Quoridor 客户端

1. 规则介绍（以PPT为准）

数据：
横方向为 x 轴沿右方增大，纵方向为 y 轴沿上方增大。棋盘为 9×9，棋子坐标范围为 $[1,9]$ 之中的整数，总共 81 个棋子位置，坐标为 $(1...9, 1...9)$。挡板以起始坐标和结束坐标表示，坐标范围为 $[0,9]$ 之中的整数，除了最边缘的边界外其它所有沟壑内均可添加挡板，只能为横竖两个方向，每个挡板长度为 2 格，例如挡板 $((4,1),(6,1))$ 表示一个挡板在棋子坐标 $(5,1)$ 和 $(6,1)$ 之上，在 $(5,2)$ 和 $(6,2)$ 之下。本次游戏中挡板不可重叠放置（例如放置了 $((4,1),(6,1))$ 之后不能放置 $((3,1),(5,1))$ 和 $((5,1),(7,1))$），可交叉（例如放置了 $((4,1),(6,1))$ 之后可以放置 $((5,0),(5,2))$）。图示见PPT。

规则：

1. 己方棋子从初始位置到达对面任一位置即为胜利（若初始位置在 $(5,1)$ 则到达 $(1...9,9)$ 为获胜；若初始位置在 $(5,9)$ 则到达 $(1...9,1)$ 为获胜）；
2. 3次违规直接判负；
3. 若既不移动自己的棋子也不放置挡板表示无操作，无操作记一次违规；
4. 若放置挡板与现有挡板冲突（重复，重叠，堵死对方或坐标错误）记一次违规；
5. 若既移动自己的棋子又放置挡板视为移动操作；
6. 若移动位置不可达，记一次违规；
7. 单步计算时间（执行一次 `nextStep()` 加通信时间）超时（暂定为≥1秒）记一次违规；（调试时所用服务端不检查超时）
8. 违规3次直接判负；
9. 为防止两位玩家反复给出往复步骤难分高下，本实验设置每场对局每人最多100步，步数超出以用时短者获胜；
10. 比赛期间超时直接踢出比赛。

2. 游戏服务端环境安装

JRE安装：

大群内群文件中下载'jre-8u231-windows-x64.exe'，根据提示不要修改任何设置（一切都默认不改）安装即可。安装后可打开命令提示符程序（CMD）输入java进行测试，若无提示找不到命令则安装成功，本步骤完成。

服务端文档详见server目录下README.md。

3. 注意事项

3.0. 初始测试 (Windows + Visual Studio) 图形化指示参考 `Quoridor初始测试.pptx`

1. 拿到客户端代码后，请直接在 Windows 下双击执行 Quoridor\bin\QuoridorUI.exe，检查是否能正常回放测试 log 文件；
2. 在未修改任何代码的情况下：
    1. 安装并配置好 JDK1.8 环境，在有 server.jar 的目录下命令行执行命令 java -jar server.jar -p 19330 -n 2 运行服务端 jar 可执行文件；
    2. 双击运行 Quoridor\bin\BaselineClient.exe；
    3. 运行 Visual Studio 2019 打开解决方案 Quoridor\Quoridor.sln，编译并运行，体会开发流程；
3. 按照自己的想法修改 Quoridor\QuoridorClient\MyPlayer.cpp(.h) 并调试。

服务端文档详见server目录下README.md。

3.1. 编译须知

若只需要使用 Visual Studio 开发，无需考虑这一部分。

1. Visual Studio 下编译若出现 printf_s 安全警告等类似问题，请自行添加宏定义 #define _CRT_SECURE_NO_WARNINGS ；
2. Windows 下 MinGW 环境直接在 Quoridor\ 目录下 make 即可在 Quoridor\bin 目录得到可执行文件；
3. Linux 下需将 Quoridor/makefile_linux 替换 Quoridor/makefile 再 make ；
4. 由于系统差异，makefile 中并未实现 make clean，有需要的同学请自行实现。

3.2. 提交须知

1. 提交的文件必须严格按照 PPT 中所述，未按要求若出现问题请学生自行承担；
2. 非 Visual Studio 需要提交 makefile 和可执行文件，并说明自己平台和编译环境；
3. 添加编译参数 -O3 可能提高程序执行速度，但会降低编译速度，类似的，Visual Studio 编译的 Release 模式，建议在比赛时使用。

3.3. 自动回放配置

自动回放机制由终端命令实现，需要将 `Quoridor\bin\QuoridorUI.exe` 和 UI 配置文件（`Quoridor\bin\uiconfig.json` 和 `Quoridor\bin\panel`）三个文件放到 `QuoridorClient` 能直接找到的目录下(Visual Studio中调试是放在 `Quoridor\QuoridorClient\` 目录下)。

3.4.

`QuoridorUI.exe` 具有两个可选输入参数，不建议修改其配置文件实现日志文件读取，详见 `Quoridor\QuoridorUI\QuoridorUI.cpp` 中 `getConfig()` 函数。

```
QuoridorUI.exe # 默认读取 uiconfig.json 配置文件,并从配置文件中取得日志文件名进行回放
QuoridorUI.exe log_xx_xx.csv # 读取 log_xx_xx.csv 日志文件回放
QuoridorUI.exe uiconfig.json # 读取 uiconfig.json 配置文件
QuoridorUI.exe log_xx_xx.csv uiconfig.json # 读取 uiconfig.json 配置文件并读取 log_xx_xx.csv 日志文件回放
QuoridorUI.exe uiconfig.json log_xx_xx.csv # 同上
```

3.5.

`Quoridor\QuoridorUI\uiconfig.json` 必须与 `Quoridor\QuoridorUI\panel` 相互配合，无特殊要求**请勿修改**这两个文件，详细信息见 QuoridorUI 游戏回放程序一节。

4. QuoridorUtils 游戏基础数据结构

请务必仔细阅读源文件 `Quoridor\QuoridorUtils\QuoridorUtils.h`

4.0. 数据结构概览

`QuoridorUtils.h` 总共包含 `GameStatus` 一个枚举类和 `Location`，`BlockBar`，`ChessboardChange`，`Step` 四个结构体。

注意：所有枚举类与结构体均在 `namespace QuoridorUtils` 中，若在其它命名空间中使用这些结构，请使用类似于 `QuoridorUtils::Location` 的标识符或加上 `using namespace QuoridorUtils;`。

4.1. 几个常量与 GameStatus 枚举类

简单明了，无需多言。

状态中有几个输赢的状态标明了“不经玩家处理”说明这些状态会在 `QuoridorUtils::Configuration` 中处理，例如：当一个玩家胜利的时候 `QuoridorUtils::Configuration` 会自动处理这条胜利的消息并等待重新开局，并调用玩家实现的 `restart()` 函数重置玩家状态。

```
extern const int SIZE;        // 棋盘大小 9
extern const int BLOCK_SUM;   // 玩家拥有的挡板总数 10
extern const int BLOCK_LEN;   // 挡板长度 2

enum class GameStatus {
    Ok = 0,                   // 正常
    Win = 1,                  // 胜利，不经玩家处理
    Lost = 2,                 // 失败，不经玩家处理
    Timeout = 3,              // 超时
    EnemyClosed = 4,          // 对手故障，不经玩家处理
    RulesBreaker = 5,         // 违规多次判负，不经玩家处理
    InsufficientBlock = 6,    // 挡板已用完
    InvalidStep = 7,          // 棋子不可达或挡板坐标错误
    None = 2000,              // 未定义状态，请勿使用
};
```

4.2. Location 坐标

该结构体为该程序提供基础坐标结构，初始化默认坐标为 $(-1,-1)$，$(-1,-1)$ 在游戏中表示空数据。棋盘中棋子的坐标x与y的范围均为 $[1,SIZE]$ 中的整数，见 PPT 坐标定义。重载 `==` 操作符便于判断两个坐标是否相等 `a==b`，未提供 `!=` 操作符，请使用 `!(a==b)`。两个常量 `PLAYER0_LOC`，`PLAYER0_LOC` 表示游戏双方开局棋子所在坐标，需要自行判断自己初始所在坐标。

```
struct Location {
    int x;
    int y;
    Location(const int x = -1, const int y = -1) {
        this->x = x;
        this->y = y;
    };
    bool operator==(const Location& rLoc) const {
        return this->x == rLoc.x && this->y == rLoc.y;
    }
    int distance(const Location& rLoc) const {  // 计算街区距离
        return (abs(this->x - rLoc.x) + abs(this->y - rLoc.y));
    }
};

extern const Location PLAYER0_LOC;  // 玩家0的初始位置(5, 1)
extern const Location PLAYER1_LOC;  // 玩家1的初始位置(5, SIZE)

// 示例代码
Location myLoc;          // 我的坐标(-1, -1)
Location enemyLoc(5,1);  // 敌人坐标(5, 1)
myLoc = PLAYER1_LOC;     // 我的坐标(5, 9)
if (!(myLoc == enemyLoc)) {  // 判断两个坐标不等
    std::cout << "City Block distance: " << myLoc.distance(enemyLoc) << std::endl;
} else {
    std::cout << "equivalence. \n";
}
```

4.3. BlockBar 挡板

挡板由开始点和终结点两个坐标表示，坐标示例见 PPT 。游戏中定义的挡板坐标范围为 $[0, SIZE]$ 中的整数，挡板长度为 `BLOCK_LEN` ，任何不合规格挡板都可以通过 `blockBar.isNan()` 判断为 `true` 。在处理挡板数据时，可能需要使用 `isH()` 判断是否为水平方向（ `isV()` 即为竖直方向）。另外， `normalization()` 用于标准化挡板数据，便于两个挡板比较是否相等。

```
struct BlockBar {
    Location start;  // 挡板开始坐标
    Location stop;   // 挡板结束坐标
    BlockBar(const int startX = -1, const int startY = -1,
             const int stopX = -1, const int stopY = -1) {
        this->start.x = startX;
        this->start.y = startY;
        this->stop.x = stopX;
        this->stop.y = stopY;
    };
    BlockBar(const Location start, const Location stop = Location(-1, -1)) {
        this->start = start;
        this->stop = stop;
    };
    bool operator==(const BlockBar& rLoc) const {
        return this->start == rLoc.start && this->stop == rLoc.stop;
    }
    bool isNan() const {  // 判断是否有挡板，返回true表示数据不合法表示没有挡板
        return this->start.x < 0 || this->start.x > SIZE ||  // 数据不合法
               this->start.y < 0 || this->start.y > SIZE ||
               this->stop.x < 0 || this->stop.x > SIZE ||
               this->stop.y < 0 || this->stop.y > SIZE ||
               !(isH() || isV()) ||                          // 挡板倾斜
               (start.distance(stop) != BLOCK_LEN);      // 挡板长度不对
    };
    bool isH() const { // 判断挡板方向是否为横向
        return this->start.y == this->stop.y && this->start.x != this->stop.x;
    };
    bool isV() const { // 判断挡板方向是否为纵向
        return this->start.x == this->stop.x && this->start.y != this->stop.y;
    };
    void normalization() {  // 标准化为 start < stop，只可能会交换 start 和 stop，不会修改数据
        if (this->start.x >= this->stop.x && this->start.y >= this->stop.y) {
            const Location tmp = this->start;
            this->start = this->stop;
            this->stop = tmp;
        }
    }
};
```

```
// 示例代码
BLockBar aBlock(4, 1, 6, 1);
BLockBar bBlock(6, 1, 4, 1);
if ((!aBlock.isNan()) && (!bBlock.isNan())) {
    aBlock.normalization()  // (4, 1, 6, 1)
    bBlock.normalization()  // (4, 1, 6, 1)
    if (aBlock == bBlock) {
        std::cout << "equivalence!" << std::endl;
    }
} else {
    std::cout << "Illegal BlockBar!" << std::endl;
}
```

4.4. ChessboardChange 棋盘变更

棋盘变更结构提供给玩家的 `Step nextStep(ChessboardChange&)` 函数获取棋盘变化数据。

具体操作体现在 `Location` 和 `BlockBar` 两个数据结构的操作上，此处不提供示例代码。

```
struct ChessboardChange {
    GameStatus status = GameStatus::Ok;  // 我的上一步执行状态
    Location myLoc;                      // 我的棋子当前位置
    Location enemyLoc;                   // 敌方棋子当前位置
    BlockBar newEnemyBlockBar;  // 若敌方在放置了隔板则该项为隔板的起始终结点，
                                // 未放置则默认为(-1, -1, -1, -1)
    bool isFinish() const {     // 判断游戏是否结束
        return isWin() || isLost();
    }

    bool isWin() const {        // 判断是否胜利
        switch (this->status) {
        case GameStatus::EnemyClosed:
        case GameStatus::Win:
            return true;
        default:
            return false;
        }
    }

    bool isLost() const {       // 判断是否失败
        switch (this->status) {
        case GameStatus::Lost:
        case GameStatus::RulesBreaker:
            return true;
        default:
            return false;
        }
    }
};
```

4.5. Step 玩家步骤

玩家在 `Step nextStep(ChessboardChange&)` 函数需要返回的数据结构。游戏中只能提供选择一种操作（要么移动，要么放板）；若玩家 `myNewLoc` 和 `myNewBlockBar` 数据都为 -1，表示玩家无操作，服务端作违规处理；若玩家 `myNewLoc` 和 `myNewBlockBar` 数据都不为 -1，表示玩家移动棋子。

具体操作体现在 `Location` 和 `BlockBar` 两个数据结构的操作上，此处不提供示例代码。

```
struct Step {
    Location myNewLoc;          // 自己移动的目的坐标，不移动则默认为(-1, -1)
    BlockBar myNewBlockBar;     // 若自己在放置了隔板则该项为隔板的起始点和终结点，
                                // 未放置则默认为(-1, -1, -1, -1)
    Step(const Location loc = Location(), const BlockBar block = BlockBar()) {
        this->myNewLoc = loc;
        this->myNewBlockBar = block;
    }
    Step(const ChessboardChange chessboard) {
        this->myNewLoc = chessboard.enemyLoc;
        this->myNewBlockBar = chessboard.newEnemyBlockBar;
    }
    bool isMove() const {       // 只要 myNewLoc 范围正确，则必然为移动步骤
        return this->myNewLoc.x >= 1 && this->myNewLoc.x <= SIZE &&
               this->myNewLoc.y >= 1 && this->myNewLoc.y <= SIZE;
    }
    bool isNan() const {        // myNewLoc 和 myNewBlockBar 都为空，则无操作
        return !this->isMove() && this->myNewBlockBar.isNan();
    }
};
```

4.6. 其它

上述一个枚举类和四个结构体均提供了标准流输出操作，可直接使用类似于 `std::cout << loc` 的代码输出。

为 `Location` 和 `BlockBarHash` 分别提供了哈希函数，便于玩家使用底层结构为哈希表的 STL 对象。

```
extern std::ostream& operator<<(std::ostream& os, const QuoridorUtils::GameStatus status);
extern std::ostream& operator<<(std::ostream& os, const QuoridorUtils::Location loc);
extern std::ostream& operator<<(std::ostream& os, const QuoridorUtils::BlockBar block);
extern std::ostream& operator<<(std::ostream& os, const QuoridorUtils::ChessboardChange change);
extern std::ostream& operator<<(std::ostream& os, const QuoridorUtils::Step step);

struct LocationHash {
    size_t operator()(const Location& loc) const {
        return std::hash<int>{}(loc.x) + std::hash<int>{}(loc.y);
    }
};
struct BlockBarHash {
    size_t operator()(const BlockBar& block) const {
        return std::hash<int>{}(block.start.x) + std::hash<int>{}(block.start.y) +
            std::hash<int>{}(block.stop.x) + std::hash<int>{}(block.stop.y);
    }
};

// 若使用 Location 或 BlockBar 作为 unordered_map/unordered_set 的 key，请使用此哈希，用法如下:
std::unordered_map<QuoridorUtils::Location, valClass, QuoridorUtils::LocationHash> map_obj;
std::unordered_set<QuoridorUtils::BlockBar, QuoridorUtils::BlockBarHash> set_obj;
```

5. QuoridorClient 游戏客户端

源文件 Quoridor\QuoridorClient\MyPlayer.cpp(.h)

同学们需要实现的函数在 `Quoridor\QuoridorClient\MyPlayer.cpp` 中，可以自定义自己需要的函数在该类中。

5.1. 客户端整体框架

客户端主函数在 `Client.cpp` 中实现，实现流程为：读取配置文件，连接网络，新建玩家，开始游戏，游戏回放。

客户端整体框架见 PPT 备片。

5.2. 编程实现

为了引导同学们使用底层数据结构，增加对游戏规则的理解， `MyPlayer.cpp(.h)` 中已经实现了一个随机乱走版本的玩家。下面的面向对象和面向过程两部分所使用的算法一模一样，只是在结构上有些许差别，请选择自己熟悉的风格结构实现。

面向对象玩家实现

同学们可以在 `MyPlayer.h` 文件中添加自己所需的方法和属性，在这个例子中，除了必须实现的3个函数(`MyPlayer(string)` 构造函数 `nextStep(ChessboardChange)` 决策函数 `restart()` 棋盘状态清零函数)之外，还实现了 `randomWalk(Location, Location)` 函数用于随机选择一个可

以走的方向，属性 `targetY` 存储自己的目的地，属性 `blocks` 存储所有挡板。其中 `randomWalk(Location, Location)` 会使用 `blocks` 中的挡板来判断自己当前的位置哪些方向被挡板挡住。

当然，这个示例为简单示例，同学们有复杂函数或深层次抽象可能要更多的类（意味着更多的文件）来实现自己想要的算法。

```
#include <vector>
#include "Player.h"

namespace QuoridorUtils {
class MyPlayer final : public Player{
private:
    std::vector<BlockBar> blocks;                                          // 实例所需要，可删
    int targetY = 0;                                                       // 实例所需要，可删
    Location randomWalk(const Location& myLoc, const Location& enemyLoc);   // 实例所需要，可删
public:
    MyPlayer(const std::string& key) : Player(key) {};            // 必须存在，无需修改
    Step nextStep(const ChessboardChange& newChange) override;    // 必须自行实现
    void restart() override;                                      // 必须自行实现
};
}
```

其 cpp 结构如下，所有函数均在 `MyPlayer` 限定符下，所有函数均可以通过 `this` 指针访问到数据成员 `targetY` 和 `blocks` 。

```
#include "MyPlayer.h"
#include ...

namespace QuoridorUtils {
// 面向对象实现开始
Location MyPlayer::randomWalk(const Location& myLoc, const Location& enemyLoc) { ... }

Step MyPlayer::nextStep(const ChessboardChange& newChange) {
    // 须知：本示例代码仅作为 API 使用以及游戏规则了解范例，甚至保证不了正确性，切勿照抄照搬
    ...
}

void MyPlayer::restart() {
    this->blocks.clear();
    this->targetY = 0;
}
// 面向对象实现结束
}
```

面向过程玩家实现

若不会面向对象编程方法，在 `MyPlayer.cpp(.h)` 中面向过程也是可以的。当然，由于底层抽象是通过面向对象实现的，不可避免地会用到面向对象的东西（比如使用 `vector` ），但自己实现的结构可以是面向过程的。

下面的代码为 `MyPlayer.h` 的全部有效内容，在使用面向过程编程方法时，h文件不需要添加任何其它声明，所有声明和实现均在 cpp 中实现。

这个 `MyPlayer.h` 为最小可用版本，也可以在此 h 文件的基础上添加内容面向对象来实现 `MyPlayer` 玩家类。

```
#include "Player.h"

namespace QuoridorUtils {
class MyPlayer final : public Player {
public:
   MyPlayer(const std::string& key) : Player(key) { };          // 必须存在，无需修改
   Step nextStep(const ChessboardChange& newChange) override;   // 必须自行实现
   void restart() override;                                     // 必须自行实现
};
}
```

`MyPlayer.cpp` 的代码结构如下所示，详细代码见源文件注释部分。可见，代码中仅有 `MyPlayer::nextStep(...)` `MyPlayer::nextStep()` 两个函数有类名限定符 `MyPlayer` ， `randomWalk(...)` 和 `blocks` 都不是类的成员，但类中的成员依然可以调用它们实现相同的功能。

同样，对比面向对象的 `restart()` 函数，在这里不是使用 `this` 指针来调用自己所需要的数据。

```cpp
#include "MyPlayer.h"
#include ...

// 面向过程实现开始
int targetY = 0;
std::vector<QuoridorUtils::BlockBar> blocks;
QuoridorUtils::Location randomWalk(const QuoridorUtils::Location& myLoc, const QuoridorUtils::Location& enemyLoc) { ... }
// 面向过程实现结束

namespace QuoridorUtils {
// 面向过程实现开始
Step MyPlayer::nextStep(const ChessboardChange& newChange) {
   // 须知：本示例代码仅作为 API 使用以及游戏规则了解范例，甚至保证不了正确性，切勿照抄照搬
   ...
}
void MyPlayer::restart() {
   blocks.clear();
   targetY = 0;
}
// 面向过程实现结束
}
```

5.3. 程序调试

由于该程序较为庞大，需要有服务端和两位玩家参与，故在调试之前请确保通过 **3.0. 初始测试**。

6. QuoridorUI 游戏回放程序

学有余力的同学可以实现实时加载器和图形界面重放器。

客户端 UI 使用控制台实现，在游戏的底层数据结构 `QuoridorUtils.cpp/h` 上定义了两个抽象类 `DataLoader` `GameView` 作为接口。 `DataLoader` 是用来加载数据的抽象类，本项目中实现了其子类 `LogLoader` 用来加载 csv 格式的 log 文件， `GameView` 是用来显示棋盘的，其子类 `ConsoleView` 通过输出字符来显示。

在此基础之上，定义了类 `Relayer` 作为游戏重放控制器，连接两个接口，接受用户输入，控制 `DataLoader` 对象的输入和 `GameView` 对象的输出。

6.1. DataLoader 数据加载虚类

该虚类的代码如下。总共分为2个步骤：1. 读取玩家名；2. 依此读出一条条的数据，若读到了返回 `true` ，否则返回 `false` 。

```cpp
#include <string>
#include "../QuoridorUtils/QuoridorUtils.h"

namespace QuoridorUtils {
class DataLoader {
public:
    virtual ~DataLoader() = default;
    // player0Name 是坐标在 QuoridorUtils::PLAYER0_LOC 开始的玩家;
    // player1Name 是坐标在 QuoridorUtils::PLAYER1_LOC 开始的玩家;
    virtual void getPlayerName(std::string& player0Name, std::string& player1Name) = 0;
    virtual bool getNextStep(std::string& playerName, QuoridorUtils::Step& nextStep, QuoridorUtils::GameStatus& status) = 0;
    virtual int getRemainingStep() = 0;
};
}
```

6.2. GameView 界面显示虚类

与 `DataLoader` 相反， `GameView` 需要做的步骤为：1. 输入玩家名；2. 压入一条条的数据，玩家名错误则返回 `false` ；3. 显示前一步和后一步，成功显示返回 `true` 。

```cpp
#include <string>
#include "../QuoridorUtils/QuoridorUtils.h"

namespace QuoridorUtils {
class GameView {
public:
    virtual ~GameView() = default;
    // 用户名注册，仅能两位玩家参与游戏.
    //  firstPlayer 是坐标在 QuoridorUtils::PLAYER0_LOC 开始的玩家;
    // secondPlayer 是坐标在 QuoridorUtils::PLAYER1_LOC 开始的玩家;
    virtual void playerRegister(const std::string& firstPlayer, const std::string& secondPlayer) = 0;
    // 输入下一步建立地图，若 playerName 未注册则不建立.
    virtual bool putStep(const std::string& playerName, const Step& nextStep, const GameStatus& status) = 0;
    virtual bool showNextStep() = 0;
    virtual bool showPrevStep() = 0;
};
}
```