vector<自定义类>中,自定义类的拷贝构造/赋值

2015年11月29日 星期日　　下午8:52

一、简介

vector支持vector级别的赋值操作，如vector<int>v1,v2;v1=v2;
vector<基本类型>v1，执行此操作没问题；但是vector<自定义类>v1,v2;v1=v2;会调用自定义类的拷贝构造函数，(比如你没有显式实现，而是使用默认的，是浅拷贝，可能会存在隐患)。

另外v1.push_back(t1)的时候也是调用拷贝构造函数，而v1[1]=t2，是调用赋值函数。(t1、t2是自定义类的对象)

这部分内容网上没有很好的资料，主要靠实验。

二、实验现象及结论

1、vector之间=赋值

1.1 调用自定义类的拷贝构造函数

vector之间支持v1=v2的操作，会把v2的内容拷给v1，这个拷贝的过程是调用vector元素的拷贝构造函数。自定义类Type的拷贝构造函数默认是浅拷贝，遇到成员变量有指针类型可能会出问题。(详见另一篇笔记"拷贝构造函数/赋值函数")

1.2 调用拷贝构造函数而不是赋值函数

不论vector级别调用的是拷贝构造函数(如:vector<Type>v1(v2))还是赋值函数，还是赋值函数(如:vector<Type>v1,v2;v1=v2;)，都是会调用Type的拷贝构造函数，而不是赋值函数。仔细想想就知道，其实不论，怎样把v2拷给v1，都是把v1清空，v2的每个元素再逐个拷贝进来。而不会是穿插一段v2的元素对v1的元素的赋值操作。
这个结论是实验得出的。

2、vector的push_back操作

vector<Type>v1;Type t1;v1.push_back(t1);
这个是调用Type的拷贝构造函数。
其实也好理解，因为拷贝之前v1中没有t1的空间，t1来了之后v1要新开一个空间，最适合调用拷贝构造函数。

3、vector的按下标赋值操作

函数

vector<Type>v1; Type t1; v1[1] = t1;(在v1空间允许的情况下，使用下标做赋值)
这个一般调用Type的赋值函数。
分情况讨论：
A 如果拷贝构造函数和赋值函数都没有写，使用默认的 => 使用默认赋值函数
B 如果拷贝构造函数和赋值函数都实现了 => 使用赋值函数
C 如果拷贝构造函数实现了，赋值函数没写，要使用默认的 => 使用拷贝构造函数

C的用法不提倡，比较少见，因为一般拷贝构造函数和赋值函数有一个需要重载的话，另一个也需要。但是实验得出C的结论。

三、实验地址

1、实验地址

work@nj01-nlp-test01.nj01.baidu.com
/home/work/tianzhiliang/test/cpp/vector_copy_constructor
详见附图

2、实验方法

可以通过对一些代码注释/去掉注释，来尝试不同的策略，形成对比实验，这些代码包含：

A 重载拷贝构造函数(声明/实现)
B 重载赋值函数(声明/实现)
C 这两个函数中深/浅拷贝

3、可以观察现象的地方

A v2、v3、v4改变之后，v1~v4值改变的情况
B test push_back功能里，通过打印log，看调用的是哪个函数
C test = operator功能里，通过打印log，看调用的是哪个函数

四、参考资料

这部分内容网上没有很好的资料，主要靠实验，下面这两篇博文参考意义也不大

1、vector间赋值 http://blog.chinaunix.net/uid-20760757-id-1872378.html
2、vector间赋值 http://blog.csdn.net/earbao/article/details/44492185

五、附图(代码)

class.h

```h
#include <stdlib.h>
#include <stdio.h>

#include <vector>
#include <string>
#include <iostream>
#include <new>

class Type{
    public:
        Type();
        Type(int id, int len, int num1, int num2, int num3, std::string name);
        Type(const Type& type);
        Type &operator=(const Type& type);

        int print();

        int _id;
        int _len;
        int* _nums;
        std::string _name;

};
```

class.cpp

```cpp
#include "class.h"

Type::Type(){
}

Type::Type(int id, int len, int num1, int num2, int num3, std::string name){
    _id = id;
    _len = len;
    _nums = new int[len];
    _nums[0] = num1;
    _nums[1] = num2;
    _nums[2] = num3;
    _name = name;
}

int Type::print(){
    std::cout << "name:" << _name << " id:" << _id << " len: " << _len << " nums:";
    for (int i = 0; i < _len; i++)
    {
        std::cout << _nums[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}

Type::Type(const Type& type){
    std::cout << "Type's Copy Constructor" << std::endl;
    _id = type._id;
    _len = type._len;
    _name = type._name;

    //shallow copy
    //_nums = type._nums;

    //deep copy
    _nums = new int[type._len];
    for (int i = 0; i < type._len; i++)
    {
        _nums[i] = type._nums[i];
    }
}

Type& Type::operator = (const Type& type){
    //shallow copy
    /*_id = type._id;
    _len = type._len;
    _name = type._name;
    _nums = type._nums;*/

    //deep copy
    if (this == &type)
    {
```

```
54          return *this;
55      }
56
57      delete[] _nums;
58
59      std::cout << "Type's Assign Operator" << std::endl;
60      _id = type._id;
61      _len = type._len;
62      _name = type._name;
63      _nums = new int[type._len];
64      for (int i = 0; i < type._len; i++)
65      {
66          _nums[i] = type._nums[i];
67      }
68
69      return *this;
70  }
```

test.cpp

```
23  //int print_type_vector(const std::vector<Type>& vec, std::string name){
24  int print_type_vector(std::vector<Type>& vec, std::string name){
25      int len = vec.size();
26      std::cout << "Name:" << name << std::endl;
27      for (int i = 0; i < len; i++)
28      {
29          vec[i].print();
30      }
31      std::cout << std::endl;
32
33      return 0;
34  }
35
36  int test_my_class(){
37      Type t1(1, 3, 0, 1, 2, "t1");
38      Type t2(2, 3, 3, 4, 5, "t2");
39      Type t3(3, 3, 6, 7, 8, "t3");
40
41      //t1.print();
42      //t2.print();
43      //t3.print();
44
45      std::vector<Type>v1;
46      v1.push_back(t1);
47      v1.push_back(t2);
48
49      std::vector<Type>v2 = v1;//vector's copy construstor, C style, will call Type's copy construstor
50      std::vector<Type>v3(v1);//vector's copy construstor, C++ style, will call Type's copy construstor
51
52      std::vector<Type>v4;//vector's assign operator, but it will call Type's copy construstor
53      v4.push_back(t2);
54      v4 = v1;
55
56      print_type_vector(v1, "v1");
57      print_type_vector(v2, "v2");
58      print_type_vector(v3, "v3");
59      print_type_vector(v4, "v4");
60
61      v2[0]._nums[0] = 1002;//change value to test
62      v3[0]._nums[0] = 1003;
63      v4[0]._nums[0] = 1004;
64
65      print_type_vector(v1, "v1");
66      print_type_vector(v2, "v2");
67      print_type_vector(v3, "v3");
68      print_type_vector(v4, "v4");
69
70      std::cout << "test push_back" << std::endl;
71      v4.push_back(t3);
72      std::cout << std::endl;
73      //v4.push_back(t2);
74      std::cout << std::endl;
75      v4.push_back(t1);
76      //Conclusion: vector's push_back will call Type's copy construstor
77
78      std::cout << "test = operator" << std::endl;
79      v4[0] = t1;
80      //Conclusion: vector's assgin will call Type's assign operator
81      return 0;
82  }
83
84  int main(){
85      //test_base_type();
```

```
86      test_my_class();
87      return 0;
88  }
```