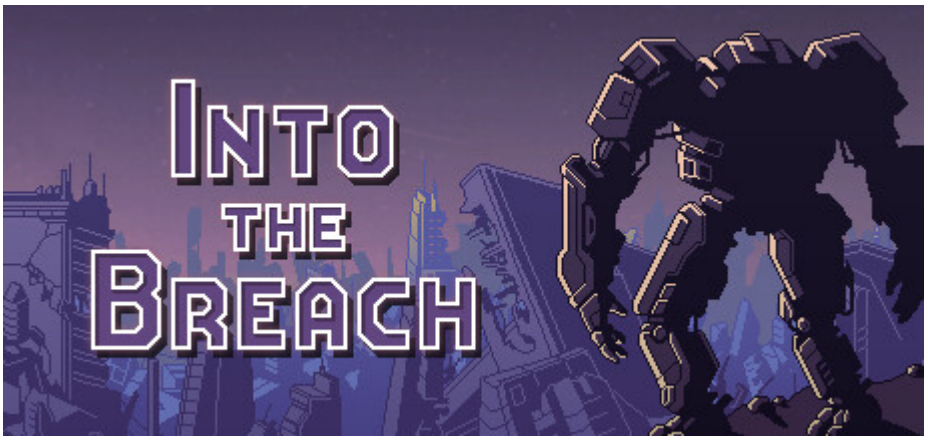


Into the Breach

基本介绍

本次大作业需要你山寨回合制战棋类游戏《陷阵之志》(Into the Breach)。在`8 x 8`的战场(`Field`)上，玩家控制己方人类单位通过移动和攻击来消灭敌方异虫单位。如果玩家成功消灭了所有的敌方单位则玩家获胜，否则失败。敌方单位的行动将由AI控制。

《陷阵之志》游戏截图

基本任务（25分）

该游戏由C++编写，其基本框架已经搭建完毕，但是关键的功能没有实现。本次大作业是**个人作业**，你需要实现游戏功能，并通过测试案例。注意，通过所有测试案例可以拿到全部的基础总分（25分），但**最终的大作业成绩还取决于能否通过隐藏测试案例**(10个案例共5分)。

为了保证你能通过隐藏测试案例，你需要使用我们提供的参考程序(demo)，对标**自行测试**（包括边缘情况），你的测试考虑周到与否将决定你是否能通过隐藏测试案例。

我们提供的文件结构如下：

```
judger_student
|- BattleField     // 提供的程序框架
|- readme          // 本说明文档和游戏手册
|- source          // 各个任务源程序的位置
|- judger.py       // 测试用Python脚本（本次测试只有一个脚本，用法参见"测试游戏"章节）
|- data            // 各个任务测试用数据
|- maps            // 自行测试用地图
|- demo            // 参考程序
```

热身（0分）：阅读并理解程序结构

你首先需要对该项目的结构有基本的了解。游戏项目的源程序结构如下：

```
BattleField
|- main.cpp          // 程序入口
|- unit.h            // 单位类头文件（访问单位的接口）
```

```
|- unit.cpp          // 单位类的实现
|- terrain.h         // 地形类头文件（访问地形的接口）
|- terrain.cpp       // 地形类的实现
|- field.h           // 战场类头文件（访问战场的接口）
|- field.cpp         // 战场类实现
|- Grid.h            // 二维网格模板类及其实现
|- actions.h         // 单位行动头文件（行动处理函数的接口）
|- actions.cpp       // 行动处理函数的实现
|- algorithms.h      // 寻路算法头文件
|- algorithms.cpp    // 寻路算法实现
|- engine.h          // 游戏引擎接口
|- engine.cpp        // 游戏引擎的实现（包括和用户交互）
```

其主要分为以下几个部分：

1. 单位类的接口和实现（`unit.h`和`unit.cpp`）。定义了名为`Unit`的类表示单位。类中关键成员数据包括单位的`side`（其中`side = true`时为玩家控制的单位，否则属于敌方单位）。
2. 地形类的接口和实现（`terrain.h`和`terrain.cpp`）。定义了名为`Terrain`的类表示地形。
3. 二维网格模板类及其实现（`Grid.h`）。详细描述及示例见下方二维网格类章节。
4. 战场类的接口和实现（`field.h`和`field.cpp`）。详细描述见下方战场类章节。
5. 单位行动的接口和实现（`actions.h`和`actions.cpp`）。定义单位能够采取的行动种类以及采取行动的效果。
6. 寻路算法的接口和实现（`algorithms.h`和`algorithms.cpp`）。定义了单位确定移动范围的寻路算法，详见接下来的任务描述。
7. 游戏引擎（`engine.h`和`engine.cpp`）。定义了回合制游戏进行的主要循环过程。给定一个初始战场`field`，调用`play`函数开始游戏。具体实现中还包括如何和用户进行交互，推动游戏进行。
8. 程序入口(`main.cpp`)。定义了一个初始战场，通过函数`loadMap`加载地形和单位，并调用`play`函数开始游戏。

你需要在总体上掌握上述代码结构，为下面的任务打下基础。我们建议在`BattleField`路径下进行开发与手动调试，完成一个任务后将源代码复制到`source`路径下对应的文件夹中，具体见后续的“测试游戏”和“提交格式部分”

除了本文档外，我们另外提供了一个游戏手册（`game_manual`），记录了游戏的总体设计，在进行如下任务时，可以参考该手册以理解具体的游戏细节和规则。

下面具体介绍较为复杂的二维网格类和战场类。

二维网格类

二维网格是实现战棋游戏的关键数据结构。二维网格类`Grid<T>`具有2部分关键信息：它的大小`n x m`，表示`n`行`m`列；它的每个格子中存的类型为`T`的数据，使用下标（从0开始）访问，比如`grd[i][j]`表示`grd`中第`i`行第`j`列的数据。在初始化一个网格时，它的大小会被固定下来，后续我们可以访问和修改每个格子的数据。示例如下：

```
// 声明一个大小是0 x 0，存储bool类型数据的网格
Grid<bool> empty_grd;

// 声明一个大小是8 x 8，存储int类型数据的网格
Grid<int> int_grd(8, 8);

// 声明一个大小是8 x 8，存储bool类型数据且每个格子被初始化为true的网格
Grid<bool> int_grd2(8, 8, true);
```

```
// 将int_grd2中第0行第1列中的数据赋值为false
int_grd2[0][1] = false;

// 获取网格的行数和列数
cout << int_grd2.numRows() << endl; // 输出 8
cout << int_grd2.numCols() << endl; // 输出 8

// 检查传入的行列坐标是否在该网格的合法范围内
if (!int_grd2.inBounds(8, 7))
    cout << "not in bounds" << endl; // 行数8超出范围，因此会输出 "not in bounds"
```

战场类

战场 `Field` 是一块 `8 x 8` 棋盘，每个格子上分布着单位和地形。因此，二维网格 `Grid` 非常适合用来存储单位和地形信息。类 `Field` 的成员数据如下：`units` 存储单位信息，如果 `units[i][j]==nullptr` 时，表示坐标 `(i,j)` 无单位，否则 `units[i][j]` 包含指向坐标 `(i,j)` 单位的指针。`terrains` 存储地形信息，`terrains[i][j]` 表示坐标 `(i,j)` 处的地形。战场类的一些基础方法已经实现好，你需要根据后续任务要求增加新的方法或修改已有方法，来达到功能需求。

```
class Field {
public:
    // some methods
private:
    // Store the units
    Grid<Unit*> units;
    // Store the terrains
    Grid<Terrain> terrains;
};
```

剩余的章节描述你需要完成的任务，**所有需要完成的任务将由黑体表粗显示**。

1. 实现地图的加载和单位的移动（5分）

1.1 地图加载

在初始程序中，我们在 `main` 函数中定义了一个大小为 `8 x 8` 的空地图。空地图默认地形为平原(`PLAIN`)，并且没有任何单位。**你需要实现函数 `loadMap`，从输入流中装载一个新的战场地图。**该函数应该定义在 `engine.cpp` 中，其原型定义在 `engine.h` 中：

```
#include <iostream>
#include "field.h"

// Load map
void loadMap(std::istream& is, Field& field);
```

给定一个提供地图信息的输入流 `is`，该函数需要从中读取地形和单位，修改 `field` 的 `units` 和 `terrains` 成员变量。因此，**你还需要在Field类中实现对应的成员函数用于加载地形和单位。**注意，`is` 可以绑定在 `cin` 上，也可以绑定在文件输入流上，因此 `loadMap` 可以通过读取 `cin` 或者文件来加载地图。

输入流对应的文件提供地图信息，并遵从一定的格式，我们以如下例子说明地图格式：

```
2 3
3 3 M
6 6 O
2 3 S
2 5 S
6 3 T
```

- 第一行`NT NU`:代表一共有`NT`个特殊地形（除平原）和`NU`个单位。在上面的例子中，`NT=2`，`NU=3`。
- 接下来`NT`行特殊地形信息，每行格式为`R C T`，`R`代表行数（row），`C`代表列数（column）。这一行输入代表在`(R,C)`坐标有一个地形为`T`。`T = O`代表海洋(`OCEAN`)，`T = M`代表山脉(`MOUNTAIN`)。如果某个坐标没有指定任何类型，则该坐标为默认地形平原(`PLAIN`)。地形类声明和定义在`terrain.h`和`terrain.cpp`。在上面的例子中，坐标`(3,3)`为山脉，`(6,6)`为海洋，其余为平原。
- 接下来`NU`行代表单位信息，每行格式为`R C U`，代表在`(R,C)`坐标有一个单位为`U`。`U = S`代表士兵(`SOLDIER`)，`U = T`代表坦克(`TANK`)。在本次任务中，只有士兵和坦克两种单位，*后续任务会增加新的单位*。单位类声明和定义在`unit.h`和`unit.cpp`。在上面的例子中，坐标`(2,3)`和`(2,5)`为士兵，`(6,3)`为坦克。
- 我们提供的地图不会出现单位和单位、特殊地形和特殊地形出现在同一个位置的情况。

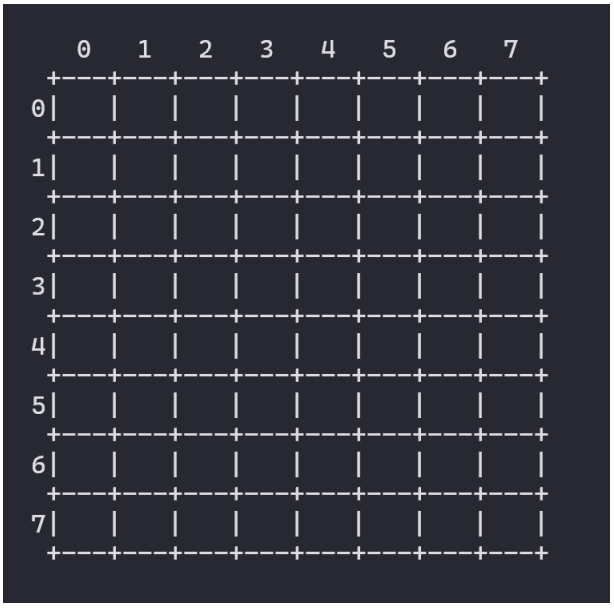
1.2 打印地图

游戏过程需要将地图信息打印出来，以便玩家进行下一步的决策。打印地图的函数`displayField`的基本框架已经提前实现好了，原型声明在engine.h:

```
// Display the battle field
void displayField(std::ostream& os, const Field& field,
                  const Grid<bool>& grd = Grid<bool>(), dp_mode dp = DP_DEFAULT);
```

`displayField`有四个输入参数，调用其的效果是将`field`战场中的信息打印到输出流`os`中。类似任务1.1的`loadMap`函数的输入流参数`is`，`os`可以绑定标准输出`cout`，也可以绑定文件输出流。`displayField`的第三和第四个参数`grd`和`dp`可以控制`displayField`打印地图上的额外信息。这两个参数将在任务1.3中被用到，当前任务可以先忽略。

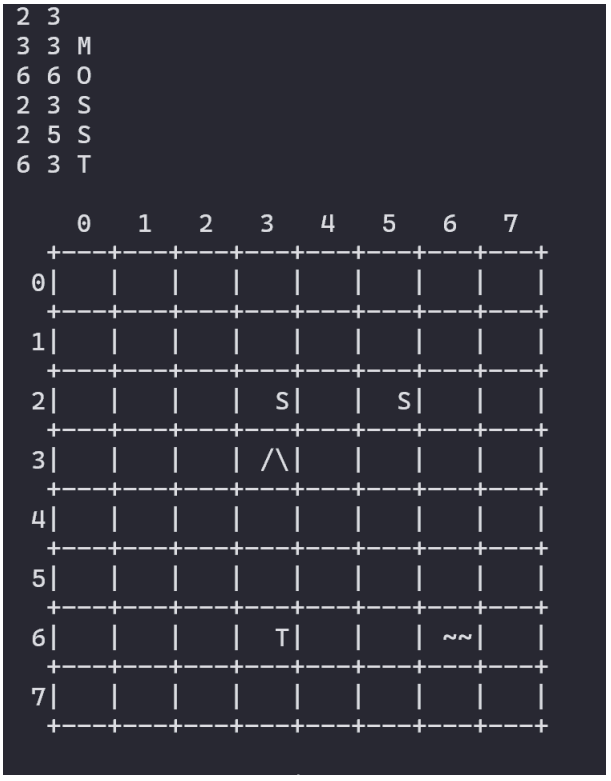
运行函数`displayField`将打印如下的空地图：



本任务的目标是在打印地图时显示在任务1.1中加载的特殊地形和单位：

- 特殊地形有如下显示效果：海洋显示为`~~`，山脉显示为`/\`。
- 单位根据其类型和阵营显示首字母。己方单位显示大写字母，敌方单位显示小写字母。任务1只包含士兵和坦克，它们的默认阵营都是己方单位。
- 如果该格子上存在单位，优先打印单位的符号；如果没有单位，则打印地形符号。

以前面的输入为例，在正确实现`loadMap`后调用`displayField`后地图应显示为



在具体实现中，`displayField`会调用`Unit`类的`getSymbol`方法和`Terrain`类的`getSymbol`方法，来获得相应的符号信息：

```
void displayField(ostream& os, const Field& field, const Grid<bool>& grd, dp_mode dp) {
    // some operation

    // print information in (n, m) of field,
    if (u != nullptr) sym += u->getSymbol();
    else sym += t.getSymbol();
    os << setw(width) << sym;

    // other operation
}
```

在上述代码中，`u`和`t`分别表示`field`坐标`(n, m)`上的单位指针和地形对象。你需要仔细阅读`displayField`的实现并理解打印地图的流程，**通过修改`Unit`类的`getSymbol`方法和`Terrain`类的`getSymbol`方法（`displayField`本身不需要修改），以显示单位和地形。**

1.3 移动单位

在加载好地图（`loadMap`）后，游戏开始（`play`）。函数`play`的定义在`engine.cpp`中：

```
// Main loop for playing the game
void play(Field& field, istream& is, ostream& os) {
    while (is) {
      displayField(os, field);

      // 你需要在这里添加代码

      if (is.eof()) break;
    }
}
```

函数`play`的作用是控制游戏的流程：通过输出流`os`与用户进行交互，并根据`is`读到的用户输入，调用`field`的方法（后续你需要根据任务增加新的方法）修改其状态，循环往复。当前的`play`函数只是在循环打印地图。**你需要修改`play`函数，实现用户对单位行动的操控**。本次任务中，行动有2种：1. 移动（Move）；2. 取消选取（Skip）。有关行动的声明和定义在`actions.h`和`actions.cpp`中。

每一轮操控单位对应着`play`函数内`while`的一次循环，其流程为：

1. 在每轮操控开始时，游戏会打印出当前的地图（`displayField(os, field);`）
2. 游戏询问玩家选取单位，打印`Please select a unit:`并换行
3. 游戏读取玩家输入坐标`R C`。若玩家选取的坐标`(R,C)`没有单位，则打印`No unit at (R, C)!`并换行，回到步骤2
4. 游戏会根据位于`(R,C)`的单位可以执行的行动，以`1.A 2.B ...`的格式（每个选项之间用空格分隔）打印可选项并换行，并继续询问玩家想要执行的行动，打印`Select your action:`并换行。任务1中，己方单位可以执行移动和取消选取，因此在打印可选项时会打印`1.Move 2.Skip`（`Move`在前，`Skip`在后）
5. 游戏读取玩家输入的行动`N`。若玩家选择的行动的数字`N`不在选择列表中（比如0, -1, 4），则打印`Invalid action!`并换行，继续询问玩家想要执行的行动，打印`Select your action:`并换行。然后重复步骤5，直至`N`在选择列表中
6. 游戏执行行动`N`，在执行过程中可能会继续和玩家交互。
    - 若玩家选择的行动是移动:

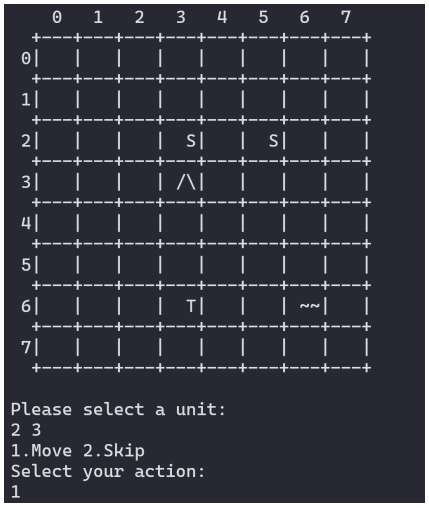
      6.1. 游戏*调用寻路算法得到单位可以移动到的目标坐标，并打印附带目标坐标信息的地图*
      6.2. 游戏询问玩家移动的目标坐标，打印`Please enter your destination:`并换行
      6.3. 游戏读取玩家输入坐标`R1 C1`。若`(R1,C1)`不可达，打印`Not a valid destination`并换行，回到6.2
      6.4. 游戏移动单位到`(R1,C1)`
    - 若玩家选的行动是取消选取，什么都不用做

关于异常输入数据，大作业的所有测试案例中只会出现本文档提到过的情况（比如上面的“选取单位时发现坐标无单位”），之后的任务2、3和4也遵循这一规则。在任务1中，正常输入数据包含完整的1或多轮操控单位，每轮不会出现选取了单位或是选择了移动行动后没有后续输入的情况。

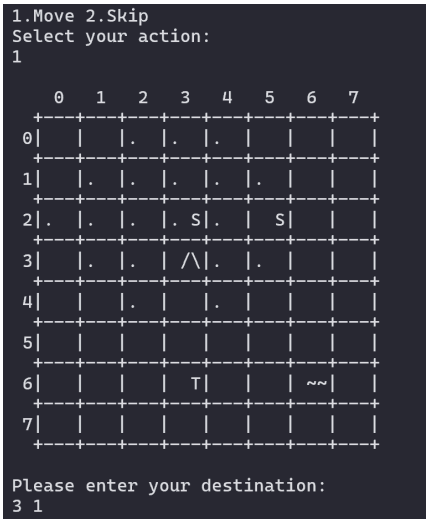
我们对上述规则中提到的*可以移动到的目标坐标*作进一步解释。单位根据自身类型会有不同的移动力，移动力会在每次行动完成后恢复。每到达一个格子，单位会消耗移动力，消耗的点数由单位的位置类型（比如是"陆地"还是"空中"）和该格子上的地形类型决定（例如，空中单位可以越过海洋，但是陆地单位不行）。每种单位是空中还是陆地单位，以及它们固定的移动力是多少，在游戏手册中有详细介绍。可以移动到的目标坐标是指单位从自身位置出发，到达该坐标时移动力依然大于等于0的坐标。

一次执行移动的示例如下面三张图所示。套用上述的流程，这次移动的交互过程可以被解释为：

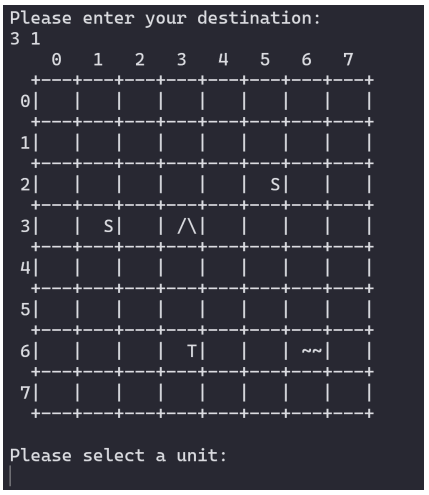
1. 游戏打印地图
2. 游戏询问玩家选取单位，打印`Please select a unit:`
3. 游戏读取玩家输入坐标`2 3`
4. 游戏根据位于`(2,3)`的单位可以执行的行动，打印可执行的行动`1.Move 2.Skip`，并继续询问玩家想要执行的行动，打印`Select your action:`
5. 游戏读取玩家输入的行动`1`（移动）
6. 游戏打印地图，位于`(2,3)`的士兵所有可以移动到的目标坐标都有一个点来标识，询问玩家移动的目标坐标，打印`Please enter your destination:`，并读取玩家选择移动的目标坐标`(3,1)`，使单位的移动生效。



移动示例：步骤1，2，3，4，5



移动示例：步骤6，执行移动时的交互



移动示例：下一轮操控的步骤1和2

本次任务中，较为复杂的是实现执行移动时的交互过程（也就是示例中的步骤6）。我们提供了位于`engine.cpp`的框架函数`performMove`供你参考，其中步骤6.1已基本实现。步骤6.1可以被拆分成两步，计算所有可以移动到的坐标和将这些坐标通过地图打印出来。在`performMove`中，第一步可以通过调用已经提供的寻路算法`searchReachable`完成，第二步可以通过调用`displayField`完成。

```
// Perform the move action
// The implementation is incomplete
bool performMove(ostream& os, istream& is, Field& field, Unit* u)
{
    // 步骤6.1的基本框架
    Grid<bool> grd =
        searchReachable(getFieldCosts(field, u), u->getRow(), u->getCol(), u-
>getMovPoints());

    displayField(os, field, grd, DP_MOVE);

    // 你需要在这里添加代码，实现步骤6执行移动的剩余步骤
```

```
    return true;
}
```

在上述函数中，我们首先调用`searchReachable`来计算单位`u`所有可以移动到达的坐标，用一张地图`grd`保存结果，其中`grd[n][m]==true`表示坐标`(n,m)`可达。接着，我们调用`displayField`函数（注意前面提到的`displayField`原型的后两个参数），并将存储可达坐标信息的`grd`和展示模式`DP_MOVE`（展示移动信息）传入。这个意思是说，`displayField`函数在打印位于`(n,m)`的格子的信息时，除了打印`(n,m)`的单位或特殊地形的符号，还会检查`grd[n][m] == true`是否成立，如果成立，则按照展示模式，在符号信息前附加展示信息。比如在这里，`DP_MOVE`（展示移动信息）被传入，那么`displayField`函数在打印地图时，需要把单位所有的可达格子打上`.`的标记。

为了实现上述功能，我们需要了解寻路算法`searchReachable`的使用方式（注意你不需要修改该算法，只需要调用它实现寻路功能）。其函数原型在algorithms.h中：

```
// Given movement points (pts), calculate
// which squares can be reached starting from (row, col)
Grid<bool> searchReachable(const Grid<int>& costs, int row, int col, int pts);
```

其中，`costs`是一张移动力消耗地图，`row`和`col`表示寻路算法的起始坐标，`pts`表示起始的移动力。函数`searchReachable`的作用是计算从`(row,col)`出发所有消耗在移动力范围内的坐标，返回一张类型是`Grid<bool>`的可达坐标地图。

以上面执行位于`(2,3)`的士兵的移动为例（即在`performMove`函数中，变量`u`指向该士兵），为其调用`searchReachable`需要提供上述的四个参数。其中，`row=2`，`col=3`。查游戏手册知道士兵的移动力为3，因此**调用**`u->getMovPoints()`应得到`pts=3`。，查游戏手册可以知道士兵为陆地单位，根据当前地图，**调用**`getFieldCosts`应得到如下移动力消耗地图`costs`：

```
  0 1 2   3 4   5   6 7
0 1 1 1   1 1   1   1 1
1 1 1 1   1 1   1   1 1
2 1 1 1   0 1 100   1 1
3 1 1 1 100 1   1   1 1
4 1 1 1   1 1   1   1 1
5 1 1 1   1 1   1   1 1
6 1 1 1 100 1   1 100 1
7 1 1 1   1 1   1   1 1
```

其中，`(2,3)`是士兵所处的坐标，其移动力消耗为0；移动力消耗为1的格子是原始地图中没有单位的平原地形；移动力消耗为100的格子存在着其他单位或特殊地形。以`(2,3)`为起点，`searchReachable`会根据`costs`不断消耗`pts`并记录达到的坐标，直到`pts=0`，返回如下的网格地图（也即`performMove`中`grd`的值），表示的所有可达坐标（这里使用1简写`true`，记录可达坐标）：

```
  0 1 2 3 4 5 6 7
0 0 0 1 1 1 0 0 0
1 0 1 1 1 1 1 0 0
2 1 1 1 1 1 0 0 0
3 0 1 1 0 1 1 0 0
4 0 0 1 0 1 0 0 0
5 0 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0 0
7 0 0 0 0 0 0 0 0
```

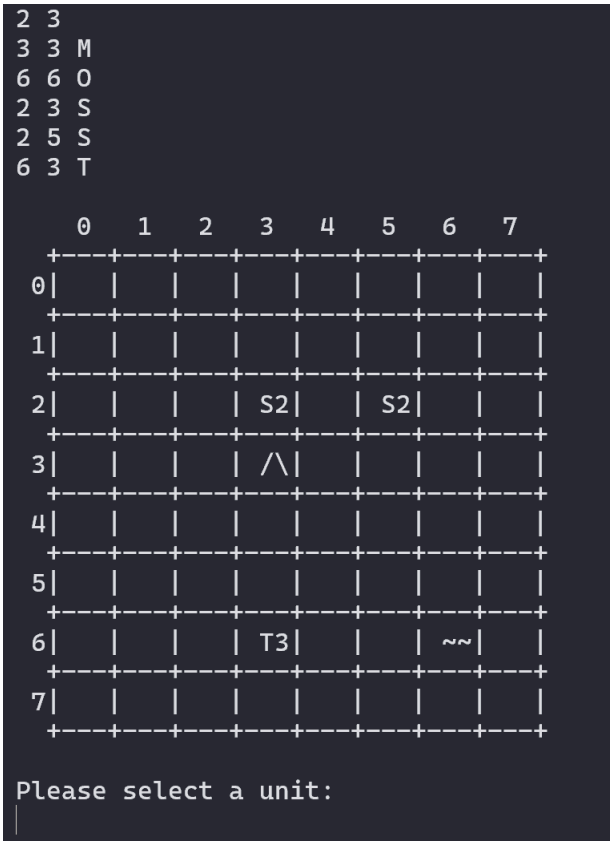
回到步骤6，你需要完成的任务如下。当前版本的`performMove`调用了未完全实现的函数`getFieldCosts`和`Unit`类的成员方法`getMovPoints`，以及`displayField`函数中调用了未完全实现的函数`getDpSymbol`（请查看`displayField`的源码），导致在打印前面例子中步骤6的交互过程时，得到的可达目标坐标地图和步骤1中的打印结果相同，即没有格子有`.`标记。因此，在本次任务中，**你需要修改函数`getFieldCosts`，`getMovPoints`和`getDpSymbol`，以达到步骤6.1正确的显示效果，并在此基础上，为`performMove`添加新的代码，实现操控流程步骤6执行移动的剩余步骤**。

2. 单位的生命值和攻击系统（6分）

本任务要求你设计每个单位的生命值，添加敌方单位，并实现攻击效果。

2.1 单位的生命值

每个单位有自身的生命值（HP），在加载地图时需要根据单位的类型初始化单位的生命值，并在打印地图时显示生命值信息。在下图的例子中，士兵的生命值初始为2，在地图中会显示`S2`；坦克的生命值初始为3，会显示`T3`。**你需要修改`Unit`类来完成这个功能**。



*注意：*在大作业的所有测试用例中，游戏运行过程中所有单位的生命值可能的范围是1到9。每种单位拥有不同的初始生命值，具体数值在游戏手册中有详细介绍。

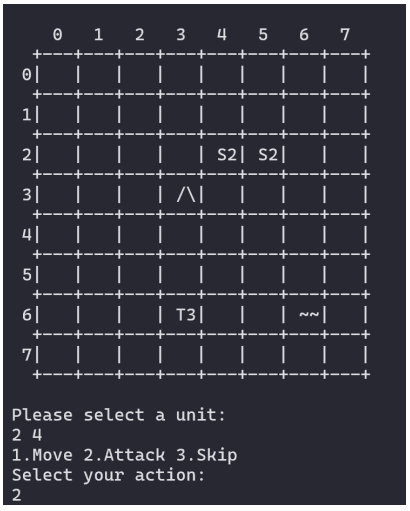
2.2 增加单位的新行动：攻击

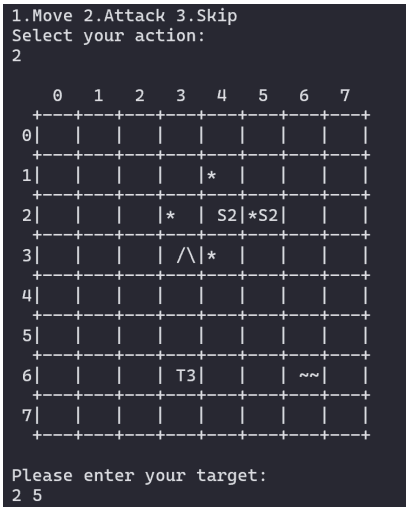
在任务1中，每个单位的行动有移动和取消选取，现在你需要为单位实现攻击(Attack)行动。你需要修改的步骤和增加的功能如下:

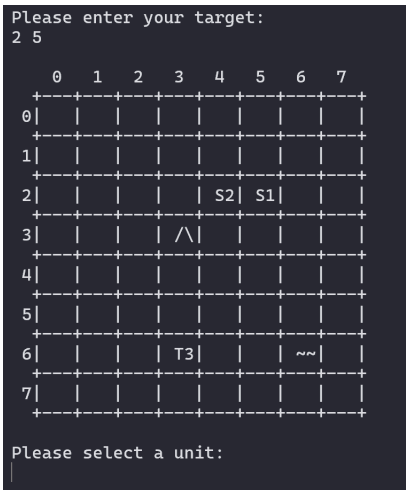
1. 加载地图。在加载地图时需要根据单位的类型初始化单位的攻击力；
2. 询问玩家选择行动。在询问玩家选择行动时，需要加上攻击的选项。三种行动的显示优先级为Move > Attack > Skip，即`1.Move 2.Attack 3.Skip`;
3. 在执行行动时和玩家进行的交互。
    - 如果玩家选择的行动是攻击，游戏会根据单位的攻击方式的范围和当前地图，计算得到可以攻击的目标坐标，并打印附带攻击目标坐标的地图，**可攻击坐标在地图上用`*`标识**
    - 游戏询问玩家攻击的目标坐标，打印`Please enter your target:`并换行。然后读取玩家输入的攻击坐标，若该坐标不在攻击范围内，游戏打印`Not a valid target`并换行。重复本步骤，直到玩家的输入坐标在攻击范围内

- 当玩家选定攻击的坐标后，游戏根据单位的攻击效果执行攻击
- 攻击效果执行完毕后，游戏将生命值小于等于0的单位移除战场

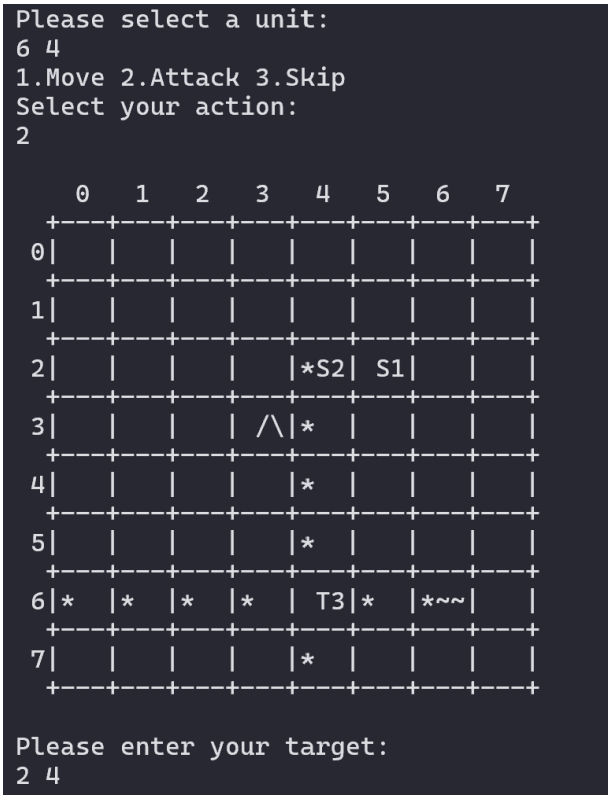
我们通过下面例子演示攻击效果。该例子中位于`(2,4)`的士兵攻击位于`(2,5)`的士兵。士兵是一种近战单位，只能攻击其相邻格子中的一个格子。可以看到，当玩家选择攻击行动时，游戏会打印出该士兵单位所有可以攻击的目标坐标，在地图上用`*`表示。士兵的攻击效果是对目标单位造成等同于自身攻击力的伤害，因此位于`(2,5)`的士兵的生命值变为`2-1=1`。如果HP降低到0，则该单位会死亡，从地图上消失。

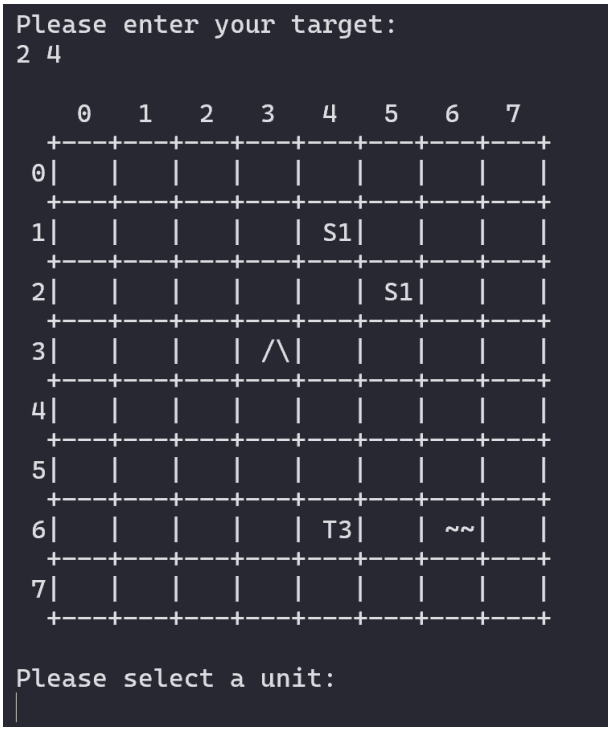






接下来，我们演示坦克是如何进行攻击的。坦克是一种远程攻击单位，可以攻击上下左右任意距离的一个单位或地形，并对目标单位造成等同于自身攻击力的伤害。注意坦克的攻击范围没有穿透效果，其射程沿着一个方向碰到单位或者非平原的地形就会被阻挡，如下面左图`*`所示。坦克还有特殊的击退效果，被攻击的目标单位将会后退一个格子。下面的例子显示，位于`(6,4)`的坦克攻击`(2,4)`的士兵的效果。原本位于`(2,4)`的士兵的生命值变为`2-1=1`，并且被击退到了`(1,4)`。





此外，坦克的攻击可能对地形造成破坏。比如下面的例子展示了位于`(6,3)`的坦克攻击位于`(3,3)`的山脉造成山脉被夷为平原。

```
Please select a unit:
6 3
1.Move 2.Attack 3.Skip
Select your action:
2

    0   1   2   3   4   5   6   7
  +---+---+---+---+---+---+---+---+
 0|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 1|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 2|   |   |   | S2|   | S2|   |   |
  +---+---+---+---+---+---+---+---+
 3|   |   |   |*/\|   |   |   |   |
  +---+---+---+---+---+---+---+---+
 4|   |   |   |*  |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 5|   |   |   |*  |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 6|*  |*  |*  | T3|*  |*  |*~~|   |
  +---+---+---+---+---+---+---+---+
 7|   |   |   |*  |   |   |   |   |
  +---+---+---+---+---+---+---+---+

Please enter your target:
3 3
```

```
Please enter your target:
3 3

    0   1   2   3   4   5   6   7
  +---+---+---+---+---+---+---+---+
 0|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 1|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 2|   |   |   | S2|   | S2|   |   |
  +---+---+---+---+---+---+---+---+
 3|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 4|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 5|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 6|   |   |   | T3|   |   | ~~|   |
  +---+---+---+---+---+---+---+---+
 7|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+

Please select a unit:
```

击退效果可能和地形产生互动，造成其他后果。下面的例子演示了位于`(4,3)`的坦克单位被坦克攻击，生命值减1（`3-1=2`）并且被击退。由于其身后是山脉，击退效果导致山脉碰撞崩塌，该坦克保持原地不动。此外，由于撞击导致其生命值减1（`2-1=1`）。如果HP降低到0，则该单位会死亡，从地图上消失。

```
Please select a unit:
6 3
1.Move 2.Attack 3.Skip
Select your action:
2

    0   1   2   3   4   5   6   7
  +---+---+---+---+---+---+---+---+
 0|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 1|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 2|   |   |   | S2|   |   |   |   |
  +---+---+---+---+---+---+---+---+
 3|   |   |   | /\|   |   |   |   |
  +---+---+---+---+---+---+---+---+
 4|   |   |   |*T3|   |   |   |   |
  +---+---+---+---+---+---+---+---+
 5|   |   |   |*  |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 6|*  |*  |*  | T3|*  |*  |*~~|   |
  +---+---+---+---+---+---+---+---+
 7|   |   |   |*  |   |   |   |   |
  +---+---+---+---+---+---+---+---+

Please enter your target:
4 3
```

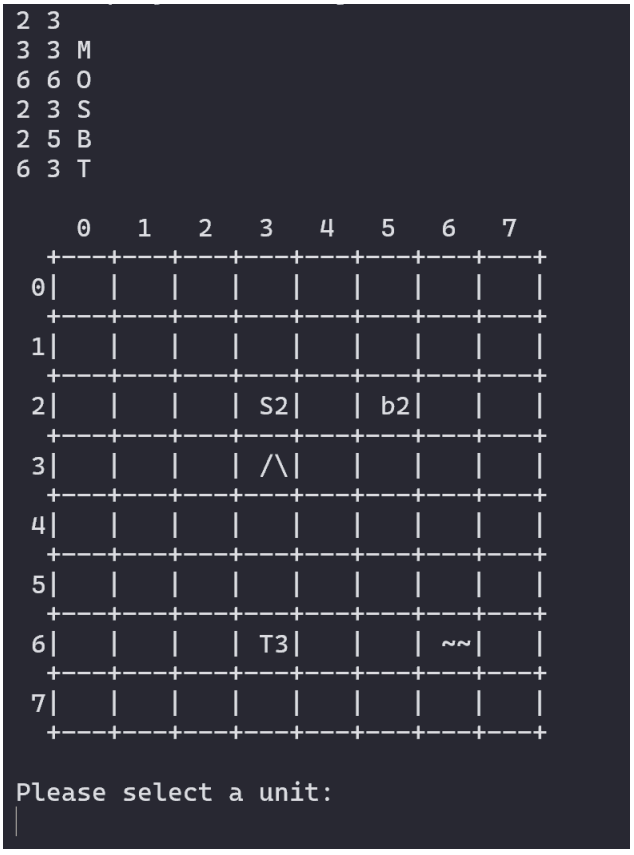
```
Please enter your target:
4 3

    0   1   2   3   4   5   6   7
  +---+---+---+---+---+---+---+---+
 0|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 1|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 2|   |   |   | S2|   |   |   |   |
  +---+---+---+---+---+---+---+---+
 3|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 4|   |   |   | T1|   |   |   |   |
  +---+---+---+---+---+---+---+---+
 5|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
 6|   |   |   | T3|   |   | ~~|   |
  +---+---+---+---+---+---+---+---+
 7|   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+

Please select a unit:
```

上面的例子并没有覆盖所有的击退效果（例如，如果被击退的陆地单位掉入海中则会立即死亡）。其他可能的击退效果参见游戏手册。需要注意的是，如果被击退到的坐标有单位存在，不管该坐标是什么地形，两个单位的生命值减1，并保持原地不动。

本任务要求你需要修改Unit类，加入攻击力数值，并且修改游戏引擎实现上述攻击效果。每种单位的攻击力、攻击类型、攻击范围、特殊机制在游戏手册中有详细的介绍。

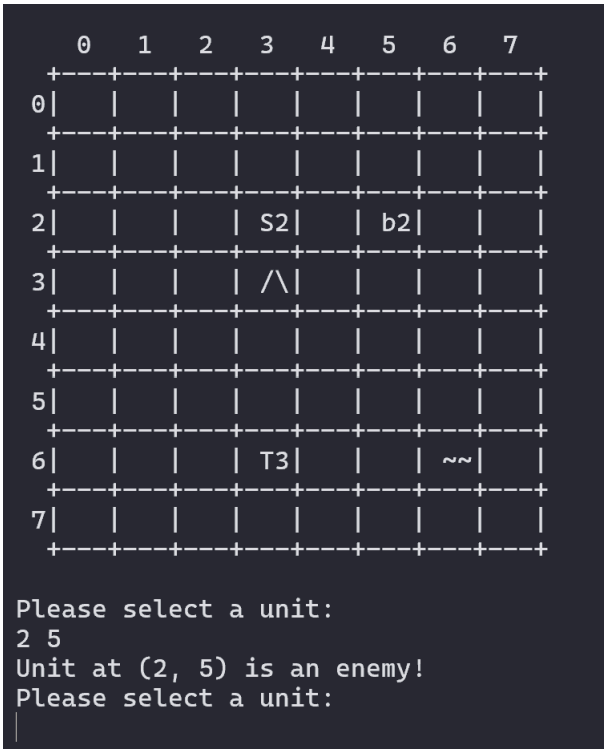
2.3 增加敌方单位：蜜蜂

在加载地图中，可能会出现新单位蜜蜂，蜜蜂是敌方单位。一个加载蜜蜂的示例如下，其中，输入的`2 5 B`表示在战场的`(2,5)`坐标加载蜜蜂单位。在打印地图时，因为蜜蜂是敌方单位，所以用小写字母`b`标识。



由于增加了敌方单位，需要额外的异常输入检查：

- 在询问玩家选取单位时，若玩家输入的坐标`(R, C)`是敌方单位，打印`Unit at (R, C) is an enemy!`并换行，重新询问玩家选取单位。一个例子如下图所示，坐标`(2,5)`是蜜蜂，因此输出`Unit at (2, 5) is an enemy!`，重新询问玩家选取单位。



本任务要求你需要修改Unit类，根据游戏手册加入蜜蜂的数值信息，并且修改游戏引擎实现上述异常输入效果。

3. 回合制和敌方AI（6分）

在本次任务中，我们需要让敌方单位动起来，并发起攻击。在此之前，我们首先要明确划分玩家回合和敌方回合。在玩家回合中，玩家可以操控单位的行动；在敌方回合中，游戏内的AI操控敌方单位的行动。

完成本次任务后，函数`play`的流程大致如下：

```
void play(Field& field, istream& is, ostream& os) {
    while (is) {
      打印地图
      检查胜利条件(3.2)
      玩家回合(3.1)
      敌方回合(3.3)
    }
}
```

3.1 玩家回合

在任务1和任务2中，玩家操控己方单位的一次行动，被称为**一轮**。玩家的**一回合**由0至多轮组成。每回合中，每个己方单位可以执行最多1次移动行动，最多1次攻击行动，两者没有先后顺序的要求。除"取消选取"行动外，如果单位还可以执行别的行动，则称该单位是**可行动的**。

每回合的流程如下：

1. 重置所有己方单位的行动状态，即每个己方单位在回合的初始阶段都没有执行过移动和攻击行动。
2. 检查是否还有可行动的单位。如果有，进入3；如果没有；打印`No more actable units.`并换行，结束本回合。
3. 打印地图。**需要特别注意的是，可行动单位在地图上用`+`标识**。例如，HP为3的坦克`T`在地图上打印的结果是`+T3`。

4. 询问玩家是否结束当前回合。打印`End this turn (y,n)?`并换行。如果玩家输入`y`，结束本回合；如果玩家输入`n`，进入5。
5. 进入每一轮操控单位的流程

   5.1 询问玩家选取单位。若玩家选取的坐标`(R,C)`没有单位，则打印`No unit at (R, C)!`。若玩家输入的坐标`(R, C)`是敌方单位，打印`Unit at (R, C) is an enemy!`。若玩家选择的坐标`(R,C)`是不可行动的友方单位，打印`Unit at (R, C) is not actable!`并换行。重新执行5.1，直至选取的是可行动的友方单位

   5.2 询问玩家选择行动。只打印还可以执行的行动。比如选取的单位已经执行过攻击，但还没有执行过移动，则打印`1.Move 2.Skip`

   5.3 执行选取的行动，如果移动（或攻击），记录选取单位已执行过该行为，不可以再次移动（或攻击）

   5.4 重新进入步骤2。

完成任务3.1后，函数`play`的结构大致如下：

```
void play(Field& field, istream& is, ostream& os) {
    while (is) {
      打印地图
      玩家回合
    }
}
```

我们以下面6张图为例展示实现玩家回合流程后的游戏交互过程，其文字解释为：

a. 游戏读入初始地图，进入`play`函数的`while`循环，并打印地图，进入玩家回合

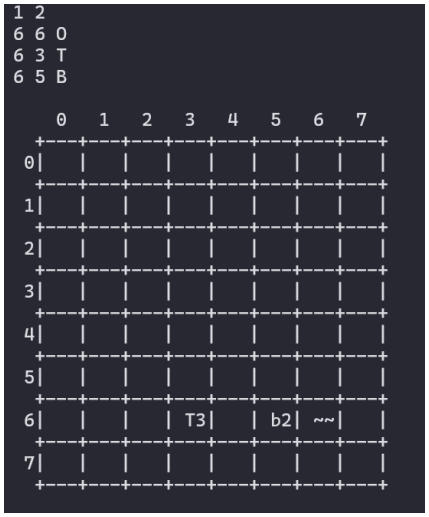
b. 在玩家回合，游戏初始化所有己方单位为*可行动单位*，并进入玩家回合内的循环。游戏检查还有可行动的己方单位（`(6,3)`的坦克），打印附带可行动信息的地图，并询问玩家是否要结束当前玩家回合。在例子中，玩家选择不结束当前回合

c. 玩家操控位于`(6,3)`的坦克攻击了位于`(6,5)`的蜜蜂，操控的流程和任务2一致。游戏会记录位于`(6,3)`的坦克在本玩家回合中已经进行过攻击

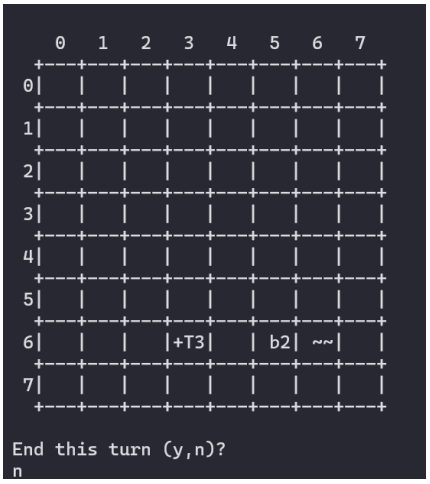
d. 游戏检查还有可行动的己方单位（`(6,3)`的坦克），打印附带可行动信息的地图，并询问玩家是否要结束当前玩家回合。在例子中，玩家选择不结束当前回合

e. 这次玩家操控位于`(6,3)`的坦克执行移动。需要注意的是，这次游戏打印可执行行动列表时，打印的是`1.Move 2.Skip`，因为该坦克已经执行过攻击

f. 游戏检查发现已经没有可行动的友方单位，打印`No more actable units.`，自动结束当前玩家回合

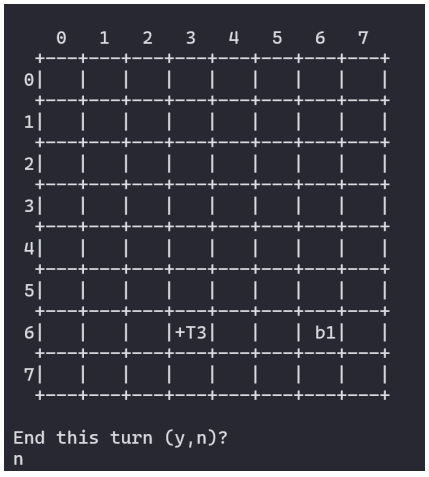


a.加载地图，进入play函数的循环，打印地图

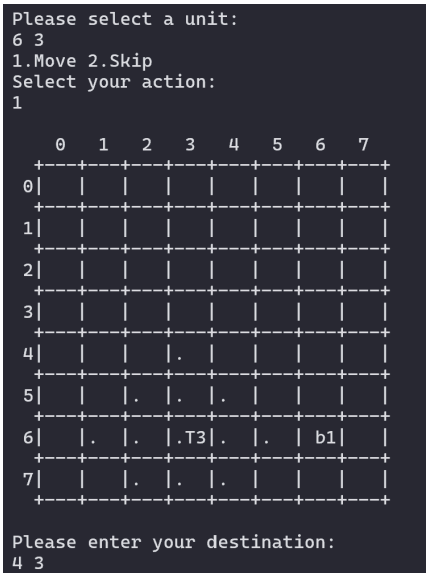


b.进入玩家回合，对应步骤1，2，3，4

c.操控单位的攻击行动，对应步骤5



d.因为还有可行动的单位，继续执行步骤2,3,4



e.操控单位的移动行动，对应步骤5

f.无可行动单位，结束玩家回合，进入play的下一轮循环

在本次任务中，**你需要修改`play`函数的逻辑，为`Unit`类增加相关成员数据记录其执行过的行动，来实现玩家回合的交互流程**。

3.2 检查胜利条件

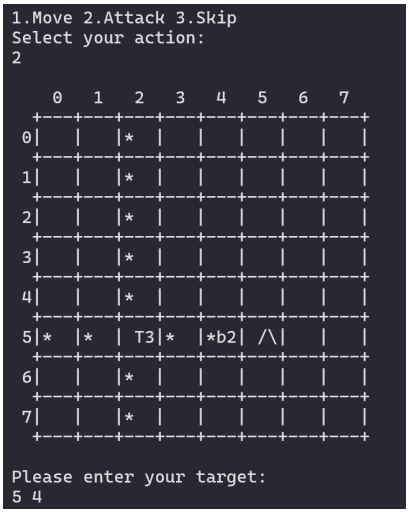
在每次玩家的回合开始前，检查地图上是否还存在敌方单位。

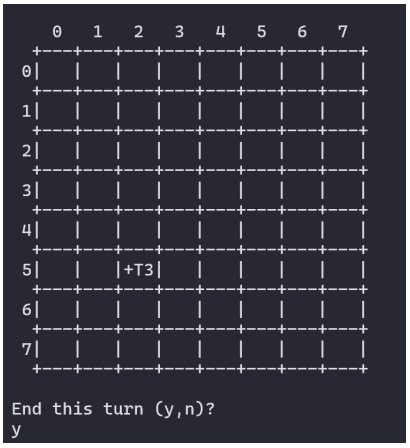
1. 若没有敌方单位（无论是否还有己方单位），打印`won`并换行，结束本场游戏；
2. 若还存在敌方单位，
   2.1 若已没有己方单位，打印`Failed`并换行，结束本场游戏；
   2.2 若还存在己方单位，进入玩家回合。

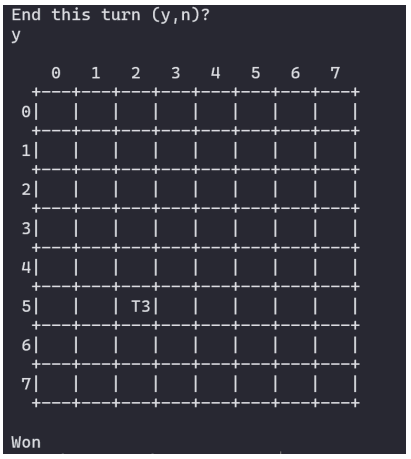
完成任务3.2后，函数`play`的流程大致如下：

```
void play(Field& field, istream& is, ostream& os) {
    while (is) {
      打印地图
      检查胜利条件
      玩家回合
    }
}
```

下面这个例子展示达到胜利条件时游戏的效果。在某个玩家回合中，玩家操控位于`(5,2)`的坦克攻击了位于`(5,4)`的蜜蜂，接着选择结束回合。在`play`的下一轮循环中，游戏打印地图，并且检查到战场上无敌方单位，因此打印`Won`并结束游戏。







下面这个例子展示达到失败条件时游戏的效果。在游戏经过0或若干次`play`内的`while`循环后，战场上只有敌方单位存在。在`while`循环的下一轮开始时，游戏打印地图，并检查到失败条件达成，因此打印`Failed`并结束游戏。

在本次任务中，**你需要修改`play`函数，实现检查胜利条件的逻辑**。

3.3 敌方回合（AI）

我们现在实现敌方回合的游戏AI，为此我们需要明确每个敌方单位的行动顺序，以及按顺序操控敌方单位。

游戏按照敌方单位坐标的偏序顺序，依次控制敌方单位的行动。假设`(R1,C1)`和`(R2,C2)`存在敌方单位，若`R1 < R2`或`R1 == R2 && C1 < C2`，则位于`(R1,C1)`的单位先行动；其他情况下位于`(R2,C2)`的单位先行动。

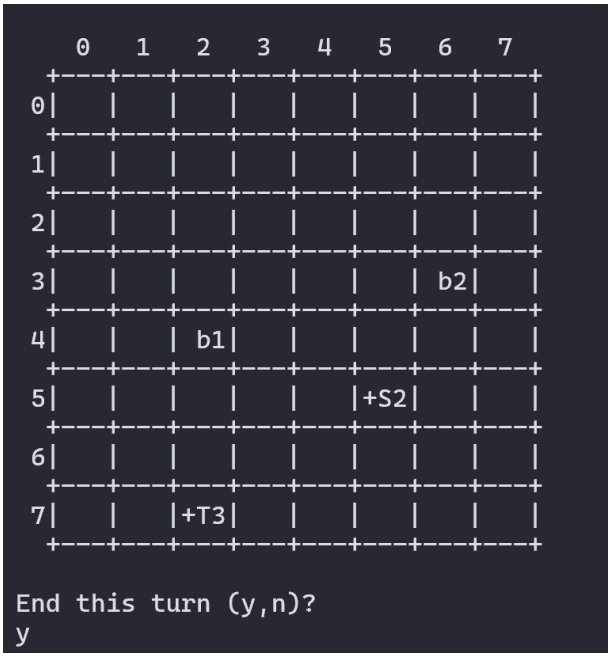
对于每一个敌方单位（下面用`e`指代这个敌方单位），游戏按照下面的顺序操作该单位：

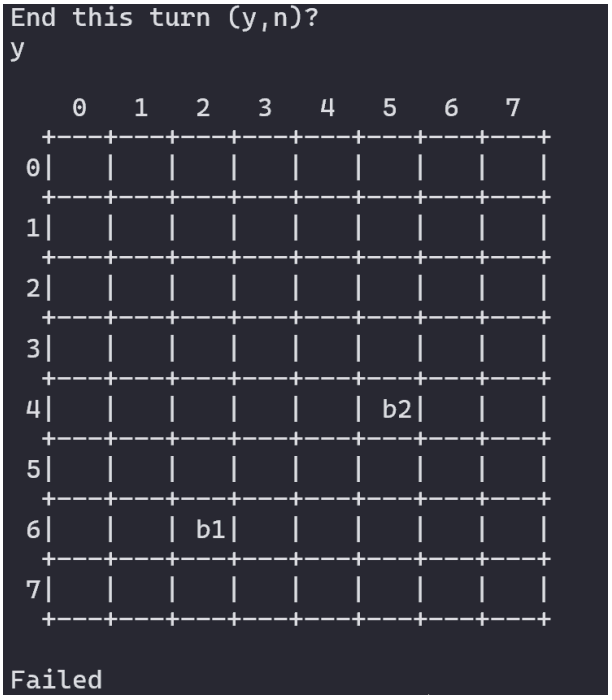
1. 如果不存在己方单位，则`e`不做任何动作，跳过下面的移动和攻击。
2. 移动。游戏通过调用寻路算法`searchReachable`获取`e`所有能够到达的目标坐标，并从中选出“最优”的目标坐标，将`e`移动到该位置。在解释最优坐标之前，首先定义坐标之间的距离。给定坐标`(R1,C1)`和`(R2,C2)`，两者的距离为行坐标差的绝对值和列坐标差的绝对值之和（`|R1-R2| + |C1-C2|`），比如`(2,3)`和`(1,5)`的距离为`1+2=3`。最优坐标的选择逻辑如下。假设`(R1,C1)`和`(R2,C2)`是能够到达的目标坐标，并且`(X1,Y1)`和`(X2,Y2)`分别是距离`(R1,C1)`和`(R2,C2)`最近的己方单位的坐标，其中`(X1,Y1)`可能等于`(X2,Y2)`。`(R1,C1)`**优于**`(R2,C2)`当且仅当，`(R1,C1)`和`(X1,Y1)`的距离**小于**`(R2,C2)`和`(X2,Y2)`的距离。若存在多个目标坐标有着相同距离的最近己方单位，则按照偏序关系选取最小的目标坐标。稍后我们将介绍一个具体的例子。
3. 攻击。在移动后，若根据`e`的攻击方式存在可以攻击的己方单位，则游戏操控`e`发动攻击。若存在多个可以攻击的己方单位，则选取按照偏序关系最小的坐标进行攻击。

完成任务3.3后，函数`play`的流程大致如下：

```
void play(Field& field, istream& is, ostream& os) {
    while (is) {
      打印地图
      检查胜利条件
      玩家回合
      敌方回合
    }
}
```

我们用下面的例子演示游戏中AI的效果。在左图中，游戏进行到了某个玩家回合，玩家选择结束本回合。接着在右图，游戏进入敌方回合，控制敌方单位的行动，并在下一次`play`函数的`while`循环打印地图，检查胜利条件。





我们拆解一下游戏进入敌方回合后，是如何控制敌方单位行动的。回到上面的左图，共有两个敌方单位，位于`(3,6)`和位于`(4,2)`的蜜蜂。`(3,6)`的蜜蜂当前生命值为`2`，`(4,2)`的蜜蜂生命值为`1`。首先，游戏确定控制这两个单位的顺序：按照偏序关系，`(3,6)`小于`(4,2)`因此游戏会先控制`(3,6)`的蜜蜂，再控制`(4,2)`的蜜蜂。

控制`(3,6)`的蜜蜂时，按照上述流程，游戏先控制它移动，再控制它攻击。移动时，需要在该蜜蜂所有可达坐标中选出一个“最优”的坐标。蜜蜂的移动力是`3`，下图的圆点展示了游戏为蜜蜂调用寻路算法后得到的可达目标，**注意该图只是起到解释作用，不是你需要在游戏中实现的打印效果**。在这些坐标中，`(4,5)`和`(5,6)`有着相同最小距离的最近己方单位（`(5,5)`的士兵）。按照偏序关系，`(4,5)`小于`(5,6)`，因此游戏会控制`(3,6)`的蜜蜂移动到`(4,5)`。接着，游戏继续控制同一个蜜蜂的攻击。按照控制单位攻击的规则，游戏检查到移动后的蜜蜂可以攻击`(5,5)`的士兵，因此控制其发动攻击。蜜蜂的攻击力为`3`，因此士兵的生命值减3（`2-3=-1`）。如果移动后的单位在检查攻击阶段存在多个可以攻击的己方单位，则选取按照偏序关系最小的坐标进行攻击。接下来，游戏将操控`(4,2)`的蜜蜂进行移动和攻击。按照控制的规则，最后达到上面右图的效果。

游戏控制(3,6)的蜜蜂的移动

在本次任务中，**你需要仿照己方单位的移动和攻击实现，修改`play`函数，以实现敌方回合AI的逻辑**。

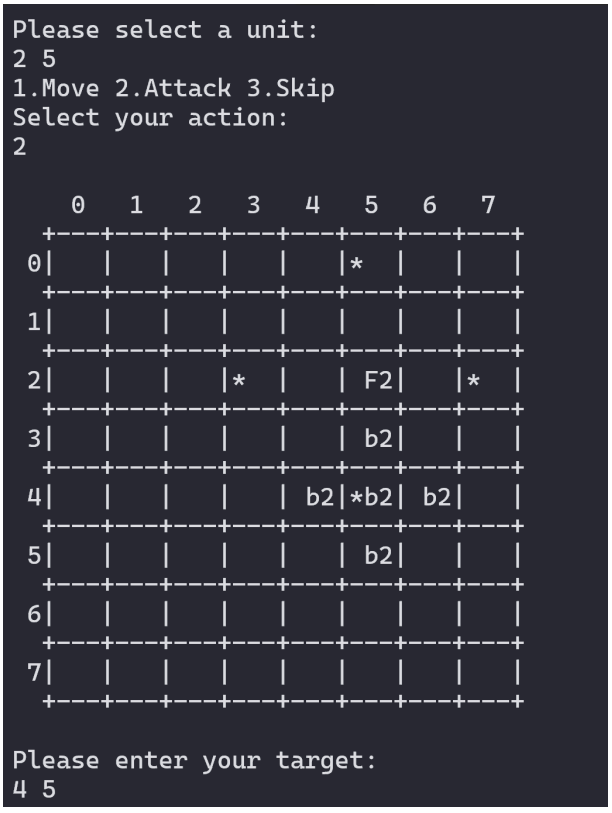
4. 高级单位和地形（8分）

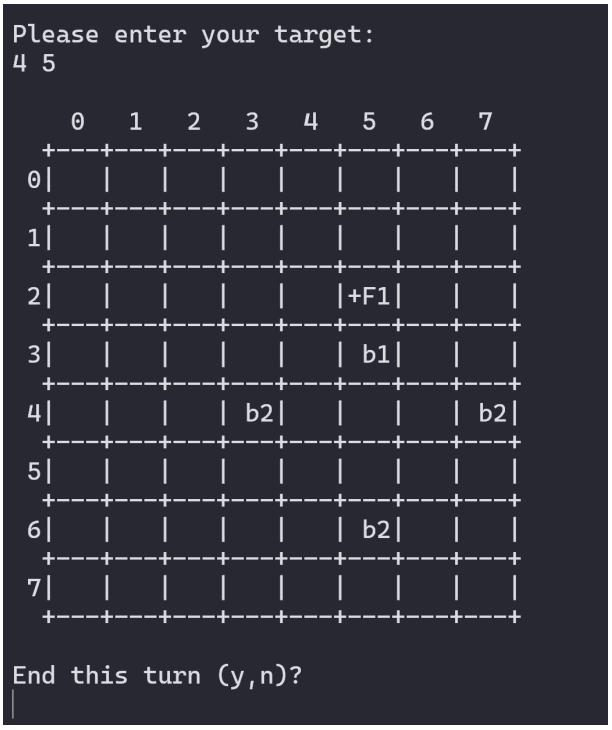
本次任务在任务3的基础上，增加了新的己方单位、敌方单位和地形。

4.1 战斗机

战斗机是一个新的己方单位，在加载地图时输入`R C F`将其载入战场。战斗机拥有`2`点攻击力。它的攻击有些特殊，可以选择上下左右距离为2的格子中的一个进行攻击，并且会击退目标坐标四周的单位（如果存在的话）。看下图的例子展示了位于`(2,5)`的战斗机的一次攻击。左图展示了`(2,5)`的战斗机可攻击的坐标是`(0,5)`,`(2,3)`,`(2,7)`和`(4,5)`。右图是当选择攻击`(4,5)`后的效果，可以被拆解为以下两部分：

1. 位于`4 5`的蜜蜂受到来自战斗机攻击力的伤害，因此其生命值减2（`2-2=0`）
2. 坐标`(4,5)`四周的单位，即`(3,5)`,`(4,4)`,`(4,6)`和`(5,5)`，被击退。由于`(3,5)`被击退的位置`(2,5)`存在单位，根据游戏手册被击退的描述，位于`(3,5)`和`(2,5)`的单位生命值减1，并且坐标保持不变。

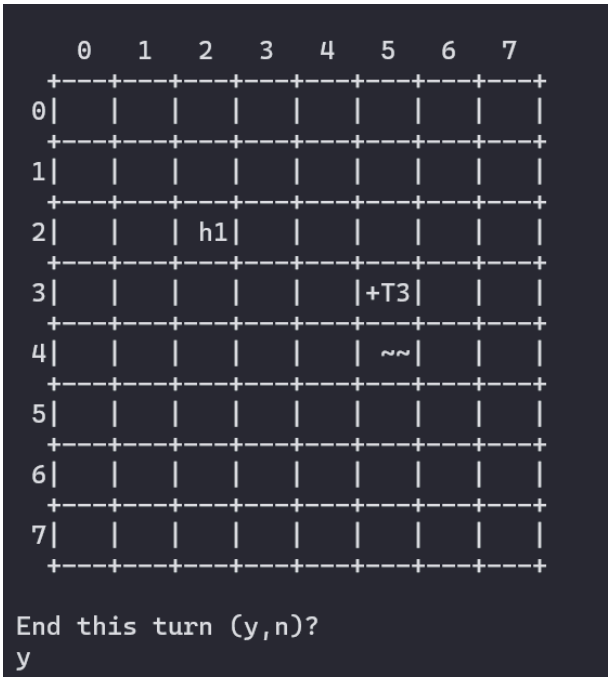


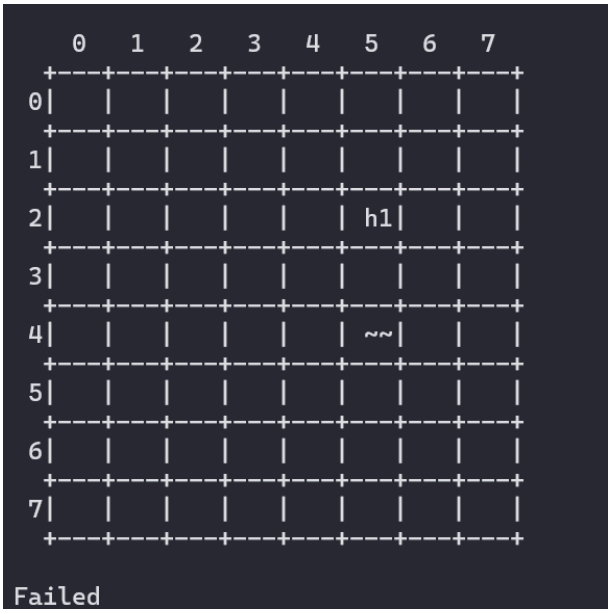


在本次任务中，**你需要为`Unit`类增加战斗机单位，并实现战斗机的攻击逻辑**。

4.2 刺蛇

刺蛇是一个新的敌方单位，在加载地图时输入`R C H`将其载入战场。刺蛇拥有`2`点攻击力，是一个近战单位，在攻击时对目标造成伤害，并拥有和坦克相同的击退效果。下图的例子展示了在敌方回合AI控制位于`(2,2)`的刺蛇移动并攻击了位于`(3,5)`的坦克。坦克受到来自刺蛇攻击力的伤害，生命值减2，并且被击退到了`(4,5)`的海洋，生命值降为`0`，效果如右图所示。



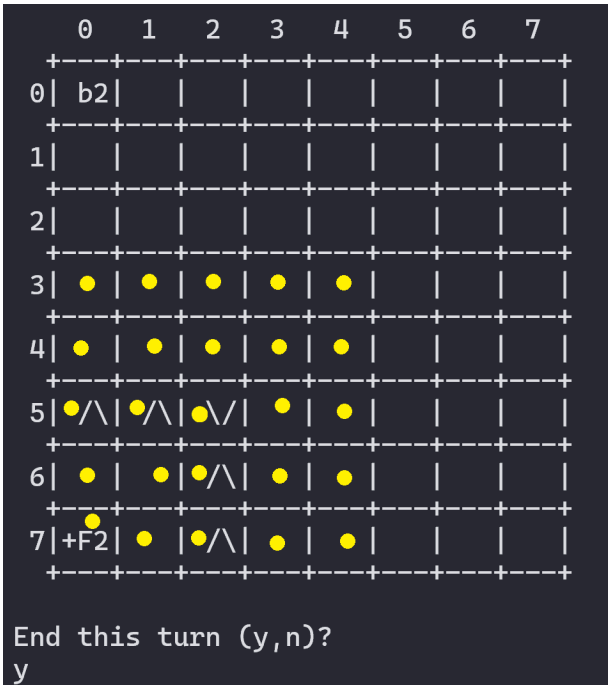


在本次任务中，**你需要为`Unit`类增加刺蛇单位，并实现刺蛇的攻击逻辑**。

4.3 森林

森林是一个新的地形，在加载地图时输入`R C W`将其载入战场。森林是具有特殊效果的地形。在每次敌方回合结束后，森林会以自身坐标为中心，使长度为5的正方形内的所有单位增加1点生命值。在下图的例子中，位于`(5,2)`的是森林地形（用`\/`表示），它能够起作用的范围是图中圆形标记的格子（该图起解释作用，**你不需要在游戏中实现打印圆形的功能**）。图中的过程可以被拆解为：

1. 玩家选择结束回合
2. AI控制位于`(0,0)`的蜜蜂移动到了`(3,0)`并结束了敌方回合；
3. 森林的效果触发，使得范围内所有的单位的生命值增加1。因此，`(3,0)`的蜜蜂的生命值由`2`增加到`3`，`(7,0)`的战斗机的生命值由`2`增加到`3`。



在本次任务中，**你需要为`Terrain`类增加森林地形，并修改`play`函数的逻辑，实现森林的特殊效果**。

调试游戏

为调试游戏，你首先需要准备好一个地图，使用`loadMap`通过文件输入流加载地图，然后在标准输入端输入命令，观察输出端的结果。主函数的示例结构如下：

```
int main() {
    Field f(8, 8);
    string filename = "map.txt";
    ifstream ifs;
    // 打开地图文件
    ifs.open(filename);
    if (!ifs) {
        cout << "Cannot open the file: " << filename << endl;
        assert(false);
    }

    // 通过文件输入流加载地图
    loadMap(ifs, f);
    // 通过标准输入输出流进行游戏
    play(f, cin, cout);

    ifs.close();
    return 0;
}
```

`maps`目录下准备好了2张测试地图`map1.txt`和`map2.txt`。你也可以设计自己的地图用于调试。需要注意的是，地图文件应和可执行程序在同一目录下，否则会出现无法打开地图文件的情况。

测试游戏

我们使用`judger.py`脚本做最终的测试，为此你需要将`main`函数改为**从`cin`中读取地图及之后的用户命令**，然后将结果输出到`cout`。每个任务我们准备了对应的测试案例，放在`data`文件夹中。**你的程序必须通过所有测试案例才能拿到对应任务的满分**。每完成一个任务，你需要将`BattleField`目录下的代码拷贝到`source`目录下对应的任务文件夹中，让`judger`成功编译你的程序。

```
source
|- 1_task1
|  |- main.cpp
|  ...
|
|- 2_task2
|  |- main.cpp
|  ...
|
|- 3_task3
|  |- main.cpp
|  ...
|
|- 4_task4
```

```
|  |- main.cpp
|  ...
|
```

然后在Windows命令行中运行

```
python judger.py -1  // 1 代表第1个任务，同理可测试2、3、4任务
```

如果测试通过，输出结果

```
[T1 c1] Correct
[T1 c2] Correct
...
```

如果测试不通过，则会显示输出不对应的地方。为了测试所有的结果，可以直接调用

```
python judger.py
```

隐藏测试（5分）

本次大作业有一部分隐藏测试用来测试**所有任务都完成的程序**可能出现的极端情况，**通过所有隐藏测试才可以得到满分**。需要注意，隐藏测试案例的重点是检查程序在逻辑上是否表现正确。，**不会**出现本文档中没有提到过的异常输入情况，比如非法的地图信息（两个单位或两个特殊地形处于同一坐标，坐标的值为负数）、非法的选取单位或行动的格式（选取单位时只输入一个数字）等等。

此外，本次大作业没有反馈的轮次，因此隐藏测试案例将不会透露给你，所以请特别注意自行测试各类边缘情况。自行测试的结果可以和参考程序相对比，参考程序在`demo`文件夹下面，有2个版本：

- `demo1.exe`从同目录下的`map.txt`中读取地图文件，然后和用户通过标准输入输出进行交互。
- `demo2.exe`从同目录下的`in.txt`中读取地图文件和所有用户输入，将结果输出到`out.txt`。

你可以修改`map.txt`和`in.txt`，通过运行`demo1.exe`和`demo2.exe`观察输出结果。在相同的输入情况下，如果你的程序和`demo2.exe`的输出内容一致，就能通过隐藏案例。注意，这里的`demo`程序只是方便你进行输出的对比，你的程序还是需要从标准输入`cin`读入数据，并且将结果输出到标准输出流`cout`。

提交文件格式

你需要提交的文件结构应该类似如下形式：

```
<your student number>.zip
|- 1_task1
|  |- main.cpp
|  ...
|
|- 2_task2
|  |- main.cpp
|  ...
|
|- 3_task3
|  |- main.cpp
```

```
|  ...
|
|- 4_task4
|  |- main.cpp
|  ...
|
```