面向对象程序设计实践（C++）报告

班级：2021211301 姓名：任飞 学号：2021210724 专业：计算机科学与技术

# 基础题目

## C++基础知识实验

本实验主要包括C++的基本语法、输入输出、动态内存申请等基础知识。

代码如下。

#include <iostream>  
const int row = 4, col = 5;  
  
void input(int \*mat) {  
 for (int i = 0; i < row; i++) {  
 for (int j = 0; j < col; j++) {  
 std::cin >> mat[i \* col + j];  
 }  
 }  
}  
  
void output(int \*mat) {  
 for (int i = 0; i < row; i++) {  
 std::cout << mat[i \* col];  
 for (int j = 1; j < col; j++) {  
 std::cout << " " << mat[i \* col + j];  
 }  
 std::cout << std::endl;  
 }  
}  
  
void add(int \*dest, int \*mat1, int \*mat2) {  
 for (int i = 0; i < row; i++) {  
 for (int j = 0; j < col; j++) {  
 dest[i \* col + j] = mat1[i \* col + j] + mat2[i \* col + j];  
 }  
 }  
}  
  
void sub(int \*dest, int \*mat1, int \*mat2) {  
 for (int i = 0; i < row; i++) {  
 for (int j = 0; j < col; j++) {  
 dest[i \* col + j] = mat1[i \* col + j] - mat2[i \* col + j];  
 }  
 }  
}  
  
int main() {  
 int \*A1 = new int[row \* col];  
 int \*A2 = new int[row \* col];  
 int \*A3 = new int[row \* col];  
 input(A1);  
 input(A2);  
 add(A3, A1, A2);  
 output(A3);  
 sub(A3, A1, A2);  
 output(A3);  
 delete[] A1;  
 delete[] A2;  
 delete[] A3;  
}

## 类与对象实验

### 第一部分

第一部分主要包括C++中类和对象的基础用法，以及对构造函数和析构函数的理解。

通过本实验可以直观地观察到，在本实验中，构造时会先调用Point的构造函数，再调用Circle的构造函数；析构时会先调用Circle的析构函数，再调用Point的析构函数。此外在值传递参数时可以构造了一个新的对象（拷贝构造函数）。

代码如下。

#include <cmath>  
#include <iostream>  
  
class Point {  
 public:  
 Point(double x = 0, double y = 0)  
 : x(x), y(y) {  
 std::cout << "Point constructor called." << std::endl;  
 }  
  
 ~Point() {  
 std::cout << "Point destructor called." << std::endl;  
 }  
  
 double dis(Point a) {  
 double dx = x - a.x;  
 double dy = y - a.y;  
 return std::sqrt(dx \* dx + dy \* dy);  
 }  
  
 private:  
 double x, y;  
};  
  
class Circle {  
 public:  
 Circle(Point center, double radius)  
 : center(center), radius(radius) {  
 std::cout << "Circle constructor called." << std::endl;  
 }  
  
 ~Circle() {  
 std::cout << "Circle destructor called." << std::endl;  
 }  
  
 bool intersect(Circle a) {  
 int dis = center.dis(a.center);  
 return dis <= radius + a.radius && dis >= std::abs(radius - a.radius);  
 }  
  
 private:  
 Point center;  
 double radius;  
};  
  
int main() {  
 double x1, y1, r1, x2, y2, r2;  
 std::cout << "Please input the center of circle 1: ";  
 std::cin >> x1 >> y1;  
 std::cout << "Please input the radius of circle 1: ";  
 std::cin >> r1;  
 std::cout << "Please input the center of circle 2: ";  
 std::cin >> x2 >> y2;  
 std::cout << "Please input the radius of circle 2: ";  
 std::cin >> r2;  
  
 Circle c1 = Circle(Point(x1, y1), r1);  
 Circle c2 = Circle(Point(x2, y2), r2);  
  
 if (c1.intersect(c2)) {  
 std::cout << "Circle 1 intersects with circle 2." << std::endl;  
 } else {  
 std::cout << "Circle 1 does not intersect with circle 2." << std::endl;  
 }  
}

### 第二部分

第二部分练习了C++中的RAII技术（利用构造函数和析构函数自动管理资源的获取和释放）、拷贝构造函数和拷贝赋值函数的使用。

拷贝赋值函数的实现是一个易错点。首先要检查自赋值，然后要释放旧内存，最后再申请内存并拷贝数据。另外，为了与原生拷贝运算符的行为相一致，拷贝赋值函数还要返回对自身的引用。

另外，在C++中，在函数参数中传递大对象的较好的方法是使用const引用。

我的实现中还具有三项额外特色：

1. 实现了Matrix类的移动构造函数和移动赋值函数，更加符合现代C++的编写标准，提高了效率。
2. 矩阵相加和矩阵相减的功能使用重载运算符实现，更加直观易用。
3. 矩阵相加和矩阵相减操作中对矩阵大小的检查使用assert断言实现。这样，在调试模式下如果出现矩阵大小不同的情况会输出调试信息，而在发布模式下这一检查会被移除，提高效率。

代码如下。

#include <cassert>  
#include <iostream>  
  
class Matrix {  
 public:  
 Matrix(int lines, int rows)  
 : lines(lines), rows(rows) {  
 data = new int[lines \* rows];  
 }  
  
 ~Matrix() {  
 delete[] data;  
 }  
  
 Matrix(const Matrix &a) {  
 lines = a.lines;  
 rows = a.rows;  
 data = new int[lines \* rows];  
 for (int i = 0; i < lines \* rows; i++) {  
 data[i] = a.data[i];  
 }  
 }  
  
 Matrix(Matrix &&a) {  
 data = a.data;  
 a.data = nullptr;  
 }  
  
 Matrix &operator=(const Matrix &a) {  
 if (this == &a) {  
 return \*this;  
 }  
 delete[] data;  
 lines = a.lines;  
 rows = a.rows;  
 data = new int[lines \* rows];  
 for (int i = 0; i < lines \* rows; i++) {  
 data[i] = a.data[i];  
 }  
 return \*this;  
 }  
  
 Matrix &operator=(Matrix &&a) {  
 if (this == &a) {  
 return \*this;  
 }  
 delete[] data;  
 lines = a.lines;  
 rows = a.rows;  
 data = a.data;  
 a.data = nullptr;  
 return \*this;  
 }  
  
 void input() {  
 for (int i = 0; i < lines; i++) {  
 for (int j = 0; j < rows; j++) {  
 std::cin >> data[i \* rows + j];  
 }  
 }  
 }  
  
 void output() {  
 for (int i = 0; i < lines; i++) {  
 std::cout << data[i \* rows];  
 for (int j = 1; j < rows; j++) {  
 std::cout << " " << data[i \* rows + j];  
 }  
 std::cout << std::endl;  
 }  
 }  
  
 Matrix operator+(const Matrix &a) {  
 assert(lines == a.lines && rows == a.rows);  
 Matrix ans(lines, rows);  
 for (int i = 0; i < lines \* rows; i++) {  
 ans.data[i] = data[i] + a.data[i];  
 }  
 return ans;  
 }  
  
 Matrix operator-(const Matrix &a) {  
 assert(lines == a.lines && rows == a.rows);  
 Matrix ans(lines, rows);  
 for (int i = 0; i < lines \* rows; i++) {  
 ans.data[i] = data[i] - a.data[i];  
 }  
 return ans;  
 }  
  
 private:  
 int lines, rows;  
 int \*data;  
};  
  
int main() {  
 Matrix A1(4, 5), A2(4, 5), A3(4, 5);  
 A1.input();  
 A2.input();  
 A3 = A1 + A2;  
 A3.output();  
 A3 = A1 - A2;  
 A3.output();  
  
 Matrix \*pA1 = new Matrix(4, 5), \*pA2 = new Matrix(4, 5), \*pA3 = new Matrix(4, 5);  
 pA1->input();  
 pA2->input();  
 \*pA3 = \*pA1 + \*pA2;  
 pA3->output();  
 \*pA3 = \*pA1 - \*pA2;  
 pA3->output();  
 delete pA1;  
 delete pA2;  
 delete pA3;  
}

## 继承与派生实验

本实验主要练习了C++中的继承与派生的用法。值得注意的是在这里还没有使用虚函数，调用哪个函数取决于对象声明为什么类型，没有实现运行时多态。

代码如下。

#include <cmath>  
#include <iostream>  
class Shape {  
 public:  
 double area() {  
 return 0;  
 }  
};  
  
class Rectangle : public Shape {  
 public:  
 Rectangle(double width, double height)  
 : width(width), height(height) {  
 std::cout << "Rectangle constructor called." << std::endl;  
 }  
  
 ~Rectangle() {  
 std::cout << "Rectangle destructor called." << std::endl;  
 }  
  
 double area() {  
 return width \* height;  
 }  
  
 private:  
 double width, height;  
};  
  
class Square : public Rectangle {  
 public:  
 Square(double length)  
 : Rectangle(length, length) {  
 std::cout << "Square constructor called." << std::endl;  
 }  
 ~Square() {  
 std::cout << "Square destructor called." << std::endl;  
 }  
};  
  
class Circle : public Shape {  
 public:  
 Circle(double radius)  
 : radius(radius) {  
 std::cout << "Circle constructor called." << std::endl;  
 }  
  
 ~Circle() {  
 std::cout << "Circle destructor called." << std::endl;  
 }  
  
 double area() {  
 return 3.14159265358979323846 \* radius \* radius;  
 }  
  
 private:  
 double radius;  
};  
  
int main() {  
 Rectangle s1(3, 2);  
 std::cout << s1.area() << std::endl;  
 Square s2(4);  
 std::cout << s2.area() << std::endl;  
 Circle s3(4);  
 std::cout << s3.area() << std::endl;  
}

## I/O 流实验

本实验主要练习了C++中输入输出流的高级用法。主要的难点在于正确地从错误输入中恢复。

从错误输入中恢复用到了std::cin.clear()和std::cin.ignore()两个函数。std::cin.clear()用于清除错误标志位，std::cin.ignore()用于忽略输入流中的一定数量的字符，向std::cin.ignore()中传入std::numeric\_limits<std::streamsize>::max()表示忽略输入流中的所有字符。这样一来就可以完全清除错误输入，使得程序可以继续运行。

#include <cstdlib>  
#include <ctime>  
#include <iostream>  
#include <limits>  
  
const long long max\_size = std::numeric\_limits<std::streamsize>::max();  
  
int main() {  
 std::srand(std::time(0));  
 int num = std::rand() % 1000 + 1;  
 int guess;  
 do {  
 std::cout << "Make a guess: ";  
 if (!(std::cin >> guess)) {  
 std::cout << "Wrong input." << std::endl;  
 std::cin.clear();  
 std::cin.ignore(max\_size, '\n');  
 } else if (guess > num) {  
 std::cout << "The price you guess is too high." << std::endl;  
 } else if (guess < num) {  
 std::cout << "The price you guess is too low." << std::endl;  
 }  
 } while (guess != num);  
 std::cout << "You are right!" << std::endl;  
}

## 重载实验

### 第一部分

本实验主要练习了C++中虚函数和抽象类的用法。

与“继承与派生实验”相比，本实验使用了虚函数。其中基类Shape的函数area为纯虚函数，而派生类中的area覆盖了基类的area函数。现在，即使对象声明为基类类型，调用area函数时也会调用派生类中的area函数，实现了运行时多态。

注意在C++中必须使用指针或引用才能使用虚函数特性，而值拷贝后派生类的信息会丢失。

C++中没有单独的抽象类语法，只要包含纯虚函数的类就是抽象类。抽象类不能实例化，只能作为基类使用。抽象类中的纯虚函数必须在派生类中实现，否则派生类也是抽象类。

代码如下。

#include <cmath>  
#include <iostream>  
class Shape {  
 public:  
 virtual double area() = 0;  
};  
  
class Rectangle : public Shape {  
 public:  
 Rectangle(double width, double height)  
 : width(width), height(height) {  
 std::cout << "Rectangle constructor called." << std::endl;  
 }  
  
 ~Rectangle() {  
 std::cout << "Rectangle destructor called." << std::endl;  
 }  
  
 double area() override {  
 return width \* height;  
 }  
  
 private:  
 double width, height;  
};  
  
class Square : public Rectangle {  
 public:  
 Square(double length)  
 : Rectangle(length, length) {  
 std::cout << "Square constructor called." << std::endl;  
 }  
 ~Square() {  
 std::cout << "Square destructor called." << std::endl;  
 }  
};  
  
class Circle : public Shape {  
 public:  
 Circle(double radius)  
 : radius(radius) {  
 std::cout << "Circle constructor called." << std::endl;  
 }  
  
 ~Circle() {  
 std::cout << "Circle destructor called." << std::endl;  
 }  
  
 double area() override {  
 return 3.14159265358979323846 \* radius \* radius;  
 }  
  
 private:  
 double radius;  
};  
  
int main() {  
 Rectangle s1(3, 2);  
 std::cout << s1.area() << std::endl;  
 Square s2(4);  
 std::cout << s2.area() << std::endl;  
 Circle s3(4);  
 std::cout << s3.area() << std::endl;  
}

### 第二部分

本实验主要练习了C++中运算符重载，尤其是重载自增和自建运算符的用法。

前置自增/自减运算符的行为是先自增/自减，然后返回自身的引用。后置自增/自减运算符的行为是先保存一份自身的拷贝，然后自增/自减，最后返回保存的拷贝。

C++中用于区分重载的是前置还是后置的自增/自减运算符的语法也值得关注。前置自增/自减运算符的声明中没有参数，而后置自增/自减运算符的声明中有一个int类型的形式参数。这个形式参数只是用来区分前置和后置，不会被使用。

代码如下。

#include <iostream>  
  
class Point {  
 public:  
 Point(double x, double y)  
 : x(x), y(y) {}  
  
 void output() {  
 std::cout << "(" << x << ", " << y << ")" << std::endl;  
 }  
  
 Point &operator++() {  
 x += 1;  
 y += 1;  
 return \*this;  
 }  
  
 Point operator++(int) {  
 Point tmp = \*this;  
 operator++();  
 return tmp;  
 }  
  
 Point &operator--() {  
 x -= 1;  
 y -= 1;  
 return \*this;  
 }  
  
 Point operator--(int) {  
 Point tmp = \*this;  
 operator--();  
 return tmp;  
 }  
  
 private:  
 double x, y;  
};  
  
int main() {  
 Point p(1, 2);  
 (++p).output();  
 (p++).output();  
 p.output();  
}

# 综合题目

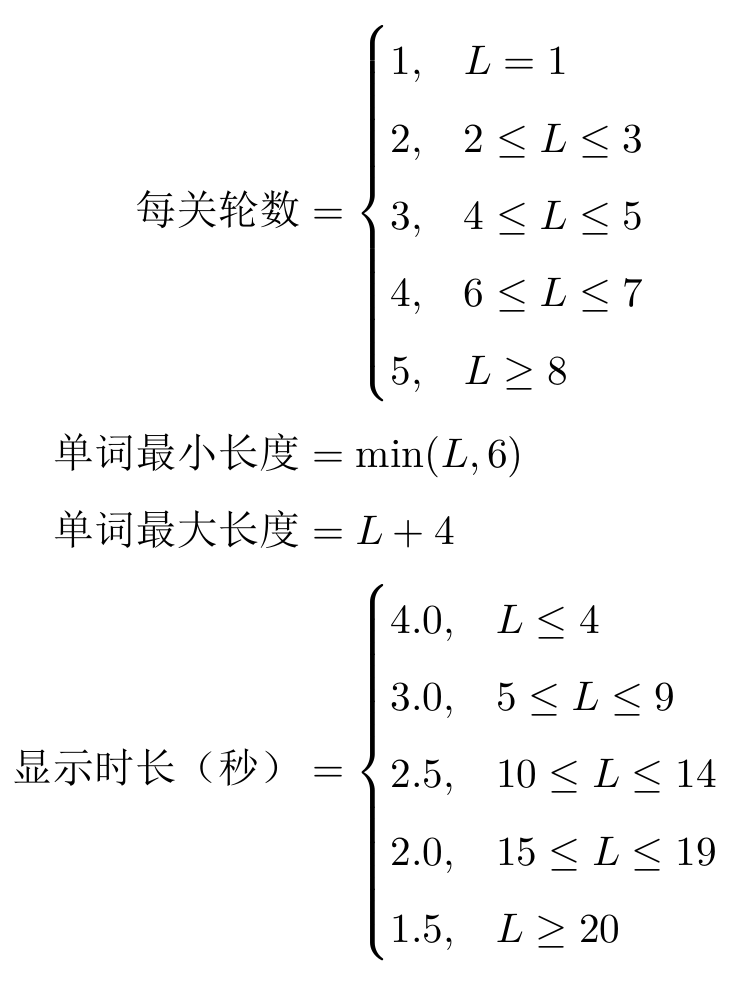
## 完成的内容

综合实验完成了三个题目的全部内容以及双人对战游戏的拓展功能。

在单人对战中，每关由若干轮构成。每通过一关玩家就可获得一定经验。获得的经验值与关卡号和所用时间有关，关系为：

在单人游戏中提供两次重试机会。输入答案错误后可以消耗重试机会再次尝试本关（题目为再次随机得到的）。若两次重试机会都耗尽则只能结束游戏。

随着关卡号的提升，游戏的难度也会从每关轮数、单词长度、显示时长三个方面逐渐提升。设关卡号为，则关系如下：



而在多人游戏中，游戏固定为第八关的难度，但游戏轮数变为10轮，采用抢答制，每一轮单独计分。抢答正确获得的经验值，对方失去的经验值。抢答错误失去的经验值，对方不获得也不失去经验值。

游戏默认的词库是牛津三千词。

## 技术选型

### 客户端界面

最先考虑的是客户端的显示界面问题。考虑到本游戏需要实现单词在一段时间后隐藏等功能，用普通的命令行界面较难实现。而用纯C++编写图形界面难度较大，QT是一个常见的选择，但过于笨重，且要把界面做美观也很困难。最后经过搜索我找到了[FTXUI](https://github.com/ArthurSonzogni/FTXUI)这个库。FTXUI是一个基于C++的字符界面库，使用简单，功能强大，且界面美观，非常适合本游戏的需求。因此我最终选择了使用FTXUI来实现客户端的显示界面。

### 通信模型

在设计程序时，我从一开始就考虑了客户端-服务端架构，围绕网络通信展开开发，使得开发过程总体而言较为顺利。

并发网络通信传统上使用多线程实现，每个线程各自处理一个socket。但是多线程编程的难度较大，容易出现各种问题，并且现在已有更加现代化的异步网络编程技术，因此我最终选择了使用asio这一C++异步网络库来完成网络通信。

服务端只使用一个线程，通过asio的事件循环来实现异步编程。

客户端由于已经使用了FTXUI的事件循环，如果要再加入asio的事件循环就必须使用另一个线程，但是考虑到两个线程之间需要频繁通信，且在客户端异步编程需要同时维护UI和网络的状态难度较大，因此我最终选择了使用同步的方式来实现客户端的网络通信。这样的好处是客户端可以保持单线程，且代码更加简单。缺点在于客户端不能及时收到服务端主动发来的消息。好在本程序中服务端主动发消息的场景不多，只有双人对战中会涉及，我使用轮询的方法来解决这个问题。

不过，客户端还存在一个计时的问题。客户端需要在一段倒计时后隐藏单词，以及前述的轮询也需要一个时钟来计时。为了提供一个时钟，最终客户端还是使用了多线程，专门开设了一个线程作为时钟，每隔100ms向主线程发送一个信号。由于这个线程和主线程的互动非常有限，因此相对较好控制，没有出现各种多线程的竞争问题。

### 通信协议

我选择了TCP长连接进行通信。TCP长连接的优点是提供了可靠的传输，且可以把登录状态绑定在了连接上，而不必每次都附带身份认证信息，缺点是需要维护连接，且需要自行分隔消息。

最开始，我想使用二进制协议来传输消息，也就是直接将一个结构体放入缓冲区，这样可以减少传输的数据量，提高效率。但是后来发现，这样做一方面灵活度低，另一方面由于不同平台的对齐、字节序不同，直接将一个结构体放入缓冲区的做法严格来说是不标准的。经过考虑，我换成了使用文本协议。

我设计的文本协议如下：

1. 每个消息以\0结尾，这与C/C++中字符串\0结尾的约定相符。
2. 每个消息内部的不同部分由换行符（\n分隔），其中第一行固定为消息类型，之后的内容根据消息类型不同而不同。

由于消息中保证不出现\0和\n，因此不需要转义。

完整的通信协议参见压缩包内的protocol.md文件，文件中消息类型标题后的C表示这一类消息由客户端发出的，S表示这一类消息由服务端发出。文本中小括号括起来的内容表示数字，中括号括起来的内容表示字符串，大括号括起来的内容表示前面定义过的数据结构。

### 数据存储

由于我对数据库的使用不熟悉，而且本程序的数据量不是非常大，因此我选择了使用tsv文件来存储数据。数据格式为每行一个条目，每个字段之间用制表符（\t）分隔。

运行时，数据存储在内存中，每次修改后都会更新文件。

题目数据在内存中用三个数据结构存储。

std::vector<Problem> m\_problems;  
std::vector<std::vector<int>> m\_problemIndexByLength;  
std::unordered\_set<std::string> m\_wordSet;

m\_problems为所有题目的列表。m\_problemIndexByLength为以单词长度为关键字的索引，方便随机获取特定长度范围内的单词。m\_wordSet是所有单词的无序集合，用于快速查找某个单词是否已经在在题库里了。

## 客户端项目结构

客户端以页面组织代码。客户端包含以下页面：

1. 连接服务器界面
2. 登录界面
3. 注册界面
4. 主界面
5. 单人游戏界面
6. 对战界面
7. 排行榜界面

具体每个界面的设计不再详述，可参见源码。

每个界面都是一个类，继承自共同的基类PageBase，PageBase又继承自ftxui的组件基类ftxui::ComponentBase。

除此以外我还自行编写了一个ftxui的路由组件，用于管理页面之间的跳转。值得注意的一点是，如果在跳转时直接将旧页面移除，由于旧页面的std::shared\_ptr引用计数归零，会被直接析构，容易出现“我删我自己”的情况导致程序崩溃。因此我在路由组件中使用了延迟删除的方法，在跳转页面时不立即删除旧页面，而是继续用一个std::shared\_ptr指向旧页面避免其引用计数归零，直到再次跳转页面才析构旧页面。此外这个路由组件还附带了一个弹出对话框的功能，使路由内的页面可以使用alert函数方便地弹出对话框。

为了方便查看调试信息，我还实现了FTXUI中调试信息的输出。我将标准错误流（stderr）重定向到了FTXUI组件中，在客户端按F1键可以开关调试信息的输出。

我使用依赖注入的思想，将全局用户状态信息、路由组件、网络组件等打包为**全局上下文**（GlobalContext）作为参数传入每个页面，使得每个页面都可以访问这些全局资源。

客户端运行在FTXUI的事件循环下。所有网络通信都由客户端主动发起。客户端主动发出一个消息后，如果期待应答，就阻塞并等待应答。对于需要主动接受的消息，客户端会用轮询的方法处理。

在需要轮询或计时的页面中，一个时钟线程会被启用。时钟线程每隔100ms向主线程发送一个信号，主线程收到信号后会进行一轮轮询并更新游戏的倒计时。为了避免页面析构后时钟线程仍然在运行，页面和时钟线程都持有一个指向堆内存里的std::atomic\_bool的std::shared\_ptr，当页面析构时会将这个标志置为false，时钟线程在每次发送信号前都会检查这个标志，如果为false则停止发送信号，时钟线程退出，同时标志本身所占内存也被清除。

## 服务端项目结构

服务端由下面几大模块组成：

1. Session类处理连接与单人游戏
2. Battle类处理双人对战
3. User定义了用户信息和相关方法。其中User类为基类，闯关者Challenger和出题者Author均为User类的派生类。
4. Problem类定义了题目信息和相关方法。
5. Database类包含数据库的相关方法，用于从tsv文件中读取和写入数据。

服务端采用asio事件循环的基本框架。每当建立一个连接时，就会创建一个Session对象，Session对象中包含了这个连接的所有信息，包括连接的socket、用户信息、用户状态等。

每个Session都维护一个状态机，表示该连接当前处于什么状态。状态机的状态包括：

1. init：初始状态，表示连接已建立，但用户还未登录。
2. challengerLogined：已以闯关者的身份登录。
3. authorLogined：已以出题者的身份登录。
4. inGame：正在单人游戏。
5. waitForRetry：单人游戏结束，等待用户选择是否重试。
6. matching：正在匹配双人对战。
7. battle：正在双人对战。

状态机的设计让服务端的逻辑更加清晰，每个状态都转交给一个单独的函数处理，更加明确了“什么时候应该收到什么类型的消息”，避免了一些潜在的bug。

单人游戏逻辑直接由Session处理，而多人游戏会先进行匹配，匹配成功后会创建一个Battle对象，双方的Session都会把控制转交给这个Battle对象，由其处理双人对战逻辑。

## 项目对C++特性的运用

作为C++课程的大作业，本项目有意识地运用了C++的各种特性。包括但不限于：

1. 面向对象思想。本项目将程序的运行视作对象和对象之间的交互，合理地使用了private，protected和public修饰符实现了各个类的**封装**，使用了**继承**和**多态**实现了代码的复用。还简单使用了工厂模式、依赖注入等设计模式。
2. 继承和虚函数。除了题目要求的让闯关者和出题者继承同一个基类，我还在客户端的UI系统中使用了继承。
3. 标准模板库（STL）。项目中大量使用了std::vector，std::unordered\_map等模板容器，以及std::sort等模板算法。
4. 引用。项目中大量使用了C++中引用这一方便的特性，相比指针更加安全、便利。
5. 现代C++特性。本项目基于C++17标准开发，使用的FTXUI和asio这两个库也都积极地使用了现代C++特性，因此本项目中也普遍地使用了现代C++特性，包括但不限于：
   1. auto关键字。使用auto关键字可以让编译器自动推导变量的类型，减少代码量，提高可读性。
   2. lambda表达式。lambda表达式不仅方便了服务端异步代码的编写，也对客户端UI的函数式编程风格有很大帮助。
   3. 智能指针。项目中没有手动使用new和delete，完全使用智能指针进行资源管理。将shared\_from\_this与lambda捕获列表结合的模式可以很好地解决异步编程中资源管理的问题。
   4. 移动语义。使用移动语义将socket的所有权转交给Session。
   5. 线程库、时间库、随机数库等新标准新增内容。

# 心得体会

基础题目较为简单，练习了一些C++的基本语法内容。

综合题目有一定难度，需要较强的C++编程能力和项目开发能力。我从中得到了很多收获。

首先，我体验了进行面向对象的程序架构的设计，学会了如何将程序分为不同的模块，如何设计类和类之间的交互，如何利用多态实现代码的复用，如何封装等。在设计的过程中，我体会到了一些经典的设计思想。例如，在服务端一开始的设计中，Session和Battle两个类相互依赖，这在C++中是不允许的，即使使用了前置声明也不能通过编译。这样的情况就表明了两个类出现了过度耦合。作为修正，Battle类现在不再依赖于Session类，而是通过回调函数来实现双方的交互。这样一来，Battle类就可以独立于Session类，消除了耦合。

其次，管理内存是C++的一大难题。在开发过程中，我体会到了智能指针等技术对内存管理的帮助。在FTXUI和asio中std::shared\_ptr都得到了广泛的应用，因此我在自己开发中也延续了这一风格。在客户端，所有自定义组件也都使用std::shared\_ptr进行管理。在服务端std::shared\_ptr可以方便异步编程的环境下维持Session的生命期，User的传递使用的也是std::shared\_ptr。

在开发程序中要考虑很多细节问题。例如在实现根据属性筛选用户时，在根据用户名筛选时用户期待的是子串匹配，但在根据等级筛选时，用户通常期待的不是子串匹配而是精确匹配。在登陆系统中需要加入用户是否重复登陆的判断。这样的细节问题在开发过程中还有很多，需要根据实际应用场景仔细考虑，才能做出真正符合实际的软件。

此外，设计游戏的难度曲线也是比较有趣的一部分。

# 对课程的建议

希望课程可以更多地介绍现代C++，即C++11和更高标准的内容，以满足当下的发展趋势。