Solution5

191220008 陈南疃

概念题

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++所提供的隐式赋值操作执行的是缺省定义的缺省的赋值运算。所谓缺省的赋值运算，是指对象中的所有位于stack中的域，进行相应的复制。但是，如果对象有位于heap上的域的话，其不会为拷贝对象分配heap上的空间，而只是指向相同的heap上的同一个地址。这样，当在一个对象中对这些成员进行操作时，会对另一个对象产生影响。

解决上面问题的办法是自己定义赋值操作符重载函数：如果对象会在 heap 上存在内存域，则我们必须重载赋值运算符，从而在进行对象的赋值操作时，使不同对象的成员域指向不同的 heap 地址。

2、请简述拷贝构造函数与赋值操作符"="重载函数的区别。

如果对象还不存在，在声明的同时将另一个已存在的对象赋给它，就会调用拷贝构造函数。

如果对象已经存在，然后将另一个已存在的对象赋给它，调用的就是赋值运算符重载函数。

3、为什么会对操作符new和delete进行重载？

系统提供的new和delete操作所涉及的空间分配和释放是通过系统的堆区管理系统来进行的，效率常常不高。

可以对操作符new和delete进行重载，使得程序能以自己的方式来实现动态对象空间的分配和释放功能。

4、C++是如何实现λ表达式的？

在C++中，λ表达式是通过函数对象来实现的：

对于λ表达式：`[...](int x)->int { ..... }`

首先，隐式定义一个类：

- 数据成员对应用到的环境变量，用构造函数对其初始化。
- 重载了函数调用操作符，重载函数按相应λ表达式的功能来实现。

然后，创建上述类的一个临时对象（设为obj）

最后，在使用上述λ表达式的地方用该对象来替代：

- 作用于实参进行函数调用 cout << obj(3);
- 传给其它函数 f(obj);

编程题

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。

```
class Matrix
{
    ? 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); //构造函数
    ~Matrix(); //析构函数
    ? &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)
};
```

```
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); //构造函数
    Matrix(const Matrix& m); //拷贝构造函数
    ~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)
{
    row = r;
    col = c;
    p_data = new int* [row];
    for (int i = 0; i < row; i++)
    {
        p_data[i] = new int[col];
        for (int j = 0; j < col; j++)
            p_data[i][j] = 0;
    }
}
```

```
Matrix::Matrix(const Matrix& m)
{
    row = m.row;
    col = m.col;
    p_data = new int* [row];
    for (int i = 0; i < row; i++)
    {
        p_data[i] = new int[col];
        for (int j = 0; j < col; j++)
        {
            p_data[i][j] = m.p_data[i][j];
        }
    }
}

Matrix::~Matrix()
{
    for (int i = 0; i < row; i++)
        delete[]p_data[i];
    delete[]p_data;
}

int*& Matrix::operator[] (int i)
{
    return p_data[i];
}

Matrix& Matrix::operator = (const Matrix& m)
{
    if (row != m.row || col != m.col)
    {
        this->~Matrix();
        p_data = new int* [m.row];
        for (int i = 0; i < m.row; i++)
        {
            p_data[i] = new int[m.col];
            for (int j = 0; j < m.col; j++)
                p_data[i][j] = 0;
        }
        row = m.row;
        col = m.col;
    }
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            p_data[i][j] = m.p_data[i][j];
        }
    }
    return *this;
}

bool Matrix::operator == (const Matrix& m) const
{
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
```

```
            if (p_data[i][j] != m.p_data[i][j])
                return false;
        }
    }
    return true;
}

Matrix Matrix::operator + (const Matrix& m) const
{
    Matrix matrix(row, col);
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            matrix.p_data[i][j] = p_data[i][j] + m.p_data[i][j];
        }
    }
    return matrix;
}

Matrix Matrix::operator * (const Matrix& m) const
{
    Matrix matrix(row, m.col);
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < m.col; j++)
        {
            for (int k = 0; k < col; k++)
            {
                matrix.p_data[i][j] += p_data[i][k] * m.p_data[k][j];
            }
        }
    }
    return matrix;
}
```

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）

```
const int NUM = 32;

class A
{
public:
    static void* operator new(size_t size);
    static void operator delete(void* p);
private:
    static A* p_free;
    A* next;
};

struct Head // 存储块头的结点
{
    A* head;
```

```
    Head* next;
};

A* A::p_free = NULL;
Head* q = NULL; // 存储块头的链表的表头

void* A::operator new(size_t size)
{
    A* p;
    if (p_free == NULL)
    {
        p_free = (A*)malloc(size * (NUM + 1)); // 多申请一个空间，作为块头
        if (q == NULL) // 链接块头
        {
            q = new Head;
            q->head = p_free;
            q->next = NULL;
        }
        else
        {
            Head* r = q;
            while (r->next != NULL)
                r = r->next;
            r->next = new Head;
            r->next->head;
            r->next->next = NULL;
        }
        p_free++; // 修正p_free
        for (p = p_free; p != p_free + NUM - 1; p++)
            p->next = p + 1;
        p->next = NULL;
    }
    p = p_free;
    p_free = p_free->next;
    memset(p, 0, size);
    return p;
}

void A::operator delete(void* p)
{
    ((A*)p)->next = p_free;
    p_free = (A*)p;
}

void delete_myheap()
{
    Head* p = q;
    Head* pre = p;
    while (p != NULL)
    {
        delete[]p->head; // 归还当前存储块
        pre = p;
        p = p->next;
        delete pre; // 归还存储当前块头指针的结点
    }
}
```