程设大作业项目报告

组长：石亦烜，计41
组员：孙炜伦，新雅52

OJ测试提交编号为：665895
测试结果：

提交号	用户名	题号	课程号	作业号	状态	代码风格	语言	运行时间	占用空间	提交时间
665895	2023010302	2434	36	1357	评测通过 (100)	...	CPP11	2 ms	4208 KB	2025.11.28 14:58:11

1. 项目设计思路

2.1 总体架构思路

作为一个简单的游戏开发，本项目采用了类似 **MVC (Model-View-Controller)** 的设计模式来解耦各个模块，从而实现更良好的可维护性以及为后续提供更好的可拓展性。根据作业要求，可以把整个任务分为下面三个部分：

2.1.1 模型层：存储数据 (Model)

模型层主要实现的是各个数据类型的封装和对外接口实现。整个任务实质上涉及两类数据的存储：

1. 游戏关卡。无论是直接硬编码还是本地存储，都需要封装关卡相应的编号、关卡要求等信息
2. 游戏运行过程中的状态。需要一步步地在模拟运行过程中进行更新迭代。

因此，分别对其进行封装，实现以下两个类：

- **Level**：存储关卡信息。包含关卡名称、输入序列、目标输出序列、空地数量限制以及允许使用的指令集。
- **Robot**: 代表游戏的动态状态。它不包含业务逻辑，只存储游戏过程中的状态（当前输入输出传送带、空地状态、当前执行指令等）。其中的`reset`方法允许根据`Level` ID 重置状态。

此外为了进一步封装，提高可维护性，还对指令和游戏运行结果进行了封装

- **Instruction**: 对操作指令的封装，这里不仅仅是提高可维护性，更是提供了简单的方法使得在解析阶段对指令完成静态合法性检查（如检查参数是否为非负整数），通过`is_valid`标志位记录。(具体可见2.2节)

- **RunResult**：对运行结果封装，包括SUCCESS、FAIL、ERROR，对于ERROR，标记非法指令位置

2.1.2 控制层：整体调度 (Controller)

为了对关卡列表的操作和运行过程中的更新，采用了两个控制类对其进行操作。

- **LevelManager**：用于管理关卡列表和本地解锁状态的持久化。
  - 其中存储一个静态的关卡列表，在第一个LevelManager对象构造时创建所有关卡。
  - 封装了获取只读(const)关卡，加载本地存储通关记录的接口
- **Actuator** (执行器): 这是核心逻辑引擎，实现游戏状态的更新和所有对象的调度。其中存储一个私有的Robot对象，并对其进行更新。具体来说，执行器封装了以下方法：
  - 解析：从用户输入（或文件）中解析指令字符串为 `Instruction` 列表（`parse_instruction_line`）。
  - 运行游戏: 即`run()` 方法，它根据 `robot.pc` 读取指令，通过 `switch-case` 分发执行逻辑。在运行过程中对指令合法性进行动态检查。并统一输出执行结果RunResult
  - 内部辅助函数：例如判断当前空地是否合法的方法 `is_valid_empty_space_arg`

2.3 UI层：显示数据 (View)

在 `view.h` 中定义了所有的UI方法：将 `Robot` 和 `Level` 的状态渲染到终端。它不修改数据，只读取数据进行展示。

- `show_level_info` 实现关卡信息界面
- `initialize_view` 实现关卡选择菜单
- `show_one_step` 实现游戏画面的实时刷新。
  此外还有一些辅助函数。

2.2 核心功能设计实现

2.2.1 指令解析与合法性设计

根据需求文档，异常情况包括：

- 不在指令表中的未定义指令。

- 不属于当前关卡固定可用指令集的指令。
- 不符合语法或参数要求的指令（操作数非整数、参数个数错误、负数参数等）。
- 指令特定运行时错误（如空地不存在、无当前积木等）。

为此，以MVC式的架构以及对指令类 `Instruction` 的单独封装，将指令合法性的检查机制分为了两层：

1. 解析阶段（静态检查）
   在执行器的 `parse_instruction_line`方法中，在指令加载阶段实现指令的静态检查，解析出指令名和参数字符串，并将指令名统一转为小写，以提高输入的鲁棒性。同时检查：
   - 指令名是否在已知集合（`inbox/outbox/add/sub/copyto/copyfrom/jump/jumpifzero`）中。
   - 参数个数是否为 0 或 1，额外参数被视为非法。
   - 参数是否为整数
     如果存在解析错误，就调用 `Instruction::set_error()` 将 `is_valid` 置为 `false`。

这里实现了对指令输入比较鲁棒的检查，如果输入非指令的字符串或者不符合基本要求的参数都会标记错误

2. 运行阶段（动态语义与关卡规则检查）
   在 `Actuator::run()` 中，对于每条待执行指令，依次进行：
   - `pc` 越界检查：若 `pc < 0` 或 `pc >= program.size()`，说明程序已经自然结束。
   - 指令合法性检查：若 `instr.is_valid == false`，立即返回错误。
   - 关卡允许指令集检查：通过 `is_instruction_allowed(level, instr)` 将 `instruction_type` 映射为字符串，再查当前关卡的 `valid_instructions` 集合。
   - 按不同的指令类型进行运行时语义检查

一旦出错，记录出错行号 `error_index = pc + 1`，并立即结束。

通过“解析阶段 + 运行阶段”的分层，既保障了对输入程序的鲁棒性，又与题目描述中的错误处理规则严格对应。

2.2.2 关卡本地存储

通过关卡管理类 `LevelManager` 的实现，将关卡的本地存储与解析抽象为两个接口：

- `set_local_level_unlocked` 方法：每次通过一个关卡，就调用该接口将对应关卡（若还是未解锁状态）解锁并存储在本地记录 `code/local_levels/level_unlock_status.txt`
- `load_local_levels` 方法：在初始化首个 LevelManager 对象时初始化关卡，并调用该接口加载存储在本地的已通关记录并设置

2.2.3 输入鲁棒性处理

为了提高程序的鲁棒性，项目绝大多数的输入均采用了 `getline()`读取整行+字符串处理+`stoi`捕获异常的方式进行，不预先设想用户输入，而是全部读取为字符串后再做处理
例如，对于用户从命令行输入指令，一开始要求先输入总的指令数量，处理方式如下：

C++

```
string ins_num_str;
getline(cin, ins_num_str);
try{
    stoi(ins_num_str);
} catch (const invalid_argument&) {
     cout<<"指令数量输入无效，请重新输入。"<<endl;
     continue;
}
```

现将用户输入作为字符串读入，再使用 `stoi` 进行安全的类型转换，并捕获异常进行处理。
在处理指令的静态检查时，也用了类似的方法检查指令操作数的合法性。

2. 项目工程结构
如前面所述，项目整体采用MVC的架构组织，并且将函数和类的声明和定义分离，分别置于头文件和源文件中。通过模块化的处理，将核心逻辑、数据模型、界面显示和程序入口分离，以提高代码的可维护性和可扩展性。主要文件结构如下：

```
|-code
|————————local_levels   # 关卡记录本地存储
|    |————————level_unlock_status.txt    # 存储通关记录，关卡初始化时加载
|    |———————— ...      # 后续可进行进一步扩展，如记录用户通关所用的策略、游戏记录等。
|
|————————src            # 项目源代码
|    |————————level.h/level.cpp          # 管理关卡类的定义和相应接口
|    |————————robot.h/robot.cpp          # Robot类，存储每一时刻的游戏状态
|    |————————actuator.h/actuator.cpp    # 核心执行器Actuator类和辅助工具,其run()接口
| | 实现游戏的核心模拟逻辑
|    |————————game.cpp                   # 游戏入口，游戏骨架
|    |————————view.h                     # 命令行UI显示的函数，后续对UI的优化修改可以通
| | 过调整这里进行扩展
|    |————————conf.h                     # 存储配置，比如加载文件的目录等
|
|————————test_level     # 关卡的测试指令库，用户需要通过文件输入指令时，将指令放置于此文件
| 夹下
|————————main.cpp       # OJ测试代码，直接复用了项目中的类和接口
```

3. 游戏界面的设计
游戏界面采用 **CLI（命令行界面）** 实现，但通过 ASCII 字符画和 ANSI 转义序列实现美化。
3.1 核心运行主界面设计
1. 具体指函数：`void show_one_step(const Robot& robot)`。
2. 功能：
- 每执行一步指令后，立即刷新整个界面，让玩家直观看到状态变化。
- 将界面拆为左右两部分：
    - 左侧：输入传送带、机器人、当前积木、输出传送带、空地。
    - 右侧：程序指令列表与当前执行行标记。
3. 主要元素布局：
    - **顶部标题栏** ：关卡头
    - **传送带和机器人区域**（左侧）：
        - 使用 ASCII 画框实现类似“积木盒子”的效果。
        - 机器人使用 ASCII 艺术。
    - 空地状态显示在界面下方
    - **指令列表区域**（右侧）：最多显示前24条指令，避免程序过长导致界面溢出，后面通过“... 共 N 条指令，已省略后续”提示。底部还有“执行步数：X”统计信息。

4. **交互节奏**
     - 每次调用 `show_one_step` 时，先通过 `"\033[2J\033[H"` 清屏，然后重绘完整界面。
     - 最后输出“按 Enter 执行下一步...”，等待用户按 Enter 才继续执行下一条指令（在 OJ 模式下关闭界面模块）。

整体显示界面如下：

3.2 交互设计
*   **动态刷新**：使用 `\033[2J\033[H` ANSI 转义序列实现清屏和光标复位，使得每一步执行看起来像是动画刷新，而不是简单的日志滚动。
*   **菜单系统**：在 `initialize_view` 中实现了基于键盘方向键（↑↓）的菜单选择功能，并实时显示关卡的解锁状态（“已解锁”/“未解锁”）。
3.3 其他界面
在 `view.h` 中还实现了游戏初始化界面、菜单界面、游戏结果界面的显示。

```
---

# 4. 创新关卡的设计

在现有的实现框架下，本项目的关卡系统具有高度的可扩展性，支持通过代码配置轻松添加具有不同逻辑挑战的关卡。
在此次项目中，注意到所有指令中只有加法(add)和减法(sub)的指令，而没有乘法。于是我们以乘法为线索，经过深入研究设计了两个创新关卡。
## 4.1 模拟乘法I：放大器
要实现乘法，则必然需要通过多次加法来实现。第四关要求实现的是一个以硬编码的常数作为其中一个乘数的乘法。也就是实现一个放大器，对输入序列的每个数放大 $k$ 倍。
```

第四关：模拟乘法I：三倍放大

输入序列：3, 1, -5, 4, 0, -2
目标输出序列：9, 3, -15, 12, 0, -6
空地数量：1
允许指令：inbox,outbox,add,copyto,copyfrom,jump

也就是对输入序列的每个积木*3

实现时，只需将每个积木先缓存一份在空地中，然后add即可。最终解决方案为：

```txt
inbox
copyto 0
add 0
add 0
outbox
jump 1
```

4.2 模拟乘法II:两数相乘

进一步，我们希望拜托硬编码乘数的限制，实现对输入序列中的两个变量进行乘法运算。深入研究，发现了两个辅助的方法。

4.2.1 计数器

要实现对变量做乘法。回想c++程序里的设计，如果只用加法来实现必然经过这样的循环：

```C++
//m * n
for(int i=m;i>1;i--){
    n+=n;
}
```

换言之，其中一个乘数在其中担任"计数器"的角色，控制另一变量自加的次数。

因此，我们需要三个空地存储三个信息：`[乘数1]`，`[乘数2（计数器）]`，`[当前总和]`。另外，要实现记数的功能，必须要一个辅助的1，来每次对计数器进行减1操作。

综上，一共需要4块空地。

4.2.2 利用跳转指令实现函数

注意到输入指令中有两条跳转指令： `jump`， `jumpifzero`。这两条指令和汇编语法里实现分支和循环的指令 `jmp` 和 `je` 有接近的作用。因此可以尝试模块化的实现小段的指令块，然后在核心逻辑里进行分支跳转，实现近似于调用函数的结果。
具体关卡设计为：

```TXT
# 第5关：模拟乘法II：两数相乘

输入序列：3, 4, 2, 5, 0, 1
目标输出序列：12, 10, 0
空地数量：4
允许指令：inbox,outbox,add,copyto,copyfrom,jump,jumpifzero

# 也就是对输入的每两个数相乘后输出，注意首两个输入积木差值需要为1
```

答案设计为：（实际输入要去掉注释）

TXT

```
# 第一遍输入，注意这一次需要利用4-3的到辅助计数器减法的1
inbox
jumpifzero 17   # 第一个乘数为0时
copyto 0
inbox
jumpifzero 20   # 第二个乘数为0时
copyto 1
copyto 3
sub 0
copyto 2
# 乘法逻辑片段
copyfrom 0
sub 2
jumpifzero 23
copyto 0
copyfrom 3
add 1
copyto 3
jump 10
# 处理第一个乘数为0
outbox
inbox
jump 25
# 处理第二个乘数为0
outbox
jump 25
# 乘法结束输出
copyfrom 3
outbox
# 后续的每两个输入积木
inbox
jumpifzero 18
copyto 0
inbox
jumpifzero 21
copyto 1
```

```
copyto 3
jump 10
```

5. 游戏测试

5.1 游戏运行

这里给出游戏的运行方法，关于调试过程中遇到的问题，见第7节问题和心得

先解压缩项目，随后进入Windows的cmd，进入到源代码所在目录 `/code/src` 下：

```shell
C:\Users\User_name> cd 根目录/code/src
```

> ⚠ **Attention**
>
> 该项目应当在**Windows**环境下的**cmd**中运行
>
> - 因为游戏使用的是中文UI界面，后续需要指定编码方式为utf-8，经测试在Powershell中会出现编码方式未能正确改变而导致的乱码
> - 游戏中用到了上下键和回车键进行动态选择的菜单系统，具体实现时使用的是Windows环境下的配置

进入目录后，进行编译得到可执行文件。注意这里最好指定为静态链接 `-static`，否则可能出现动态链接库 `stringstream` 访问失败的情况。同时采用 `-std=c++11`

```shell
g++ level.cpp robot.cpp actuator.cpp game.cpp -o game.exe -std=c++11 -static
```

随后指定中文编码为utf-8以正确显示中文：

```shell
chcp 65001
```

运行程序即可：

```shell
game.exe
```

5.2 输入指令

1. 如果选择直接从命令行输入指令，则输入样式和文档中一致，游戏中也进行了相应的指引和鲁棒性处理，也就是先要求输入指令数量 M ，剩余M行依次输入指令，最后按回车开始运行。

2. 如果选择从文件中读入指令，则注意：
   - 文件输入指令的格式和命令行输入不同，不需要先输入指令数量，直接一行一条地输入所有指令即可。（可见 `code/test_level/` 目录下的示例）
   - 在项目目录的 `code/test_level` 目录下，目前已有 `level1.in` ~ `level5.in` 五个指令输入文件，它们分别是关卡1至关卡5的标准答案（SUCCESS）
   - 文件路径建议使用相对路径，例如：

```
../test_level/level2.in
```

6. 小组分工

组长：石亦烜，负责工作：

- 整体项目框架的搭建，相关类和接口的声明
- `Robot` 类， `Actuator` 类和辅助类中接口的实现，以及主程序 `game()` 的实现
- 与孙炜伦合作完成核心界面 `show_one_step` 函数
- 项目报告的撰写
- 程序的调试debug

组员：孙炜伦，完成工作：

- 关卡列表的初始化的实现，即 `LevelManager` 中接口的实现
- 与组长石亦烜合作完成核心界面 `show_one_step` 函数
- 最终的游戏录像和剪辑

7 问题和心得

7.1 遇到的问题和解决

除了一般的代码逻辑bug的调试外，在实验过程中我还遇到了下面几个问题：

1. **清屏操作失灵。**一开始，我使用了Windows系统函数 `system("cls")` 进行清屏。但直接导致进程卡死，且无法正常中断，最终重启才得以恢复。
   解决：多方检查后发现，是因为VScode的终端似乎不支持`system("cls")`的清屏，从而使得进程始终挂起。随后也是为了调试需要，我将清屏的实现修改为了ANSI的"伪清屏"，也就是：

   ```C++
   cout << "\033[2J\033[H";
   ```

   并在Windows自己的终端运行，解决了问题

2. **中文编码问题。**本次项目我全程使用了中文来实现各项界面。结果在Windows的cmd和powershell中运行均显示乱码，后来查阅资料后我得知，目前Windows系统的很多地方对于中文仍采用GBK编码，而VScode中的默认编码是更通用的UTF-8。
   解决：为了以最小的改动解决问题，选择在运行程序前显示要求Windows的shell以UTF-8编码读取显示。即：

```SHELL
chcp 65001
```

7.2 心得

通过此次大作业，我们进一步地认识到了对于项目开发和合作开发来说，前期搭建一个模块化的项目框架的重要性。不仅可以极大程度上提高项目的可维护性，方便协作开发和后期维护，而且从长远来看还具有很好的扩展性，在很多功能上都有办法实现更进一步的开发。

此次大作业整体耗时越6天完成，其中3天时间都在进行框架的搭建，但很大程度上节省了后续debug的耗时。这也说明：必要的前期投入从长远来看是提高效率的手段

再次感谢老师和各位助教的辛苦付出！