Assignment 5

Fu-yun Wang

191300051

一.概念题

1.1 C++所提供的隐式赋值操作存在什么样的问题？如何解决这样的问题?

C++中对两个对象的赋值操作，是将其中一个对象的所有数据成员逐个赋值给另外一个对象。对于普通的数据成员，将使用普通的赋值操作。对于成员对象，则调用该成员对象的赋值操作。

问题：

如果出现了指针数据成员，那么会导致两个对象的指针成员都指向同一块内存区域，修改操作会相互影响，同时消亡时还会导致空间被释放两次。

1.2 请简述拷贝构造函数与赋值操作符"="重载函数的区别.

> *注意return reference 是一个lvalue，所以当对等于号进行操作符重载时，最好返回引用，这样能够满足(a=b)=c. 实际上a=b=c不论是return value还是reference都能够成立，但是即使是这样，returnvalue仍然会两次的调用拷贝构造函数，导致速度较慢。*

拷贝函数是用在已存在对象（的各成员当前值）来创建一个相同的新对象（尚不存在，正在创建）。

赋值运算符重载函数要把一个已存在的对象（的各成员当前值）赋值给另一个已存在的同类对象（已经存在，不需要再创建）。

1.3 为什么会对操作符new和delete进行重载?

主要是为了提高效率。以教材中代码为例，我们可以利用new和delete的重载实现一种高效的堆区管理结构，如下代码所示。

1.4 C++是如何实现λ表达式的?

使用对于类中函数调用操作符的重载--函数对象实现。

- 首先隐式的定义一个类
  - 数据成员对应到该类的环境变量，使用构造函数对其进行初始化
  - 重载函数调用操作符，重载函数按照λ表达式对应的功能来实现。
- 创建上述类的一个临时对象
- 在使用λ表达式的地方用该对象来代替。

(lambda表达式不仅可以作为参数，传给sort等函数，还可以使用auto func=[] (int a,int b)->int{return a+b;})，然后调用func()来使用。

二.编程题

2.1 完成int型矩阵类Matrix的实现，要求补充 '?' 处内容并完成如下 的接口.

```
#include <iostream>
#include <cstring>
#include <assert.h>
using namespace std;
//一个类本身就是自己的友元.
class Matrix
{
    int *p_data;  //表示矩阵数据
    int row, col; //表示矩阵的行数和列数
public:
    Matrix(int r, int c, const int a[]);
    void display();
    Matrix(int r, int c);                       //构造函数
    ~Matrix();                                  //析构函数
    int &operator[](int i);                     //重载[], 对于Matrix对象m, 能够通过m[i][j]访问第i+1行、第j+1列元素
    Matrix &operator=(const Matrix &m);         //重载=, 实现矩阵整体赋值, 若行/列不等, 归还空间并重新分配
    bool operator==(const Matrix &m) const;   //重载==, 判断矩阵是否相等
    Matrix operator+(const Matrix &m) const; //重载+, 完成矩阵加法, 可假设两矩阵满足加法条件(两矩阵行、列分别相等)
    Matrix operator*(const Matrix &m) const; //重载*, 完成矩阵乘法, 可假设两矩阵满足乘法条件(this.col = m.row)
```

```
};
Matrix::Matrix(int r, int c)
{
    p_data = new int[r * c];
    row = r;
    col = c;
}
Matrix::Matrix(int r, int c, const int a[])
{
    row = r;
    col = c;
    p_data = new int[r * c];
    for (int i = 0; i < r * c; i++)
    {
        p_data[i] = a[i];
    }
}
void Matrix::display()
{
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            cout << p_data[i * col + j] << " ";
        }
        cout << endl;
    }
}
Matrix::~Matrix()
{
    delete p_data;
}
int &Matrix::operator[](int i)
{
    return p_data[i * row];
}
Matrix &Matrix::operator=(const Matrix &m)
{
    if (m.col != col || m.row != row)
    {
        col = m.col;
        row = m.row;
    }
    memcpy(p_data, m.p_data, col * row * sizeof(int));
}
bool Matrix::operator==(const Matrix &m) const
{
    if (col != m.col || row != m.row)
    {
```

```
69          return false;
70      }
71      for (int i = 0; i < row * col; i++)
72      {
73          if (p_data[i] != m.p_data[i])
74          {
75              return false;
76          }
77      }
78      return true;
79  }
80  Matrix Matrix::operator+(const Matrix &m) const
81  {
82      assert(m.col == col && m.row == row);
83      Matrix tmp(row, col);
84      for (int i = 0; i < col * row; i++)
85      {
86          tmp.p_data[i] = p_data[i] + m.p_data[i];
87      }
88      return tmp;
89  }
90  Matrix Matrix::operator*(const Matrix &m) const
91  {
92      Matrix tmp(row, m.col);
93      memset(tmp.p_data, 0, row * m.col * sizeof(int));
94      for (int i = 0; i < row * m.col; i++)
95      {
96          int r = i / m.col;
97          int c = i % m.col;
98          for (int j = 0; j < col; j++)
99          {
100             tmp.p_data[i] += p_data[r * col + j] * m.p_data[j *
    m.col + c];
101         }
102     }
103     return tmp;
104 }
105 int main()
106 {
107     const int a[6] = {1, 1, 1, 1, 1, 1};
108     Matrix A(2, 3, a);
109     Matrix B(3, 2, a);
110     A.display();
111     B.display();
112     cout << (A == B) << endl;
113     Matrix C(1, 1);
114     C = A * B;
115     C.display();
116 }
```

```
117
```

2.2 设计一种能解决教材例6-10中把存储块归还到堆空间的方法.(提 示：可以在每次申请存储块时多申请一个能存储一个指针的空间，用 该指针把每个存储块链接起来.)

```
#include <cstring>
#include <iostream>
#include <vector>
#include <stdlib.h>
using namespace std;
const int NUM = 32;
class A
{
    //对于静态的成员函数与变量，只在声明的时候加入static，在定义的时候不加。
    //同时，必须将其声明预定义分开写。
public:
    static void *operator new(size_t size);
    static void operator delete(void *p);
    static void give_back();

private:
    A *next;           //用于组织A类对象自由空间结点的链表。
    static A *p_free; //用于指向A类对象的自由空间链表头。
    static vector<A *> blocks;
};
A *A::p_free = NULL;
vector<A*> A::blocks;
void *A::operator new(size_t size)
{
    if (p_free == NULL)
    {                                          //申请NUM个A类对象的大空间。
        p_free = (A *)malloc(size * NUM); //一个动态数组
        blocks.push_back(p_free);
        //在大空间上建立自由结点链表。
        for (int i = 0; i < NUM - 1; i++)
            p_free[i].next = &p_free[i + 1];
        p_free[NUM - 1].next = NULL;
    }
    //从链表中给当前对象分配空间
    A *p = p_free;
    p_free = p_free->next;
    memset(p, 0, size);
    return p;
}
void A::operator delete(void *p)
{
```

```
42      ((A *)p)->next = p_free;
43      p_free = (A *)p;
44  }
45  void A::give_back()
46  {
47      printf("give back!.blocksize=%d",blocks.size());
48      for (int i = 0; i < blocks.size(); i++)
49      {
50          free(blocks[i]);
51      }
52      blocks.erase(blocks.begin(), blocks.end());
53  }
54
55  int main()
56  {
57      A *q1 = new A;
58      A *q2 = new A;
59      delete q1;
60      A::give_back();
61  }
```