

|  |
| --- |
| 程序设计语言原理 |
| 项目说明 |

**姓名： 梁伊雯**

**学号： 1951095**

**专业： 软件工程**

**日期：** 2020年 8 月 22 日

目录

[1 整体设计思路 3](#_Toc48899582)

[2 类结构及功能介绍 5](#_Toc48899583)

[2.1 菜单类 5](#_Toc48899584)

[2.2 棋盘类 5](#_Toc48899585)

[2.3 判断类 6](#_Toc48899586)

[2.4 A星类 7](#_Toc48899587)

[3 算法介绍 9](#_Toc48899588)

[3.1消除及计分算法 9](#_Toc48899589)

[3.2 A星算法 9](#_Toc48899590)

[3.3其他寻路算法 10](#_Toc48899591)

[3.4对算法改进的设想 10](#_Toc48899592)

[4系统工作流程 11](#_Toc48899593)

[4.1工作流程 11](#_Toc48899594)

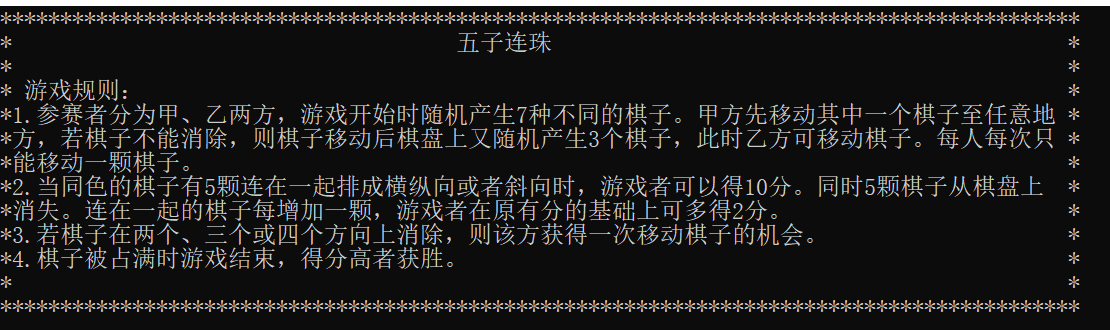
[4.2工作细节 12](#_Toc48899595)

[5总结 13](#_Toc48899596)

# 整体设计思路

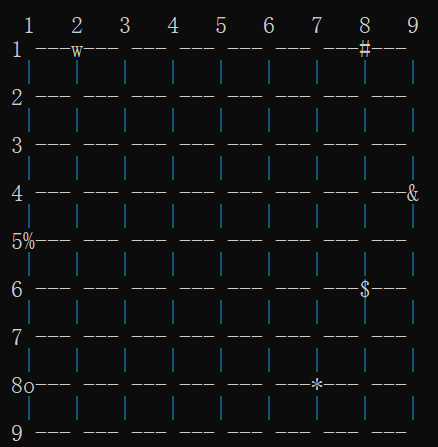
1. 设计菜单

菜单内为本游戏的游戏规则，便于玩家阅读及了解。



（2）选择棋盘样式及棋子形状

棋盘大小为9\*9，棋子共有7种形状，我分别用 'w', '#', '\*', '%' , '$' , 'o' , '&'这几种符号代表不同的7种棋子。棋盘设计如下，棋子分布在棋盘交线的空隙中。



（3）设计棋子移动机制

首先比赛方输入棋子的初始坐标和目的坐标，判断输入数据是否有效。若有效，则判断能否由该位置移动到目标位置，只依据上、下、左、右四方向判断路径有效性。若能够移动，则调用移动函数。

（4）设计随机棋子

随机棋子共有4种情况：在随机位置产生七个不同的棋子，在空余位置产生三个棋子、两个棋子或一个棋子。产生随机棋子时要考虑是否能生成相同类型的随机棋子，若是七个棋子则不能相同，三个棋子和两个棋子的情况可相同。还要考虑产生的位置是否是空白可下棋部分。

（5）设计计分及消除机制

对目标位置的周围八个方向进行遍历，计数某一方向上的连续相同的棋子。上与下为一组，左和右为一组，以此类推，若一组中的与目标位置相同的棋子为4个以上，则可以调用消除和计分的函数。该机制主要有两种方式：单方向与多方向。单方向为多方向的基础，后者可调用前者函数，但多方向消除可获得多一次移动棋子的机会。

（6）判断游戏结束及公布比赛结果

每一方对棋盘操作完成后遍历一次棋盘，若所有棋点全被占满，则游戏结束。进行双方分数的比较，最后公布双方总分及比赛结果。

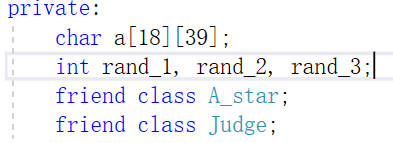
# 类结构及功能介绍

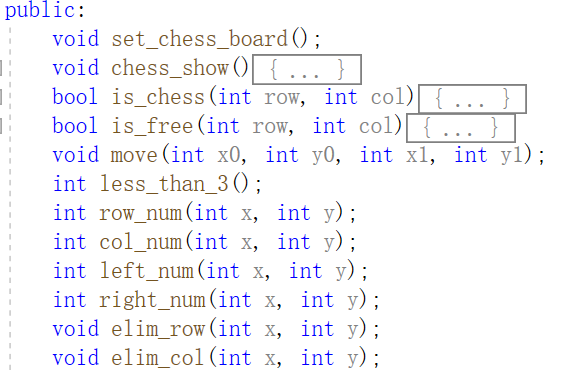
## 菜单类

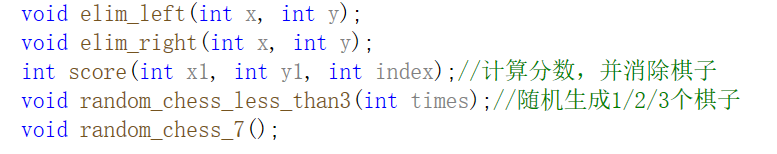
菜单类中含有setmenu()和setxy()两个函数，主要用来显示菜单和xy坐标轴。

## 棋盘类

a\_chess\_board类为棋盘类，含有4个数据成员、2个友元类及19个公有成员函数。







a[18][39]中存放棋盘中的字符，rand\_1、rand\_2和rand\_3是3个随机变量，用于随机棋子的生成。A\_star和Judge类为本类的友元类，便于访问其私有成员及函数。

set\_chess\_board()函数为棋盘进行初始化，包括棋盘边线对应数字的设置和边界形状设置。

Chess\_show()函数能够展示棋盘形状。

is\_chess()和is\_free()分别判断该位置是否为棋点和该位置是否是空闲的。后者要以前者的判断为前提。

move(int x0,int y0,int x1,int y1)函数是单纯的移动棋子函数，但要将用户输入的数据通过计算转换为棋盘上对应的数字。

less\_than\_3()作用为判断棋盘上的棋子是否少于3个，若少于，则后面会随机产生2个或1个棋子。

row\_num, col\_num, left\_num, right\_num四个函数分别用来计算目标点周围四个方向（横、竖、左斜、右斜）相连相同的棋子数。

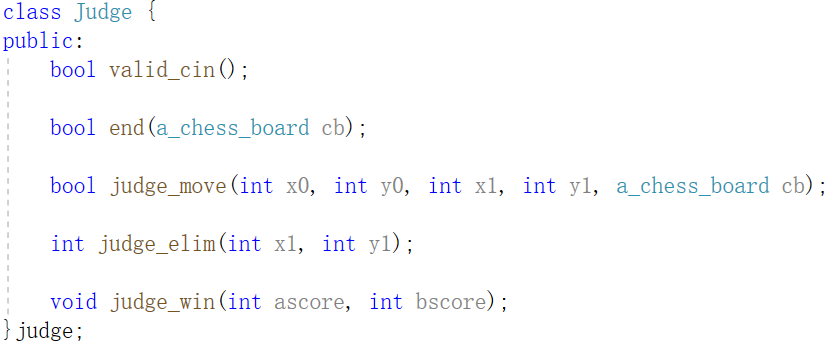
elim\_row, elim\_col, elim\_left, elim\_right四个函数分别是消除目标点周围四个方向相连相同的棋子，该函数实现需要以上一个计算函数的结果为前提。

Score(int x1,int y1,int index)函数是用来计分的，其中的参数index为Judge类中的judge\_elim函数提供，index为消除方式的代号，棋子不同的相连方法有不同的消除方式，总共16种。多方向消除的计分与单方向消除计分有些许不同，多方向得分为：（消除棋子数-5）\*2+10。而单方向的为：消除棋子数\*2。

最后2个函数为产生随机棋子的函数，两个算法大体相同，都是判断产生的随机坐标是否符合要求，再产生随机的棋子形状。若系统产生随机棋子后正好能消去，则调用消去函数，但不加分。而产生7个随机棋子的函数与前者有些不同，必须要保证生成的棋子不重复，所以每次生成随机棋子形状时，还要对其前面的棋子遍历，看是否与该棋子形状相同。若相同，则要重新生成。

## 判断类

该类主要用来进行游戏中的一些判断。内含5个成员函数。



valid\_cin()函数用来判断输入数据的有效性，定义为int类型的数据若输入的是char类型，则会抛出char类型异常，捕获到后首先要清空输入缓冲流，使用cin.clear()和cin.ignore()函数，然后重新输入数据。

end()作用为判断棋盘是否被占满，若被占满则游戏结束

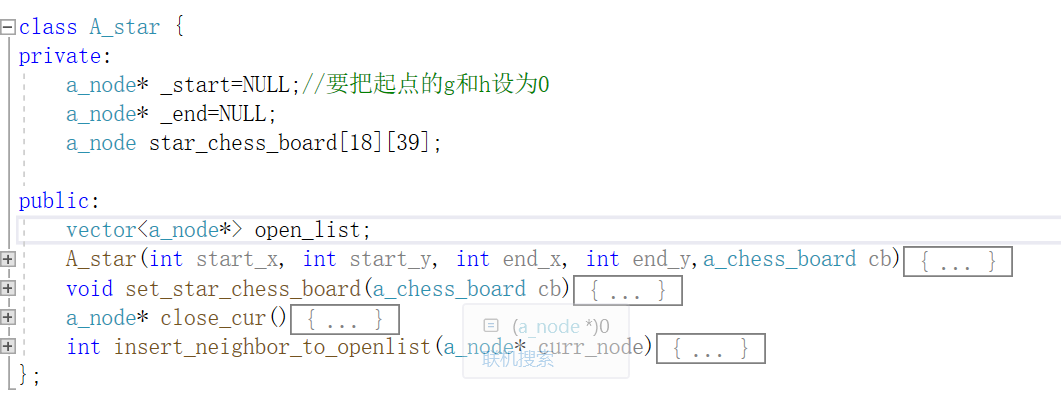
judge\_move函数判断输入输出位置是否能够满足移动需求以及是否超出棋盘范围。

judge\_elim函数主要是判断移动棋子到目标位置后是否能够消除，并给出是哪一种消除方式，按消除次数与顺序的不同组合得到总共有15种方式。

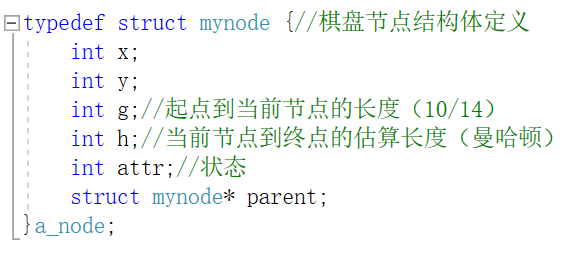
Judge\_win函数比较参赛双方的比分，给出比赛结果并输出游戏结束的标志。

## A星类

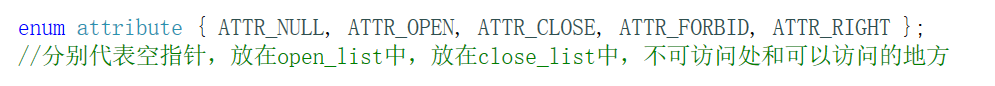
A星类主要用来判断移动路径是否有效，内含3个私有数据成员和1个公有成员及4个成员函数。



\_start是指向起始点的指针，\_end是指向终点的指针，它们都是a\_node类型的。



A\_node类型是关于棋盘节点的结构体，x、y代表节点坐标，g、h是A星算法中的两个重要参数，g为从起点到当前点的代价，h为从当前点到终点的估算代价。Attr是所在节点的状态，总共有5种可能。



Parent是指向该节点父类的指针。

star\_chess\_board是建立与chess\_board类似的棋盘数组，只是将原来的char类型改为了a\_node类型，便于实现A星寻路。

open\_list是A星算法中一个重要的容器，用来存放路径中所有有可能经过的点。而A星算法还需要一个close\_list用来存放路径已经过的节点，本程序的算法并未设置close\_list，只将节点的状态变为close。

A\_star是其构造函数，要用a\_chess\_board类型的棋盘来初始化A\_star中的节点棋盘。Set\_star\_chess\_board（）也是用来进行初始化的函数。

Close\_cur()将父节点的状态设置为close，即已作为过父节点的点不能再加入open\_list中了，避免下次重复遍历。

Insert\_neighbor\_to\_openlist函数为该类的核心函数，主要功能是将有效的节点放进open\_list中，若目标节点被找到并放入了open\_list中，则从起始节点能够走到目标节点，路径有效。否则会一直寻找，直到open\_list为空。

# 算法介绍

## 3.1消除及计分算法

该算法以目标位置的坐标作为中心进行八个方向的遍历，若棋子连续相同且均不为空格，计算与其相同的棋子个数，最后再计算出得分。消除算法与计分算法类似，都要计数单方向相同棋子数，仅最后操作不同。多方向消除以单方向消除为基础，都要调用单方向计分及消除算法，最后计分公式为：（各个方向相同棋子数+1-5）\*2+10。多方向消除还能够获得一次额外移动的机会，主要用的是goto关键字，跳转到循环起始点，让该方再进行一次输入。

## 3.2 A星算法

A\*算法是比较流行的启发式搜索算法之一，被广泛应用于路径优化领域。在 A\* 中，我们从起点开始，检查其相邻的方格，然后向四周扩展，直至找到目标。这一算法需要两个容器：open\_list与close\_list，open\_list中装的是待检查的位置，其中的位置是路径可能会是沿途经过的，也有可能不经过。close\_list中装的是已无需检查、无需计算的位置。设起点为A，A四周相邻的可走的节点则加入open\_list中，A点移出放入close\_list中，同时将A设置为open\_list中节点的父节点。设置父节点的作用为帮助我们还原整条路径，但在本程序中无需该种操作，只需判断能否到达终点即可。

A星算法能够通过遍历计算出最短路径，主要依靠的是式子f=g+h。其中的g是从起点 A 移动到指定方格的移动代价，设一次横向或纵向移动的代价为10，沿对角线的移动代价为14（对角移动距离为直线的根号二倍）。由于本程序中只考虑上下左右四个方向的移动路径，所以仅需要代价为10的数据即可。从起点到当前位置移动了n格子（不包含斜线路径），则其g值就为n\*10。

h值为从指定的方格移动到终点 B 的估算成本。这里采用的估算方法为曼哈顿方法，即计算从当前方格横向或纵向移动到达目标所经过的方格数，忽略对角移动，然后把总数乘以 10。之所以叫做曼哈顿方法，是因为这很像统计从一个地点到另一个地点所穿过的街区数，而你不能斜向穿过街区。计算h时，要忽略路径中的障碍物。

g与h值相加为f值，是该位置的总代价。将open\_list中所有节点的f值进行排序，每次都取f值最小的节点作为父节点放到close\_list中，对齐周围的位置进行检查，将有效位置放入open\_list中。当open\_list为空时，即所有有可能的点都检查过了，但找不到目标节点时，表明棋子不能移动到目标位置。若检查某一父节点的周围找到了目标节点，则棋子能够移动到目标位置。否则，若没有找到目标节点且open\_list不为空，就会一直重复上述搜索步骤。

查找目标节点是否在open\_list中需要用到c++迭代器和find函数，find函数会返回一个迭代器，若找不到则会返回指向open\_list末尾元素的下一个位置的迭代器。若某一节点被棋子占据，则将该节点设置为FORBID状态，禁止访问。而其他下棋点且为空的则设为NULL状态，允许访问。

## 3.3其他寻路算法

其他的寻路算法还有广度优先算法、Dijkstra算法、最好优先贪婪算法等，他们都与A星算法相似。这些算法都有open\_list和close\_list两个容器装载节点。

广度优先算法缺少了计算与排序的环节，它将起始点一层层向外拓展，直至所有可到达的节点都被扫描。对于每一个被扫描的节点，为其添加一个指向父节点的方向箭头。从地图上任意一点开始，只要沿着箭头的方向移动，总能走到起始点，而且走过的路径必然是最短路径之一。Dijkstra算法也有计算排序环节，只是计算公式与A星算法略有不同，只计算从起点到扫描点的移动总消耗。当A星算法的h值对g值影响微乎其微时，A星算法就退化为Dijkstra算法。最好优先贪婪算法常采用计算当前点与终点间的曼哈顿距离的方法。曼哈顿距离越小的节点，优先级越高。Dijkstra与最好优先贪婪算法的综合就是A星算法。

## 3.4对算法改进的设想

我认为还可以从以下方面对算法进行改进：

（1）本程序中采用的是4方向的A星算法，只考虑上下左右四个方向。后面可以改进为8方向，让棋子能够沿斜线移动。

（2）本程序中父节点的用处还未体现出来，可以增加起点到终点路径的输出来展现最优路径的走法。

# 4系统工作流程

## 4.1工作流程

输入起始点、终点坐标

**否**

判断输入是否正确有效

**否**

**是**

判断路径是否有效

**否**

**是**

判断是否能消除

判断游戏是否结束

**是**

游戏结束

## 4.2工作细节

1. 采用异常处理机制捕获输入错误并处理。
2. 设置的输入错误主要包括：起始点无棋子、目标位置已有棋子、输入范围错误、输入类型错误（如输入char类型）。
3. 若移动棋子后只剩余两个或一个空格，则会产生2/1个随机棋子。然后判断生成随机棋子后能否消除。若能，则消除后继续输入；若不能，生成棋子布满棋盘后结束游戏。
4. 随机棋子生成函数中有判断能否消除的功能，若能消除不会为比赛双方加分。

# 

# 5总结

为了制作该五子连珠游戏，我上4399试玩了一下同名游戏，了解规则。然后将功能及编写顺序列好，再进行一个一个难点攻克。我认为比较难的点在计分消除及路径判断上。原本我将计分与消除函数连在一起，但在多方向消除中遇到了困难，所以只能分开。这两个算法大同小异，如果今后能找到一种合适的方法将这两个组合，可以避免相似代码的重复编写。

在本项目中，我终于理解到了类先声明其中的函数，再在类外定义的好处。因为本项目的a\_chess\_board类和Judge类要相互调用，无论哪一个定义在前都会报错。后来我采用了先在头文件中声明这两个类中的成员函数，再在源文件中定义它们，这样就不会报错了。本来还想再设置一个棋手类，但因为在主函数中要对其得分进行操作，不能将其得分设置为私有成员，所以并未设置该类。

程序错误列表中显示了这样一句话“函数使用了堆栈中的…字节:超过了 /analyze:stacksize。请考虑将某些数据移到堆中。”而我对这句话的内容还不是特别了解，解决该问题目前对于我来说还较为困难。不过我也会不断地去学习与尝试解决该类问题。

学习本项目的路径判断算法，我也是受益匪浅，顺便练习了一下不算熟悉的vector和iterator的用法。虽然花了比较长的时间学习和调试，但是自己的收获也挺多的，这个项目也还有很多可以改进的空间。