華中科技大學

课 程 实 验 报 告

课程名称：面向对象程序设计

实验名称：智能车控制系统的设计与实现（实验三）

院　　系 ：计算机科学与技术

专业班级 ：计算机本硕博 2301 班

学　　号 ：U202315763

姓　　名 ：王家乐

指导教师 ：辜希武

2024 年 12 月 18 日

一、需求分析

1. 题目要求

设计一个 C++程序，实现智能车控制系统的执行器 Executor 组件控制指令功能。要求利结合面向对象程序设计的特性：多态性、接口/抽象类、运算符重载和函数式编程特性：Lambda 表达式实现，以提高系统的扩展性

2. 需求分析

Executor 组件可以执行如下的移动指令：

M: 前进，1 次移动 1 格

L: 左转 90 度，位置不变

R: 右转 90 度，位置不变

F: 加速指令，接收到该指令，车进入加速状态，该状态下：

M：前进 2 格（不能跳跃，只能一格一格前进）

L：先前进 1 格，然后左转 90 度

R：先前进 1 格，然后右转 90 度

再接收一次 F 指令，对应的状态取消

B: 倒车指令，接收到该指令，车进入倒车状态，该状态下：

M：在当前朝向上后退一格，朝向不变。注：比如朝向为 N 时收到 M 指令，y 坐标减 1，朝向保持 N

L：右转 90 度，位置不变

R：左转 90 度，位置不变

B 和 F 两个状态可以叠加，叠加状态下：

M：倒退 2 格（不能跳跃，只能一格一格后退）

L：先倒退一格，然后右转 90 度

R：先倒退一格，然后左转 90 度

再接收一次 B 指令，对应的状态取消。

要求可以执行上面这些指令的组合序列，如 MLMRMFMBBFF。

二、系统设计

1. 概要设计

给出系统的总体设计。要求给出系统实现包括哪些类/接口，每个类/接口的功能是什么？

1.1 系统总体设计

该系统基于命令模式（Command Pattern），其中不同的操作（如前进、转向等）被封

装成命令对象，执行器（Executor）根据输入指令来执行相应的操作。系统包括以下主要组件：方向控制（Direction 类）、车辆状态管理（PoseHandler 类）、命令接口与具体命令（Command 类）、执行器（Executor 类）。

1.2 类和接口说明

1.2.1 Point 类

表示车辆的位置，使用 x 和 y 坐标进行定位。

1.2.2 Direction 类

表示车辆的方向（东、南、西、北），并提供与方向相关的操作。

1.2.3 PoseHandler 类

管理车辆当前的状态，处理与车辆姿态相关的操作。

1.2.4 具体命令(Command)类

每个命令类封装了一种具体的操作行为，通过重载 operator()实现 Command 接口。

MoveCommand 类：根据是否处于加速或倒车状态，控制车辆前进或后退。

TurnLeftCommand 类：控制车辆左转。

TurnRightCommand 类：控制车辆右转。

FastCommand 类：切换加速模式。

ReverseCommand 类：切换倒车模式。

1.2.5 Executor 接口

定义智能车执行器的接口，允许执行一系列控制命令。

1.2.6 ExecutorImpl 类

实现 Executor 接口，负责执行具体的控制指令。

1.3 命令模式

该系统使用命令模式来将请求封装为对象，从而允许将请求发送者和请求接收者解耦。每个控制命令（如前进、转向、加速、倒车等）都被封装为一个命令对象，通过 Executor 执行。命令类负责调用 PoseHandler 提供的接口来改变车辆的状态。不同的命令类实现了 ICommand 接口，确保可以通过统一的方式来执行车辆的动作。

2. 详细设计

给出每个类/接口的具体实现。具体介绍每个类接口的数据成员及其作用、函数成员及其作用。建议以表格的形式给出，清晰明了。对于关键函数成员，给出方法的流程图。

2.1 Point 类

成员	类型	描述
x	int	车辆的横坐标
y	int	车辆的纵坐标

表 2-1-1 Point 类数据成员

方法	描述
GetX()	返回车辆的横坐标
GetY()	返回车辆的纵坐标
operator+=	用于实现位置的加法操作（增量移动）
Operator-=	用于实现位置的减法操作（减量移动）

表 2-1-2 Point 类函数成员

2.2 Direction 类

成员	类型	描述
index	unsigned	方向索引 0 1 2 3
heading	char	方向字符 E S W N

表 2-2-1 Direction 类数据成员

方法	描述
Move()	根据车辆当前的方向返回对应的移动增量（如在 N 方向时，返回 (0, 1)）
LeftOne()	获取左转 90 度后的方向
RightOne()	获取右转 90 度后的方向
GetHeading()	获取当前方向的字符表示

表 2-2-2 Direction 类函数成员

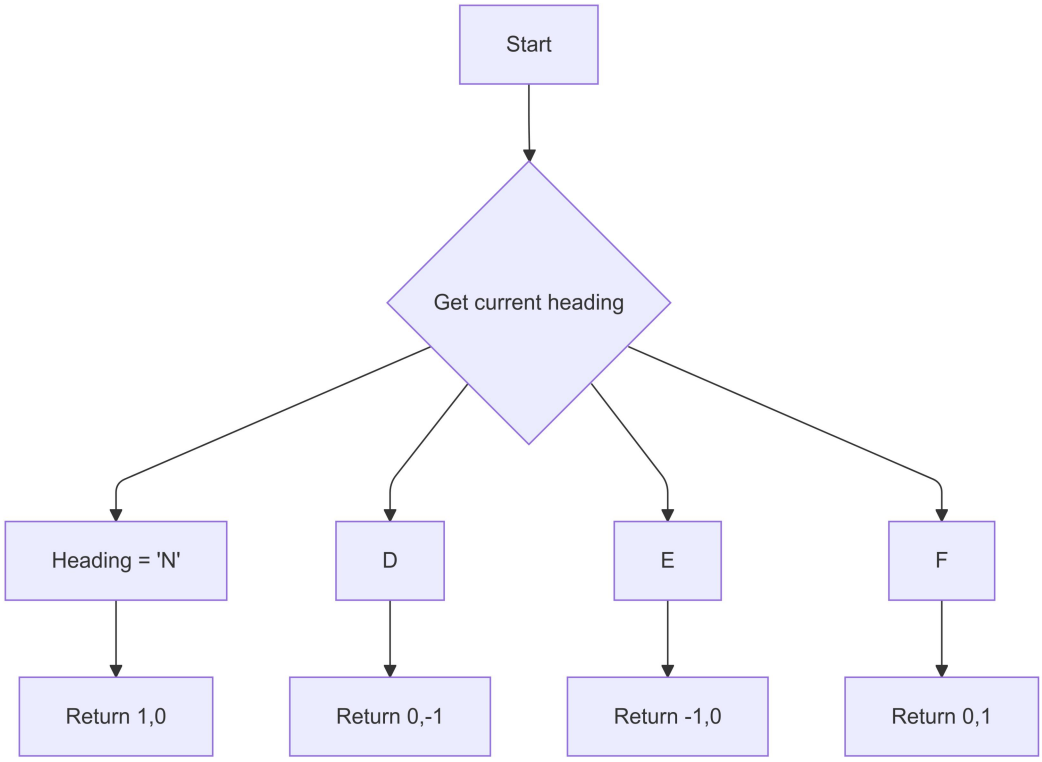

图 2-2-1 Move()方法流程图

2.3 PoseHandler 类

成员	类型	描述
point	Point	车辆的位置
facing	Direction	车辆的方向
isfast	bool	是否处于加速状态
isreverse	bool	是否处于倒车状态

表 2-3-1 PoseHandler 类数据成员

方法	描述
Forward()	根据当前方向前进 1 格
Backward()	根据当前方向后退 1 格
TurnLeft()	左转 90°
TurnRight()	右转 90°
Fast()	切换加速状态
Rerverse()	切换倒车状态
isFast()	查询是否是加速状态
isReverse()	查询是否是倒车状态
Query()	查询车辆的当前状态，包括位置和方向

表 2-3-2 PoseHandler 类函数成员

2.4 具体命令(Command)类

MoveCommand 类：

方法	描述
operator()(PoseHandler &poseHandler)	执行前进操作，依据车辆的状态（加速或倒车）来前进对应格数。

表 2-4-1 MoveCommand 类函数成员

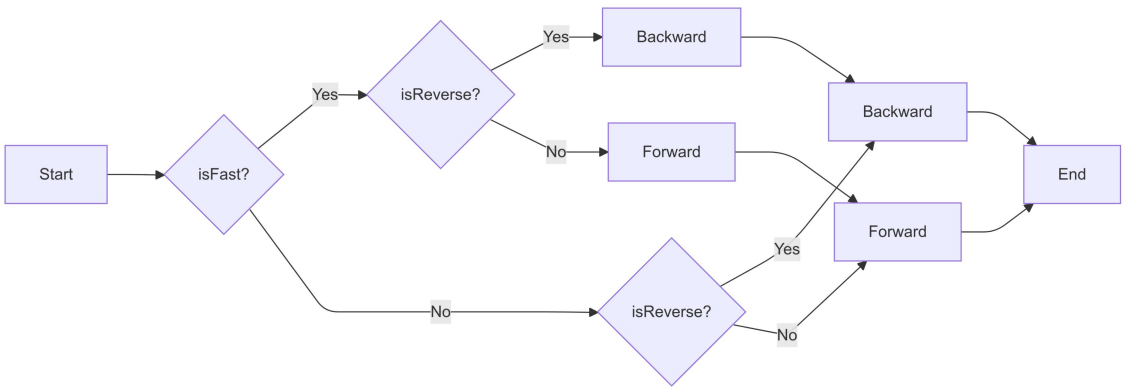

图 2-4-1 MoveCommand 方法流程图

TurnLeftCommand 类：

方法	描述
operator()(PoseHandler &poseHandler)	执行左转操作，依据车辆的状态（加速或倒车）来转向。

表 2-4-2 TurnLeftCommand 类函数成员

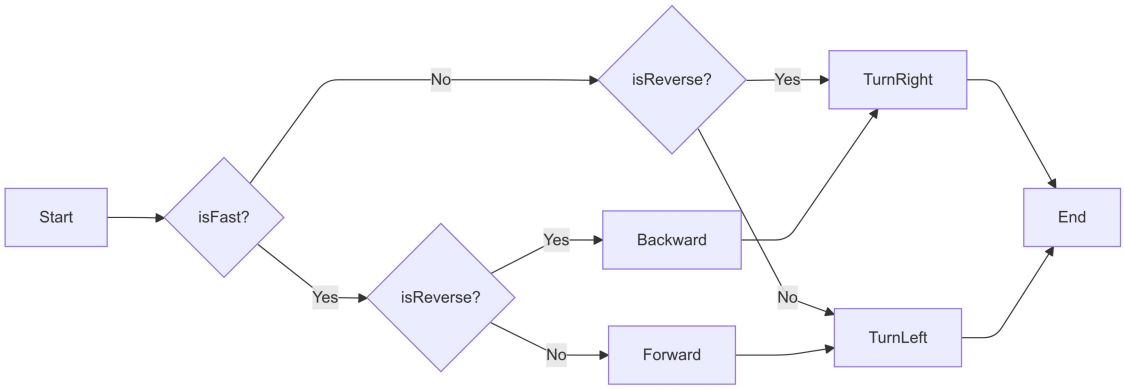

图 2-4-2 TurnLeftCommand 方法流程图

TurnRightCommand 类：

方法	描述
operator()(PoseHandler &poseHandler)	执行右转操作，依据车辆的状态（加速或倒车）来转向。

表 2-4-3 TurnRightCommand 类函数成员

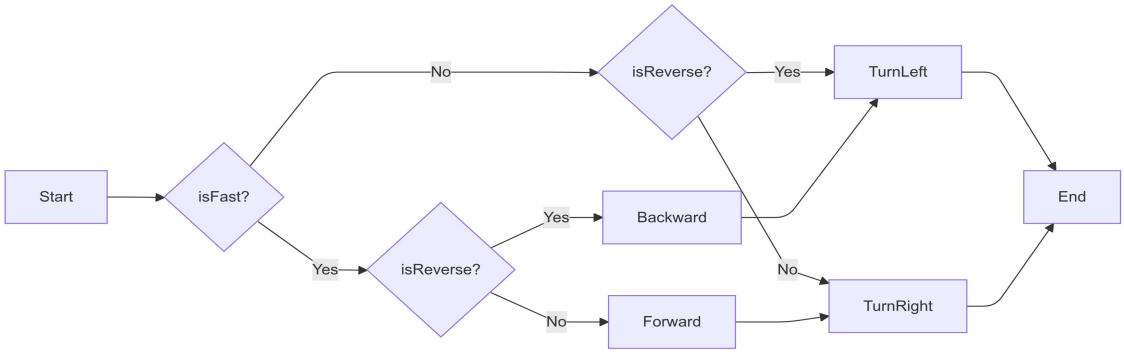

图 2-4-3 TurnRightCommand 方法流程图

FastCommand 类：

方法	描述
operator()(PoseHandler &poseHandler)	切换加速模式

表 2-4-4 FastCommand 类函数成员

ReverseCommand 类：

方法	描述
operator()(PoseHandler &poseHandler)	切换倒车模式

表 2-4-5 ReverseCommand 类函数成员

2.5 Executor 接口

方法	描述
Query()	查询当前车辆的姿态，返回车辆 Pose。
Execute(const std::string &command)	接受一个字符串的命令，并根据命令控制车辆的动作。

表 2-5-1 Executor 接口函数成员

2.6 ExecutorImpl 类

成员	类型	描述
poseHandler	PoseHandler	管理当前车辆的状态（位置、朝向等）。

表 2-6-1 ExecutorImpl 类数据成员

方法	描述
ExecutorImpl(const Pose &pose)	构造函数，初始化 poseHandler，并将初始位置传入。
Query()	查询当前车辆的姿态，返回车辆 Pose。
Execute(const std::string &command)	根据输入的命令执行车辆操作。

表 2-6-2 ExecutorImpl 类函数成员

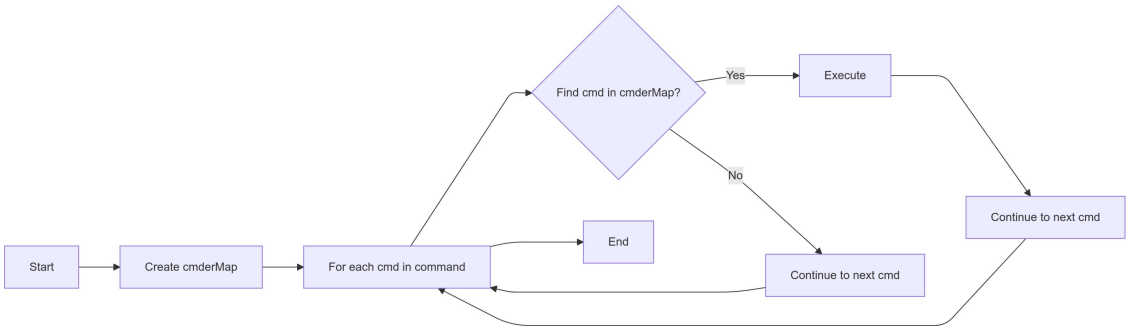

图 2-6-1 Execute 方法流程图

三、软件开发

简单介绍采用什么开发环境，如何编译、连接生成可执行文件。使用了什么调试工具。篇幅不要长。

开发环境：VSCode 与 CMake，配合 mingw64 作为编译器。

编译链接：使用 CMake 管理项目构建，通过配置 CMakeLists.txt 文件指定源文件和生成目标。首先，脚本会检查是否存在 build 目录。如果没有，它会创建一个。其次通过 cmake -G"MinGW Makefiles" -DCMAKE_BUILD_TYPE=Debug ..生成适用于 MinGW 的 Makefiles，并设置构建类型为 Debug。再使用 mingw32-make.exe 进行编译，将源代码链接为可执行文件，

生成的.exe 文件会被放置在 bin 目录下。

调试工具：在 VSCode 中，调试通过集成的调试器进行，通常使用 GDB（GNU 调试器）。VSCode 的调试插件配置了 GDB 调试工具，可以在程序运行时设置断点、查看变量状态、逐行执行代码，帮助调试程序逻辑问题。

四、软件测试

要求基于 gtest 进行功能测试。要求以表格的形式给出所有测试用例，包括：测试用例名称、测试用例功能。并给出测试用例运行结果的截图。

ExecutorTest

测试用例名称	测试用例功能
should_return_init_pose_when_without_command	测试无命令时，Executor 是否返回初始姿态{0, 0, 'E'}
should_return_default_pose_when_init_and_command	测试没有给定初始姿态时，Executor 是否返回默认姿态{0, 0, 'N'}
should_return_x_plus_1_given_command_is_M_and_facing_is_E	朝向 E，执行 M 命令时，期望返回{1, 0, 'E'}
should_return_x_minus_1_given_command_is_M_and_facing_is_W	朝向 W，执行 M 命令时，期望返回{-1, 0, 'W'}
should_return_x_plus_1_given_command_is_M_and_facing_is_N	朝向 N，执行 M 命令时，期望返回{0, 1, 'N'}
should_return_x_minus_1_given_command_is_M_and_facing_is_S	朝向 S，执行 M 命令时，期望返回{0, -1, 'S'}
should_return_facing_N_given_command_is_L_and_facing_is_E	朝向 E，执行 L 命令时，期望返回{0, 0, 'N'}
should_return_facing_S_given_command_is_L_and_facing_is_W	朝向 W，执行 L 命令时，期望返回{0, 0, 'S'}
should_return_facing_W_given_command_is_L_and_facing_is_N	朝向 N，执行 L 命令时，期望返回{0, 0, 'W'}
should_return_facing_E_given_command_is_L_and_facing_is_S	朝向 S，执行 L 命令时，期望返回{0, 0, 'E'}
should_return_facing_S_given_command_is_R_and_facing_is_E	朝向 E，执行 R 命令时，期望返回{0, 0, 'S'}
should_return_facing_N_given_command_is_R_and_facing_is_W	朝向 W，执行 R 命令时，期望返回{0, 0, 'N'}
should_return_facing_E_given_comman	朝向 N，执行 R 命令时，期望返回{0, 0,

d_is_R_and_facing_is_N	'E'}
should_return_facing_W_given_command_is_R_and_facing_is_S	朝向 S，执行 R 命令时，期望返回{0, 0, 'W'}

表 4-1 ExecutorTest 测试用例

```
[==========] Running 25 tests from 4 test suites.
[----------] Global test environment set-up.
[----------] 14 tests from ExecutorTest
[ RUN      ] ExecutorTest.should_return_init_pose_when_without_command
[       OK ] ExecutorTest.should_return_init_pose_when_without_command (0 ms)
[ RUN      ] ExecutorTest.should_return_default_pose_when_init_and_command
[       OK ] ExecutorTest.should_return_default_pose_when_init_and_command (0 ms)
[ RUN      ] ExecutorTest.should_return_x_plus_1_given_command_is_M_and_facing_is_E
[       OK ] ExecutorTest.should_return_x_plus_1_given_command_is_M_and_facing_is_E (0 ms)
[ RUN      ] ExecutorTest.should_return_x_minus_1_given_command_is_M_and_facing_is_W
[       OK ] ExecutorTest.should_return_x_minus_1_given_command_is_M_and_facing_is_W (0 ms)
[ RUN      ] ExecutorTest.should_return_x_plus_1_given_command_is_M_and_facing_is_N
[       OK ] ExecutorTest.should_return_x_plus_1_given_command_is_M_and_facing_is_N (0 ms)
[ RUN      ] ExecutorTest.should_return_x_minus_1_given_command_is_M_and_facing_is_S
[       OK ] ExecutorTest.should_return_x_minus_1_given_command_is_M_and_facing_is_S (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_N_given_command_is_L_and_facing_is_E
[       OK ] ExecutorTest.should_return_facing_N_given_command_is_L_and_facing_is_E (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_S_given_command_is_L_and_facing_is_W
[       OK ] ExecutorTest.should_return_facing_S_given_command_is_L_and_facing_is_W (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_W_given_command_is_L_and_facing_is_N
[       OK ] ExecutorTest.should_return_facing_W_given_command_is_L_and_facing_is_N (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_E_given_command_is_L_and_facing_is_S
[       OK ] ExecutorTest.should_return_facing_E_given_command_is_L_and_facing_is_S (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_S_given_command_is_R_and_facing_is_E
[       OK ] ExecutorTest.should_return_facing_S_given_command_is_R_and_facing_is_E (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_N_given_command_is_R_and_facing_is_W
[       OK ] ExecutorTest.should_return_facing_N_given_command_is_R_and_facing_is_W (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_E_given_command_is_R_and_facing_is_N
[       OK ] ExecutorTest.should_return_facing_E_given_command_is_R_and_facing_is_N (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_W_given_command_is_R_and_facing_is_S
[       OK ] ExecutorTest.should_return_facing_W_given_command_is_R_and_facing_is_S (0 ms)
[----------] 14 tests from ExecutorTest (96 ms total)
```

图 4-1 ExecutorTest 运行结果

ExecutorFastTest

测试用例名称	测试用例功能
should_return_x_plus_2_given_status_is_fast_command_is_M_and_facing_is_E	Fast 状态下，朝向 E，执行 M 命令时，期望返回{2, 0, 'E'}
should_return_N_and_x_plus_1_given_status_is_fast_command_is_L_and_facing_is_E	Fast 状态下，朝向 E，执行 L 命令时，期望返回{1, 0, 'N'}
should_return_S_and_x_plus_1_given_status_is_fast_command_is_R_and_facing_is_E	Fast 状态下，朝向 E，执行 R 命令时，期望返回{1, 0, 'S'}
should_return_y_plus_1_given_command_is_FFM_and_facing_is_N	朝向 N，执行 FFM 命令时，期望返回{0, 1, 'N'}

表 4-2 ExecutorFastTest 测试用例

```
[----------] 4 tests from ExecutorFastTest
[ RUN      ] ExecutorFastTest.should_return_x_plus_2_given_status_is_fast_command_is_M_and_facing_is_E
[       OK ] ExecutorFastTest.should_return_x_plus_2_given_status_is_fast_command_is_M_and_facing_is_E (0 ms)
[ RUN      ] ExecutorFastTest.should_return_N_and_x_plus_1_given_status_is_fast_command_is_L_and_facing_is_E
[       OK ] ExecutorFastTest.should_return_N_and_x_plus_1_given_status_is_fast_command_is_L_and_facing_is_E (0 ms)
[ RUN      ] ExecutorFastTest.should_return_S_and_x_plus_1_given_status_is_fast_command_is_R_and_facing_is_E
[       OK ] ExecutorFastTest.should_return_S_and_x_plus_1_given_status_is_fast_command_is_R_and_facing_is_E (0 ms)
[ RUN      ] ExecutorFastTest.should_return_y_plus_1_given_command_is_FFM_and_facing_is_N
[       OK ] ExecutorFastTest.should_return_y_plus_1_given_command_is_FFM_and_facing_is_N (0 ms)
[----------] 4 tests from ExecutorFastTest (31 ms total)
```

图 4-2 ExecutorFastTest 运行结果

ExecutorReverseTest

测试用例名称	测试用例功能
should_return_x_minus_1_given_status_is_back_command_is_M_and_facing_is_E	Reverse 状态下，朝向 E，执行 M 命令时，期望返回{-1, 0, 'E'}
should_return_S_given_status_is_reverse_command_is_L_and_facing_is_E	Reverse 状态下，朝向 E，执行 L 命令时，期望返回{0, 0, 'S'}
should_return_N_given_status_is_reverse_command_is_R_and_facing_is_E	Reverse 状态下，朝向 E，执行 R 命令时，期望返回{0, 0, 'N'}
should_return_y_plus_1_given_command_is_BBM_and_facing_is_N	朝向 N，执行 BBM 命令时，期望返回{0, 1, 'N'}

表 4-3 ExecutorReverseTest 测试用例

```
[----------] 4 tests from ExecutorReverseTest
[ RUN      ] ExecutorReverseTest.should_return_x_minus_1_given_status_is_back_command_is_M_and_facing_is_E
[       OK ] ExecutorReverseTest.should_return_x_minus_1_given_status_is_back_command_is_M_and_facing_is_E (0 ms)
[ RUN      ] ExecutorReverseTest.should_return_S_given_status_is_reverse_command_is_L_and_facing_is_E
[       OK ] ExecutorReverseTest.should_return_S_given_status_is_reverse_command_is_L_and_facing_is_E (0 ms)
[ RUN      ] ExecutorReverseTest.should_return_N_given_status_is_reverse_command_is_R_and_facing_is_E
[       OK ] ExecutorReverseTest.should_return_N_given_status_is_reverse_command_is_R_and_facing_is_E (0 ms)
[ RUN      ] ExecutorReverseTest.should_return_y_plus_1_given_command_is_BBM_and_facing_is_N
[       OK ] ExecutorReverseTest.should_return_y_plus_1_given_command_is_BBM_and_facing_is_N (0 ms)
[----------] 4 tests from ExecutorReverseTest (32 ms total)
```

图 4-3 ExecutorReverseTest 运行结果

ExecutorReverseFastTest

测试用例名称	测试用例功能
should_return_x_minus_2_given_status_is_fast_and_reverse_command_is_M_and_facing_is_E	Fast + Reverse 状态下，朝向 E，执行 M 命令时，期望返回{-2, 0, 'E'}
should_return_S_and_x_minus_1_given_status_is_fast_and_reverse_command_is_L_and_facing_is_E	Fast + Reverse 状态下，朝向 E，执行 L 命令时，期望返回{-1, 0, 'S'}
should_return_N_and_x_minus_1_given_status_is_fast_and_reverse_command_is_R_and_facing_is_E	Fast + Reverse 状态下，朝向 E，执行 R 命令时，期望返回{-1, 0, 'N'}

表 4-4 ExecutorReverseFastTest 测试用例

```
[----------] 3 tests from ExecutorReverseFastTest
[ RUN      ] ExecutorReverseFastTest.should_return_x_minus_2_given_status_is_fast_and_reverse_command_is_M_and_facing_is_E
[       OK ] ExecutorReverseFastTest.should_return_x_minus_2_given_status_is_fast_and_reverse_command_is_M_and_facing_is_E (0 ms)
[ RUN      ] ExecutorReverseFastTest.should_return_S_and_x_minus_1_given_status_is_fast_and_reverse_command_is_L_and_facing_is_E
[       OK ] ExecutorReverseFastTest.should_return_S_and_x_minus_1_given_status_is_fast_and_reverse_command_is_L_and_facing_is_E (0 ms)
[ RUN      ] ExecutorReverseFastTest.should_return_N_and_x_minus_1_given_status_is_fast_and_reverse_command_is_R_and_facing_is_E
[       OK ] ExecutorReverseFastTest.should_return_N_and_x_minus_1_given_status_is_fast_and_reverse_command_is_R_and_facing_is_E (0 ms)
[----------] 3 tests from ExecutorReverseFastTest (29 ms total)
```

图 4-4 ExecutorReverseFastTest 运行结果

五、特点与不足

1. 技术特点

1）请说明如何利用表驱动方式消除大量冗余的 if 条件判断语句；并解释这样做的好处。

方法：

本系统创建了一个 unordered_map（无序映射），其键值为命令字符（如 'M'），值为相应的操作函数（如 MoveCommand()）。std::function<void(PoseHandler&)> 是一个可以存储函数对象或可调用对象的类型，这里存储了每个命令的处理函数。再通过命令字符 cmd 查找表中的对应操作函数。如果找到了对应的命令，就执行它。

好处：

•减少冗余代码：表驱动模式避免了重复的 if 或 switch 语句，所有的命令都被统一管理在一个表中。当需要增加新命令时，只需向映射表中添加一个新的键值对，而不需要修改复杂的判断逻辑。

•提高代码可维护性：表驱动方法将命令与执行操作分开，使得代码更加模块化和可扩展。如果某个命令的行为发生变化，只需要修改对应的处理函数，而不必改动判断语句。

•提升性能：在查找操作时，std::unordered_map 提供了平均常数时间复杂度（O(1)）。相比于 if 或 switch 语句，表驱动方式可以更高效地进行命令的查找和执行，尤其是在命令种类较多时，性能优势更加明显。

2）请说明如何通过状态抽象和计算属性解决复杂状态流转问题。

状态抽象：

状态抽象是指通过将系统的状态进行抽象，使得每个状态都可以通过简单的对象或枚举值进行表示，而不需要大量的条件判断。

在 ExecutorImpl 类中，不同的行驶状态：普通行驶、加速 (F) 和倒车 (B) 状态。每种状态下执行的指令（M、L、R）具有不同的行为。使用状态抽象可以将这些行为统一管理，避免复杂的条件判断。

该系统使用一个了状态管理类（PoseHandler）来管理当前的状态，并根据当前状态执行相应的行为。每个状态对象负责处理状态下的具体行为。这样，可以根据当前状态的不同，调用不同的状态处理逻辑，而不需要大量的 if 或 switch 判断。

计算属性：

计算属性指的是通过当前状态计算出新的值，避免了繁琐的条件判断。它可以根据状态直接计算出要执行的动作，而不需要逐一检查每个可能的条件。

对于执行的指令（M、L、R）和当前状态（普通、加速、倒车等），我们可以计算出对应的行为。例如，朝向为 E 时，M 指令会让车辆向 x 轴正方向前进；在加速状态下，M 指令应该让车辆前进 2 格，而在倒车状态下，M 指令让车辆向反方向前进。

2. 不足和改进的建议

不足①：

PoseHandler 类可能同时处理了多种任务，比如计算汽车的当前位置、管理状态等，导致类变得复杂。

建议①：

可以将 PoseHandler 中的状态管理和位置更新逻辑分离成不同的类。PositionManager 类仅负责管理位置更新，StateManager 类负责管理车辆状态。使用状态模式来管理不同的状态。每个状态类只处理该状态下的特定行为，减少 PoseHandler 类中的复杂逻辑。

不足②：

每次执行命令调用时都会生成一个 cmderMap，导致效率问题。此外，新增指令时需要修改 Executor、 command 两个地方，增加了维护成本。

建议②：

指令对象生成与执行分开：先生成指令列表，再执行指令；抽取指令对象处理为 command 层：在 command 层中处理所有与指令对象相关的操作；使用单体对象：在 command 层中只生成一个 cmderMap，避免重复生成。

不足③：

不能支持多种类型的车。

建议③：

增加一个枚举类 ExecutorType 来区分不同类型的汽车，构造汽车对象时指定其类型。并为不同类型的汽车构造不同的移动适配器。

六、过程和体会

1. 遇到的主要问题和解决方法

问题①：

在 ExecutorImpl.cpp 等文件中，可能存在大量的 if-else 或 switch 语句来处理不同命令和状态的切换。这会导致代码的重复性高，且在添加新功能时容易出现冗余代码，影响代码的可维护性和可扩展性。

解决方法①：

通过构建一个命令与执行逻辑的映射表来消除复杂的条件判断。这可以将多个 if-else 或 switch 转换成通过查表直接进行处理的方式，从而简化代码逻辑。

问题②：

如果将所有逻辑硬编码在单一的类或方法中，那么新增功能（如添加新的命令或新的状态）将变得非常困难。任何新的命令或状态的加入都需要修改现有的代码，容易导致代码膨胀，甚至引入 bug。

解决方法②：

使用命令模式（Command Pattern）将每种操作封装成一个命令对象，指令的执行由命令对象完成，从而解耦命令的调用者和执行者；同时使用状态模式（State Pattern）管理不同状态下的行为，将每个状态的行为封装在独立的状态对象中。

问题③：

在处理多个状态（如 "快速"、"倒车" 等）时，状态之间的流转可能会变得非常复杂。状态的切换和行为的执行可能被分散在多个地方，导致逻辑混乱。

解决方法③：

使用状态模式使得每个状态的行为封装在一个独立的类中，状态之间的转换通过上下文类来控制。这样，每个状态只负责自己相关的行为，不会影响到其他状态的处理。状态切换清晰且易于扩展。

2. 课程设计的体会

在智能车控制系统的设计与实现过程中，我深入体会到了如何将理论与实践相结合，解决实际问题。首先，系统的复杂性让我意识到设计清晰、模块化结构的重要性。通过拆解任务并合理分配职责，避免了代码的冗长和重复，提升了系统的可维护性。特别是在状态管理和命令执行方面，通过引入表驱动和设计模式（如状态模式、命令模式），我学会了如何高效地处理不同状态下的逻辑切换，并且确保系统具备良好的可扩展性。这些模式的应用，不仅简化了代码结构，还提高了开发效率，使得新增功能能够在不破坏现有代码结构的基础上轻松实现。测试工作也让我认识到质量保障的重要性。通过单元测试和集成测试的结合，确保了系统每个模块的正确性。整个课程设计的过程不仅是对我编程能力的锻炼，更是对我系统思维、问题分析与解决能力的提升。我不仅对 C++面向对象编程有了更深入的了解，还学到了如何高效地组织代码和管理复杂系统，为将来的工程实践打下了坚实的基础。