

问题

实现二维矩阵 `Matrix` : 构造 `Matrix(r, c, val)`、`at(i, j)` 越界抛异常、`A+B` 维度不匹配抛异常、`transpose()` ; 主程序会把 `Matrix` 放进 `vector` 里频繁 `push_back/resize/赋值` (触发拷贝/移动/重分配)。规模 `r*c` 可到 `1e7` , 要求 move 友好、无泄漏、无 double free, C++20。

回答

1) 题意一句话

你要写一个“可被 `vector` 安全搬来搬去”的矩阵类型：资源所有权清晰、拷贝深拷贝、移动偷资源且 `noexcept`。

2) 致命点清单 (≥ 4)

1. `delete` 用错：`new[]` 必须配 `delete[]` , 现在 `~Matrix(){ delete p; }` 是 UB, 可能崩/泄漏。
 2. 浅拷贝 + `double free` : 编译器生成的拷贝/赋值只会拷 `p` 指针；两个对象析构都会释放同一块内存 → 偶现炸。
 3. `vector` 重分配触发拷贝：`v.resize(10)` 可能导致重新分配；老元素会被拷贝/移动到新内存。你没写正确移动/拷贝，默认浅拷贝 → 重分配后旧对象析构把 `p` 释放，新对象还指向已释放内存 → 访问/析构必炸。
 4. 缺少拷贝赋值的自赋值安全：`A=A` 时若先 `delete[] p` 再读 RHS, 会读到被删资源（或破坏自身）。
 5. 异常安全薄弱：深拷贝/赋值若中途抛异常（如分配失败），应保证原对象不被破坏（强保证）。
-

3) 修复方案 A | 最小修复：保留裸指针 + 正确 Rule of Five (含自赋值)

```
#include <bits/stdc++.h>
using namespace std;

struct Matrix {
    int r=0, c=0;
    double* p=nullptr;

    Matrix(int rr=0, int cc=0, double v=0): r(rr), c(cc) {
        size_t n = 1ULL*r*c;
        if(n==0) { p=nullptr; return; }
        p = new double[n];
        for(size_t i=0;i<n;i++) p[i]=v;
    }

    ~Matrix(){ delete[] p; }
```

```

// 深拷贝
Matrix(const Matrix& o): r(o.r), c(o.c) {
    size_t n = 1ULL*r*c;
    if(n==0) { p=nullptr; return; }
    p = new double[n];
    memcpy(p, o.p, n*sizeof(double));
}

// move : 偷资源 ; noexcept 让 vector 重分配优先 move
Matrix(Matrix&& o) noexcept: r(o.r), c(o.c), p(o.p) {
    o.r=o.c=0; o.p=nullptr;
}

void swap(Matrix& o) noexcept {
    std::swap(r,o.r); std::swap(c,o.c); std::swap(p,o.p);
}

// copy assign : copy-and-swap (自赋值安全 + 强异常保证)
Matrix& operator=(const Matrix& o){
    if(this==&o) return *this;
    Matrix tmp(o);
    swap(tmp);
    return *this;
}

// move assign : 释放旧资源并接管
Matrix& operator=(Matrix&& o) noexcept {
    if(this==&o) return *this;
    delete[] p;
    r=o.r; c=o.c; p=o.p;
    o.r=o.c=0; o.p=nullptr;
    return *this;
}

double& at(int i,int j){
    if(i<0||i>=r||j<0||j>=c) throw out_of_range("OOB");
    return p[1LL*i*c+j];
}
const double& at(int i,int j) const{
    if(i<0||i>=r||j<0||j>=c) throw out_of_range("OOB");
    return p[1LL*i*c+j];
}

friend Matrix operator+(const Matrix& a,const Matrix& b){
    if(a.r!=b.r||a.c!=b.c) throw invalid_argument("DIM");
    Matrix s(a.r,a.c);
    size_t n = 1ULL*a.r*a.c;
}

```

```

        for(size_t i=0;i<n;i++) s.p[i]=a.p[i]+b.p[i];
    return s;
}

Matrix transpose() const{
    Matrix t(c,r);
    for(int i=0;i<r;i++) for(int j=0;j<c;j++)
        t.p[1LL*j*r+i] = p[1LL*i*c+j];
    return t;
}
};

int main(){
    vector<Matrix> v;
    v.reserve(3);
    v.push_back(Matrix(2,3,1.0));
    v.push_back(Matrix(2,3,2.0));
    auto t = v[0].transpose();
    v.resize(10);
    cout << t.at(1,0) << "\n";
}

```

4) 修复方案 B | 现代 C++ : Rule of Zero (推荐)

用 `vector<double>` 承载数据：拷贝/移动/析构都由 STL 正确完成；移动只搬指针/size，`vector` 重分配时更稳。

```

#include <bits/stdc++.h>
using namespace std;

struct Matrix {
    int r=0,c=0;
    vector<double> a; // 连续存储

    Matrix(int rr=0,int cc=0,double v=0): r(rr), c(cc), a(1ULL*rr*cc, v) {}

    double& at(int i,int j){
        if(i<0||i>=r||j<0||j>=c) throw out_of_range("OOB");
        return a[1LL*i*c+j];
    }
    const double& at(int i,int j) const{
        if(i<0||i>=r||j<0||j>=c) throw out_of_range("OOB");
        return a[1LL*i*c+j];
    }
}

```

```

friend Matrix operator+(const Matrix& x, const Matrix& y){
    if(x.r!=y.r||x.c!=y.c) throw invalid_argument("DIM");
    Matrix s(x.r,x.c);
    for(size_t i=0;i<s.a.size();i++) s.a[i]=x.a[i]+y.a[i];
    return s;
}

Matrix transpose() const{
    Matrix t(c,r);
    for(int i=0;i<r;i++) for(int j=0;j<c;j++)
        t.a[1LL*j*r+i] = a[1LL*i*c+j];
    return t; // RVO/Move
}
};

int main(){
    vector<Matrix> v;
    v.reserve(3);
    v.push_back(Matrix(2,3,1.0));
    v.push_back(Matrix(2,3,2.0));
    auto t = v[0].transpose();
    v.resize(10);
    cout << t.at(1,0) << "\n";
}

```

5) 复杂度

- `at` : 时间 O(1), 空间 O(1)
- `+ / transpose` : 时间 O(rc), 额外空间 O(rc)

6) ≥6 个边界测试 (输入要点→期望现象)

1. `Matrix(0,0) + resize/push_back` → 不崩、不越界 (数据为空)。
2. `Matrix(0,5) / Matrix(5,0) → at` 必抛异常；`transpose` 维度互换但 size 仍为 0。
3. 超大 `Matrix(1e5,1e2)` 放入 `vector` 多次增长 → 只要内存够, 不崩；重分配走 move。
4. `A=A` (自赋值) → 数据不变、不泄漏。
5. `A+B` 维度不等 → 抛 `invalid_argument`。
6. `at(-1,0) / at(r,0) → out_of_range`。
7. `A.transpose().transpose()` → 元素回到原位置 (值相等)。
8. 异常路径：构造大矩阵触发 `bad_alloc` (若环境限制) → 不泄漏 (A 用 copy-swap / B 由 STL 保证)。

7) 可复用小模板 (≤6行)

```

template<class T>
void chk(bool ok, const char* msg){

```

```
    if(!ok) throw std::out_of_range(msg);  
}
```