文字王国历险记-续

贪心潜入迷宫太深的人，最终会被迷宫吞噬。

本PJ负责助教：唐傑伟 22302010060@m.fudan.edu.cn

“宝藏，嘿嘿，宝藏……”闹钟声把小黄猛的拉回现实。梦中的金银财宝骤然离他而去，意识到现实的它愤恨地敲击着桌面（以头抢地尔）。也许是用力过猛，消失的秘境居然又重新出现在他眼前。原封不动的场景，甚至被他打翻的宝箱都还在。“这到底是不是梦？”

一、回顾&总览

PJ2是基于PJ1进行的。如果没有额外说明，你需要保证PJ1中已经实现的功能不受影响。

项目链接

让我们先回顾你已经做了什么：

- 你实现了一个具有多种地块的地图的数据结构
- 你实现了一个可以与地块交互的虚拟角色小黄
- 你在程序中预置了两个地图
- 你的程序可以响应用户输入，操作小黄
- 你的程序可以输出探索的统计信息

你需要做的有：

- 从文件加载地图。
- 实现撤销和恢复。
- 能够保存进度。
- 记录已经完成的关卡。
- 结构化组织你的代码。（重要！）

还有一些具有挑战性的选做附加功能：

- 实现多层地图：当小黄移动到传送门时，可以切换上下层地图。
- 路线规划。
- 存档加密。

二、必做部分

0. PJ1内容

你可以删去自定义地图的功能，也可以保留。

撤销和恢复

我们希望你能够实现一个撤销功能，当用户输入Z时，小黄会回到上一步的位置，体力值也会回到上一步的值。撤销需要支持一直撤销直到游戏开始状态。

注：已获取的宝藏也需要撤销（或恢复）。但PJ1中可以不撤销。

当用户输入Y时，如果有被撤销的下一步。小黄会恢复到下一步的位置，体力值也会恢复到下一步的值。如果用户输入Z后又输入了一个合法的移动指令，那么撤销的操作会被清空（如图）。

建议你用链表（以及基于链表的数据结构）实现这一功能。

链表里应该放哪些内容？我们注意到，除了小黄自己（体力值，坐标）以外，似乎下一步要消耗的体力值，地图上的宝箱也会被修改。我们不妨用一个双向链表记录这些内容（我们要求实现的操作列表似乎也可以被记录在上面哦）。

```
struct Op {
    int currentEffort;
    int totalEffort;
    Player player; // 这里为什么不是指针?
    int treasures_found[4][2]; // 最多同时找到三个宝箱，但是设置成4
    struct Op *prev;
    struct Op *next;
}
```

思考：为什么这里用Player，而不是指向Player的指针？

按下Y，Z时，实际上就是某一个指针在这个双向链表的节点上左右移动，把当前状态设置成指针所指的对象内部记录的状态。

当发生有效操作时，我们free所有的当前指针右侧（next可达）的节点，让一个描述新状态的新节点成为next指针指向的实体。这样就实现了上图中的新操作覆盖旧的被撤销的操作的功能。

还有一种链表的设计，使用宏定义，出自Linux内核，可以把链表结构和数据结构体分离，感兴趣可参见附录2。

本功能仅在实时模式下被测试。

2. 地图加载

在PJ1中，我们通过硬编码（hard-code）的方式把地图写在了数组里，这种方式是不便于维护的，我们可以把地图的信息保存在文件里。

请你使用文件相关操作，程序运行时，读取exe文件所处文件夹下的maps文件夹中的文件生成关卡，准备三个关卡即可。例如，假设文件夹下有如下文件：

- 平凡之路.map
- 康庄大道.map
- 魔王之旅.map

那么，游戏的主界面现在将成为：

```
小黄的奇妙探险!

> 开始<平凡之路>
  开始<康庄大道>
  开始<魔王之旅>
```

```
  退出

按<Enter>选择
```

操作模式的选择不受影响。

3. 进度保存

我们的地图变得越来越庞大，有时用户想要暂时离开游戏，但又不想失去自己的进度。用户通关后，我们也希望进度可以保存，从而给用户带来成就感。

另外，我们目前的程序都是基于程序永远只能正常退出的原则设计的。事实上，我们永远无法预测程序运行时会发生什么。例如：家里突然停电，导致电脑关机；某个程序员写的程序管理不善，发生了内存泄漏导致内存枯竭；程序被任务管理器kill掉……这就对数据的持久化提出了挑战。

请你使用文件相关操作，完成进度的保存功能。

如果你对自己有更高要求，请让我们的程序对于正常退出和非正常退出的状态，都具有合理的恢复现场的能力。

不需要保存撤销和恢复的信息。

TODO: 请你设计文件内保存结构，使得： 当用户输入Q退出时自动保存进度，并在关卡选择界面作出（上次）提示。

```
小黄的奇妙探险!

> 开始<平凡之路>（上次）
  开始<康庄大道>
  开始<魔王之旅>
  退出

按<Enter>选择
```

用户选择了保存了进度的关卡后，跳出提示，包含上次退出的时间和进度信息。

```
是否加载上次的进度?
上次游玩的时间：2024-11-11 15:00
寻得的宝箱数：3/27

> 是
  否

按<Enter>选择
```

如果用户选择是，应该恢复到上次退出之前的状态，包括输入模式。如果用户选择否，清空存档，并要求用户选择输入模式（参见PJ1）。如果你已经忘记了板块之间怎么互相联系，你可以参考实现提示中的UI转换一览板块。

三、实现提示

虽然是建议，但是占30%分数。

在软件工程中，良好的代码组织结构是保证程序可维护性、可读性和扩展性的关键。通过合理地使用函数来组织源代码，并利用结构体、联合以及枚举等数据类型来表示复杂的数据结构，可以有效地提高程序设计的质量。以下将以迷宫游戏为例，探讨如何应用这些编程技巧。你不需要和我们的例子采用相同的设计，我们甚至期待你想出比下面的例子更好的设计。

思考：所以，C语言函数的用处都有哪些？

使用函数进行组织

将程序分解成多个小而专一的函数是一种非常有效的做法。每个函数应该只负责完成一项具体任务。这样做不仅可以让主程序更加简洁易懂，同时也方便了后期的功能测试与调试。例如，在一个迷宫游戏中，我们可以定义如下几个函数：

- `initMaze(&maze)`：初始化迷宫布局。
- UI系列函数。见子板块3。
- `movePlayer(direction, &player, &maze)`：根据给定的方向移动玩家位置。
- `checkCollision(playerPos, &maze)`：检查是否发生碰撞（如撞墙或遇到敌人）。
- `isWinningConditionMet(&maze)`：判断是否达到胜利条件。

通过这种方式，各个功能被清晰地区分开来，使得整个项目的架构变得更加清晰。

利用结构体、联合及枚举表示数据

结构体 (Struct)

在迷宫游戏中，可以使用结构体来存储游戏角色的信息。比如定义一个Player结构体用来保存小黄的位置、体力值等属性。这样的设计能够让不同类型的角色具有各自独特的属性集合（虽然我们还没有不同类型的角色就是了）。

```
typedef struct {
    int x;
    int y;
    int health; // 体力值
} Player;

struct info {
    char** maze;
    Player* player;
    char* game_info;
} Info;
```

枚举 (Enum)

对于一些固定选项或者状态变量来说，使用枚举是一个很好的选择。它能够提供更直观且易于理解的状态标识。例如，在迷宫游戏中，可以定义方向枚举以简化参数传递：

```
typedef enum {
    UP,
    DOWN,
    LEFT,
    RIGHT
} Direction;
```

这样，在调用`movePlayer()`函数时就可以直接传入`UP`,`DOWN`等值，而不是依赖于难以记忆的数字常量。

一种建议UI实现

UI变得越来越复杂，你应当实现UI和游戏逻辑分离的设计。UI的职责是获取用户输入和展示数据。

如果你的实现足够优雅，那么游戏逻辑里不应该有直接的`getch()`, `getchar()`等读取输入的函数。

在游戏中，我们可以调用自己定义的`get_ch()`，从而把它放进UI。在一回合结束后，我们需要刷新UI。

```
void repaint(struct info* game_info);
```

但是，在一些选择界面，不存在回合的概念，在这里，我们想要把控制权完全交给UI代码，让它自动读取输入，刷新UI。游戏逻辑只关心用户最后选择了哪个选项，或者是输入了什么。

不难想到，很多界面可以被抽象成这样的结构：

```
<头>是否加载上次的进度?
<信息列表>上次游玩的时间: 2024-11-11 15:00
<信息列表>寻得的宝箱数: 3/27
<选项集合>是
<选项集合>否
<操作提示>按<Enter>选择
```

于是，这些字段的设置逻辑可以被封装成一系列的函数：

```
// UI.h
#include <stdbool.h>

void set_header(char* text_str);
void set_info_list(char** text_str_list);
void set_map_visible(bool visible);
void start_selection(); // 展示选择菜单
void end_selection(); // 收起选择菜单
void show_input_box(int max_length);
void set_selection_options(char** options, int max_key); // 设置菜单
int get_selection_index(); // 获得选择的是哪个选项，阻塞直至输入<Enter>
void get_input_name(const char* out); // 获得输入的文字，阻塞直至输入<Enter>
char get_ch();
```

```
void set_hint(char* text_str);
void repaint(struct info* game_info); // 一回合结束触发手动重绘，更新info
void _repaint(); // UI内部函数，表示自动重绘
```

把游戏中和游戏前后不同的刷新逻辑一起放到时间序列图上表示，可以是这样的（什么是时间序列图？）。

输入字符（任务点3.3）和第二个流程是相似的，在此省略。

你会发现，在进行面板选择时，控制流一直在UI这个抽象实体上。这样，你的游戏逻辑中就不会出现各种状态变量与if-else。

在UI中，我们保存这些字符串的指针，当任何字符串发生更新时，_repaint函数可以直接获取指针对应的内容。

关于`get_selection_index`：我们有很多地方都有选择分支的UI。我们可以专门开发一个函数来实现它，获取当前选择的对象，以减少工作量。

这是一种实现的方式，仅供参考。如果是勇者辛美尔的话，一定会这样做的吧。

评分点4.1将重点关注你的代码结构。请务必不要在代码中出现过多的全局变量（职责内聚），过大过长的函数和文件，浪费的空间，无法理解的命名和注释，冗余的代码以及不合理的高复杂度算法。无法快速让读者理解的代码可能无法让你在这一得分点得分。

思考：我们现在在每次响应用户输入时刷新UI。假设游戏里有一个会自己周期性移动的NPC，那怎么刷新界面效果才会好呢？

思考：结构化UI设计一定好吗？会不会在某一些情况下，我们受迫要作出一些非结构化设计呢？

UI转换一览

无括号表示触发条件，尖括号表示按键输入，方括号表示条件，斜杠后面表示程序采取的动作（仅为提示）。（怎么看懂状态机？）

四、评分标准

本PJ占期末成绩中的15分。在计算成绩时，四舍五入到小数点后1位。

分数构成

参见下表。

评分点	测试内容	分数占比	是否支持部分得分
1	撤销恢复	25%	
2	地图加载	15%	
2.1	地图名称正确	5%	否
2.2	能够加载地图	10%	否
3	进度保存	15%	
3.1	恢复时展示统计数据	6%	是

评分点	测试内容	分数占比	是否支持部分得分
3.2	恢复时恢复游戏进度	6%	是
3.3	允许用户从头开始	3%	否
4	文档和代码结构	45%	
4.1	代码结构	30%	是
4.2	文档（包含本文档中的思考题）	15%	是
5	附加功能	+10%MAX	是
5.1.1	能否显示新地图元素	+2%	是
5.1.2	能否显示升降选择	+2%	是
5.1.3	能否响应升降选择	+3%	是
5.1.4	整体行为逻辑不变	+3%	是
5.2.1	实现这种体力消耗模式	+3%	是
5.2.2	路线规划	+5%	是
5.2.3	使用一定的动画展示这一路线	+2%	是
5.3	排行榜	+5%	是
5.4	你可以和助教沟通做其他内容	+10%	是

有关完整的编程规范可以参考：Making The Best Use of C

五、提交

你的提交是一个压缩包，内部结构如下。

文件结构

你提交的文件目录应该符合以下格式：

```
- <你的学号>_pj2
  ├ src
  │ ├ main.c(pp)
  │ └ 其他代码文件
  ├ doc
  │ ├ readme.pdf
  │ └ 其他有助于助教理解你的代码的文档
  └ build
    ├ main.exe
    └ 其他编译生成的文件
```

截止时间

提交的截止时间是2024/12/28 08:00AM。请你打包成一个压缩包提交到学习通平台。

每晚交24小时，会在你的得分上扣除10%分数（不满24小时按24小时计算），扣完为止。

如果提交时间晚于期末考试五天后，直接记录为0分。

附加内容不受此限制，但必须和整个PJ一起提交。

面试

本次PJ有面试。请提前和助教在下面的共享文档中预约时间。

腾讯文档：PJ2面试预约

TBC：面试地点会在之后发布。

六、附加内容（选做）

我们希望给不同能力的同学提供适应自身的体验，因此设计了附加功能。即使不完成附加功能，也仍然可能获得满分。附加功能首先填充本PJ不足的分数，再填充PJ1不足的分数。溢出的忽略。

附加内容的分数可以加到PJ1与PJ2的得分之和上，但不能使得这两个板块的合计分数超过PJ总分。

自动迷宫解题（10%）

小黄在文字王国的迷宫里玩得不亦乐乎，突然踩中一个陷阱，不明气体喷出，它失去意识。

醒来的时候，它发现自己正站在一个迷宫的入口处，眼睛一亮，刚想迈步进去探险，突然，一个超级大巨人“轰”地一下出现在它面前，挡住了去路。

小黄心里一紧，以为要战斗，那人却笑眯眯地说：“哇塞，小黄你好幸运哦！居然遇到了奖励关！宝藏就在迷宫里面藏着，完全自助哦！”

“真的吗？太棒啦！”小黄开心得跳了起来，尾巴都翘得老高。

“嗯哼，不过嘛，你得有本事把宝藏带回家哦！”那人眨了眨眼睛，神秘地说完，就“呼”地一下消失了。

小黄一听，乐得嘴巴都合不拢了，它迫不及待地冲进了迷宫，开始了它的寻宝大冒险！

迷宫-奖励关中有许多宝箱。小黄打开宝箱后，就能获得宝箱里的宝物。然而，宝物都是货真价实的真金白银，小黄在携带宝物时，体力值的消耗会增加相应的百分比。

小黄可以同时携带多个宝物，每个宝物增加的体力消耗是独立的（都基于由移动造成的基础体力消耗）。由于撞墙或者是调入陷阱而增加的体力消耗不受携带的宝藏的影响。

小黄的体力变为有限。如果小黄在体力耗尽前没有回到迷宫的入口处（出生点），它就会永远被困在迷宫中(X_X)。

小黄可以多次进入迷宫，每次都会回到迷宫的入口处。

实现这种体力消耗模式，你可以自由的修改输入输出格式和主界面UI，并提供一种自动规划的路线，使小黄能够获得尽可能多的宝物，又不至于被困在迷宫中。使用一定的动画展示这一路线。

基于二维即可。由于任务点1和3有冲突，建议择一完成。

排行榜（5%）

勇者被世界记住。

如果用户找到了所有宝箱。在退出前，你可以让用户输入自己的名字，程序会记录这一信息。

```
恭喜你，小黄找到了所有宝藏！

行动路径：UUUUUUIIIIUUUUUUDDDDDDDDDDDDDDLLLLLLLLLLLLLLLLRRRRRRRRRRRRRRRR
消耗的体力：1048576
找到的宝箱的数量：1

输入你的名字：_____
<按回车键继续>
```

在主界面上增加一个排行榜界面。

```
小黄的奇妙探险！

> 开始<平凡之路>（上次）
  开始<康庄大道>
  开始<魔王之旅>
  排行榜
  退出

按<Enter>选择
```

进入排行榜后，按体力消耗从低到高展示这一关卡的排行榜，按回车返回主界面。

```
<平凡之路> 排行榜
排名     挑战者          消耗体力
1        小黄              134
2        小绿              166
3        小蓝              192
4        小黄              200
5        小彩              459

> 退出

按<Enter>选择
```

多层地图（10%）

由于引入了多层地图，地图的表示方法会发生变化，请参见附录。

传送门在地图中用P(Portal)表示。当小黄站到传送门上时，按T(Teleport)之后进入传送模式，使用W，S键，地图会自动在不同层间切换。在传送模式中不能上下左右移动，再次使用T退出传送模式。

T操作需要在最后的操作中展示出来。

两层地图之间如果通过传送门连接，传送门之于这两层地图的坐标是相同的。（和直梯差不多）

传送门的数量和每一个传送门能够连接的层数是不确定的。

当小黄移动到传送门并进入传送模式时，需要展示出楼层选择界面。

请选择层数，按T离开传送门：

> 3F 2F 1F 退出传送模式时，该界面消失。

因此，地图上现在有这些实体：

实体	符号	说明
空地	空格	小黄可以自由移动
墙壁	W	小黄无法穿过
陷阱	D	小黄走到陷阱上，体力消耗增加
宝藏	T	小黄的目标
小黄	Y	闪闪发光的小黄，永远在地图最上层
传送门	P	可以在不同层间移动

七、学术诚信

我们鼓励你讨论这个项目，但是你的代码必须是你自己的。你可以从网上获取代码，但是你必须理解这些代码并且在你的代码中注明引用。

关于大语言模型的使用

我们不允许使用大型语言模型（如GPT）来生成代码。如果我们发现你的代码是由大型语言模型生成的，你将会被判为违反学术诚信。

抄袭

我们会检查你的代码是否与其他同学的代码相似。如果我们发现你的代码与其他同学的代码相似，你们两人都会被判为违反学术诚信。

后果

任何违反学术诚信的行为都将被根据复旦大学学生纪律处分条例中有关学术诚信的条款进行处理。

八、附录

地图下载

点我下载三张地图

数据约定

如果你没有做附加功能3，可以忽略本附录。

所有的坐标都从0开始。坐标原点在地图的左上角。第一个坐标是横坐标，向右为正方向；第二个坐标是纵坐标，向下为正方向。

第一行四个数字表示地图的宽、高、层数和传送门的数量m，层数从1开始。第二行三个数字表示角色的初始位置坐标和所在的层数。

第三行到第3+m-1行描述每个传送门的信息。第一个数字n描述传送门可以去的楼层的总数，第二、第三个数字描述传送门的位置：第一个坐标是横坐标，向右为正方向；第二个坐标是纵坐标，向下为正方向。接下来的n个数字描述具体可以去的楼层。传送门所在的地方必然是空地。

之后开始的每一行用空格分割的数字表示地图的具体内容。从最低层1层开始，到最高n层。

其中0表示空地，1表示墙，2表示陷阱，3表示宝藏的位置。

传送不消耗体力。

参考文件

```
15 15 3 2
1 1 1
2 5 1 1 2
3 13 13 1 2 3
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 0 0 0 0 0 2 0 0 0 0 0 1
1 2 1 1 1 1 1 1 1 1 1 1 1 0 1
1 0 0 0 0 0 1 0 0 0 0 0 1 0 1
1 1 1 1 1 0 1 0 1 1 1 0 1 3 1
1 0 2 0 0 2 1 2 0 2 1 0 0 0 1
1 0 1 1 1 1 1 0 1 0 1 1 1 0 1
1 0 2 0 0 0 1 0 1 0 0 0 1 0 1
1 1 1 1 1 0 1 1 1 0 1 0 1 1 1
1 0 0 0 1 0 0 0 1 0 1 2 0 0 1
1 1 1 0 1 1 1 0 1 1 1 1 1 2 1
1 2 0 0 0 0 1 0 0 0 1 0 2 0 1
1 2 1 1 1 1 1 1 1 0 1 0 1 0 1
1 0 0 0 0 0 0 2 0 0 0 0 1 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 1 0 0 0 0 0 2 0 1
1 1 1 1 1 0 1 0 1 1 1 1 1 1 1
1 0 0 0 3 0 1 0 0 2 0 0 0 0 1
1 0 1 1 1 1 1 0 1 1 1 1 1 0 1
1 0 1 0 0 0 0 0 1 0 0 0 0 0 1
1 0 1 1 1 1 1 1 1 0 1 1 1 0 1
1 0 0 2 0 2 0 0 1 0 1 0 1 0 1
1 1 1 1 1 1 1 1 0 1 0 1 0 1 0 1
1 2 2 0 0 0 1 0 0 2 1 0 1 0 1
1 0 1 1 1 0 1 1 1 1 1 0 1 0 1
1 0 2 0 1 0 0 0 2 0 0 0 1 0 1
1 0 1 0 1 1 1 1 1 1 1 1 1 0 1
```

```
1 0 1 0 0 0 0 0 0 2 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 2 2 2 0 1 0 0 0 0 2 0 0 1
1 1 1 1 1 0 1 2 1 1 1 1 1 0 1
1 0 0 0 1 2 0 0 1 0 0 0 1 0 1
1 0 1 0 1 1 1 1 1 0 1 0 1 0 1
1 0 1 0 0 0 0 3 1 0 1 0 0 0 1
1 1 1 1 1 1 1 0 1 2 1 1 1 1 1
1 0 0 2 0 0 0 0 1 0 0 0 1 0 1
1 0 1 1 1 1 1 1 1 1 1 0 1 0 1
1 0 1 0 0 0 0 0 0 0 1 0 1 0 1
1 0 1 0 1 1 1 1 1 0 1 0 1 0 1
1 0 0 0 1 0 0 0 1 0 1 0 2 0 1
1 0 1 1 1 1 1 0 1 0 1 1 1 0 1
1 0 0 0 2 0 0 0 1 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

container_of实现的列表

```
typedef unsigned long long u64;
#define offset_of(type, member) ((u64)(&((type *)NULL)->member))
#define container_of(mptr, type, member)                        \
    ({                                                          \
        const typeof(((type *)NULL)->member) *_mptr = (mptr); \
        (type *)((u64 *)_mptr - offset_of(type, member));      \
    })

typedef struct ListNode_ {
    ListNode *prev;
    ListNode *next;
} ListNode;

struct Op {
    ListNode *node;
    int currentEffort;
    int totalEffort;
    Player player;
    int treasures_found[4][2];
}
```

这个环状双向链表用三个函数即可维护：

```
void init_list_node(ListNode *node)
{
    node->prev = node;
    node->next = node;
}

ListNode *_merge_list(ListNode *node1, ListNode *node2)
{
```

```
    if (!node1)
        return node2;
    if (!node2)
        return node1;

    ListNode *node3 = node1->next;
    ListNode *node4 = node2->prev;

    node1->next = node2;
    node2->prev = node1;
    node4->next = node3;
    node3->prev = node4;

    return node1;
}

ListNode *_detach_from_list(ListNode *node)
{
    ListNode *prev = node->prev;

    node->prev->next = node->next;
    node->next->prev = node->prev;
    init_list_node(node);

    if (prev == node)
        return NULL;
    return prev;
}
```

我们可以这样方便的获取对象：

```
struct Op * op = container_of(<某个链表节点的指针>, struct Op, node);
```

在更底层的软件设计里，我们会更多的看到这种形式，因为我们有时会更希望数据Op被组织在同一个地方。在这个PJ中，使用这种形式并没有明显的好处。不过在系统程序设计、操作系统、编译课上你们可能会遇到这种形式，便事先提及。