps << " --Copy assignment" << endl;
    return *this;
}
```

```
myString &myString::operator=(myString &&other) {//移动赋值运算符重载
    if (this != &other) {
        delete[] ps;
        ps = other.ps;
        other.ps = nullptr;
    }
    cout << ps << "--Move assignment" << endl;
    return *this;
}
```

对象移动7

std::move 显式调用，强制移动。int &&r = std::move(r1);
调用move以后，对r1只能赋值或者销毁，r1中的内容不再有意义。

```
myString operator+(const myString &a, const myString &b) {
    int length = strlen(a.c_str()) + strlen(b.c_str());
    char * p = new char[length + 1];
    strcpy(p, a.c_str());
    strcat(p, b.c_str());
    myString tmp(p);
    delete[] p;
    return tmp;
}

myString fun() {
    myString tmp("allok");
    return tmp;
}
```

```
abc --Str constructor
123 --Str constructor
abc --Copy constructor
1====================
ok --Str constructor
2====================
abc123 --Str constructor
abc123 --Move constructor
ps is NULL --Destructor
3====================
allok --Str constructor
allok --Move constructor
ps is NULL --Destructor
4====================
allok --Move constructor
00000000
5====================
allok --Destructor
ps is NULL --Destructor
abc123 --Destructor
ok --Destructor
abc --Destructor
123 --Destructor
abc --Destructor
请按任意键继续. . .
```

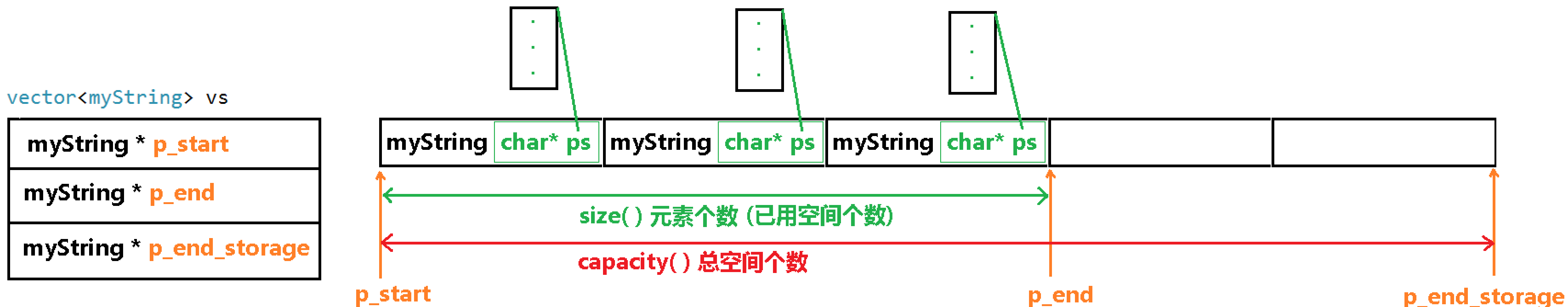
```
int main() {
    myString s1("abc");
    myString s2("123");
    myString s3 = s1; //拷贝构造
    cout << "1====================" << endl;
    myString s4(myString("ok"));
    cout << "2====================" << endl;
    myString s5 = s1 + s2;
    // s1 + s2 是临时量(右值),调用移动构造给s4,马上析构
    cout << "3====================" << endl;
    myString s6 = fun();
    // fun()返回值 是临时量(右值),调用移动构造给s6,马上析构
    cout << "4====================" << endl;
    myString s7 = std::move(s6); //std::move函数，#include <utility>
    cout << (void*)s6.c_str() << endl; //s6中的资源ps=NULL了。
    cout << "5====================" << endl;
    return 0;
}
```

vector的动态增长1

vector类似动态数组，所以在内存中是一段连续的内存。
观察 sizeof(vector<myString>) 、sizeof(vector<int>)、sizeof(vector<string>) 的大小，猜测vector的内存结构。

vector<myString> vs;
vs.size(); //此函数返回vector中的元素个数（已用空间数）
vs.capacity(); //此函数返回vector中的总空间个数
vs.reserve(n); //此函数预先分配一块指定大小为n个的内存空间。

思考：vector如何支持动态增长？



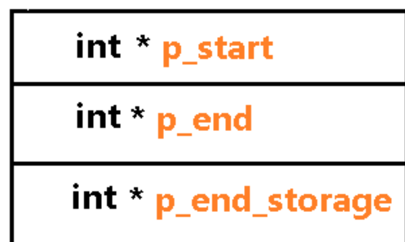
vector的动态增长2

添加元素时，如果vector空间大小不足，则会以原大小的1.5倍重新分配一块较大的新空间。

思考：动态增长的过程中发生了什么？

```
int main() {
    vector<int> vs;
    cout << "size=" << vs.size() << " capacity=" << vs.capacity() << endl;
    for (int i = 0; i < 10; i++) {
        vs.push_back(i);
        cout << "size=" << vs.size() << " capacity=" << vs.capacity() << endl;
    }
    cout << "====================" << endl;
    vector<int> vs1;
    vs1.reserve(4);
    cout << "size=" << vs1.size() << " capacity=" << vs1.capacity() << endl;
    for (int i = 0; i < 4; i++) {
        vs1.push_back(i);
        cout << "size=" << vs1.size() << " capacity=" << vs1.capacity() << endl;
        cout << &vs1[0] << endl; //观察连续内存的起始地址
    }
    vs1.push_back(4);
    cout << "size=" << vs1.size() << " capacity=" << vs1.capacity() << endl;
    cout << &vs1[0] << endl; //观察连续内存的起始地址
    cout << &(*vs1.begin()) << endl;
    return 0;
}
```

```
size=0 capacity=0
size=1 capacity=1
size=2 capacity=2
size=3 capacity=3
size=4 capacity=4
size=5 capacity=6
size=6 capacity=6
size=7 capacity=9
size=8 capacity=9
size=9 capacity=9
size=10 capacity=13
====================
size=0 capacity=4
size=1 capacity=4
003861B8
size=2 capacity=4
003861B8
size=3 capacity=4
003861B8
size=4 capacity=4
003861B8
size=5 capacity=6
003878B0
003878B0
请按任意键继续. . .
```



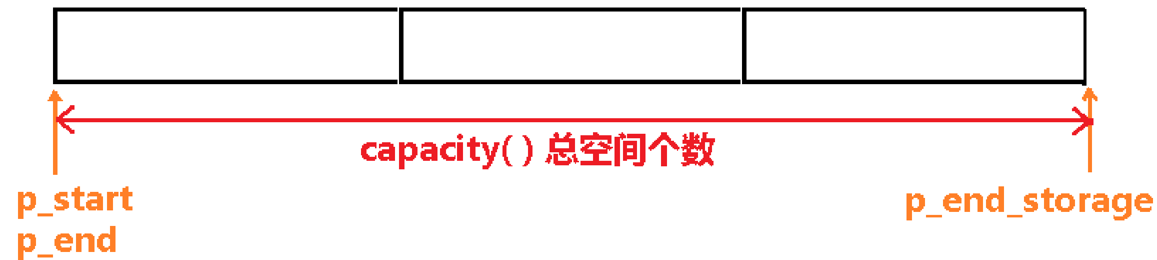
vector的动态增长3

重新分配一块较大的新空间后，将原空间内容拷贝过来，在新空间的内容末尾添加元素，并释放原空间。也就是说vector的空间动态增加大小，并不是在原空间之后的相邻地址增加新空间，因为vector的空间是线性连续分配的，不能保证原空间之后有可供配置的空间。

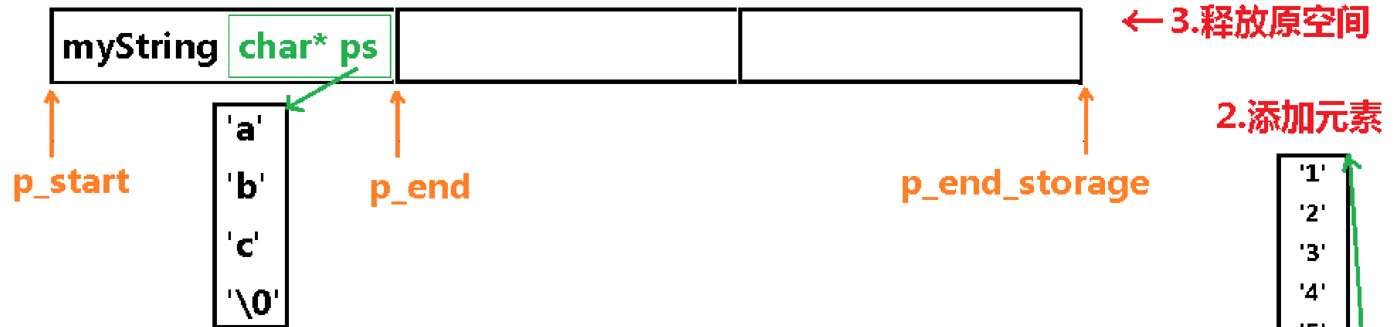
```
vector<myString> vs;
```

myString * p_start
myString * p_end
myString * p_end_storage

```
vs1.reserve(3);
```




```
vs1.push_back(myString("abc"));
```




```
vs1.push_back(myString("12345"));
```

1. 拷贝原空间数据

新地址：

vector的动态增长4

思考：
在只有拷贝构造、拷贝赋值运算符重载时，存在浪费吗？

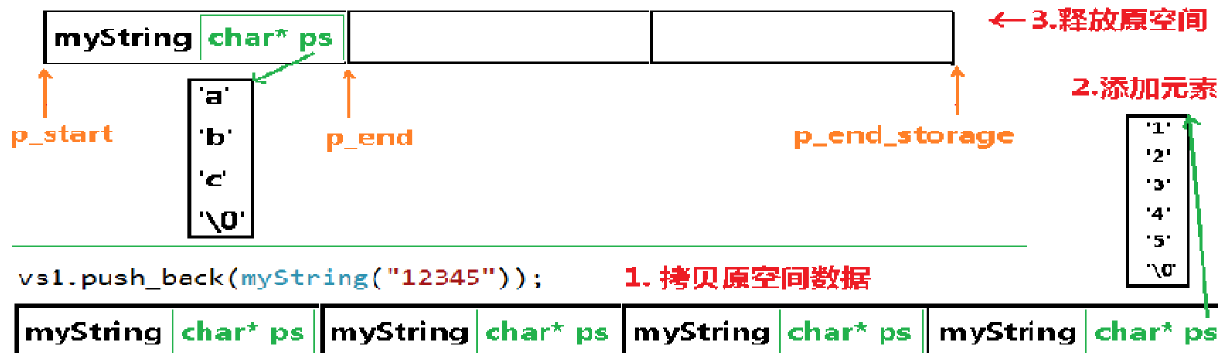
如何解决？

拷贝构造：

```
myString::myString(const myString &other){
int length = strlen(other.ps);
ps = new char[length + 1];
strcpy(ps, other.ps);
cout << ps << " --Copy constructor" << endl;
}
```

```
//myString 实现了普通构造，拷贝构造，赋值运算符重载，析构
int main() {
    vector<myString> vs1;
    vs1.reserve(3);
    cout << "size=" << vs1.size() << " capacity=" << vs1.capacity() << endl;
    for (int i = 0; i < 3; i++) {
        cout << "--------------------------" << endl;
        vs1.push_back(myString("abc"));
        cout << "size=" << vs1.size() << " capacity=" << vs1.capacity() << endl;
        cout << &vs1[0] << endl; //观察连续内存的起始地址
    }
    cout << "==========================" << endl;
    vs1.push_back(myString("12345"));
    cout << "size=" << vs1.size() << " capacity=" << vs1.capacity() << endl;
    cout << &vs1[0] << endl; //观察连续内存的起始地址
    cout << &(*vs1.begin()) << endl;
    cout << "==========================" << endl;
    return 0;
}
```

```
size=0 capacity=3
--------------------------
abc --Str constructor
abc --Copy constructor
abc --Destructor
size=1 capacity=3
003C5BF8
--------------------------
abc --Str constructor
abc --Copy constructor
abc --Destructor
size=2 capacity=3
003C5BF8
--------------------------
abc --Str constructor
abc --Copy constructor
abc --Destructor
size=3 capacity=3
003C5BF8
==========================
12345 --Str constructor
abc --Copy constructor
abc --Copy constructor
abc --Copy constructor
abc --Destructor
abc --Destructor
abc --Destructor
12345 --Copy constructor
12345 --Destructor
size=4 capacity=4
003C78B0
003C78B0
==========================
abc --Destructor
abc --Destructor
abc --Destructor
12345 --Destructor
请按任意键继续. . .
```



vector与移动1

```
//myString 实现了普通构造，拷贝构造，赋值运算符重载，析构
//         并且实现了 移动构造，移动赋值运算符重载
myString::myString(myString &&other) noexcept:ps(other.ps) { //移动构造
    other.ps = nullptr; //要保证：移后源对象能正常析构
    cout << ps << " --Move constructor" << endl;
}
int main() {
    vector<myString> vs1;
    vs1.reserve(3);
    cout << "size=" << vs1.size() << " capacity=" << vs1.capacity() << endl;
    for (int i = 0; i < 3; i++) {
        cout << "--------------------------" << endl;
        vs1.push_back(myString("abc"));
        cout << "size=" << vs1.size() << " capacity=" << vs1.capacity() << endl;
        cout << &vs1[0] << endl; //观察连续内存的起始地址
    }
    cout << "===========================" << endl;
    vs1.push_back(myString("12345"));
    cout << "size=" << vs1.size() << " capacity=" << vs1.capacity() << endl;
    cout << &vs1[0] << endl; //观察连续内存的起始地址
    cout << &(*vs1.begin()) << endl;
    cout << "===========================" << endl;
    return 0;
}
```

移动

```
size=0 capacity=3
--------------------------
abc --Str constructor
abc --Move constructor
ps is NULL --Destructor
size=1 capacity=3
00955BF8
--------------------------
abc --Str constructor
abc --Move constructor
ps is NULL --Destructor
size=2 capacity=3
00955BF8
--------------------------
abc --Str constructor
abc --Move constructor
ps is NULL --Destructor
size=3 capacity=3
00955BF8
===========================
12345 --Str constructor
abc --Move constructor
abc --Move constructor
abc --Move constructor
ps is NULL --Destructor
ps is NULL --Destructor
ps is NULL --Destructor
12345 --Move constructor
ps is NULL --Destructor
size=4 capacity=4
00955778
00955778
===========================
abc --Destructor
abc --Destructor
abc --Destructor
12345 --Destructor
请按任意键继续. . .
```

拷贝

```
size=0 capacity=3
--------------------------
abc --Str constructor
abc --Copy constructor
abc --Destructor
size=1 capacity=3
003C5BF8
--------------------------
abc --Str constructor
abc --Copy constructor
abc --Destructor
size=2 capacity=3
003C5BF8
--------------------------
abc --Str constructor
abc --Copy constructor
abc --Destructor
size=3 capacity=3
003C5BF8
===========================
12345 --Str constructor
abc --Copy constructor
abc --Copy constructor
abc --Copy constructor
abc --Destructor
abc --Destructor
abc --Destructor
12345 --Copy constructor
12345 --Destructor
size=4 capacity=4
003C78B0
003C78B0
===========================
abc --Destructor
abc --Destructor
abc --Destructor
12345 --Destructor
请按任意键继续. . .
```

利用移动语义可以实现对象的移动而非拷贝。在某些情况下，可以大幅度的提升性能。

vector与移动2

自己编写的移动函数，最好加上**noexcept**。

vector保证：在调用push_back时发生异常，vector自身不会发生改变。

push_back可能会要求vector重新分配新内存，然后将元素对象从旧内存移动或者拷贝到新内存中。
假如使用拷贝，那么拷贝到一半的时候出现异常的话，由于旧内存的所有内容都还在，所以vector可以很容易恢复到原始状态。
假如使用移动，那么移动到一半的时候出现异常的话，旧空间的部分元素已经被改变了，而新空间中未构造的元素还不存在，此时vector将不能满足自身保持不变的要求。
为了避免这样的情况发生，除非vector知道元素类型的移动函数不会抛出异常，否则在重新分配内存的时候会使用拷贝构造而不是移动构造。
所以，**通常将移动构造函数和移动赋值运算符重载标记为 noexcept**。

```
class myString {
    ......
    myString(myString &&other) noexcept; //移动构造
    ......
}
myString::myString(myString &&other) noexcept:ps(other.ps){
    other.ps = nullptr;
}
```

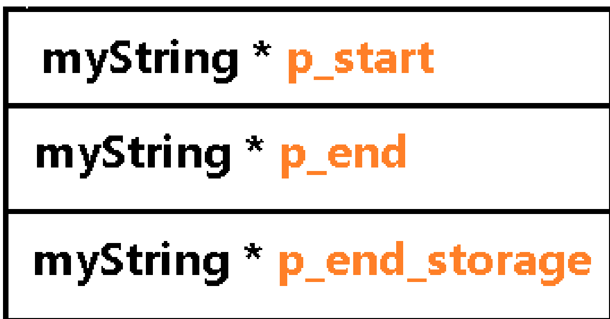
vector与移动3

vector 作为值返回，由于有移动语义，一点也不浪费！很自然的值返回。

标准库容器、string、shared_ptr类即支持移动也支持拷贝，IO类和unique_ptr类可以移动但不能拷贝。

```cpp
void fun1(vector<myString> &vs) {
    vs.push_back(myString("abc"));
    //do something...
}
vector<myString> fun2() {
    vector<myString> vs;
    vs.push_back(myString("123"));
    //do something...
    return vs;
}
int main() {
    //没有引入移动语义时，习惯这样用：
    //先准备好 vector, 然后用引用作为参数传递给函数
    vector<myString> vs1;
    fun1(vs1);
    cout << "===========================" << endl;
    //引入移动语义以后，这样写也非常ok
    vector<myString> vs2 = fun2();
    cout << "===========================" << endl;
    return 0;
}
```

```
abc --Str constructor
abc --Move constructor
ps is NULL --Destructor
=======================
123 --Str constructor
123 --Move constructor
ps is NULL --Destructor
=======================
123 --Destructor
abc --Destructor
请按任意键继续. . .
```



这里使用的是vector的移动函数。

移动小结

合成的移动函数：
1.自己没有定义拷贝构造、赋值运算符重载和析构函数；
2.类中所有非static数据成员都可移动时；同时满足上面两个条件，编译器会合成默认的移动函数。

移后源对象必须可析构；
移动右值，拷贝左值；
拷贝参数：const T& other 移动参数： T&& other
如果没有移动函数，右值也会被拷贝；

std::move
使用move可大幅提高性能，但是要小心使用 move操作，要绝对确认移后源对象没有其他用户。

为了效率，实现移动函数。比如自己写一个String类。