

C++基础知识实验

因为是4*5的矩阵，在动态申请数组的时候，需要先动态申请4个二维int指针，再对每个二维int指针动态申请五个一维int指针。

```
int **matrix = new int*[4];
for (int i = 0; i < 4; i++)
{
    matrix[i] = new int[5];
}
```


在程序运行过程中要及时释放动态申请的内存，对于二维指针来说，同样需要先释放每个一维指针，再释放4个二维指针。

```
for (int i = 0; i < 4; i++)
{
    delete []matrix[i];
}
delete []matrix;
```

以矩阵加法为例，返回值为二维int指针。**为什么在函数内部定义的变量，可以直接作为返回值呢？不会发生内存访问错误吗？**答案是不会，因为动态申请内存是在堆空间，而函数返回值之后释放的是栈空间，因此在函数内申请的动态数组不会被释放，在函数外依然有效。

```
int ** add_matrix(int **A1, int ** A2)
{
    cout << "-----执行矩阵加法-----" << endl;
    int **A3 = new int*[4];
    for (int i = 0; i < 4; i++)
    {
        A3[i] = new int[5];
    }
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            A3[i][j] = A1[i][j] + A2[i][j];
        }
    }
    return A3;
}
```

实验结果:

 C:\Users\10343\source\repos\C++程序设计\Debug\C++程序设计.exe

```
请输入4*5的矩阵:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1      2      3      4      5
6      7      8      9      10
11     12     13     14     15
16     17     18     19     20
请输入4*5的矩阵:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1      2      3      4      5
6      7      8      9      10
11     12     13     14     15
16     17     18     19     20
-----执行矩阵加法-----
2      4      6      8      10
12     14     16     18     20
22     24     26     28     30
32     34     36     38     40
-----执行矩阵减法-----
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
请按任意键继续. . .
```

类与对象实验

2.1

构造函数和析构函数的分析

有参构造

```
Circle a(3,4,5);
Circle b(0,0,1);
intersect(a, b);
```

```
Point正在调用有参构造函数...
正在调用圆心为3 4的Circle的有参构造函数...
Point正在调用有参构造函数...
正在调用圆心为0 0的Circle的有参构造函数...
Point正在调用拷贝构造函数...
Point正在调用拷贝构造函数...
Point正在调用析构函数...
Point正在调用析构函数...
相交
请按任意键继续. . .
正在调用圆心为0 0的Circle的析构函数...
Point正在调用析构函数...
正在调用圆心为3 4的Circle的析构函数...
Point正在调用析构函数...
```

一个类中的构造函数的执行顺序：

1. 创建派生类的对象，基类的构造函数优先被调用
2. 类里有成员类，成员类的构造函数优先被调用
3. 类本身的构造函数

因此在构造一个Circle之前，先把它内部的所有成员类构造了，如Point类就优先被构造

第五行开始，是执行get_center函数时，return m_center返回一个类的时候，是返回一个m_center的拷贝，因此执行构造函数。

第七行，先是在cal_distance内，函数返回的时候，析构参数Point a

```
double Point::cal_distance(Point a)
{
    int ax = a.get_x(), ay = a.get_y();
    int bx = _x, by = _y;
    return sqrt(pow(ax - bx, 2) + pow(ay - by, 2));
}
```

第八行，是析构a.get_center得到的临时Point类

后续就是正常的析构顺序

无参构造

```
Circle a;
Circle b;
intersect(a, b);
```

```

Point正在调用无参构造函数...
请输入圆心坐标的横坐标: 3
请输入圆心坐标的纵坐标: 4
正在调用圆心为3 4的Circle的无参构造函数...
请输入圆的半径: 5
Point正在调用有参构造函数...
Point正在调用析构造函数...
Point正在调用无参构造函数...
请输入圆心坐标的横坐标: 0
请输入圆心坐标的纵坐标: 0
正在调用圆心为0 0的Circle的无参构造函数...
请输入圆的半径: 1
Point正在调用有参构造函数...
Point正在调用析构造函数...
Point正在调用拷贝构造函数...
Point正在调用拷贝构造函数...
Point正在调用析构造函数...
Point正在调用析构造函数...
相交
请按任意键继续. . .
正在调用圆心为0 0的Circle的析构造函数...
Point正在调用析构造函数...
正在调用圆心为3 4的Circle的析构造函数...
Point正在调用析构造函数...

```

在执行Circle的构造时，因为没指定Point的构造函数，因此程序自动无参构造了m_center，之后输入坐标完成后，因为m_center = Point(x, y);是产生了一个临时的Point类，因此是有参构造之后析构。

后面的构造、析构顺序同有参构造分析方法相同。

2.2

本实验中，我重载了操作符+、-、>>、<<。

```

// 重构等号，实现深拷贝
Matrix& operator=(const Matrix &b)
{
    //先释放干净本对象里的堆区

    //检测了自赋值
    if (this != &b)
    {
        if (m_matrix)
        {
            for (int i = 0; i < m_rows; i++)
            {
                delete[]m_matrix[i];
            }
            delete[]m_matrix;
        }

        //深拷贝
        m_matrix = new int*[b.m_rows];
        for (int i = 0; i < b.m_rows; i++)

```

```

    {
        m_matrix[i] = new int[b.m_lines];
    }
    for (int i = 0; i < m_rows; i++)
    {
        for (int j = 0; j < m_lines; j++)
        {
            m_matrix[i][j] = b.m_matrix[i][j];
        }
    }
}
return *this;
}

Matrix operator+(Matrix &b)
{
    cout << "-----执行矩阵加法-----" << endl;
    Matrix d(m_rows, m_lines);
    if (!(m_rows == b.get_rows() && m_lines == b.get_lines()))
    {
        cout << "两个矩阵大小不一致！" << endl;
        return d;
    }
    int **d_matrix = d.get_matrix(), **b_matrix = b.get_matrix();
    for (int i = 0; i < m_rows; i++)
    {
        for (int j = 0; j < m_lines; j++)
        {
            d_matrix[i][j] = b_matrix[i][j] + m_matrix[i][j];
        }
    }
    return d;
}

Matrix operator-(Matrix &b)
{
    cout << "-----执行矩阵减法-----" << endl;
    Matrix d(m_rows, m_lines);
    if (!(m_rows == b.get_rows() && m_lines == b.get_lines()))
    {
        cout << "两个矩阵大小不一致！" << endl;
        return d;
    }
    int **d_matrix = d.get_matrix(), **b_matrix = b.get_matrix();
    for (int i = 0; i < m_rows; i++)
    {
        for (int j = 0; j < m_lines; j++)
        {
            d_matrix[i][j] = -(b_matrix[i][j] - m_matrix[i][j]);
        }
    }
    return d;
}

ostream& operator<<(ostream &cout, Matrix p)
{
    cout << "正在输出矩阵" << p._id << "的值..." << endl;
    for (int i = 0; i < p.m_rows; i++)

```

```

    {
        for (int j = 0; j < p.m_lines; j++)
        {
            cout << p.m_matrix[i][j] << '\t';
        }
        cout << endl;
    }
    return cout;
}

istream& operator>>(istream &cin, Matrix &p)
{
    cout << "请输入矩阵" << p._id << "的值..." << endl;
    int ** m = p.get_matrix();
    for (int i = 0; i < p.get_rows(); i++)
    {
        for (int j = 0; j < p.get_lines(); j++)
        {
            cin >> m[i][j] ;
        }
    }
    return cin;
}

```

其中需要注意的是，在重构操作符=的时候，需要考虑自赋值的情况。如Matrix A = A的情况，如果不检测自赋值，则重构函数会先释放本对象的堆区，此时如果再将已经释放的堆区赋值给自己，则会产生错误。因此需要在释放堆区之前检测参数矩阵的指针和this是否相同。

构造函数和析构函数的分析

```

正在执行Matrix 的有参构造函数
正在执行Matrix 的有参构造函数
正在执行Matrix 的有参构造函数
请输入矩阵 的值...
12 11 10 9 8 7 6 5 4 3 2 1
正在执行Matrix 的拷贝构造函数
正在输出矩阵 的值...
12      11      10      9
8       7       6       5
4       3       2       1
正在执行Matrix 的析构函数
请输入矩阵 的值...
1 2 3 4 5 6 7 8 9 10 11 12
正在执行Matrix 的拷贝构造函数
正在输出矩阵 的值...
1       2       3       4
5       6       7       8
9       10      11      12
正在执行Matrix 的析构函数
正在执行Matrix 的有参构造函数
正在执行Matrix 的拷贝构造函数
正在执行Matrix 的析构函数
正在执行Matrix 的析构函数
正在执行Matrix 的拷贝构造函数
正在输出矩阵 的值...
13      13      13      13
13      13      13      13
13      13      13      13
正在执行Matrix 的析构函数

```

前三行即Matrix类的构造函数，第一个拷贝构造函数是重载操作符<<参数Matrix p的拷贝构造函数，析构是参数Matrix p的析构，下面的拷贝构造和析构同理。第一个有参构造是+的重载中的Matrix d(m_rows, m_lines);，拷贝函数是返回Matrix d的时候执行的。两个析构分别析构有参构造的d 和参数b。

引申：执行拷贝构造函数的情况

1. 用类的一个对象去初始化另一个对象
2. 函数的形参是类的对象（值传递）
3. 函数的返回值是类的对象或引用

继承与派生实验

3

首先我定义了基类Shape，Shape中的函数并没有实际意义

```

class Shape
{
public:
    Shape() { cout << "正在执行Shape的构造函数" << endl; }
    ~Shape() { cout << "正在执行Shape的析构函数" << endl; }

    int cal_area()
    {
        cout << "正在执行Shape的计算面积函数" << endl;
        return 0;
    }
};

```

Rectangle类和Circle类继承于Shape类，重构了Shape类的cal_area函数

```

class Circle :public Shape
{
private:
    double m_r;
public:
    Circle(int r) {
        cout << "正在执行Circle的构造函数" << endl;
        m_r = r;
    }
    ~Circle() { cout << "正在执行Circle的析构函数" << endl; }
    double cal_area() {
        cout << "正在执行Circle的计算面积函数" << endl;
        return pow(m_r,2)*3.1415;
    }
};

class Rectangle :public Shape
{
private:
    int m_x;
    int m_y;
public:
    Rectangle(int x,int y) {
        cout << "正在执行Rectangle的构造函数" << endl;
        m_x = x;
        m_y = y;
    }
    ~Rectangle() { cout << "正在执行Rectangle的析构函数" << endl; }
    int cal_area() {
        cout << "正在执行Rectangle的计算面积函数" << endl;
        return m_x * m_y;
    }
};

```

定义Square类继承于Rectangle类，重构Rectangle类的cal_area函数

```

class Square :public Rectangle
{
private:
    int m_x;
public:
    Square(int x):Rectangle(x,x) {

```




```

        cout << "正在执行Square的构造函数" << endl;
        m_x = x;
    }
    ~Square() { cout << "正在执行Square的析构函数" << endl; }
    int cal_area() {
        cout << "正在执行Square的计算面积函数" << endl;
        return pow(m_x, 2);
    };
};
};

```

执行结果:



```

C:\Users\10343\source\repos\C++程序
正在执行Shape的构造函数
正在执行Rectangle的构造函数
正在执行Square的构造函数
正在执行Square的计算面积函数
s的面积是:9
正在执行Rectangle的计算面积函数
s的面积是:9
正在执行Shape的构造函数
正在执行Rectangle的构造函数
正在执行Rectangle的计算面积函数
r的面积是:12
正在执行Shape的构造函数
正在执行Circle的构造函数
正在执行Circle的计算面积函数
c的面积是:28.2735
请按任意键继续. . .
正在执行Circle的析构函数
正在执行Shape的析构函数
正在执行Rectangle的析构函数
正在执行Shape的析构函数
正在执行Square的析构函数
正在执行Rectangle的析构函数
正在执行Shape的析构函数

```

根据之前分析的构造函数执行顺序，可以得知，基类的构造函数优先于继承类的构造函数，因为Square继承于Rectangle继承于Shape，所以执行顺序为Shape->Rectangle->Square。析构函数的执行顺序与构造函数相反。

IO流实验

使用srand设置随机数种子

```

srand(unsigned(time(NULL)));
int ans = rand()%1000+1;

```

使用isint函数来检查输入的合法性，仅当输入为数字，且为1-1000范围内的整数才有效。

```

//检查输入的合法性
bool isint(char s[])

```

```

{
    for (int i = 0; s[i] != '\0'; i++)
        if (s[i]<1||s[i]>255||!isdigit(s[i]))
            return false;
    return true;
}

//while内部的检查部分
if (cin >> input && !isint(input))
{
    cout << "输入无效! 请输入1-1000范围内的整数! " << endl;
    getchar();
    continue;
}
guess = atoi(input);

if (guess > 1000 || guess < 1)
{
    cout << "数据范围错误! 请输入1-1000范围内的整数! " << endl;
    continue;
}

```

根据用户输入的数据与答案的差值，对用户进行提示

```

if (guess > ans) {
    if (guess - ans < 100)
        cout << "猜的价格高了一点点" << endl;
    else if(guess -ans<300)
        cout << "猜的价格高了挺多" << endl;
    else
        cout << "猜的价格高了非常多" << endl;
}
else if (guess < ans) {
    if (ans - guess < 100)
        cout << "猜的价格低了一点点" << endl;
    else if (ans - guess < 300)
        cout << "猜的价格低了挺多" << endl;
    else
        cout << "猜的价格低了非常多" << endl;
}
else
{
    cout << "恭喜你猜对了!!!" << endl;
    break;
}

```

运行结果：

C:\Users\10343\source\repos'

```
请猜猜商品的价格：
100
猜的价格低了非常多
请猜猜商品的价格：
500
猜的价格低了挺多
请猜猜商品的价格：
700
猜的价格高了一点点
请猜猜商品的价格：
650
猜的价格低了一点点
请猜猜商品的价格：
675
猜的价格低了一点点
请猜猜商品的价格：
685
猜的价格低了一点点
请猜猜商品的价格：
695
猜的价格高了一点点
请猜猜商品的价格：
690
恭喜你猜对了!!!
请按任意键继续. . .
```

5.1

虚函数

区别：可以通过基类的指针直接访问派生类的函数。

抽象类

抽象类不能直接创造对象，会报错

```
// 抽象类不能实例化
shape shape;
```

抽象类子类必须实现纯虚函数，不然还是抽象类

定义抽象类shape类，包含了纯虚函数virtual double cal_area()。

```

//包含了纯虚函数，所以是抽象类
class Shape
{
public:
    Shape() { cout << "正在执行Shape的构造函数" << endl; }
    ~Shape() { cout << "正在执行Shape的析构函数" << endl; }
    // 纯虚函数
    virtual double cal_area() = 0;
};

```

Rectangle类继承于抽象类，但是类内部实现了Shape类的所有虚函数，因此Rectangle类不再是抽象类，可以创造对象，Circle类，Square类同理。

```

class Rectangle :public Shape
{
private:
    int m_x;
    int m_y;
public:
    Rectangle(int x, int y) {
        cout << "正在执行Rectangle的构造函数" << endl;
        m_x = x;
        m_y = y;
    }
    ~Rectangle() { cout << "正在执行Rectangle的析构函数" << endl; }
    virtual double cal_area() {
        cout << "正在执行Rectangle的计算面积函数" << endl;
        return m_x * m_y;
    }
};

```

定义一个函数calArea，参数是Shape类的对象，地址传递

```

void calArea(Shape & s)
{
    cout << s.cal_area() << endl;
}

```

当Rectangle类调用它时，实际上使用基类的指针调用了cal_area函数，因为cal_area是虚函数，因此用基类指针调用时，调用的是派生类Rectangle类的cal_area

运行结果：

```

正在执行Shape的构造函数
正在执行Rectangle的构造函数
正在执行Rectangle的计算面积函数
12
正在执行Shape的构造函数
正在执行Rectangle的构造函数
正在执行Square的构造函数
正在执行Square的计算面积函数
25
请按任意键继续. . .

```

5.2

```

class Point {
    friend ostream& operator<<(ostream &cout, Point p);
private:
    int x;
    int y;
public:
    Point(int xx, int yy);
    ~Point() { cout << "Point正在调用析构函数..." << endl; };
    void set_x(int xx) { x = xx; }
    void set_y(int yy) { y = yy; }
    int get_x(void) { return x; }
    int get_y(void) { return y; }
    double cal_distance(Point a);
    //前置
    Point& operator++()
    {
        cout << "正在执行前置++操作...\n";
        x++;
        y++;
        return *this;
    }
    //后置
    Point operator++(int)
    {
        cout << "正在执行后置++操作...\n";
        Point temp = *this;
        x++;
        y++;
        return temp;
    }
    Point& operator--()
    {
        cout << "正在执行前置--操作...\n";
        x--;
        y--;
        return *this;
    }
    Point operator--(int)
    {
        cout << "正在执行后置--操作...\n";

```

```

        Point temp = *this;
        x--;
        y--;
        return temp;
    }
};

```

前置++

返回值是class的引用，因为如果返回的是值，则遇到++(++a)的情况，再cout<<a只体现了自增一次的结果，因为括号内返回的不是a而是a的拷贝

先进行++运算再返回自身

后置

在参数里加int——占位参数，用于区分前置和后置，避免重定义（返回值不同不是函数重载的条件）

先保存原数据，再递增，再返回

后置递增返回值，如果返回引用会返回局部变量的引用，会报错

基础验收改进部分

相交函数未考虑两个圆包含的情况

重载= 的自赋值问题

原来的代码

```

// 重构等号，实现深拷贝
Matrix& operator=(const Matrix &b)
{
    //先释放干净本对象里的堆区
    if (m_matrix)
    {
        for (int i = 0; i < m_rows; i++)
        {
            delete[]m_matrix[i];
        }
        delete[]m_matrix;
    }

    //深拷贝
    m_matrix = new int*[b.m_rows];
    for (int i = 0; i < b.m_rows; i++)
    {
        m_matrix[i] = new int[b.m_lines];
    }
    for (int i = 0; i < m_rows; i++)
    {
        for (int j = 0; j < m_lines; j++)
        {

```

```

        m_matrix[i][j] = b.m_matrix[i][j];
    }
}
return *this;
}

```

如果是a=a的情况，则先delete掉了自己的二维指针，再用已经释放的二维指针给自己赋值，则会产生段错误。

解决方法

加一句判定，if (this != &b)

```

// 重构等号，实现深拷贝
Matrix& operator=(const Matrix &b)
{
    //先释放干净本对象里的堆区
    //检测了自赋值
    if (this != &b)
    {
        if (m_matrix)
        {
            for (int i = 0; i < m_rows; i++)
            {
                delete[] m_matrix[i];
            }
            delete[] m_matrix;
        }

        //深拷贝
        m_matrix = new int*[b.m_rows];
        for (int i = 0; i < b.m_rows; i++)
        {
            m_matrix[i] = new int[b.m_lines];
        }
        for (int i = 0; i < m_rows; i++)
        {
            for (int j = 0; j < m_lines; j++)
            {
                m_matrix[i][j] = b.m_matrix[i][j];
            }
        }
    }
    return *this;
}

```