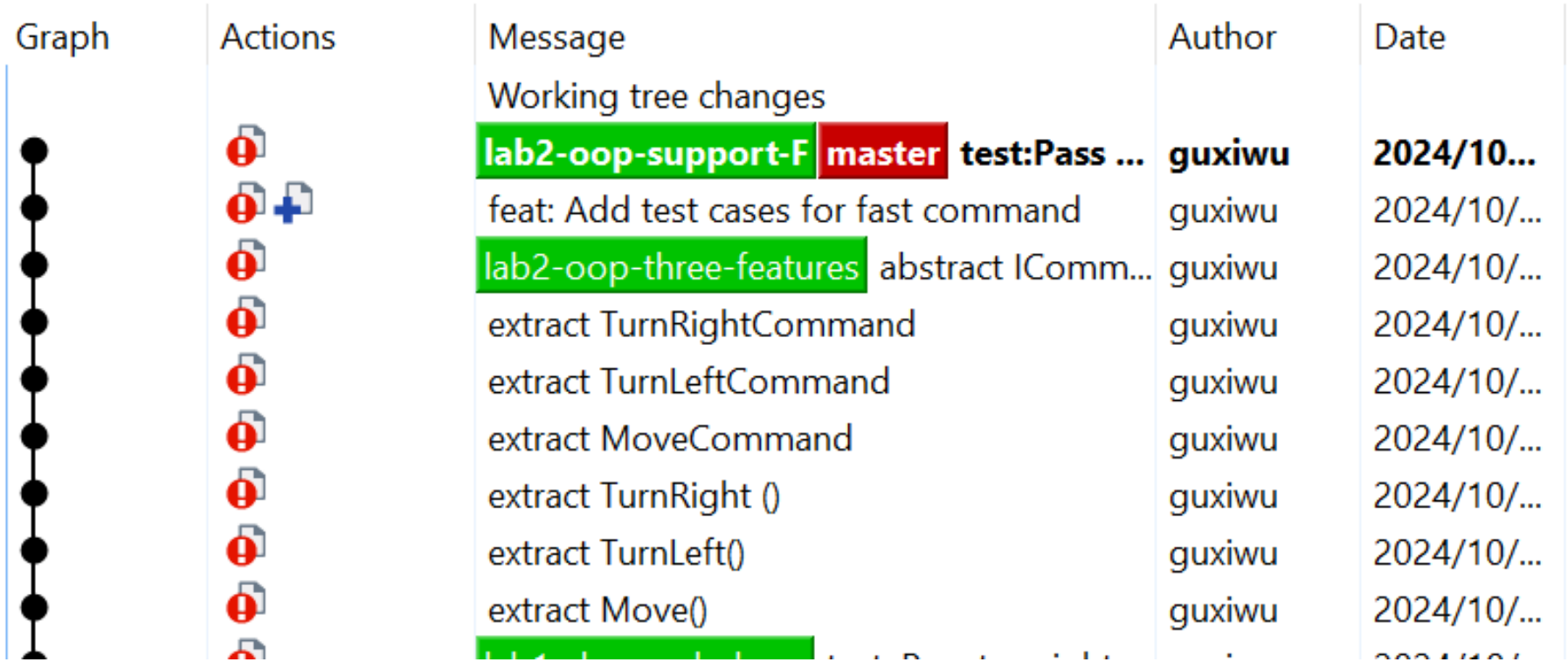
实验三 StepbyStep

实验二完成后，应该已经在目录里创建lab2-oop-support-F分支



Graph	Actions	Message	Author	Date
		Working tree changes		
		lab2-oop-support-F master test:Pass ...	**guxiwu**	**2024/10...**
		feat: Add test cases for fast command	guxiwu	2024/10/...
		lab2-oop-three-features abstract IComm...	guxiwu	2024/10/...
		extract TurnRightCommand	guxiwu	2024/10/...
		extract TurnLeftCommand	guxiwu	2024/10/...
		extract MoveCommand	guxiwu	2024/10/...
		extract TurnRight ()	guxiwu	2024/10/...
		extract TurnLeft()	guxiwu	2024/10/...
		extract Move()	guxiwu	2024/10/...

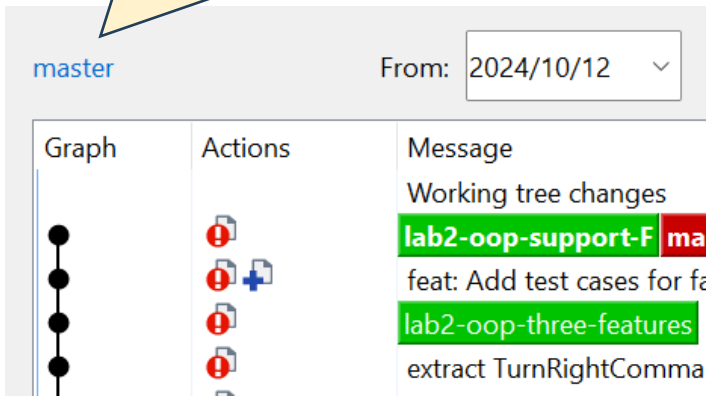
如果没有对应分支，请打开Terminal，输入下面命令创建分支

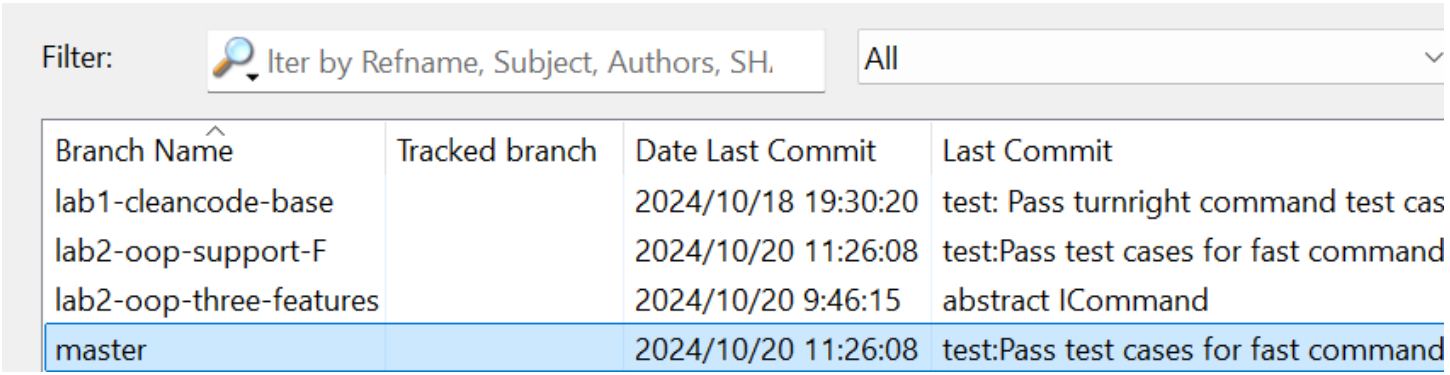
git branch lab2-oop-support-F

HUAWEI

目前已经创建了三个分支

点击master可以查看所有分支





也可以打开Terminal，输入下面命令查看分支

git branch

```
Active code page: 65001
PS E:\CodeProject\CppProject\Experiment4Student> git branch
  lab1-cleancode-base
  lab2-oop-support-F
  lab2-oop-three-features
* master
PS E:\CodeProject\CppProject\Experiment4Student>
```

HUAWEI

C++企业软件开发实践

实验3-1 面向对象编程

HUAWEI

前言

欢迎参加C++企业软件开发实践课程。本课程旨在通过完整开发案例，分享企业软件开发中的实践、经验和要求，帮助在校学生提升软件开发技能，养成良好的软件开发习惯。

实践课程共有4次实验，本课程为实验3，您将继续实践面向对象编程，包括：

- 表驱动：掌握面向对象多态特性融合C++ STL库的使用，进一步提升代码的扩展性
- 分层设计与抽象：掌握有效解耦代码中的循环依赖，从而提高代码的可维护性和可扩展性

同时，实践面向函数式编程，包括：

- 函数式编程、Lambda：掌握C++新特性带来的代码简洁性收益，写出简洁优雅的代码
- 对象生命周期安全：关注C++使用过程中编程陷阱可能导致的问题，对背后原理究根，学习C++特性整体体系

目标

通过本课程的学习，您将能够：

- 表驱动和STL容器、智能指针
  - 掌握C++ STL库的融合使用：通过学习如何有效地使用STL容器和智能指针，提升代码的可扩展性和灵活性
  - 提高代码的可扩展性：通过表驱动的设计模式，增强代码的可维护性和可读性
- 解耦循环依赖
  - 分层设计和抽象化：学习如何通过分层设计和抽象化来有效地解耦代码中的循环依赖
  - 提高代码的可维护性和可扩展性：通过解耦循环依赖，提升代码的可维护性和可扩展性，使代码更易于理解和修改

这些学习目标旨在帮助您更好地掌握C++编程的最佳实践，建立高质量编程的意识，养成良好的开发习惯，提高代码质量和开发效率。

目录

1. 实验2回顾
2. 面向对象项目实践
    2.1 数据抽离解耦循环依赖
    2.2 表驱动提升扩展性
    2.3 状态抽象提升可读性
3. 函数式项目实践
    2.1 函数式编程
    2.2 Lambda对象生命周期安全
    2.3 操作符重载
4. 总结

实验2回顾

```
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    for(const auto cmd:commands){
        //声明一个ICommand类型的智能指针
        std::unique_ptr<ICommand> cmder = nullptr;
        if(cmd == 'M'){
            //智能指针指向子类MoveCommand实例，不用担心delete的问题了
            cmder = std::make_unique<MoveCommand>();

        }
        else if (cmd == 'L') {
            //智能指针指向子类TurnLeftCommand实例，不用担心delete的问题了
            cmder = std::make_unique<TurnLeftCommand>();

        }
        else if (cmd == 'R') {
           //智能指针指向子类TurnRightCommand实例，不用担心delete的问题了
            cmder = std::make_unique<TurnRightCommand>();
        }
        else if(cmd == 'F'){
            cmder = std::make_unique<FastCommand>();
        }

        if(cmder){
            //多态，当cmder指向不同子类实例，调用的是不同的命令
            cmder->DoOperate(*this);
        }
    }
}
```

- 通过封装、父类抽象，完成了指令处理结构的建立
- F指令的处理，依然需要修改指令处理主干逻辑

目录

1. 实验2回顾
2. 面向对象项目实践
   - 2.1 数据抽离解耦循环依赖
   - 2.2 表驱动提升扩展性
   - 2.3 状态抽象提升可读性
3. 函数式项目实践
   - 2.1 函数式编程
   - 2.2 Lambda对象生命周期安全
   - 2.3 操作符重载
4. 总结

面向对象编程：指令处理的类独立到文件中

当前指令处理的类作为ExecutorImpl的内部类定义在src/ExecutorImpl.hpp中

```
/*
    Executor的具体实现
*/
class ExecutorImpl: public Executor {
private:
    //定义所有具体Command对象的抽象基类ICommand
    class ICommand {…
    //定义一个嵌套类MoveCommand，完成Move动作（M指令）
    class MoveCommand final : public ICommand {…

    class TurnLeftCommand final : public ICommand {…

    class TurnRightCommand final : public ICommand {…

    class FastCommand final :public ICommand {…
```

面向对象编程：指令处理的类独立到文件中

创建src/Command.hpp，将指令处理的类复制到该文件中

```
#pragma once

#include "ExecutorImpl.hpp"

namespace adas {
    //定义所有具体Command对象的抽象基类ICommand
    class ICommand {…

    //定义一个嵌套类MoveCommand，完成Move动作（M指令）
    class MoveCommand final : public ICommand {…

    class TurnLeftCommand final : public ICommand {…

    class TurnRightCommand final : public ICommand {…

    class FastCommand final :public ICommand {…
}
```

面向对象编程：修改依赖方法访问权限

src/Command.hpp文件会报错，这是由于ExecutorImpl的Move、TurnLeft、TurnRight、Fast、IsFast成员函数是私有的，将其改为公有

```
public:
    void Move(void) noexcept;              // 执行M指令的逻辑现在封装在函数Move里
    void TurnLeft(void) noexcept;          // 执行L指令的逻辑现在封装在函数TurnLeft里
    void TurnRight(void) noexcept;         // 执行R指令的逻辑现在封装在函数TurnRight里
    void Fast(void) noexcept;              // 切换加速状态
    bool IsFast(void) const noexcept;      // 查询当前是否处于加速状态
```

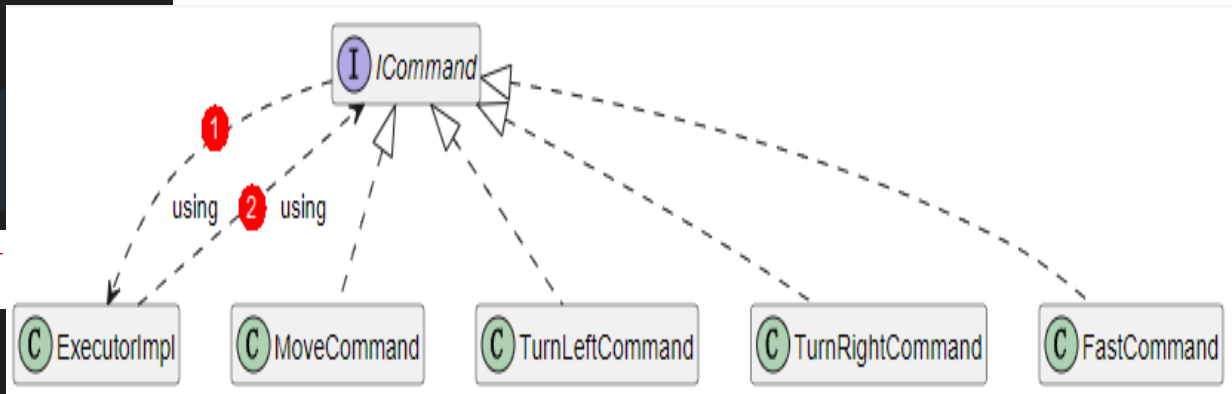
src/ExecutorImpl.cpp文件也会报错，需要引入指令处理的类

编译运行验证后，代码及时入库
git add .
git commit -m "command to extract independent file"

面向对象编程：代码问题分析

```cpp
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    for(const auto cmd:commands){
        //声明一个ICommand类型的智能指针
        std::unique_ptr<ICommand> cmder = nullptr;
        if(cmd == 'M'){
            //智能指针指向子类MoveCommand实例，不用担心delete的问题了
            cmder = std::make_unique<MoveCommand>();

        }
        else if (cmd == 'L') {…
        else if (cmd == 'R') {…
        else if(cmd == 'F'){…
```

ExecutorImpl 依赖于 ICommand




```cpp
//定义所有具体Command对象的抽象基类ICommand
class ICommand {
public:
    virtual ~ICommand() = default;
    virtual void DoOperate(ExecutorImpl& exector) const noexcept = 0;
};
```

ICommand 依赖于 ExecutorImpl

Command <-> ExecutorImpl 循环依赖，该如何解决?

面向对象编程：循环依赖的根源分析

```
//定义一个嵌套类MoveCommand，完成Move动作（M指令）
class MoveCommand final : public ICommand {
public:
    //执行Move动作，需要委托ExecutorImp&执行器来完成动作
    void DoOperate(ExecutorImpl& executor) const noexcept {
        if(executor.IsFast()) {
            executor.Move();  //如果是F状态，多执行一次MOVE
        }
        executor.Move();  //通过执行器
    }
};
```

ICommand 依赖的是 ExecutorImpl 的车状态数据，即成员函数 Move、TurnLeft、TurnRight、Fast、IsFast。
只要将其抽离出来作为 ICommand 和 ExecutorImpl 的共同依赖即可解决循环依赖问题

面向对象编程：解耦循环依赖，封装抽离ExecutorImpl状态数据

创建src/PoseHandler.hpp

```hpp
#pragma once

#include "Executor.hpp"

namespace adas {

    class PoseHandler final {
        public:
            PoseHandler(const Pose& pose) noexcept;
            PoseHandler(const PoseHandler&) = delete;                   // 禁止拷贝构造
            PoseHandler& operator=(const PoseHandler&) = delete;        // 禁止拷贝赋值

        public:
            void Move(void) noexcept;               // 向前移动
            void TurnLeft(void) noexcept;           // 向左转
            void TurnRight(void) noexcept;          // 向右转
            void Fast(void) noexcept;               // 切换模式
            bool IsFast(void) const noexcept;       // 是否处于快速模式
            Pose Query(void) const noexcept;        // 查询当前位置

        private:
            Pose pose;              // 当前位置
            bool fast{false};       // 是否处于快速状态
    };

}
```

创建src/PoseHandler.cpp

```cpp
#include "PoseHandler.hpp"

namespace adas {

    PoseHandler::PoseHandler(const Pose& pose) noexce

    void PoseHandler::Move() noexcept {…

    void PoseHandler::TurnLeft() noexcept {…

    void PoseHandler::TurnRight() noexcept {…

    void PoseHandler::Fast() noexcept {…

    bool PoseHandler::IsFast() const noexcept {…

    Pose PoseHandler::Query(void) const noexcept {…

}
```

代码请自行实现

面向对象编程：解耦循环依赖，封装抽离ExecutorImpl状态数据

修改src/Command.hpp

添加依赖 PoseHandler

对应参数改为 PoseHandler

具体实现改为 PoseHandler

```cpp
#pragma once

#include "PoseHandler.hpp"

namespace adas {

    //定义所有具体Command对象的抽象基类ICommand
    class ICommand {
    public:
        virtual ~ICommand() = default;
        virtual void DoOperate(PoseHandler& poseHandler) const noexcept = 0;
    };

    //定义一个嵌套类MoveCommand，完成Move动作（M指令）
    class MoveCommand final : public ICommand {
    public:
        //执行Move动作，需要委托ExecutorImp&执行器来完成动作
        void DoOperate(PoseHandler& poseHandler) const noexcept {
            if(poseHandler.IsFast()) {
                poseHandler.Move();  //如果是F状态，多执行一次MOVE
            }
            poseHandler.Move();  //通过执行器完成Move动作
        }
    };

    class TurnLeftCommand final : public ICommand {…

    class TurnRightCommand final : public ICommand {…

    class FastCommand final :public ICommand {…
}
```

TurnLeftCommand 、TurnRightCommand 、FastCommand函数由大家自行修改

面向对象编程：解耦循环依赖，封装抽离Executo

修改src/ExecutorImpl.hpp

```
#pragma once

#include "Executor.hpp"
#include "PoseHandler.hpp"
#include <string>

namespace adas {

    /*
        Executor的具体实现
    */
    class ExecutorImpl: public Exe
    public:
        // 构造函数
        explicit ExecutorImpl(cons
        // 默认析构函数
        ~ExecutorImpl() noexcept =

        // 不能拷贝
        ExecutorImpl(const Executo
        // 不能赋值
        ExecutorImpl &operator=(const
    public:
        // 查询当前汽车姿态，是父类抽象方法Query的具体实现
        Pose Query(void) const noexcept override;
        // 第二阶段新增加的纯虚函数，执行一个用字符串表示的指令
        void Execute(const std::string &commands) noexcept override;

    private:
        PoseHandler poseHandler;     // 状态管理类

    };

}
```

添加依赖 PoseHandler

删除状态管理函数成员，只保留实现父类的两个虚函数

状态管理交给 PorseHandler

修改src/ExecutorImpl.cpp

初始化状态改为初始化状态管理器

```
ExecutorImpl::ExecutorImpl(const Pose &pose) noexcept : poseHandler(pose) {}

Pose ExecutorImpl::Query() const noexcept {
    return poseHandler.Query();
}

void ExecutorImpl::Execute(const std::string
    for(const auto cmd:commands){
        //声明一个ICommand类型的智能指针
        std::unique_ptr<ICommand> cmder = nullptr;
        if(cmd == 'M'){…
        else if (cmd == 'L') {…
        else if (cmd == 'R') {…
        else if(cmd == 'F'){…

        if(cmder){
            //多态，当cmder指向不同子类实例，调用的是不同的命
            cmder->DoOperate(poseHandler)
        }
    }
}
```

查询状态改为
查询状态管理器

修改状态改为
修改状态管理器

编译运行验证后，代码及时入库
git add .
git commit –m “extract PoseHandler, decouple ExecutorImpl, Command interdependencies”

本节小结

通过数据抽离和代码分层策略提高代码可扩展性和可维护性：

- 抽离数据：
  - 演示如何通过抽离数据来解耦代码中的循环依赖
  - 强调数据抽离在简化代码结构和减少依赖关系中的作用
- 代码分层策略：
  - 介绍代码分层策略及其在解决相互依赖问题中的应用
  - 通过具体示例展示如何分层代码以实现更清晰的结构和更低的耦合度
- 封装和抽离Executor的状态数据：
  - 解释如何通过封装和抽离Executor的状态数据来消除循环依赖
  - 展示这种方法如何提高代码的可扩展性和可维护性
- 提升代码质量：
  - 强调通过数据抽离和代码分层策略，可以显著提升代码的可扩展性和可维护性

目录

1. 实验2回顾
2. 面向对象项目实践
   - 2.1 数据抽离解耦循环依赖
   - 2.2 表驱动提升扩展性
   - 2.3 状态抽象提升可读性
3. 函数式项目实践
   - 2.1 函数式编程
   - 2.2 Lambda对象生命周期安全
   - 2.3 操作符重载
4. 总结

课程实训需求2-2 支持倒车指令

Executor组件增加执行倒车指令：

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

面向对象编程：B指令对指令处理结构的扩展性诉求分析

```
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    for(const auto cmd:commands){
        //声明一个ICommand类型的智能指针
        std::unique_ptr<ICommand> cmder = nullptr;
        if (cmd == 'M'){…
        else if (cmd == 'L') {…
        else if (cmd == 'R') {…
        else if (cmd == 'F') {…
        else if (cmd == 'B') {
            cmder = std::make_unique<ReverseCommand>();
        }

        if(cmder){
            //多态，当cmder指向不同子类实例，调用的是不同的命令
            cmder->DoOperate(poseHandler);
        }
    }
}
```

- 通过封装、多态特性的应用，完成了指令处理结构的建立
- B指令的处理，依然需要修改指令处理主干逻辑

先不要着急进行代码实现，目前的代码仍然可以进一步优化

面向对象编程：表驱动简介

- 在C++中，可以使用std::map来存储键值对，其中键是条件，值是对应的处理函数或者对象
- 表驱动的优势：
  - 简化代码逻辑：避免了大量的条件语句，使代码更易读
  - 易于扩展：添加新的条件或和处理函数时，只需要更新表格，而不需要修改现有的逻辑
  - 提高可维护性：逻辑和数据分离，便于维护和调试

面向对象编程：应用表驱动消减圈复杂度，功能代码实现

添加依赖：#include <unordered_map>

```
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    // 表驱动
    std::unordered_map<char, std::unique_ptr<ICommand>> cmderMap;
    // 建立操作M和前进指令的映射关系
    cmderMap.emplace('M', std::make_unique<MoveCommand>());

    // 请大家自行实现：建立操作L、R、F的映射关系

    for(const auto cmd : commands){
        // 根据操作查找表驱动
        const auto it = cmderMap.find(cmd);
        // 如果找到表驱动，执行操作对应的指令
        if (it != cmderMap.end()){
            it->second->DoOperate(poseHandler);
        }
    }
}
```

建立指令和对应操作类的映射

通过表驱动避免了冗长的if-else语句

查找失败会返回cmderMap.end()
查找成功会返回指针，其中first指向key，second指向value

请大家自动添加L、R、F指令的表驱动映射，编译运行验证后，代码及时入库
git add .
git commit –m “command table-driven”

本节小结

通过面向对象编程和STL库提升代码可扩展性：

- 表驱动消减圈复杂度：
  - 通过表驱动方法减少代码中的复杂循环，提高代码的可读性和维护性
- 多态特性与STL库的融合：
  - 将多态特性与STL（标准模板库）结合使用，提升代码的灵活性和可扩展性

目录

1. 实验2回顾
2. 面向对象项目实践
   - 2.1 数据抽离解耦循环依赖
   - 2.2 表驱动提升扩展性
   - 2.3 状态抽象提升可读性
3. 函数式项目实践
   - 2.1 函数式编程
   - 2.2 Lambda对象生命周期安全
   - 2.3 操作符重载
4. 总结

面向对象编程：PoseHandler待优化代码分析

```
void PoseHandler::Move() noexcept {
    if (pose.heading == 'E')       { ++pose.x; }
    else if (pose.heading == 'W') { --pose.x; }
    else if (pose.heading == 'N') { ++pose.y; }
    else if (pose.heading == 'S') { --pose.y; }
}
```

前进操作可以理解为在当前位置(x,y)的基础上分别加上(1,0)|(-1,0)|(0,1)|(0,-1)

```
void PoseHandler::TurnLeft() noexcept {
    if (pose.heading == 'E')        { pose.heading = 'N';}
    else if (pose.heading == 'N')   { pose.heading = 'W';}
    else if (pose.heading == 'W')   { pose.heading = 'S';}
    else if (pose.heading == 'S')   { pose.heading = 'E';}
}

void PoseHandler::TurnRight() noexcept {
    if (pose.heading == 'E')        { pose.heading = 'S';}
    else if (pose.heading == 'S')   { pose.heading = 'W';}
    else if (pose.heading == 'W')   { pose.heading = 'N';}
    else if (pose.heading == 'N')   { pose.heading = 'E';}
}
```

转向操作可以理解为在E|N|W|S间相互切换

面向对象编程：前进操作优化

添加src/Point.hpp

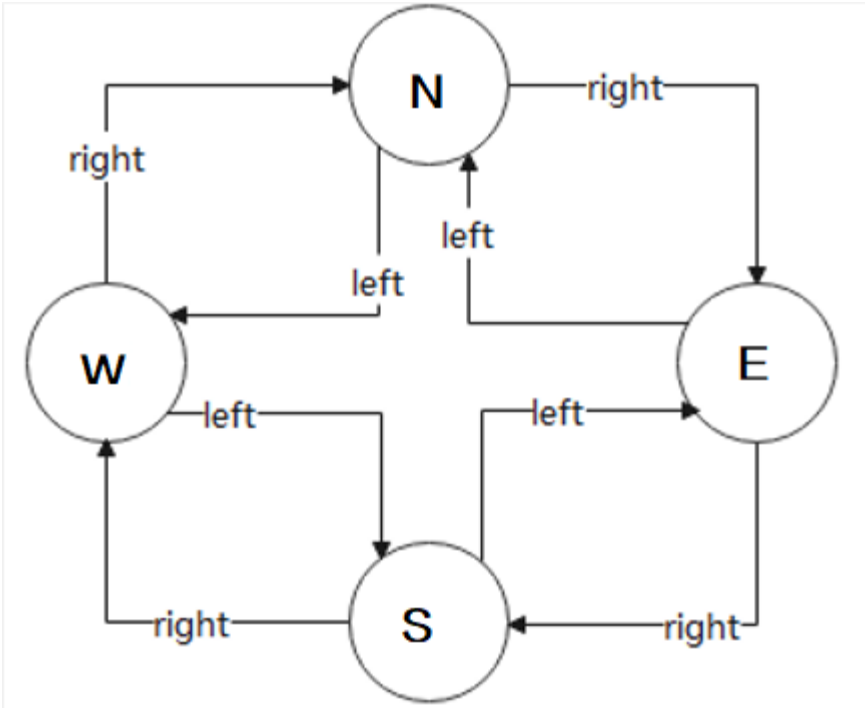
```hpp
#pragma once

namespace adas {

    class Point final {
    public:
        Point(const int x, const int y) noexcept;

        Point(const Point& rhs) noexcept;                // 拷贝构造
        Point& operator=(const Point& rhs) noexcept;     // 拷贝赋值

        Point& operator+=(const Point& rhs) noexcept;    // 移动

    public:
        int GetX(void) const noexcept;  // 获取X坐标
        int GetY(void) const noexcept;  // 获取Y坐标

    private:
        int x;      // X坐标
        int y;      // Y坐标
    };

}
```

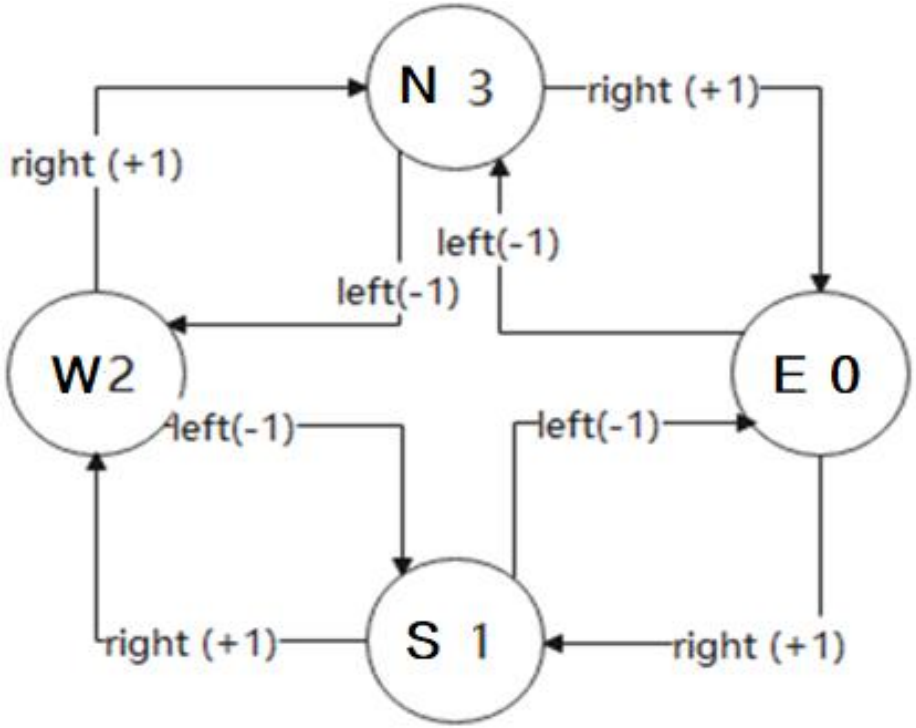
添加src/Point.cpp，实现Point.hpp中的函数，请大家自行实现

```cpp
#include "Point.hpp"

namespace adas {

    // 自行实现Point.hpp中的函数

}
```

面向对象编程：转向操作优化

用计算机的原始计算属性解决复杂状态流转问题





右转后状态编码 =（当前状态编码 + 1）% 4

左转后状态编码 =（当前状态编码 – 1 + 4）% 4 =（当前状态 + 3）% 4

面向对象编程：转向操作优化

添加src/Direction.hpp

```hpp
#pragma once

#include "Point.hpp"

namespace adas {

    /**
     * 方向类
     */
    class Direction final {
    public:
        // 根据方向字符获取方向
        static const Direction& GetDirection(const char heading) noexcept;

    public:
        Direction(const unsigned index, const char heading) noexcept;

    public:
        const Point& Move(void) const noexcept;
        const Direction& LeftOne(void) const noexcept;
        const Direction& RightOne(void) const noexcept;

        const char GetHeading(void) const noexcept;          // 获取方向字符

    private:
        unsigned index;      // 方向索引 0 1 2 3
        char heading;        // 方向字符 E S W N
    };

}
```

添加src/Direction.cpp

```cpp
#include "Direction.hpp"

namespace adas {

    // 4种方向
    static const Direction directions[4] = {
        {0, 'E'},
        {1, 'S'},
        {2, 'W'},
        {3, 'N'},
    };

    // 4种前进坐标
    static const Point points[4] = {
        {1, 0},      // E
        {0, -1},     // S
        {-1, 0},     // W
        {0, 1},      // N
    };

ction& Direction::GetDirection(const char heading) noexc

:Direction(const unsigned index, const char heading) noe

    const Point& Direction::Move() const noexcept {…

    const Direction& Direction::LeftOne() const noexcept {…

Direction& Direction::RightOne() const noexcept {…

char Direction::GetHeading() const noexcept {…
```

使用 Direction 类的初始化方法定义了4种方向的索引和字符

定义了4种方向的前进坐标

当前朝向对应的移动坐标，例如“E”应该对应“(1,0)”

当前朝向对应的转向，例如朝向为“E”时LeftOne应该返回“N”，RightOne应该返回“S”

面向对象编程：转向操作优化

directions方向数组建立了0-E、1-S、2-W、3-N的映射关系，这样就可以通过当前的方向索引计算出左转或右转后的方向索引。Points坐标数组给出了每个方向对应的移动坐标，(1,0)|(0, -1)|(-1,0)|(0,1)正好对应了E|S|W|N对应的移动坐标。

```
// 4种方向
static const Direction directions[4] = {
    {0, 'E'},
    {1, 'S'},
    {2, 'W'},
    {3, 'N'},
};

// 4种前进坐标
static const Point points[4] = {
    {1, 0},     // E
    {0, -1},    // S
    {-1, 0},    // W
    {0, 1},     // N
};
```

例如directions[0]对应的方向数据为{0,'E'}，代表索引0对应了方向“E”；

points[0]对应的移动坐标为{1,0}，代表向东移动一步坐标需要加上(1,0)；

根据右转状态编码公式：(0 + 1) % 4 = 1，direcionts[1]对应的方向数据为{1,'S'}，代表朝向为“E”时右转后的朝向为“S”；

同理左转状态编码公式：(0 + 3) % 4 = 3， direcionts[3]对应的方向数据为{3,'N'}，代表朝向为“E”时左转后的朝向为“N”

面向对象编程：转向操作优化

以下给出GetDirecion函数的示例，请自行实现剩余函数

```
const Direction& Direction::GetDirection(const char heading) noexcept {
    for (const auto& direction : directions) {
        if (direction.heading == heading) {
            return direction;
        }
    }

    return directions[3];
}
```

这里直接返回了当前heading对应的directions数组中的元素
要求Point和Direction类型的返回直接使用points和directions数组中的元素，通过当前index计算对应元素的索引

如果没有匹配的方向返回默认方向N

实现了坐标和方向后，还需要修改src/PoseHandler.hpp和src/PoseHandler.cpp

面向对象编程：PoseHandler待优化代码分析

修改src/PoseHandler.hpp

```hpp
#pragma once

#include "Executor.hpp"
#include "Direction.hpp"

namespace adas {

    /**
     * 状态管理类
     */
    class PoseHandler final {
        public:
            PoseHandler(const Pose& pose) noexcept;
            PoseHandler(const PoseHandler&) = delete;
            PoseHandler& operator=(const PoseHandler&) = delete;

        public:
            void Move(void) noexcept;           // 向前移动
            void TurnLeft(void) noexcept;       // 向左转
            void TurnRight(void) noexce
            void Fast(void) noexcept;
            bool IsFast(void) const noe
            Pose Query(void) const noex

        private:
            Point point;
            const Direction* facing;
            bool fast{false};
    };

}
```

添加依赖

使用坐标Point和方向Direction替换Pose

修改src/PoseHandler.cpp

```cpp
#include "PoseHandler.hpp"

namespace adas {

    PoseHandler::PoseHandler(const Pose& pose) noexcept
        // 初始化Pose改为初始化Point和Direction
        : point(pose.x, pose.y)
        , facing(&Direction::GetDirection(pose.heading)) {}

    void PoseHandler::Move() noexcept {
        // 将当前坐标加上方向向量
        point += facing->Move();
    }

    void PoseHandler::TurnLeft() noexcept {
        // 左转
        facing = &(facing->LeftOne());
    }

    void PoseHandler::TurnRight() noexcept {
        // 右转
        facing = &(facing->RightOne());
    }

    void PoseHandler::Fast() noexcept {…

    bool PoseHandler::IsFast() const noexcept {…

    Pose PoseHandler::Query(void) const noexcept {
        return {point.GetX(), point.GetY(), facing->GetHeading()};
    }

}
```

面向对象编程：PoseHandler待优化代码分析

修改src/PoseHandler.hpp

```hpp
#pragma once

#include "Executor.hpp"
#include "Direction.hpp"

namespace adas {

    /**
     * 状态管理类
     */
    class PoseHandler final {
        public:
            PoseHandler(const Pose& pose) noexcept;
            PoseHandler(const PoseHandler&) = delete;
            PoseHandler& operator=(const PoseHandler&) = delete;

        public:
            void Move(void) noexcept;           // 向前移动
    };

}
```

添加依赖

修改src/PoseHandler.cpp

```cpp
#include "PoseHandler.hpp"

namespace adas {

    PoseHandler::PoseHandler(const Pose& pose) noexcept
        // 初始化Pose改为初始化Point和Direction
        : point(pose.x, pose.y)
        , facing(&Direction::GetDirection(pose.heading)) {}

    void PoseHandler::Move() noexcept {
        // 将当前坐标加上方向向量
        point += facing->Move();
    }

    void PoseHandler::TurnLeft() noexcept {
        // 左转
        facing = &(facing->LeftOne());
    }

    return {point.GetX(), point.GetY(), facing->GetHeading()};
    }

}
```

编译运行验证后，代码及时入库

git add .

git commit –m “extract Point&Direction, simplify the code cyclomatic complexity in PoseHandler through state changes”

git branch lab3-oop-recfactor-final # 面向对象的代码优化完毕，创建一个新的分支

本节小结

通过状态抽象和计算属性解决复杂状态流转问题：

- 状态问题的抽象：
  - 介绍如何将复杂的状态问题进行抽象，以简化状态管理
- 利用计算机的原始计算属性：
  - 通过计算机的基本计算属性来处理复杂的状态流转
  - 展示如何将复杂的状态转换问题简化为基本的数学运算
- 小车状态的抽象：解释如何通过这种抽象方法来简化状态管理和转换
- 指令的数字化抽象：将所有指令抽象为数字，通过加减运算来处理指令

目录

1. 实验2回顾
2. 面向对象项目实践
   - 2.1 数据抽离解耦循环依赖
   - 2.2 表驱动提升扩展性
   - 2.3 状态抽象提升可读性
3. 函数式项目实践
   - 2.1 函数式编程
   - 2.2 Lambda对象生命周期安全
   - 2.3 操作符重载
4. 总结

面向对象编程问题分析

1. 继承实现多态

   通过继承机制实现的多态性可能导致较高的耦合度。这是因为子类与父类之间存在直接的关系，子类必须实现或覆盖父类的所有接口

2. 接口实现的强制性

   无论子类的具体需求如何，它都被迫实现父类定义的所有接口，这可能包括一些不必要或不相关的方法

3. 虚地址引发的问题

   多态性通常涉及虚函数表，这可能导致在运行时定位问题的难度增加，特别是在出现错误或异常时

目录

1. 实验2回顾
2. 面向对象项目实践
   - 2.1 数据抽离解耦循环依赖
   - 2.2 表驱动提升扩展性
   - 2.3 状态抽象提升可读性
3. 函数式项目实践
   - 2.1 函数式编程
   - 2.2 Lambda对象生命周期安全
   - 2.3 操作符重载
4. 总结

函数式编程

- 一种编程范式，强调对表达式的运算而非执行命令（声明式）
- 纯函数（pure function）
- 延迟计算（lazy evaluation）C++20

纯函数在C++中如何保障?

- 没有副作用：class中采用static，const关键字修饰
- 不变性Immutable

函数式编程

在C++中，函数式编程有许多优势和收益：
1. 代码简洁性：函数式编程通常使用更少的代码来实现相同的功能，使代码更加简洁和易读。
2. 可维护性：由于函数是纯函数且数据是不可变的，代码更容易理解和测试。这种特性使得代码的维护和调试更加方便。
3. 可预测性：纯函数的行为是确定的，对于相同的输入总是产生相同的输出，因此我们可以更准确地预测代码的行为。
4. 并发编程：由于函数式编程强调不可变性和无副作用，减少了并发编程中的数据竞争问题，使得多线程编程更加安全和高效。
5. 代码复用性：函数式编程中的高阶函数和柯里化等概念使得代码更具复用性和组合性，可以更方便地构建复杂的功能。

这些优势使得函数式编程在提高代码质量、减少错误和提高开发效率方面具有显著的收益。

函数式编程实现

删除Icommand，修改MoveCommand

```cpp
#pragma once

#include "PoseHandler.hpp"

#include <functional>

namespace adas {

    // class ICommand {
    // public:
    //     virtual ~ICommand() = default;
    //     virtual void DoOperate(PoseHandler& poseHandler) const noexcept = 0;
    // };

    class MoveCommand final { // : public ICommand {
    public:
        // void DoOperate(PoseHandler& poseHandler) const noexcept {
        //     if(poseHandler.IsFast()) {
        //         poseHandler.Move();
        //     }
        //     poseHandler.Move();
        // }

        // 定义函数对象operate，接收参数PoseHandler，返回void
        const std::function<void(PoseHandler& PoseHandler)> operate = [](PoseHandler& poseHandler) noexcept {
            if(poseHandler.IsFast()) {
                poseHandler.Move();
            }
            poseHandler.Move();
        };
    };
```

添加依赖

定义函数对象，该对象接收参数PoseHandler，返回void

TurnLeftCommand
TurnRightCommand
FastCommand
由大家自行修改

函数式编程实现

修改ExecutorImpl.cpp

```cpp
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    // 表驱动
    std::unordered_map<char, std::function<void(PoseHandler& PoseHandler)>> cmderMap;
    // 前进
    MoveCommand moveCommand;
    cmderMap.emplace('M', moveCommand.operate);
    // 左转
    TurnLeftCommand turnLeftCommand;
    cmderMap.emplace('L', turnLeftCommand.operate);
    // 右转
    TurnRightCommand turnRightCommand;
    cmderMap.emplace('R', turnRightCommand.operate);
    // 快速
    FastCommand fastCommand;
    cmderMap.emplace('F', fastCommand.operate);

    for(const auto cmd : commands){
        // 根据操作查找表驱动
        const auto it = cmderMap.find(cmd);
        // 如果找到表驱动，执行操作对应的指令
        if (it != cmderMap.end()){
            it->second(poseHandler);
        }
    }
}
```

修改表驱动的数据结构

注意需要先初始化对象，再获取元素

编译运行验证后，代码及时入库
git add .
git commit –m “use lambda to optimize code”

可以直接将其作为函数使用

目录

1. 实验2回顾
2. 面向对象项目实践
   - 2.1 数据抽离解耦循环依赖
   - 2.2 表驱动提升扩展性
   - 2.3 状态抽象提升可读性
3. 函数式项目实践
   - 2.1 函数式编程
   - 2.2 Lambda对象生命周期安全
   - 2.3 操作符重载
4. 总结

函数式编程实现：代码问题分析

```
class MoveCommand final { // : public ICommand {
public:
    // void DoOperate(PoseHandler& poseHandler) const noexcept {
    //     if(poseHandler.IsFast()) {
    //         poseHandler.Move();
    //     }
    //     poseHandler.Move();
    // }

    // 定义函数对象operate，接收参数PoseHandler，返回void
    const std::function<void(PoseHandler& PoseHandler)> operate = [](PoseHandler& poseHandler) noexcept {
        if(poseHandler.IsFast()) {
            poseHandler.Move();
        }
        poseHandler.Move();
    };
};
```

- **公开操作：**

使用public关键字可能破坏类的封装性，因为它允许外部代码直接访问和修改类成员变量

- **封装性破坏：**

缺乏足够的封装，业务逻辑直接暴露，从而带来安全隐患，同时也降低了模块的独立性和安全性

目录

1. 实验2回顾
2. 面向对象项目实践
   - 2.1 数据抽离解耦循环依赖
   - 2.2 表驱动提升扩展性
   - 2.3 状态抽象提升可读性
3. 函数式项目实践
   - 2.1 函数式编程
   - 2.2 Lambda对象生命周期安全
   - 2.3 操作符重载
4. 总结

确保对象生命周期安全保证：MoveCommand操作符重载

修改MoveCommand

```
class MoveCommand final { // : public ICommand {
public:
    // void DoOperate(PoseHandler& poseHandler) const noexcept
    //     if(poseHandler.IsFast()) {
    //         poseHandler.Move();
    //     }
    //     poseHandler.Move();
    // }

    // const std::function<void(PoseHandler& PoseHandler)> oper
    //     if(poseHandler.IsFast()) {
    //         poseHandler.Move();
    //     }
    //     poseHandler.Move();
    // };

    void operator()(PoseHandler& poseHandler) const noexcept {
        if(poseHandler.IsFast()) {
            poseHandler.Move();
        }
        poseHandler.Move();
    }
};
```

operator() 是一个特殊的函数，称为函数调用运算符。它允许一个对象像函数一样被调用。当你定义了一个重载了 operator() 的类时，你可以创建一个类的实例，然后像调用函数一样调用它。

TurnLeftCommand
TurnRightCommand
FastCommand
由大家自行修改

确保对象生命周期安全保证：表驱动修改

修改ExecutorImpl.cpp

```
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    // 表驱动
    std::unordered_map<char, std::function<void(PoseHandler& PoseHandler)>> cmderMap;
    cmderMap.emplace('M', MoveCommand());          // 前进
    cmderMap.emplace('L', TurnLeftCommand());      // 左转
    cmderMap.emplace('R', TurnRightCommand());     // 右转
    cmderMap.emplace('F', FastCommand());          // 快速

    for(const auto cmd : commands){
        // 根据操作查找表驱动
        const auto it = cmderMap.find(cmd);
        // 如果找到表驱动，执行操作对应的指令
        if (it != cmderMap.end()){
            it->second(poseHandler);
        }
    }
}
```

像调用函数一样调用即可

编译运行验证后，代码及时入库
git add .
git commit –m “operator overloading is used to simplify code”

简化表驱动代码：初始化语义应用

修改ExecutorImpl.cpp，初始化表驱动

```cpp
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    // 表驱动
    std::unordered_map<char, std::function<void(PoseHandler& PoseHandler)>> cmderMap {
        {'M', MoveCommand()},           // 前进
        {'L', TurnLeftCommand()},       // 左转
        {'R', TurnRightCommand()},      // 右转
        {'F', FastCommand()},           // 快速
    };
    // cmderMap.emplace('M', MoveCommand());       // 前进
    // cmderMap.emplace('L', TurnLeftCommand());   // 左转
    // cmderMap.emplace('R', TurnRightCommand());  // 右转
    // cmderMap.emplace('F', FastCommand());       // 快速

    for(const auto cmd : commands){
        // 根据操作查找表驱动
        const auto it = cmderMap.find(cmd);
        // 如果找到表驱动，执行操作对应的指令
        if (it != cmderMap.end()){
            it->second(poseHandler);
```

编译运行验证后，代码及时入库

git add .

git commit –m “initialization semantics to make the cmderMap init code more concise”

git branch lab3-fp　　# 代码修改完毕，保存新分支

目录

1. 实验2回顾
2. 面向对象项目实践
   - 2.1 数据抽离解耦循环依赖
   - 2.2 表驱动提升扩展性
   - 2.3 状态抽象提升可读性
3. 函数式项目实践
   - 2.1 函数式编程
   - 2.2 Lambda对象生命周期安全
   - 2.3 操作符重载
4. 总结

本章总结

通过本课程的学习：

- 提升代码可扩展性：利用表驱动方法减少复杂循环，结合多态特性提升代码灵活性和可扩展性
- 解耦和分层策略：通过数据抽离和代码分层策略，解耦循环依赖，显著提升代码的可扩展性和可维护性
- 状态抽象和计算属性：通过状态抽象，简化复杂状态流转问题，将指令数字化处理，提高代码的可维护性和可扩展性
- 测试用例和降维分析：解析多维降维分析方法，设计测试防护网，确保代码的可靠性和稳定性
- 函数式编程：掌握C++新特性带来的代码简洁性收益，写出简洁优雅的代码

同时关注了使用过程中编程陷阱可能导致的问题，确保C++对象生命周期安全。

这些技能和知识将为您在未来的企业软件开发中打下坚实的基础，帮助您成为一名更加专业和高效的软件开发者。

感谢您参与本课程，期待您在未来的软件开发工作中取得更大的成就！

作业

- **状态变化简化**：通过状态变化，简化PoseHandler中的代码圈复杂度，实现优雅的代码结构
- **操作符()重载**：完成function操作符()的重载优化，提升代码效率和可读性
- **实现B指令需求**：包括测试用例

B&F指令正交分解的测试用例

状态 指令	B	F	BF	BB	FF
M	当前朝向E 执行BM X−1	当前朝向E 执行FM X+2	当前朝向E 执行BFM X−2	当前朝向N 执行BBM Y+1	当前朝向N 执行FFM Y+1
L	当前朝向E 执行BL 朝向S	当前朝向E 执行FL X+1，朝向N	当前朝向E 执行BFL X−1，朝向S	NA	NA
R	当前朝向E 执行BR 朝向N	当前朝向E 执行FR X+1，朝向S	当前朝向E 执行BFR X−1，朝向N	NA	NA

1. 首先实现测试代码tests/ExecutorReverseTest.cpp

```cpp
#include <gtest/gtest.h>
#include "Executor.hpp"
#include "PoseEq.hpp"

namespace adas{
    // 测试输入：FM
    TEST(ExecutorFastTest, should_return_x_plus_2_given_status_is_fast_command_is_M_and_facing_is_E){
        // given
        std::unique_ptr<Executor> executor(Executor::NewExecutor({0, 0, 'E'}));

        // when
        executor->Execute("FM");

        // then
        const Pose target{2, 0, 'E'};
        ASSERT_EQ(target, executor->Query());
    }

    // 测试输入：FL
    TEST(ExecutorFastTest, should_return_N_and_x_plus_1_given_status_is_fast_command_is_L_and_facing_is_E){…

    // 测试输入：FR
    TEST(ExecutorFastTest, should_return_S_and_x_plus_1_given_status_is_fast_given_command_is_R_and_facing_is_E){…

    // 测试输入：FFM
    TEST(ExecutorFastTest, should_return_y_plus_1_given_command_is_FFM_and_facing_is_N){…
}
```

2. 然后实现测试代码tests/ExecutorReverseFastTest.cpp

```cpp
#include <gtest/gtest.h>

#include "Executor.hpp"
#include "PoseEq.hpp"

namespace adas {

    // 测试输入：FBM
    TEST(ExecutorReverseFastTest, should_return_x_minus_2_given_status_is_fast_and_reverse_command_is_M_and_facing_is_E) {
        // given
        std::unique_ptr<Executor> executor(Executor::NewExecutor({0, 0, 'E'}));

        // when
        executor->Execute("FBM");

        // then
        const Pose target{-2, 0, 'E'};
        ASSERT_EQ(target, executor->Query());
    }

    // 测试输入：FBL
    TEST(ExecutorReverseFastTest, should_return_S_and_x_minus_1_given_status_is_fast_command_is_L_and_facing_is_E) {…

    // 测试输入：FBR
    TEST(ExecutorReverseFastTest, should_return_N_and_x_minus_1_given_is_fast_and_reverse_given_command_is_R_and_facing_is_E) {

} // namespace adas
```

3.进行完代码改进后，新功能的添加就不需要添加大量冗长的if-else语句了，后退功能可以看作前进功能的逆向操作，因此可以针对M指令对应的前进操作进行修改

在实验二中为快速移动添加了变量fast，这里也为后退添加变量reverse

修改PoseHandler.hpp，添加变量reverse和相关函数，请自行实现

```
public:
    void Forward(void) noexcept;                // 向前移动
    void Backward(void) noexcept;               // 向后移动

    void TurnLeft(void) noexcept;               // 向左转
    void TurnRight(void) noexcept;              // 向右转

    void Fast(void) noexcept;                   // 切换模式
    void Reverse(void) noexcept;                // 切换倒车状态

    bool IsFast(void) const noexcept;           // 是否处于快速模式
    bool IsReverse(void) const noexcept;        // 是否处于倒车状态

    Pose Query(void) const noexcept;            // 查询当前位置

private:
    Point point;                    // 当前坐标
    const Direction* facing;        // 当前朝向
    bool fast{false};               // 是否处于快速状态
    bool reverse{false};            // 是否处于倒车状态
```

将原Move函数具体为Forward向前移动函数，添加Backward向后移动函数

添加变量reverse和相关函数

在实现Forward的时候用到了+=，因此可以针对坐标添加逆向操作-=

修改Point.hpp，添加+=对应的逆向操作-=，请自行修改Point.cpp

```
Point& operator+=(const Point& rhs) noexcept;   // 前进
Point& operator-=(const Point& rhs) noexcept;   // 后退
```

接下来修改ExecutorImpl.cpp中的Execute函数，为指令B添加状态操作ReverseCommand

```
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    // 表驱动
    std::unordered_map<char, std::function<void(PoseHandler& PoseHandler)>> cmderMap {
        {'M', MoveCommand()},           // 前进
        {'L', TurnLeftCommand()},       // 左转
        {'R', TurnRightCommand()},      // 右转
        {'F', FastCommand()},           // 快速
        {'B', ReverseCommand()},        // 后退
    };
```

由于ReverseCommand操作还没有实现，因此会报错

需要修改Command.hpp，添加ReverseCommand操作，同时需要修改之前的操作，以下为MoveCommand的修改示例，其余函数请大家自行实现

```
class MoveCommand final {
public:
    void operator()(PoseHandler& poseHandler) const noexcept {
        if (poseHandler.IsFast()) {
            if (poseHandler.IsReverse()) {
                poseHandler.Backward();
            } else {
                poseHandler.Forward();
            }
        }

        if (poseHandler.IsReverse()) {
            poseHandler.Backward();
        } else {
            poseHandler.Forward();
        }
    }
};
```

编译运行验证后，代码及时入库

```
git add .
git commit -m "test:support F&B commander"
git branch lab3-fp-support-B        # 代码修改完毕，保存新分支
```

学习推荐

在线参考资料网站，涵盖了C++基本概念到高级特性、标准库函数、类和模板等各个方面：

- 搜索功能：有强大的搜索功能，可以快速找到需要的函数、类或概念
- 示例代码：可以帮助理解如何使用C++特性
- 网址：https://en.cppreference.com/w/

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and organization for a fully connected, intelligent world.

HUAWEI