# 实验二StepbyStep

# 实验一已经完成，在git里面创建一个分支作为实验一的里程碑

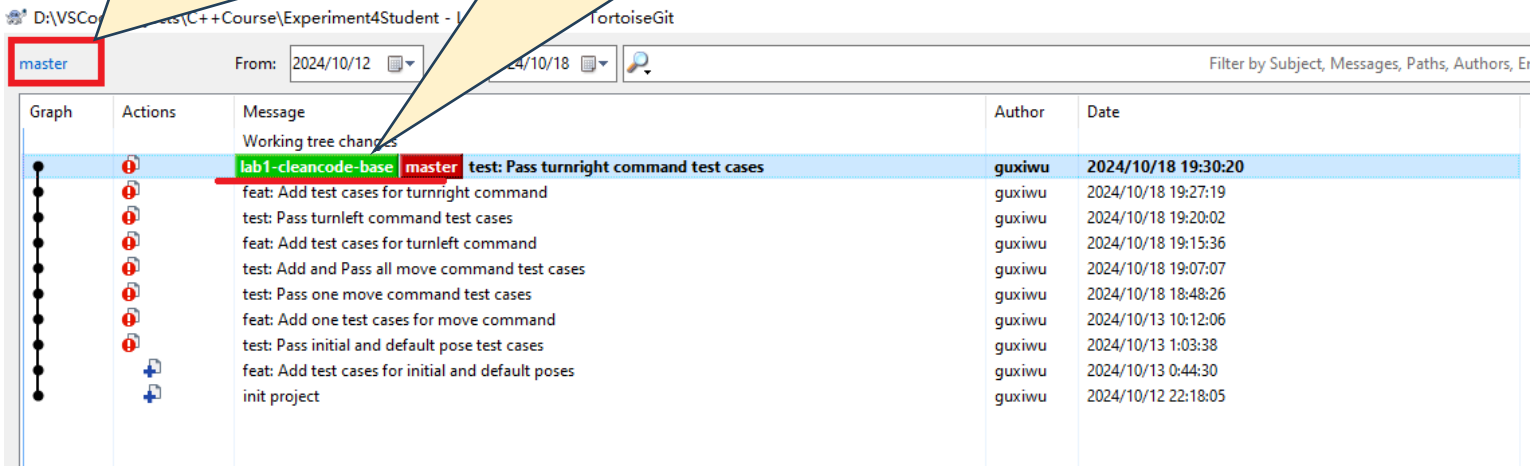
**打开Terminal，输入下面命令创建分支**

**git branch lab1-cleancode-base**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

[----------] Global test environment tear-down
[==========] 14 tests from 1 test suite ran. (312 ms total)
[  PASSED  ] 14 tests.
PS D:\VSCodeProjects\C++Course\Experiment4Student> git commit -m "test: Pass turnright command test cases"
[master 38849cf] test: Pass turnright command test cases
 1 file changed, 6 insertions(+)
PS D:\VSCodeProjects\C++Course\Experiment4Student>
PS D:\VSCodeProjects\C++Course\Experiment4Student>
PS D:\VSCodeProjects\C++Course\Experiment4Student>
PS D:\VSCodeProjects\C++Course\Experiment4Student> git branch lab1-cleancode-base
PS D:\VSCodeProjects\C++Course\Experiment4Student>
```

当前是查看的master分支，点击master可以显示所有分支

可以看到绿色的lab1-cleancode-base，这就是分支

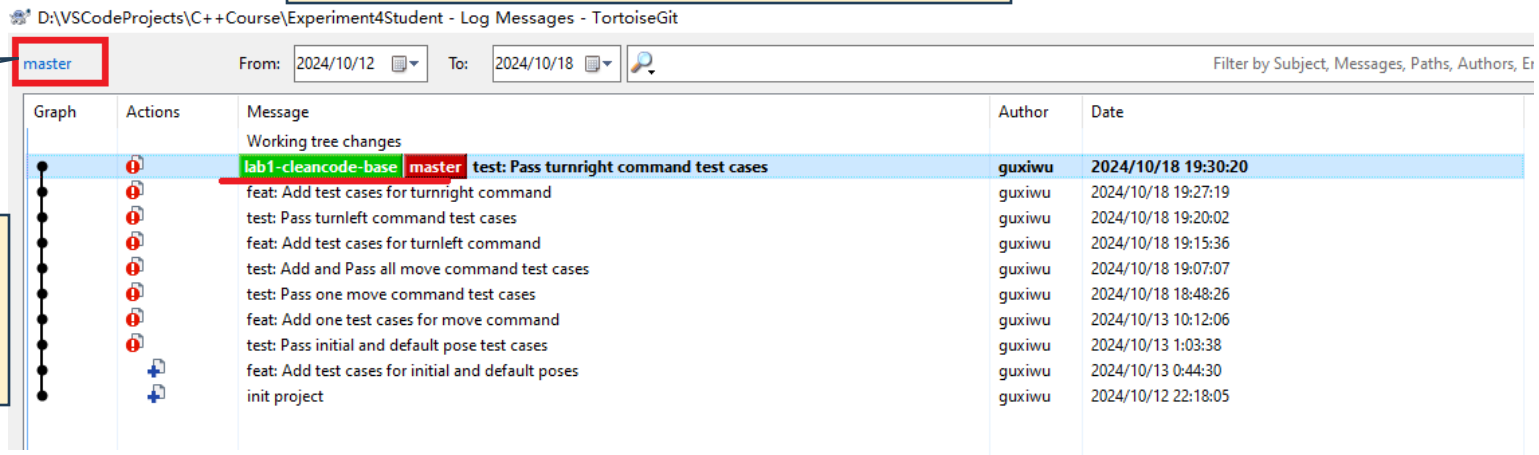



HUAWEI

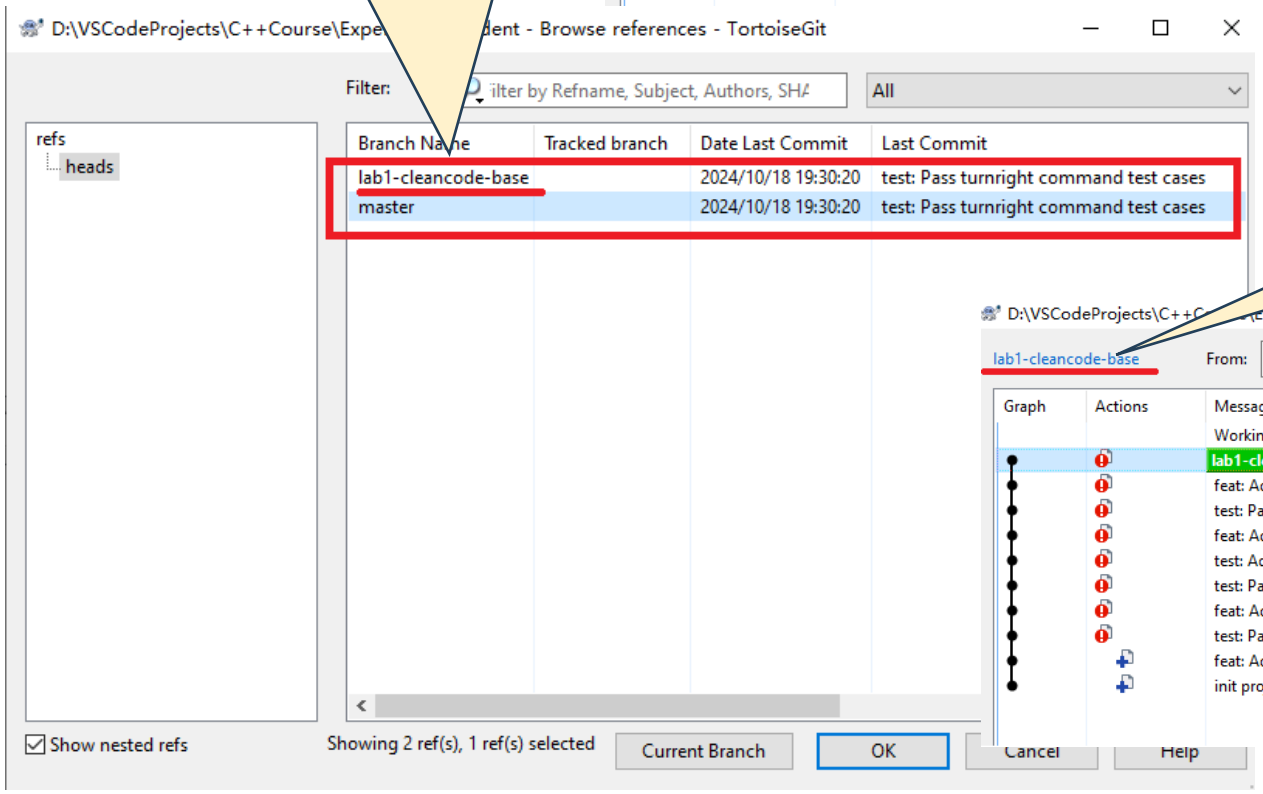
# 查看不同分支的log

在进行实验2时，请确保位于master分支

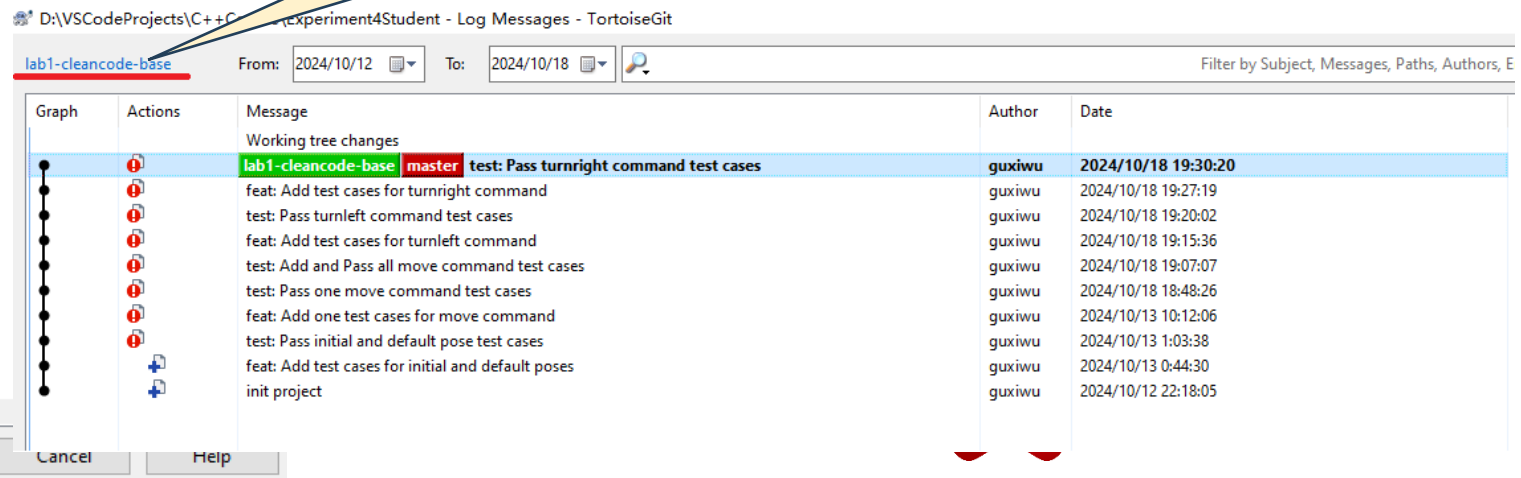
点击master可以查看所有分支




比如点击lab1-cleancode-base, 再点击ok按钮，就可以查看分支lab1-cleancode-base的提交日志




查看lab1-cleancode-base分支的提交日志



# 查看当前位于哪个分支，切换分支

在进行实验2时，请确保位于master分支。这样提交的代码才能位于master分支下面

在进行实验2时，请确保位于master分支。

输入下面的命名可以查看当前位于什么分支

**git branch**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Active code page: 65001
PS D:\VSCodeProjects\C++Course\Experiment4Student> git branch
  lab1-cleancode-base
* master
PS D:\VSCodeProjects\C++Course\Experiment4Student>
```

git branch命令会显示当前所有分支列表，其中带*的表示当前所处于的分支，例如master

输入下面的命名可以切换分支

**git checkout lab1-cleancode-base**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Active code page: 65001
PS D:\VSCodeProjects\C++Course\Experiment4Student> git branch
  lab1-cleancode-base
* master
PS D:\VSCodeProjects\C++Course\Experiment4Student> git checkout lab1-cleancode-base
Switched to branch 'lab1-cleancode-base'
PS D:\VSCodeProjects\C++Course\Experiment4Student> git branch
* lab1-cleancode-base
  master
PS D:\VSCodeProjects\C++Course\Experiment4Student> git checkout master
Switched to branch 'master'
PS D:\VSCodeProjects\C++Course\Experiment4Student> git branch
  lab1-cleancode-base
* master
PS D:\VSCodeProjects\C++Course\Experiment4Student>
```

当输入git checkout lab1-cleancode-base切换到了lab1-cleancode-base分支

当输入git checkout master切换到了master分支

# C++企业软件开发实践

## 实验2-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2，您将实践面向对象编程的基础知识，包括：

- 封装：掌握面向对象编程的基本概念，包括类和对象的封装，以及如何通过封装实现代码的复用性
- 继承与多态：掌握面向对象中，继承、多态等特性，实现代码的良好扩展性

期待您在课程中的精彩表现！

# 目标

通过本课程的学习，您将能够：

- 深入理解面向对象编程的三大特性（封装、继承、多态）的概念
- 实践并提升代码的可扩展性，充分利用面向对象编程的优势
- 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

# 目录

# 实验1回顾：左转指令测试用例设计

```
// L
TEST(ExecutorTest, should_return_facing_N_given_command_is_L_and_facing_is_E)
{
    // given
    std::unique_ptr<Executor> executor(Executor::NewExecutor({0, 0, 'E'}));

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

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

TEST(ExecutorTest, should_return_facing_W_given_command_is_L_and_facing_is_N)
{
    // given
    std::unique_ptr<Executor> executor(Executor::NewExecutor({0, 0, 'N'}));

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

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

```
TEST(ExecutorTest, should_return_facing_S_given_command_is_L_and_facing_is_W)
{
    // given
    std::unique_ptr<Executor> executor(Executor::NewExecutor({0, 0, 'W'}));

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

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

TEST(ExecutorTest, should_return_facing_E_given_command_is_L_and_facing_is_S)
{
    // given
    std::unique_ptr<Executor> executor(Executor::NewExecutor({0, 0, 'S'}));

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

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

# 实验1回顾：右转指令测试用例设计

```
// R
TEST(ExecutorTest, should_return_facing_S_given_command_is_R_and_facing_is_E)
{
    // given
    std::unique_ptr<Executor> executor(Executor::NewExecutor({0, 0, 'E'}));

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

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

TEST(ExecutorTest, should_return_facing_W_given_command_is_R_and_facing_is_S)
{
    // given
    std::unique_ptr<Executor> executor(Executor::NewExecutor({0, 0, 'S'}));

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

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

```
TEST(ExecutorTest, should_return_facing_N_given_command_is_R_and_facing_is_W)
{
    // given
    std::unique_ptr<Executor> executor(Executor::NewExecutor({0, 0, 'W'}));

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

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

TEST(ExecutorTest, should_return_facing_E_given_command_is_R_and_facing_is_N)
{
    // given
    std::unique_ptr<Executor> executor(Executor::NewExecutor({0, 0, 'N'}));

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

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

# 实验1回顾：功能代码实现

```
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    for(const auto cmd:commands){
        if(cmd == 'M'){
            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; }
        }
        else if (cmd == 'L') {
            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';}
        }
        else if (cmd == 'R') {
            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';}
        }
    }
}
```

# 目录

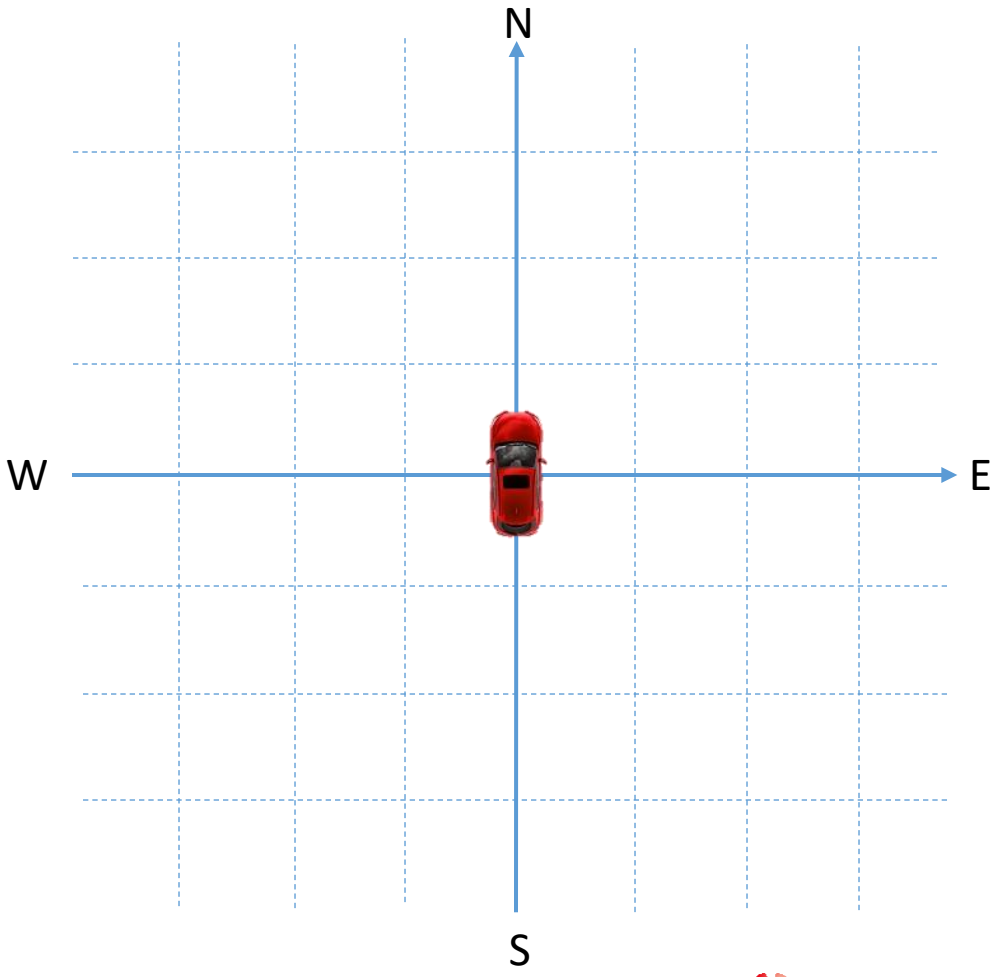
# 课程实训需求2-1 支持加速指令

**Executor组件增加支持执行：**

**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**




指令序列：　F M L M L M

当前位置：

当前朝向：

# F指令实现

请大家实现，F指令的功能代码（思考10分钟）

正常情况下先写测试用例，现在假设测试已经构建，如何实现功能代码?

# F指令实现

首先需要在ExecutorImpl添加一个实例数据成员，记录当前是不是位于加速状态

ExecutorImpl.hpp M

src > ExecutorImpl.hpp > {} adas > ExecutorImpl > isFast

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

namespace adas{
    /*
        Executor的具体实现
    */
    class ExecutorImpl: public Executor{
    public:
        //构造函数
        explicit ExecutorImpl(const Pose& pose) noexcept;
        //默认析构函数
        ~ExecutorImpl() noexcept = default;

        //不能拷贝
        ExecutorImpl(const ExecutorImpl &) = delete;
        //不能赋值
        ExecutorImpl &operator=(const ExecutorImpl &) = delete;

    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 overrid
    private:
        //私有数据成员，汽车当前姿态
        Pose pose;
        //是否处于加速状态，默认是false
        bool isFast;
    };
}
```

ExecutorImpl.cpp M

src > ExecutorImpl.cpp > {} adas

```cpp
#include "ExecutorImpl.hpp"

#include <new>

namespace adas{
    ExecutorImpl::ExecutorImpl(const Pose &pose) noexcept :pose(pose) ,isFast(false) {}

    Pose ExecutorImpl::Query(void) const noexcept{
        return pose;
    }

    /*
        std::nothrow 是 C++ 标准库中的一个常量，用于指示在分配内存时不抛出任何异常。
        它是 std::nothrow_t 类型的实例，通常用在 new 运算符和 std::nothrow 命名空间中，
        以请求内存分配器在分配失败时返回一个空指针，而不是抛出 std::bad_alloc 异常。
    */
    Executor *Executor::NewExecutor(const Pose &pose) noexcept{
        return new(std::nothrow) ExecutorImpl(pose);  //只在C++17下有效
    }

    void ExecutorImpl::Execute(const std::string &commands) noexcept {…
}
```

# F指令实现

其次需要修改ExecutorImpl::Execute

其次要修改M、L、R指令的处理逻辑：在里面需要添加条件判断是否处于加速状态

圈复杂度高，代码重复　　如何优化代码?

首先要添加处理F指令的条件分支逻辑

```
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    for(const auto cmd:commands){
        if(cmd == 'M'){
            if(!isFast){ //如果不是处于加速状态，和以前一样
                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; }
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'L') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'R') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if(cmd == 'F'){
            isFast = !isFast; //每次收到F指令，切换isFast
        }
    }
}
```

# F指令实现

其次需要修改ExecutorImpl::Execute

圈复杂度高，代码重复　如何优化代码?

代码就更复杂了，需求的迭代，对代码的扩展性提出了强烈的诉求

代码圈复杂度（Cyclomatic Complexity）是用来衡量代码复杂性的一种指标。简单来说，它表示代码中有多少条不同的执行路径。路径越多，代码就越复杂。想象一下，你在一个迷宫里，每次遇到一个岔路口（比如 if 语句或 for 循环），你就有了一个新的选择。代码圈复杂度就是计算这些岔路口的数量。数值越高，说明迷宫越复杂，走出去的难度也越大。

**为什么重要?**

- **可维护性**：复杂的代码更难理解和修改。
- **测试难度**：复杂的代码需要更多的测试用例来覆盖所有可能的路径。
- **错误风险**：复杂的代码更容易出错。

```
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    for(const auto cmd:commands){
        if(cmd == 'M'){
            if(!isFast){ //如果不是处于加速状态，和以前一样
                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; }
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'L') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'R') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if(cmd == 'F'){
            isFast = !isFast; //每次收到F指令，切换isFast
        }
    }
}
```

# 目录

# 面向对象编程-代码分析及优化思路

```
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    for(const auto cmd:commands){
        if(cmd == 'M'){
            if(!isFast){ //如果不是处于加速状态，和以前一样
                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; }
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'L') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'R') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if(cmd == 'F'){
            isFast = !isFast; //每次收到F指令，切换isFast
        }
    }
}
```

我们要把容易互相影响的、关联程度紧密的元素，都封装在一个类内部（而这正是我们老生常谈的**封装变化**的动机）；同时让类之间的关联紧密程度尽可能降低，以让类间尽可能不要相互影响。从而最终做到**局部化影响**。

首先进入我们射程的就是**重复代码**。编写重复代码不仅仅会让有追求的程序员感到乏味。真正致命的是：“重复”极度违背**高内聚**、**低耦合**原则，从而会大幅提升软件的长期维护成本。因而，对于**完全重复**的代码进行消除，合二为一，会让系统更加**高内聚**、**低耦合**。

我们暂时先不考虑F指令和加速状态，先优化代码，摆脱冗余无聊的if语句

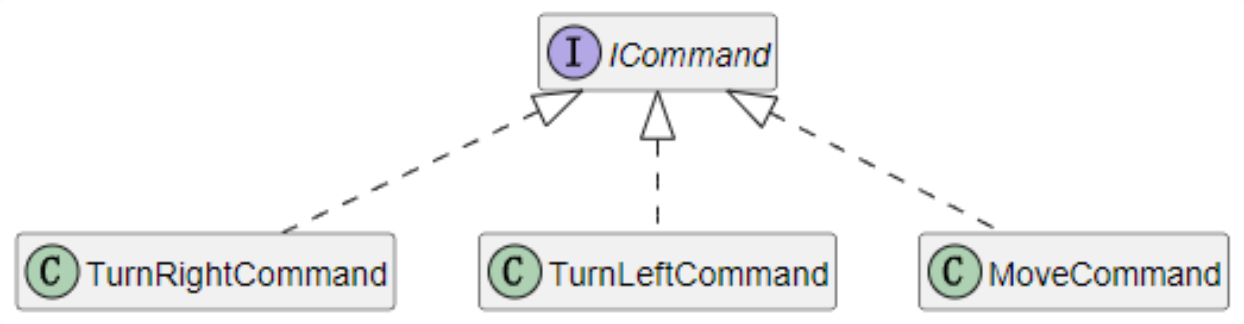
HUAWEI

# 面向对象编程-代码分析及优化思路

```
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    for(const auto cmd:commands){
        if(cmd == 'M'){
            if(!isFast){ //如果不是处于加速状态，和以前一样
                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; }
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'L') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'R') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if(cmd == 'F'){
            isFast = !isFast; //每次收到F指令，切换isFast
        }
    }
}
```

**1.指令处理划分：**按照指令处理逻辑的不同，首先将M/L/R 3个指令处理的逻辑抽取出Move、TurnLeft、TurnRight三个成员函数

**2.抽象，消减重复代码：**为了提高代码的可维护性和扩展性，我们可以使用面向对象的封装，将这三个方法统一到一个基类Icommand的 DoOperate 抽象方法中。通过创建 MoveCommand、TurnLeftCommand 和 TurnRightCommand 的子类，并在每个子类的Operate实现具体的操作，我们可以利用多态性来简化代码结构。




HUAWEI

# 面向对象编程-移动指令行为抽取为Move方法

ExecutorImpl.hpp

```
class ExecutorImpl final : public Executor
{
...

private:
   void Move(void) noexcept;
...
```

添加私有的Move方法

编译运行验证后（还是用实验一测试M指令的用例），代码及时入库:（10分钟）

```
git add .

git commit -m " extract Move() "
```

ExecutorImpl.cpp

```
void ExecutorImpl::Execute(const std::string& commands) noexcept
{
   for (const auto cmd : commands) {
      if (cmd == 'M') {
         Move();
      }
...
void ExecutorImpl::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;
   }
}
```

这里调用Move

无聊的if语句现在都被封装在Move方法里了
局部化了

# 面向对象编程-练习时间：移动/左转/右转行为抽取

- 参考Move成员函数的抽取，完成Move/TurnLeft/TurnRight3个成员函数抽取
- 每个成员函数抽取完成后，必须进行编译、运行验证，并将代码提交到库中

```
抽取出TurnLeft后，编译、运行验证正确后提交一次
git add .
git commit -m "extract TurnLeft()"
```

```
抽取出TurnRight后，编译、运行验证正确后提交一次
git add .
git commit -m "extract TurnRight ()"
```

# 面向对象编程-代码分析及优化思路

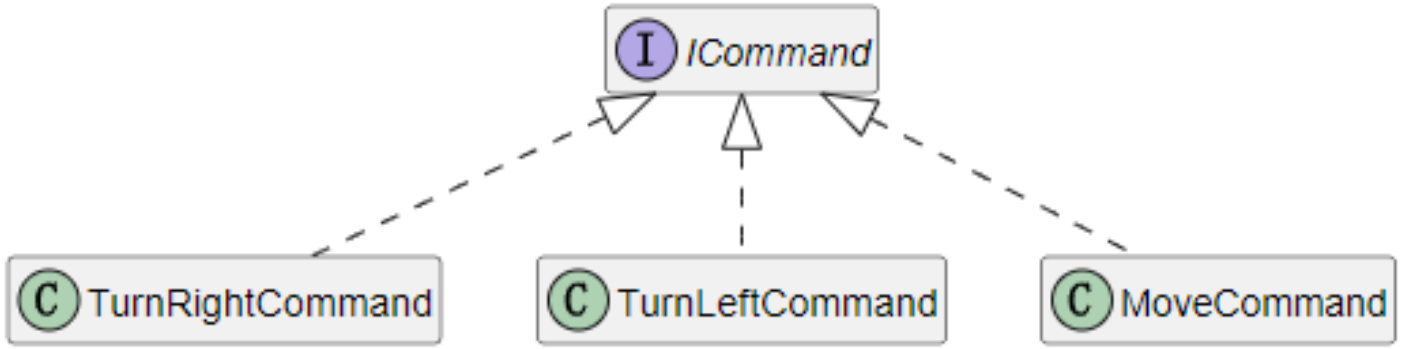
```
private:
    void Move(void) noexcept;
    void TurnLeft(void) noexcept;
    void TurnRight(void) noexcept;
```

输入输出完全一致，唯一的区别是方法名不同

```
void ExecutorImpl::Execute(const
std::string& commands) noexcept
{
    for (const auto cmd : commands) {
        if (cmd == 'M') {
            Move();
        } else if (cmd == 'L') {
            TurnLeft();
        } else if (cmd == 'R') {
            TurnRight();
        }
    }
}
```

为了提高代码的可维护性和扩展性，我们可以使用面向对象的封装，将这三个方法统一到一个基类Icommand的 DoOperate 抽象方法中。

通过创建 MoveCommand、TurnLeftCommand 和 TurnRightCommand 的子类，并在每个子类的DoOperate实现具体的操作，我们可以利用多态性来简化代码结构。



# 面向对象编程-Move方法封装到MoveCommand类中

```
void ExecutorImpl::Execute(const
std::string& commands) noexcept
{
    for (const auto cmd : commands) {
        if (cmd == 'M') {
            Move();
        } else if (cmd == 'L') {
            TurnLeft();
        } else if (cmd == 'R') {
            TurnRight();
        }
    }
}
```

```
class ExecutorImpl final : public Executor
{
...
private:
    class MoveCommand final //定义一个嵌套类MoveCommand，完成Move动作（M指令）
    {
    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
        {
            executor.Move();
        }
    };
```

```
#include <memory>
...
void ExecutorImpl::Execute(const std::string& commands) noexcept
{
    for (const auto cmd : commands) {
        if (cmd == 'M') {
            //智能指针指向MoveCommand实例，不用担心delete了
            std::unique_ptr<MoveCommand> cmder = std::make_unique<MoveCommand>();
            //*this就是ExecutorImpl实例对象，作为实参传递给DoOperate方法
            cmder->DoOperate(*this); //执行MoveCommand的DoOperate，即Move
        }
```

编译运行验证后，代码及时入库

```
git add .
git commit -m "extract MoveCommand"
```

修改的意义：每个指令的执行，都是用同样的语句：
cmder->DoOperate(*this); 是不是和多态很接近了？

# 面向对象编程-练习时间，Move/TurnLeft/TurnRight封装到类中

- 类似MoveCommand

再分别将TurnLeft方法和TurnRight方法封装到TurnLeftCommand和TurnRightCommand类里

- 每个成员函数抽取完成后，必须进行编译、运行验证，并将代码提交到库中
- 练习时间：10分钟

编译运行验证后，代码及时入库

```
git add .
git commit -m "extract TurnLeftCommand"
```

编译运行验证后，代码及时入库

```
git add .
git commit -m "extract TurnRightCommand"
```

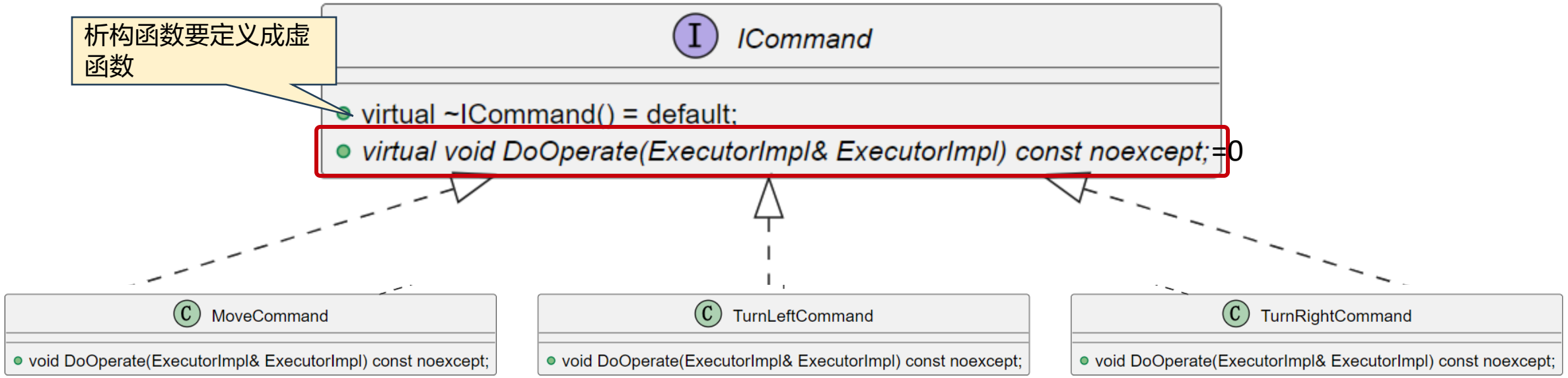
# 本节小结

**演示从面向过程代码到面向对象代码的转变：**

- **封装和复用：**
  - 展示如何通过封装提高代码的复用性，降低圈复杂度
  - 通过具体示例展示封装的实现方法

# 目录

# 面向对象编程：接口抽象、继承与多态关系设计




- 第一步封装，已完成：将指令处理的三个方法封装到MoveCommand、TurnLeftCommand和TurnRightCommand 三个类中，每个类的DoOperate方法实现具体操作，行为抽象一致
- 第二步接口抽象：通过定义抽象类ICommand的抽象方法DoOperate；
- 第三步继承与多态：建立MoveCommand、TurnLeftCommand和 TurnRightCommand的继承关系，简化代码结构。

# 面向对象编程-接口抽象，虚基类ICommand建立

ExecutorImp.hpp

```
class ExecutorImpl final : public Executor
{
...
private:
   class ICommand
   {
   public:
      //请在这里给出析构函数和纯虚函数DoOperate的声明
   };
...
```

再添加一个私有的抽象类ICommand

# 面向对象编程-指令继承关系建立

ExecutorImp.hpp

```
class ExecutorImpl final : public Executor
...
   class MoveCommand final : public ICommand
   {
   public:
      void DoOperate(ExecutorImpl& executor) const noexcept override   //给出具体实现
...

   class TurnLeftCommand final : public ICommand
   {
   public:
      void DoOperate(ExecutorImpl& executor) const noexcept override          //给出具体实现
...

   class TurnRightCommand final : public ICommand
   {
   public:
      void DoOperate(ExecutorImpl& executor) const noexcept override          //给出具体实现
...
```

现在三个具体的Command类都继承ICommand

现在三个具体的Command类都继承ICommand

现在三个具体的Command类都继承ICommand

# 面向对象编程-重复代码消减，简化代码

```
void ExecutorImpl::Execute(const std::string& commands)
noexcept
{
   for (const auto cmd : commands) {
      if (cmd == 'M') {
         std::unique_ptr<ICommand> cmder =
std::make_unique<MoveCommand>();
         cmder->DoOperate(*this);
      } else if (cmd == 'L') {
         std::unique_ptr<ICommand> cmder =
std::make_unique<TurnLeftCommand>();
         cmder->DoOperate(*this);
      } else if (cmd == 'R') {
         std::unique_ptr<ICommand> cmder =
std::make_unique<TurnRightCommand>();
         cmder->DoOperate(*this);
      }
   }
}
```

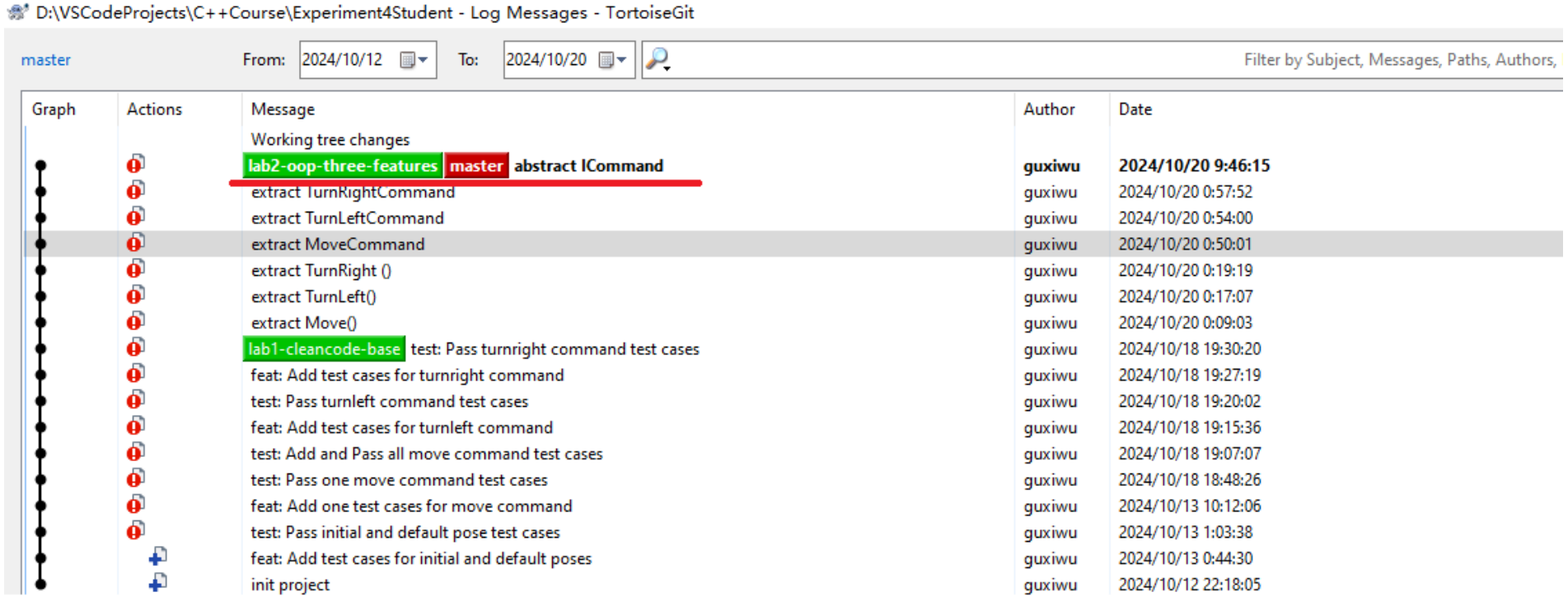
```
void ExecutorImpl::Execute(const std::string& commands)
noexcept
{
   for (const auto cmd : commands) {
      //声明一个ICommand类型的智能指针
      std::unique_ptr<ICommand> cmder;

      if (cmd == 'M') {
         //智能指针指向子类MoveCommand实例
         cmder = std::make_unique<MoveCommand>();
      } else if (cmd == 'L') {
         //请自己给出代码
      } else if (cmd == 'R') {
         //请自己给出代码
      }

      if (cmder) {
         //请自己给出代码      }
   }
```

编译运行验证后，代码及时入库

```
git add .
git commit -m "abstract ICommand "
git branch lab2-oop-three-features  #这是实验2的一个里程碑，所以再创建一个分支
```

# 实验2创建的lab2-oop-three-features 分支

编译运行验证后，代码及时入库
git add .
git commit -m "abstract ICommand "
git branch lab2-oop-three-features  #这是实验2的一个里程碑，所以再创建一个分支




继续在master分支下完成接下来的实验

# 面向对象编程重构后，F指令的实现

1. 设计F指令的测试用例，确保需求实现的完整性；
2. 创建F指令处理类FastCommand，继承自ICommand；
3. MoveCommand、TurnLeftCommand、TurnRightCommand支持F指令状态。

# F指令的用例设计

请大家思考10分钟，按照实验1开发者测试相关的原则和设计方法，设计出F指令的测试用例。

正交分解通常用于二维数据分析。在支持控制指令MLR的需求中，正交分解的维度包括指令和方向。现在引入了状态指令F，因此数据维度增加到三个。为了对三维数据进行正交分解，需要先降维，将三维数据降为二维后再进行分析。

一种常用的降维方法是将两个维度的数据进行组合。例如，将指令MLR和方向ESWN组合成一维：EM、ER、EL、SM、SR、SL、WM、WR、WL、NM、NR、NL。然后再与F状态进行正交分解，这样可以得到24个用例。

需要注意的是，F状态的影响仅对MLR指令的执行产生变化，与车辆当前朝向无关。因此，方向维度可以忽略，只需对控制指令MLR和F状态进行正交分解即可。

# F指令用例设计

请大家根据F指令正交分解的测试用例设计，以及实验1命名规范的实践，实现F指令的4个测试用例（15分钟）：

| 状态 / 指令 | F | FF |
| --- | --- | --- |
| M | 当前朝向E<br>执行FM<br>X+1, X+1 | 当前朝向N<br>执行FFM<br>Y+1 |
| L | 当前朝向E<br>执行FL<br>X+1,朝向N | NA |
| R | 当前朝向E<br>执行FR<br>X+1,朝向S | NA |

should_return_x_plus_2_given_status_is_fast_command_is_M_and_facing_is_E

should_return_N_and_x_plus_1_given_status_is_fast_command_is_L_and_facing_is_E

should_return_S_and_x_plus_1_given_status_is_fast_given_command_is_R_and_facing_is_E

should_return_y_plus_1_given_command_is_FFM_and_facing_is_N

# F指令用例设计：F状态下执行移动指令

创建tests/ExecutorFastTest.cpp，在此实现先实现第1个F指令的测试用例

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

#include "Executor.hpp"

namespace adas
{
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");  //FM: F状态下Move

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

# F指令用例设计：F状态下执行移动指令

创建tests/ExecutorFastTest.cpp，在此实现先实现第1个F指令的测试用例

但是执行.\script\build.bat后编译器报错

```
In file included from D:\VSCodeProjects\C++Course\Experiment4Student\tests\ExecutorFastTest.cpp:1:
D:/VSCodeProjects/C++Course/Experiment4Student/tests/googletest/googletest/include/gtest/gtest.h: In instantiation of 'testing::AssertionResult testing::internal::CmpHelperEQ(const char*, const char*, co
st T1&, const T2&) [with T1 = adas::Pose; T2 = adas::Pose]':
D:/VSCodeProjects/C++Course/Experiment4Student/tests/googletest/googletest/include/gtest/gtest.h:1377:23:   required from 'static testing::AssertionResult testing::internal::EqHelper::Compare(const char*
 const char*, const T1&, const T2&) [with T1 = adas::Pose; T2 = adas::Pose; typename std::enable_if<(! std::is_integral<_Tp>::value) || (! std::is_pointer<_Dp>::value))>::type* <anonymous> = 0]'
D:\VSCodeProjects\C++Course\Experiment4Student\tests\ExecutorFastTest.cpp:14:9:   required from here
D:/VSCodeProjects/C++Course/Experiment4Student/tests/googletest/googletest/include/gtest/gtest.h:1358:11: error: no match for 'operator==' (operand types are 'const adas::Pose' and 'const adas::Pose')
   if (lhs == rhs) {
       ~~~^~~~~~~~
D:/VSCodeProjects/C++Course/Experiment4Student/tests/googletest/googletest/include/gtest/gtest.h:1350:13: note: candidate: 'bool testing::internal::operator==(testing::internal::faketype, testing::interna
l::faketype)'
 inline bool operator==(faketype, faketype) { return true; }
             ^~~~~~~~
D:/VSCodeProjects/C++Course/Experiment4Student/tests/googletest/googletest/include/gtest/gtest.h:1350:13: note:   no known conversion for argument 1 from 'const adas::Pose' to 'testing::internal::faketype
'

mingw32-make.exe[2]: *** [tests\CMakeFiles\Experiment4Student-main.dir\build.make:76: tests/CMakeFiles/Experiment4Student-main.dir/ExecutorFastTest.cpp.obj] Error 1
mingw32-make.exe[1]: *** [CMakeFiles\Makefile2:176: tests/CMakeFiles/Experiment4Student-main.dir/all] Error 2
mingw32-make.exe: *** [Makefile:145: all] Error 2
PS D:\VSCodeProjects\C++Course\Experiment4Student>
```

原因是比较二个Pose是否相等是实现在ExecutorTest.cpp里， ExecutorFastTest.cpp找不到这个方法的实现，因此需要将比较二个Pose是否相等的方法单独抽取出来了

# F指令用例设计：编译问题解决，Pose相等重载独立到文件

Pose ==操作符重载从ExecutorTest.cpp中独立出来：

创建tests/PoseEq.hpp

```
#pragma once
#include "Executor.hpp"

namespace adas
{
bool operator==(const  Pose& lhs, const Pose& rhs);
}
```

创建tests/PoseEq.cpp

```
//这里请自己给出实现（参考ExecutorTest.cpp 的实现）
```

# F指令用例设计：编译问题解决，各测试用例引用Pose重载封装头文件

ExecutorTest.cpp

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

#include <memory>
// #include <tuple>

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

namespace adas
{
// bool operator==(const Pose& lhs, const Pose& rhs)
// {
//     return std::tie(lhs.x, lhs.y, lhs.heading) == std::tie(rhs.x, rhs.y, rhs.heading);
// }
```

ExecutorFastTest.cpp

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

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

现在ExecutorFastTest.cpp里先实现的第1个F指令的测试用例可以编译通过了

# F指令用例设计：继续实现F状态下执行转向指令

```
TEST(ExecutorFastTest, should_return_N_and_x_plus_1_given_status_is_fast_command_is_L_and_facing_is_E)
{
    //请自己给出实现，命令是FL，起始状态{0,0,'E'}
}

TEST(ExecutorFastTest, should_return_S_and_x_plus_1_given_status_is_fast_given_command_is_R_and_facing_is_E)
{
    //请自己给出实现，命令是FR，起始状态{0,0,'E'}
}
```

# F指令用例设计：再一次收到F指令，状态取消

```
TEST(ExecutorFastTest, should_return_y_plus_1_given_command_is_FFM_and_facing_is_N)
{
    // given
    std::unique_ptr<Executor> executor(Executor::NewExecutor()); //默认起始状态是{0,0,'N'}

    // when
    executor->Execute("FFM");                    //FFM等价于M
    // then
    const Pose target{0, 1, 'N'};
    ASSERT_EQ(target, executor->Query());
}
} // namespace adas
```

编译运行验证后，代码及时入库
git add .
git commit −m "feat: Add test cases for fast command"

# F指令功能代码实现： ExecutorImpl.hpp添加F指令接口支持及状态保存

```
class ExecutorImpl final : public Executor
{
...
private:
    ...
    void TurnRight(void) noexcept;
    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


private:
    Pose pose;
    bool fast{false};          //记录是否处于加速状态

};
```

# 请在ExecutorImpl.cpp给出这二个方法的实现

# F指令功能代码实现：FastCommand类建立

请自己给出FastCommand类的实现

请自己修改ExecutorImpl，使得可以执行F指令

# F指令功能代码实现：修改MoveCommand支持F指令状态

```
class MoveCommand final : public ICommand
{
public:
   void DoOperate(ExecutorImpl& executor) const noexcept override
   {
     //如果是F状态，多执行一次MOVE，该怎么修改
       executor.Move();
   }
};
```

参考MoveCommand，实现TurnLeftCommand、TurnRightCommand支持F指令状态

编译运行验证后，代码及时入库（应该是一个18个测试用例全部通过）
git add .
git commit -m "test:Pass test cases for fast command"
git branch lab2-oop-support-F

# F指令功能代码实现：面向对象优化后，扩展性的直观收益

```cpp
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    for(const auto cmd:commands){
        if(cmd == 'M'){
            if(!isFast){ //如果不是处于加速状态，和以前一样
                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; }
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'L') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'R') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if(cmd == 'F'){
            isFast = !isFast; //每次收到F指令，切换isFast
        }
    }
}
```

```cpp
class ExecutorImpl: public Executor{
    void TurnLeft(void) noexcept;
    void TurnRight(void) noexcept;
private:
    //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;
    };
    //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{…
};
```

```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'){…

        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);
        }
    }
}
```

HUAWEI

# F指令功能代码实现：面向对象优化后，扩展性的直观收益

```cpp
void ExecutorImpl::Execute(const std::string &commands) noexcept {
    for(const auto cmd:commands){
        if(cmd == 'M'){
            if(!isFast){ //如果不是处于加速状态，和以前一样
                if(pose.heading == 'E')         {  ++pose.x; }
                else if(pose.heading == 'W')    {  --pose.x; }
                else if(pose.heading == 'S')    {  --pose.y; }
                else if(pose.heading == 'N')    {  ++pose.y; }
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
                if(pose.heading == 'E')         {  pose.x += 2; }
                else if(pose.heading == 'W')    {  pose.x -= 2; }
                else if(pose.heading == 'S')    {  pose.y -= 2; }
                else if(pose.heading == 'N')    {  pose.y += 2; }
            }
        }
        else if (cmd == 'L') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                if (pose.heading == 'E')        { pose.heading = 'N';}
                else if (pose.heading == 'S')   { pose.heading = 'E';}
                else if (pose.heading == 'W')   { pose.heading = 'S';}
                else if (pose.heading == 'N')   { pose.heading = 'W';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if (cmd == 'R') {
            if(!isFast){//如果不是处于加速状态，和以前一样
                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';}
            }
            else{
                //如果处于加速状态，则稍微有点不一样，代码省略
            }
        }
        else if(cmd == 'F'){
            isFast = !isFast; //每次收到F指令，切换isFast
        }
    }
}
```

```cpp
class ExecutorImpl: public Executor{
    void TurnLeft(void) noexcept;
    void TurnRight(void) noexcept;
private:
    //定义所有具体Command对象的抽象基类ICommand
    class ICommand{
```

把冗长无聊的if语句封装到命令对象里，每个命令的条件分支局部最小化

利用多态消除代码重复性，实现代码的良好扩展性，子类只需关注自己的业务即可

设计模式的原则：对修改是关闭的，对扩展是开放的。
左边设计每当增加新的指令，就要修改ExecutorImpl::Execute方法，对修改不是关闭的。
而采用右边的设计方法，每当增加新的指令，只需要在ExecutorImpl里面添加新指令的方法实现，同时添加新指令对应的命令对象，唯一需要修改的地方是需要在ExecutorImpl::Execute里加一个条件分支。虽然没有完全做到对修改关闭（继续优化代码是可以做到的），但起码修改是局部最小化了。

```cpp
        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'){…

        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);
        }
    }
}
```

# 本节小结

- 继承：
  - 演示如何通过继承来扩展现有类，避免代码重复
- 多态：
  - 通过逐步优化代码，展示如何利用多态消除代码重复性，实现代码的良好扩展性
- 多维因子下的正交分解：
  - 解析多维降维分析方法，展示如何将高维问题简化为低维问题

# 目录

# 本章总结

通过本课程的学习，您已经掌握了面向对象编程的特性：

- 封装：掌握面向对象编程的基本概念，包括类和对象的封装，以及如何通过封装实现代码的复用性
- 继承与多态：掌握面向对象中，继承、多态等特性，实现代码的良好扩展性

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！

# 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