Report for Project 2: goChess

Demand Analysis

提供多种下棋模式

- goChess 提供了3种棋局模式，分别是翻转棋、五子棋和自制的“跳棋”。
- 用户可以在选择菜单中对下棋模式自由选择

为用户提供友好的操作界面

- 对功能列表菜单化，并设置多级菜单方便用户使用。
- 充分考虑程序运行中可能出现的异常输入和特殊情况，并在其发生时提供有效的提示。

棋局记录存取的实现

- 某一局中玩家希望退出，可选择是否需要保存棋局。
- 玩家开始游戏的时候，可选择开始新的游戏，或从存档开始。
- 棋局的保存通过对应文件来实现，文件名为开始新游戏的时间戳。

游戏的执行逻辑

- goChess 提供了3种棋局模式，分别是翻转棋、五子棋和自制的“跳棋”。
- 翻转棋与五子棋的逻辑已无需赘述。自制“跳棋”则类似于五子棋，不同之处在于3个棋子间隔相连即算成功。
- 对于不合法的落子位置，应让用户重新输入，直到合法为止。

整体信息的存取

- 棋局的保存通过对应文件来实现，文件名为开始新游戏的时间戳。
- 分别将属于三种模式的记录名称（时间戳）保存至以该模式命名的文件中。
- 在退出时，需要向文本写入更新过的游戏记录信息。
- 需要设计一个合理的文件层次来存储这些信息。

Module Design

`FYI：`具体设计细节在注释中给出

Reversi

Reversi类负责翻转棋的相关操作逻辑，包括：

- 游戏的初始化
- 翻转棋逻辑的控制
- 单次游戏记录的存储

```
class Reversi
{
private:
    Color board[9][9];  //8*8的棋盘
    string timestamp;   //开始时间
    int blackCnt, whiteCnt; //黑子、白子的计数
    map<pair<int, int>, vector<Position>> attached; //可落子位置->对应的本方棋子位置
    bool findAvl(int x, int y); //查找当前位置是否存在对应的可落子位置
    void refresh(); //刷新屏幕
    void layout();  //绘制棋盘
    void leave(int mode);   //离开游戏，可选保存记录至docs/Reversi/timestamp.txt
public:
    Reversi();  //两个重载的初始化函数分别对应从头开始和读取存档
    Reversi(string ts);
    int startGame(); // 返回值表明此次的退出状态，正常结束/中途突出
};
```

FiaR

FiaR类负责五子棋的相关操作逻辑，包括：

- 游戏的初始化
- 五子棋逻辑的控制
- 单次游戏记录的存储

```
class FiaR
{
private:
    Color board[9][9];  //8*8的棋盘
    string timestamp;   //开始时间
    int sum;    //棋子总数
    bool complete(int x, int y);    //判断此位置是否成功连线
    void refresh(); //刷新屏幕
    void layout();  //绘制棋盘
    void leave(int mode);   //退出，可选保存记录至docs/FiaR/timestamp.txt
public:
    FIAR(); //两个重载的初始化函数分别对应从头开始和读取存档
    FIAR(string ts);
    int startGame();
};
```

Jumpy

Jumpy类负责自制“跳棋”的相关操作逻辑，包括：

- 游戏的初始化
- 自制“跳棋”逻辑的控制
- 单次游戏记录的存储

```
class Jumpy
{
private:
    Color board[9][9];  //8*8的棋盘
    string timestamp;   //开始时间
    int sum;    //棋子总数
    bool complete(int x, int y);    //判断此位置是否成功连线
    void refresh(); //刷新屏幕
    void layout();  //绘制棋盘
    void leave(int mode);   //退出，可选保存记录至
docs/Jumpy/timestamp.txt
public:
    FIAR(); //两个重载的初始化函数分别对应从头开始和读取存档
    FIAR(string ts);
    int startGame();
};
```

Admin

Admin类负责管理员的相关操作，包括：

- 游戏的各级菜单选项的维护
- 整体游戏记录的读取和存储

```
class Admin
{
private:
    //存储三种模式各自的存档集合
    vector<string> reversi;
    vector<string> fiar;
    vector<string> jumpy;
    void subMenu(int mode);     //次一级菜单
    void recordMenu(int mode);  //游戏记录菜单
    //保存记录，分别将属于三种模式的记录名称（时间戳）保存至对应的文件中
    void storeRecords();
public:
    void mainMenu();    //主菜单，也是整个程序的入口和出口
};
```

Novelty

翻转棋落子位置

对于翻转棋中可落子的位置，以**特殊符号进行标识**。当然，**不正确的落子位置仍会进行提醒**。

Problems & Solutions

P：对于游戏如何**实现直接退出或暂停存档**，我开始有过很多方案，比如鼠标操作或是键入指定字符，但发现效果都不理想。而且没有GUI的控件辅助，在控制台实现鼠标操作极为麻烦，对实际学习也没有帮助。

S：基于此，我利用了两个非法的棋盘坐标`(0,0)`，`(0,1)`来分别代表直接退出和暂停存档，实现简单，不易出错。虽然纯粹从用户的角度来看，这种实现有讨巧之嫌，但权衡之下，我仍然认为这是不错的解决方案。

P：对于存储用户信息的**文件结构**，我最初没有清晰的构思，试图将所有信息放在同一个文件夹docs里进行存取。但事实证明，这种方案虽然不是完全不可行，但执行效率比较低，且信息夹杂在一起不利于后期的调试和维护。

S：基于此，我将三种模式的游戏记录名（即相应的时间戳）放置在以游戏模式命名的文本文件中，再为每条记录建立一个以其时间戳命名的文件存放相关的游戏存档，使文件的脉络更清晰，便于维护和查找。

P：在三种游戏模式对应的类中，有大量重复或相似的成员变量或函数；尤其是`FiaR`与`Jumpy`，仅仅是`complete()`函数的实现略有不同。对相应功能的重复实现会增添不必要的工作量，代码可读性变差，也不便于后期维护。

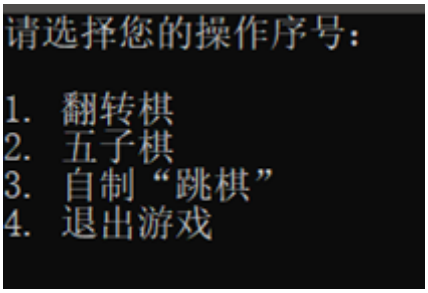
S：我对相关代码进行了多次**复用**，使工作量下降很多。其实更好的解决方案是利用C++面向对象的**继承**特性，先实现一个棋类基类，再分别就不同模式进行扩充和修改。但我一开始并没有考虑周全，也就维持现状，在以后的编程过程中，一定要事先构思好最优的代码架构，才能事半功倍。

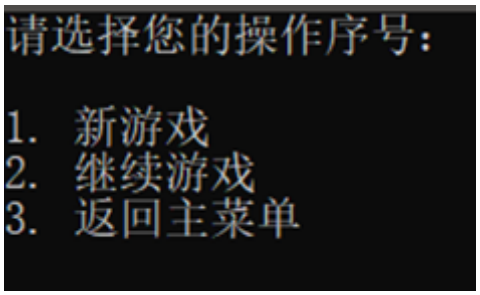
P：本程序的实现中涉及大量屏幕刷新和光标移动的操作，尤其是在棋盘上进行移动，对原始坐标进行计算虽可行，但会耗费大量额外精力，也很可能出错。

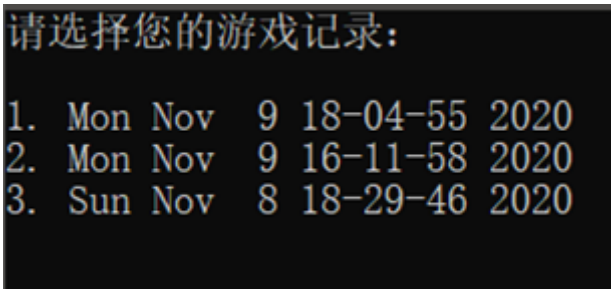
S：我实现了一个`go_to(int x, int y)`函数，两个参数分别代表棋盘的横纵坐标，大大方便了光标的移动。

Testing Snapshots

多级菜单

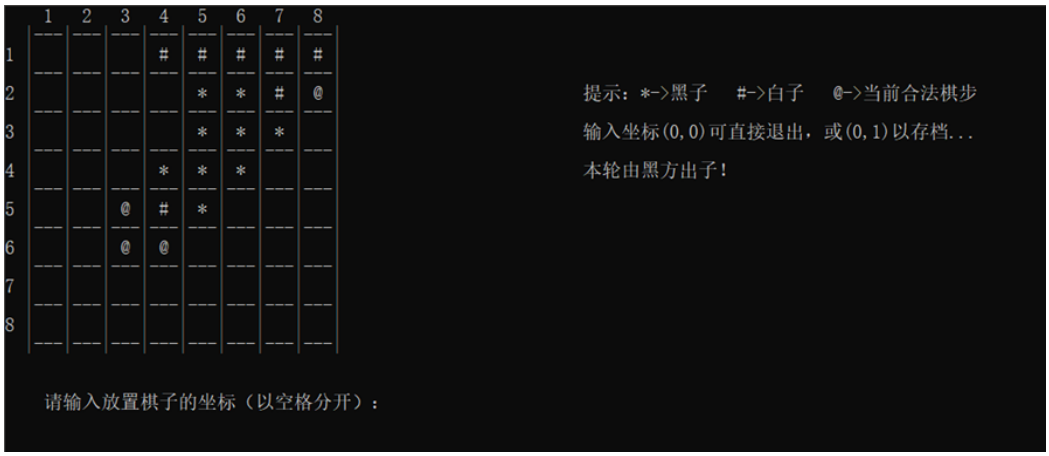




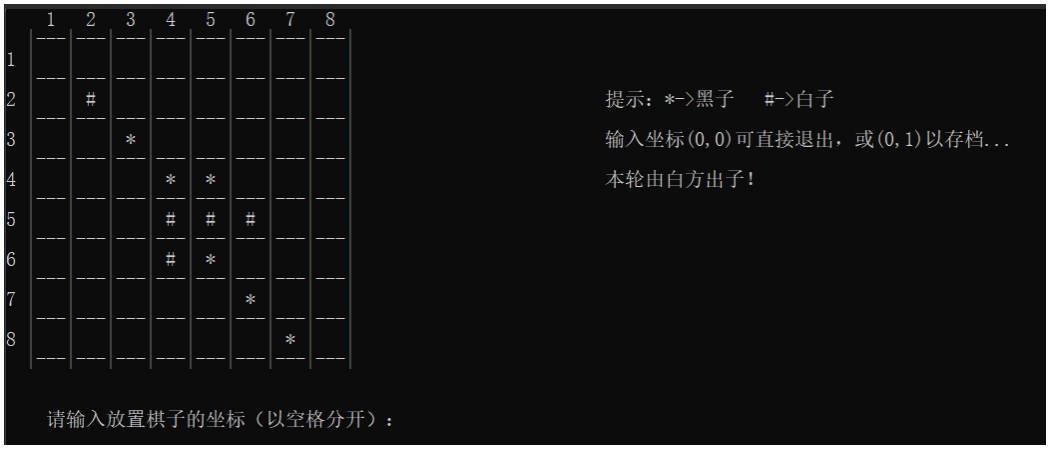


游戏界面

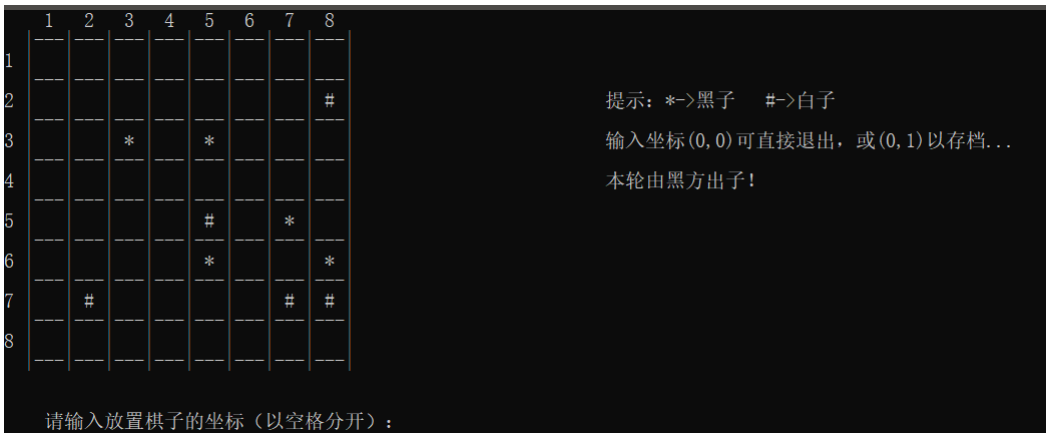
翻转棋：



五子棋：


1 2 3 4 5 6 7 8
1
2 #
3 *
4 * *
5 # # #
6 # *
7 *
8 *
提示：*->黑子 #->白子
输入坐标(0,0)可直接退出，或(0,1)以存档...
本轮由白方出子！
请输入放置棋子的坐标（以空格分开）：

自制“跳棋”：


1 2 3 4 5 6 7 8
1
2 #
3 * * *
4
5 # *
6 * *
7 # #
8
提示：*->黑子 #->白子
输入坐标(0,0)可直接退出，或(0,1)以存档...
本轮由黑方出子！
请输入放置棋子的坐标（以空格分开）：