银行家算法的目的：要求程序运行时，根据不同的要求，给予是否分配资源的回答，如果可分配，输出安全序列；否则，给出输出拒绝理由

如： **最大需求 已分配 当前需要 可用** 核心思想是：持有数量+申请数量 一定<=最大需求&&<=现有可用的资源数量

资源情况 / 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3			
P1	3	2	2	2	0	0	1	2	2			
P2	9	0	2	3	0	2	6	0	0	3	3	2
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

输出资源分配表

```cpp
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <thread>
using namespace std;

const int N = 12, M = 12;   //最大进程数，最大资源种类
int n, m;       //n个进程，m种资源
int mks[N][M];  //Max最大需求矩阵,进程i需要j类资源的数目
int alcs[N][M]; //Allocation分配矩阵，进程i的j类资源已分配的数目
int need[N][M]; //Need需求矩阵，进程i，j类资源还需要的数目
int avlb[M];    //Available可用资源向量，j类资源可用的数目
int sod[N];     //SafeOrder安全序列
int bkup_alcs1[N][M], bkup_need1[N][M], bkup_avlb1[M];  //BackUp备份
int bkup_alcs2[N][M], bkup_need2[N][M], bkup_avlb2[M];  //BackUp备份

void myinput() {
    //输入
    cout << "请输入每个进程的Max Allocation Need" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) cin >> mks[i][j];
        for (int j = 0; j < m; j++) cin >> alcs[i][j];
        for (int j = 0; j < m; j++) cin >> need[i][j];
    }
    cout << "请输入Available" << endl;
    for (int j = 0; j < m; j++) cin >> avlb[j];
}
```

```
void myoutput() {
    cout << "\n资源分配表\n\n";
    cout << "进程\\资源\tMax\t\tAllocation\tNeed" << endl;
    for (int i = 0; i < n; i++) {
        cout << "P" << i << "\t\t";
        for (int j = 0; j < m; j++) cout << mks[i][j] << " ";
        cout << "\t\t";
        for (int j = 0; j < m; j++) cout << alcs[i][j] << " ";
        cout << "\t\t";
        for (int j = 0; j < m; j++) cout << need[i][j] << " ";
        cout << endl;
    }
    cout << "Available" << endl;
    for (int j = 0; j < m; j++) cout << avlb[j] << " ";
    cout << endl;
}
```

每一次的安全性检查，实际上是对 假设给某某进程分配资源后，全部进程是否可以全部完成，所以每次对 资源表都会进行脏操作，所以需要备份和还原

```
//备份alcs,need,avlb
void backup(int bkup_alcs[N][M], int bkup_need[N][M], int bkup_avlb[M]) {
    memcpy(bkup_alcs, alcs, sizeof alcs);
    memcpy(bkup_need, need, sizeof need);
    memcpy(bkup_avlb, avlb, sizeof avlb);
}

//恢复alcs,need,avlb
void recovery(int bkup_alcs[N][M], int bkup_need[N][M], int bkup_avlb[M]) {
    memcpy(alcs, bkup_alcs, sizeof alcs);
    memcpy(need, bkup_need, sizeof need);
    memcpy(avlb, bkup_avlb, sizeof avlb);
}
```

```
//安全性检查 检查对于现在资源分配情况，是否可以完成全部任务的进行，如果再一开始的时候，就return false，则直接说 对于初始输入而言，就已经不符合银行家算法了
// 这里做的检查是针对 后序是否能够完成所有进程而言的：你对1个进程分配了资源，则当前的资源分配情况表会发生变化，我们就是先假设给目的进程进行了所需资源的分配，然后检查对于该分配之后的资源表，是否能够完成全部进程！！！  所以 这个检查会对 资源分配表进行脏操作，所以每次需要备份，还原
bool check_safe() {
    backup(bkup_alcs1, bkup_need1, bkup_avlb1); //备份alcs,need,avlb
    bool flag[N];   //已完成的进程的标志
    int cnt = 0;    //完成的进程数
    //初始化falg
    memset(flag, 0, sizeof flag);
    //run
    while (1) {
        //遍历，找到可以满足的资源
        int i;
        for (i = 0; i < n; i++) {
            if (flag[i]) continue;  //已完成的进程跳过
            int j;
            for (j = 0; j < m; j++)
                if (need[i][j] > avlb[j]) break;    //有资源不能满足，判断下个进程
            if (j == m) break;  //找到了一个可以满足的进程，跳出
        }
        //如果找不到可以满足的进程，那么不是安全状态
        if (i == n) {
            recovery(bkup_alcs1, bkup_need1, bkup_avlb1);   //恢复alcs,need,avlb
            return false;
        }
        int id = i; //取得进程编号
        //回收该进程的资源
        for (int j = 0; j < m; j++) avlb[j] += alcs[id][j];
        flag[id] = true;    //当前进程设完成
        sod[cnt] = id;  //放安全序列
        cnt++;  //已完成++
        if (cnt == n) {
            recovery(bkup_alcs1, bkup_need1, bkup_avlb1);   //恢复alcs,need,avlb
            return true;    //全部完成，返回是安全状态
        }
    }
}
```

资源分配

```
//请求资源
//返回值，0:正确，1:超需求，2：超可分配资源，3：假定分配无法通过安全性检查
int request_resource(int id, int req[M]) {
    //超需求判断
    for (int j = 0; j < m; j++) if (req[j] > need[id][j]) return 1;
    //超可分配资源判断
    for (int j = 0; j < m; j++) if (req[j] > avlb[j]) return 2;
    //假定分配
    backup(bkup_alcs2, bkup_need2, bkup_avlb2); //备份alcs,need,avlb
    for (int j = 0; j < m; j++) avlb[j] -= req[j];
    for (int j = 0; j < m; j++) alcs[id][j] += req[j];
    for (int j = 0; j < m; j++) need[id][j] -= req[j];
    //安全性检查
    if (!check_safe()) {
        recovery(bkup_alcs2, bkup_need2, bkup_avlb2);   //恢复
        return 3;
    }
    //检查通过，正式分配资源，也就不需要恢复备份了
    return 0;
}
```

main函数

```
int main() {
    cout << "请输入进程数 资源数" << endl;
    cin >> n >> m;
    myinput();  //输入参数
    myoutput(); //输出表格
    if (check_safe()) {
        cout << "是安全状态，安全序列为" << endl;
        for (int i = 0; i < n; i++) {
            cout << sod[i];
            if (i != n - 1) cout << "->";
        }
        cout << endl;
    }
    else cout << "不是安全状态" << endl;
    //输入请求资源
    while (1) {
        int id, req[M];
        cout << "\n请输入要分配的进程 要分配的资源数量" << endl;
        cin >> id;
        if (id == 9999)break;
        for (int j = 0; j < m; j++) cin >> req[j];
        //做银行家算法分配资源的四个步骤
        int res = request_resource(id, req);
```

```
        switch (res) {
        case 0: cout << "分配成功" << endl;
        myoutput();
        this_thread::sleep_for(chrono::seconds(5));
        cout << "Process Complate" << endl;
        for (int i = 0; i < m; i++) {
            avlb[i] += alcs[id][i];
            alcs[id][i] = 0;
        }
        myoutput();
        ; break;
        case 1: cout << "分配失败！请求的资源数超过最大值！" << endl; break;
        case 2: cout << "分配失败！系统中尚无足够的资源满足P" << id
            << "的申请！" << endl; break;
        case 3: cout << "分配失败！假定分配后，无法通过安全性检查！"
            << endl; break;
        }
    }
    return 0;
}
```

sleep睡5s，模拟分配资源之后，工作完成

修复备份做了两次：

在你提供的银行家算法代码中，`check_safe()` 函数确实对资源表进行了备份和还原，而 `request_resource()` 函数在安全性检查失败时也做了备份和还原。看起来似乎是重复了，但实际上它们的备份和还原操作的目的是不同的，彼此并不多余。

- **在 `request_resource()` 中的备份和还原**：这里的备份主要是为了应对**假定分配失败的情况**。也就是说，当用户请求某些资源时，程序假定把这些资源分配给进程，并在这种状态下运行安全性检查。如果安全性检查失败，则说明分配这次资源后系统可能进入不安全状态，因此需要还原到最初的状态。
- **在 `check_safe()` 中的备份和还原**：这个函数的目的是模拟资源分配后的**全局系统状态**，它会逐一检查每个进程是否能顺利完成，并回收它们占用的资源。这会对当前的资源表进行反复修改，所以必须备份并在检查完成后还原。否则，当我们回到 `request_resource()` 时，资源表已经被修改，无法回到调用 `check_safe()` 之前的状态。

简单来说，`request_resource()` 的备份是为了恢复到请求之前的状态，而 `check_safe()` 的备份是为了让 `check_safe()` 内部可以自由地修改资源表，而不影响外部调用。两者的目的不同，因此这两个地方的备份和还原都是必要的，并不多余。

Request：若进程分配资源后，检查错误，则恢复成：假设之前的资源表的情况 Check：对资源表进行大量的脏操作，执行进程，回收进程资源，然后不断循环，看是否能执行完全部进程，其实若：return false，则恢复的话，Request会进行恢复；但是，如果return true的话，因为 这里 Request没有做恢复，所以需要 恢复成： **假设分配之后，进来做安全性检查的资源分配表的情况**

```
backup(bkup_alcs2, bkup_need2, bkup_avlb2); //备份alcs,need,avlb
for (int j = 0; j < m; j++) avlb[j] -= req[j];
for (int j = 0; j < m; j++) alcs[id][j] += req[j];
for (int j = 0; j < m; j++) need[id][j] -= req[j];
```

看图：Request恢复的是 1 的资源分配表情况，此时未作任何假设处理 $\color{green}{假设失败之前的表}$ Check中恢复的是 2 的资源分配表情况，是假设给 **请求的进程** 分配了 **请求的资源**之后的 资源表的情况 $\color{green}{假设成功，做出修改之后的表}$