程序设计实践课程设计报告

1. 项目概述

本程序以1995年由Oberon Games开发的一款的棋牌类单机游戏《空当接龙》原型为蓝本，该游戏自1995年起在Windows系统发行。空当接龙（FreeCell）是一款使用52张牌进行的单人纸牌游戏，游戏目标是将整副牌搬到右上角的四个空栏中。在把牌从初始牌堆移动到目标牌区过程中，用户可用提供的四个缓冲区暂存牌。游戏逻辑将在用户使用手册中详细说明。

1. 项目分工

7班 2017211337 刘兴贤 负责后端服务器、游戏初始设计框架及部分游戏逻辑的开发。

6班 2017211325 项云松 负责游戏逻辑代码部分的编写、移动逻辑及附加功能的开发。

6班 2017211327 富俞锋 游戏界面的前端设计、附加功能开发及窗口逻辑。

1. 开发环境

Windows 10 系统个人PC机（刘兴贤、项云松）

Mac OS 个人PC机（富俞锋）

C++程序设计语言

Qt 5.12

Visual Studio集成开发环境

Qt Creater 4.8.2

1. 软件结构

服务器

Card

Controller

其中，采用MVC结构，MainWindow中实现的功能在controller中进行控制。

1. 设计模式

本程序在最初设计架构及类之间的逻辑关系加入了设计模式的考量。

本程序使用的架构是MVC模式，这是23种基础设计模式中观察者模式(Observer)、策略模式(Strategy)和组合模式(Composite)三种设计模式的组合。

其具体模式如下



该模式的主要思想是数据、控制、视图三者分离，该模式降低了程序各部分之间的耦合，使每个部分能够较为独立、代码划分较为清晰，程序的“高内聚低耦合”特性能够得以实现。

MVC是三个单词的首字母缩写，它们是Model（模型）、View（视图）和Controller（控制）。这个模式认为，程序不论简单或复杂，从结构上看，都可以分成三层。

1）最上面的一层，是直接面向最终用户的"视图层"（View）。它是提供给用户的操作界面，是程序的外壳。

2）最底下的一层，是核心的"数据层"（Model），也就是程序需要操作的数据或信息。

3）中间的一层，就是"控制层"（Controller），它负责根据用户从"视图层"输入的指令，选取"数据层"中的数据，然后对其进行相应的操作，产生最终结果。

本程序界面如下图：



本程序中的Model是纸牌Card类，每张纸牌是一个独立的对象，它用来存储纸牌的数据（点数、位置）等信息。

View是MainWindow、ScoreWindow、EntryWindow这些界面类，这些界面类直接负责与用户的交互，每张牌的位置从用户界面直接获得，并由控制类传入model。

Controller是本程序中的Controller类，该类作为控制类负责协调前端View与Model之间的协调。

MVC模式在本程序中的具体工作机制将在下面一些具体情况下给出描述：

1. 纸牌的绘图

纸牌的绘图需要其点数，但为了支持MVC模式，每一张纸牌彼此之间应当都是独立的，也就是说，只要纸牌拿到了自己的点数，就可以绘出该牌，而为该牌赋点数这一操作应当从该类中分离出去。作为Model的Card类存储自己的数据，然后利用该数据绘图，也就是说，该类的成员方法只用于绘出该牌，而不必管具体逻辑或数据是什么，Card类在构造函数中提供了一个接口，使得其对于所有的数据都做相同的操作而不必显式区分。

1. 纸牌的移动

纸牌的移动本身是使用Qt中提供的信号与槽机制完成的，该移动似乎只与Card类本身有关，但值得注意的是：我们在移动Card类时，其位置的移动依赖于其内部成员x、y坐标，而这些x、y坐标则是由View通过Controller传给该类的。在View中我们并未直接操纵Card类的位置，而是在View类中使用Controller控制位置坐标，并在加入移动逻辑的基础上，确定某牌的合法坐标，并将其传入Card类。这也就是说，Controller类本身在这里是一个中间层，用于从View中拿到用户操作，经过一系列逻辑处理后，将其交给Model。在此过程中控制流从用户界面流入Controller，再从Controller流出最终到达数据部分。通过这种方式，用户以间接方式控制数据，实现了功能部分的解构与分离，这使得程序部分之间的的耦合度降低。

实现方式：

由于在主窗口中创建的Controller的对象负责Model与View之间的协调，并且作为管控者的Cotroller应当对Model的数据负全部责任，这意味着Model中数据的正确性与安全性应当由Controller完全管控。而由于我们使用了MVC设计模式，我们将本来耦合的控制与模型分开成了两个类，这样虽然在逻辑层面更加清楚，但给编程实现上额外增加了不小的难度，难度在于为了保证Card中数据的安全性，我们将其使用private域保护起来是必需的，然而类的分离将使这些数据不可被Controller直接访问。

由于我们在前面说明了Controller对Model的责任，因此我们希望：

Controller管控Card的数据，但并不由它本身存储。

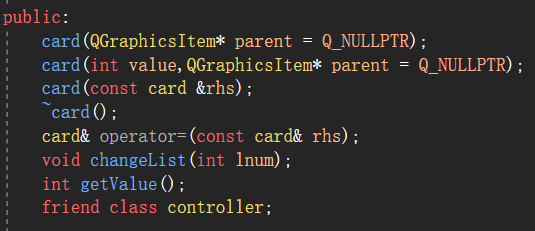
这样在C++中有3种实现方式：

（1）使用嵌套类（内部类），将Controller声明为Card的内部类，这样我们能保证Card数据的安全性，Card的private数据成员并未对外开放，而是开放给了它的内部类访问，相当于Controller作为Card类的内部成员，有权限操作Card所有的private数据成员。但这样的问题在于，我们使用MVC设计模式，花费相当精力将控制与数据分离。这样内部类的机制事实上又增加了程序模块之间的耦合性，这使得我们之前做的分离工作有相当一部分是毫无作用的。

（2） 定义Card类与Controller类之间的接口，这样毫无疑问的保证了Card的private数据成员的安全性，但这样做带来的缺点也相当明显，这意味着为了实现控制流的传递，我们要对Card的每一个private数据成员都设置get与set成员方法并将其暴露在外，这样会带来相当复杂的冗余性。事实上，这样的设计思路应当在两个无管控关系的类实现，以将那些不希望被修改的成员保护起来。但对于本程序中，Controller作为Card数据的完全管控者，Card不应当隐藏自己任何的成员变量。

（3）使Controller成为Card的友元，然后在Controller中实例化所有需要的牌。这是目前我们给出的最优的实现方法。这样避免了数据成员在两个类之间无意义的复制，也避免了设计专用额外接口带来的开销。同时也保证了数据的安全性，因为Card类只将这个权限开放给了Controller类，也只有Controller类能够直接访问其内部的私有成员。这相当于是Card给了Controller一把钥匙，这把钥匙开放了其内部私有成员的访问权与控制权。当然，这一切都建立在Card类足够信任Controller类的基础上。

具体实现代码如下图：



1. 数据结构

1、客户端

class **Card** : public QObject, public QGraphicsItem

{

Q\_OBJECT

Q\_INTERFACES(**QGraphicsItem**)

public:

**Card**(QGraphicsItem\* parent = Q\_NULLPTR);

**Card**(int value, QGraphicsItem\* parent = Q\_NULLPTR);

**Card**(const Card& rhs);

~***Card***();

Card& operator=(const Card& rhs);

void **changeList**(int lnum);

int **getValue**();

friend class Controller;

signals:

void **signalFindList**(int source, int x, int y, int v);

void **signalMoveTogether**(double x, double y);

void **signalStartBind**(double x, double y);

protected:

void ***mousePressEvent***(QGraphicsSceneMouseEvent\* event) override;

void ***mouseMoveEvent***(QGraphicsSceneMouseEvent\* event) override;

void ***mouseReleaseEvent***(QGraphicsSceneMouseEvent\* event) override;

void ***paint***(QPainter\* painter, const QStyleOptionGraphicsItem\* option, QWidget\* widget) override; //画图

QRectF ***boundingRect***() const override;

private:

int value; //牌的值，映射到花色和大小

QRectF rect;

int listNum; //所在位置编号

bool isPressed; //能否点击

bool ismovable; //能否移动

};

class **Operation**//操作的记录

{

public:

int num; //操作的牌数

int sour; //源编号

int dest; //目的地编号

};

class **Controller** : public QObject

{

Q\_OBJECT

public:

**Controller**(QObject\* parent);

~***Controller***();

void **addToList**();//初始化队列，向队列中加入牌

void **addToScene**(QGraphicsScene\* scene);

void **initialCardHeap**();//初始化牌堆，为牌申请空间

void **initialCardRec**();//设置牌的初始位置和属性

void **initialBuffer**();//初始化缓冲区

void **initialDest**();//初始化最终区域

void **shuffleCardHeap**();//洗牌

signals:

void **signalPassed**();

void **signalTimeout**();

public slots:

void **slotFindList**(int source, int x, int y, int value); //寻找目的地

void **slotClickRetractionButton**();//悔牌

void **slotBindMoveTogether**(double x, double y); //多张牌移动时绑定信号

private:

void **isPassed**();//判断是否通关

bool **isBetween**(int lhs, int rhs); //判断两张牌能否连在一起

//移动到队列

int **moveToList**(int num, int source, Card\* leaveCard, int x, int y, int value);

//移动到缓冲区

int **moveToTemp**(int source, Card\* leaveCard, int x, int y, int value);

//移动到最终区域

int **moveToDest**(int source, Card\* leaveCard, int x, int y, int value);

//撤销移动多张牌的记录

void **retractMulCard**(const Operation& last);

//撤销移动单张牌到队列的记录

void **retractFromList**(const Operation& last);

//撤销移动单张牌到缓冲区的记录

void **retractFromTemp**(const Operation& last);

//撤销移动单张牌到最终区域的记录

void **retraceFromDest**(const Operation& last);

//为单张牌寻找目的地

void **findList\_SingleCard**(int num, int source, int x, int y, int value);

//为多张牌寻找目的地

void **findList\_SeveralCard**(int num, int source, int x, int y, int value);

Card\* buffer[4]; //缓冲区

QVector<Card\*> cardHeap; //牌堆

QVector<QVector<Card\*>> cardList; //队列

QVector<QVector<Card\*>> destList; //最终区域上放的牌

Card\* dest[4]; //最终区域

bool isFull[4]; //缓冲区是否为空

int bufferEmpty; //空缓冲区数目

QVector<Operation> steps; //操作记录队列

QVector<Card\*> leaveCardVec; //被移动的牌

QTimer\* timer; //计时器

};

class **ScoreWindow** : public QWidget

{

Q\_OBJECT

public:

**ScoreWindow**( int score = 0, int time = 0, QString name = "", QWidget\* parent = Q\_NULLPTR);

~***ScoreWindow***();

void **getInfo**();//接收数据

void **sendRequest**(int time, QString name);//发送请求

void **testConnection**();//测试链接

signals:

void **signalDataReady**();//数据ready信号

void **signalTestConn**(bool isConnecting);//测试链接信号

void **signalNextGame**();//下一局按钮信号

private:

QString **readIPAddr**();//获取IP地址

private slots:

void **showData**();//显示排行

void **lookUpHost**(QHostInfo info);//传数据

void **slotNextGameButton**();//下一局游戏按钮槽函数

private:

Ui::ScoreWindow ui;

QVector<User> userList;//存储排行信息

QTcpSocket\* clientSocket;//socket

QTableView\* viewRank;//排行榜

QPushButton\* updateGame;//下一局游戏

};

class **EntryNameWindow** : public QWidget

{

Q\_OBJECT

public:

explicit **EntryNameWindow**(int time, QWidget\* parent = nullptr);

~***EntryNameWindow***();

signals:

void **signalNextGame**();//下一关信号

private slots:

void **on\_buttonSure\_clicked**();//退出键

void **slotNextGame**();//下一关按键

private:

Ui::EntryNameWindow ui;

QLabel\* labelTip;//提示信息

QLineEdit\* entryName;//输入框

QPushButton\* buttonSure;//确认按钮

ScoreWindow\* scoreRank;//排行榜窗口

int timeofUser;//用户所用时间

bool isOpenScore = false;//标志判断是否打开score窗口

};

class **MainWindow** : public QMainWindow

{

Q\_OBJECT

public:

**MainWindow**(QWidget\* parent = Q\_NULLPTR);

~***MainWindow***();

public slots:

void **slotPassed**();

void **showtime**();

void **slotNewGame**();

private:

Ui::MainWindowClass ui;

QGraphicsView\* view;

QGraphicsScene\* scene;

Controller\* contrl; //控制器

EntryNameWindow\* entryName; //输入用户名窗口

QPushButton\* retractionButton; //悔牌按钮

bool isClose;

QLabel\* displaytime; //显示时间

int timeNowSec = 0; //计时器的分钟

int timeNowMin = 0; //计时器的秒钟

};

2、服务器

class **MulServer** : public QTcpServer

{

Q\_OBJECT

public:

**MulServer**(QObject\* parent);

~***MulServer***();

void ***incomingConnection***(qintptr handle) override;

void **replyToClient**(QTcpSocket\* socket);

void **showConnection**(QTcpSocket\* socket);

void **writeToSocket**(qintptr handle, QString str);

signals:

void **readyToReply**(QTcpSocket\* socket);

void **serverReceived**(QTcpSocket\* socket);

private:

QMap<qintptr, QTcpSocket\*> clientList;

};

class DBThread : public QObject, public QRunnable

{

Q\_OBJECT

public:

**DBThread**(qintptr handle, QObject\* parent = Q\_NULLPTR);

**DBThread**(qintptr handle, QString info, QObject\* parent = Q\_NULLPTR);

~***DBThread***();

bool **connectToSql**(const QString dbName);

QString **getInfo**();

signals:

void **finishDB**(qintptr handle, QString str);

void **finished**();

protected:

void ***run***()override;

private:

qintptr handle;

QString updateInfo;

QSqlDatabase myDB;

};

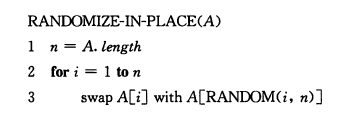
1. 算法描述

本程序中有洗牌的需求，下面给出它的问题描述：

数组顺序存储了1到52的整数，现需要找到一个算法，使得使用该算法后，数组内任意一张牌出现在任意位置的概率都相等。

对于一个长度为n的序列来说，其排列可能性应有n！种，我们的目标是等概率的给出这些排列中的一种。也即我们给出每一种排列的概率都是 ，下面使用的算法将在O（n）时间内给出一个上面的排列，使得对任一序列其被选取到的概率都是 。

该算法由Donald Knuth 在其著作《The Art Of Computer Programing》中给出，因此该算法也被称为Knuth-shuffle算法。下面给出其伪代码描述：

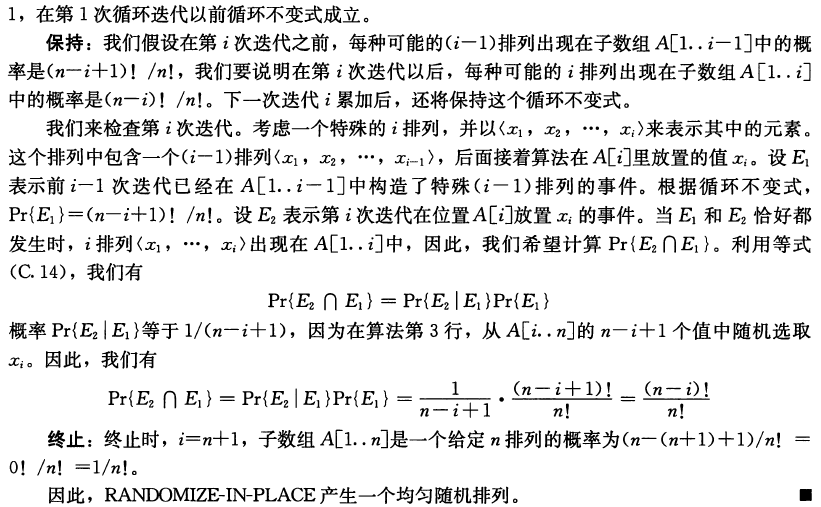


从伪代码来看，这是一个相当简洁的算法。

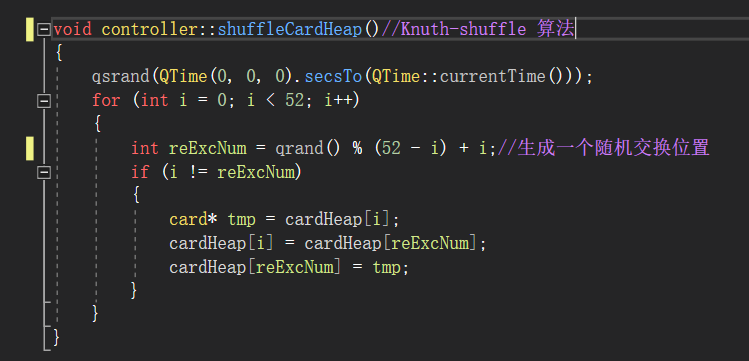
下面我们尝试使用循环不变式证明这个算法确实能给出一个均匀随机排列。

我们使用的循环不变式如下：

**在2到3行for循环的第i次迭代以前，对每个可能的（i-1）排列，子数组A[1..i-1]包含这个（i-1）排列的概率是(n-i+1)!/n!。**

我们只需证明上述不变式在第一次迭代前为真，并且循环的每次迭代能维持此不变式，并且循环终止时这个不变式提供一个有用的性质来说明正确性。

具体实现方式如下：



1. 游戏逻辑设计

1、牌的移动

当点击一张牌的时候，程序根据它的位置来判断能否进行移动。

如果在队列中，那么分为两种情况：

（1）在队列尾部，那么此时必定可以且只能移动一张牌，直接移动即可；

（2）不在队列尾部，那么此时有可能可以移动多张牌，如果从这张牌开始队列尾部的牌满足三个条件：（A）是降序放置的，（B）是红黑交替的，（C）这些牌的数目小于等于当前空缓冲区数目+1，那么就可以同时移动这些牌。为了能够实现同时移动，程序中为Card类设立了一个signalMoveTogether信号，每次点击时，首先将所有牌的该信号全部解绑，然后将要同时移动的牌的设置位置函数与被点击的牌的signalMoveTogether信号绑定，只要被点击的牌一挪动，就会发出signalMoveTogether信号，其它的牌的位置就会改变。

如果在缓冲区中，那么必定可以移动。

如果在最终区域中，那么不能移动。

另外，如果有牌可以移动，那么这些牌要放入移动队列leaveCardVec中。

2、把被移动的牌放到目的地

当释放鼠标按键时，要根据牌的当前位置为其寻找目的地。只要之前被点击的牌与某张在队列尾部的牌、某个空缓冲区、或某个最终区域有重叠的话，那么就要根据游戏规则判断这张牌（或几张牌）是否能够放到与其重叠的区域。

重叠的判断逻辑如下：在本程序中，所有的牌、缓冲区和最终区域的大小都是一样的，我们假设其高为H，宽为W，再假设被点击的牌被释放时的左上角坐标为，某个区域的左上角坐标为，那么只要满足，那么就说明这张牌与该区域重叠。

被点击的牌能否放到某一区域的判断逻辑如下：

（1）被挪动的牌只有一张

首先对8个队列尾部的牌从左到右进行遍历，如果被点击的牌与第i个队列尾部的牌c满足如下三个条件：（A）重叠，（B）被点击的牌的大小=牌c的大小-1，（C）颜色不同，那么就可以把被点击的牌放到牌c的下方，然后终止遍历（也就是说，如果被点击的牌同时与两个区域重叠，永远放在左边的区域，下同）。

如果没有发现满足上述三个条件的牌c，那么我们就对4个缓冲区从左到右进行遍历，如果被点击的牌与第i个缓冲区b满足如下两个条件：（A）重叠，（B）缓冲区b为空，那么就可以把被点击的牌放到区域b上，并设置缓冲区相关属性，然后终止遍历。

如果没有发现满足上述三个条件的缓冲区b，那么我们就对4个最终区域从左到右进行遍历，如果被点击的牌与第i个最终区域d满足如下两个条件：（A）重叠，（B）最终区域d为空 or (被点击的牌与最终区域d最上面的牌花色相同 and 被点击的牌的大小=最终区域d最上面的牌的大小+1)，那么就可以把被点击的牌放到最终区域d上,然后终止遍历。

如果没有发现满足上述三个条件的最终区域d，那么就把牌放回原位。

（2）被挪动的牌有多张

该情况下只能移动到队列，故只要对8个队列尾部的牌从左到右进行遍历即可，只要不满足上述三个条件，就可以直接把牌放回原位。唯一不同的是，当满足三个条件时，如果被点击的牌下面还有其他一同被挪动的牌，那么也按顺序搁在c下方（其他一同被挪动的牌从移动队列leaveCardVec中取出）。

另外，每次移动结束后，都要清空移动队列leaveCardVec。

3、悔牌

程序中维护了一个操作记录栈，每条记录包含所移动的牌的数目n、源位置、目的位置，每次悔牌即时从栈顶取出一条记录，并按照记录从目的位置移动n张牌到源位置。移动逻辑与上述逻辑类似。

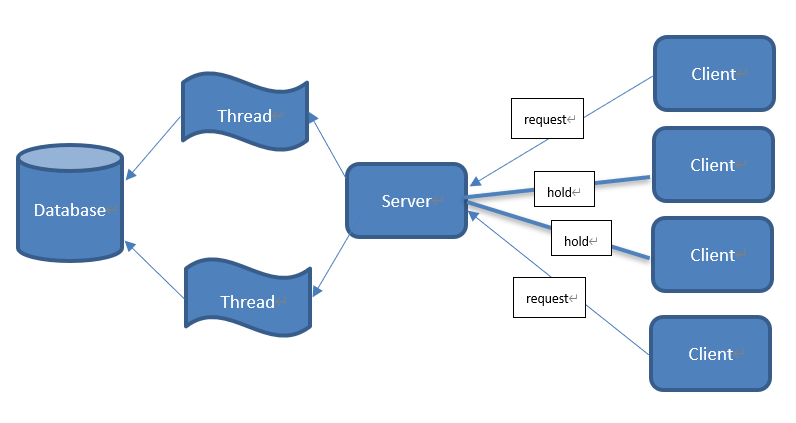
4、判断是否通关

当所有的牌都在最终区域中时，则说明该局游戏通关。

1. 并发结构设计

本程序的服务模型使用的是C/S（Client and Server）模型，因此并发任务集中在服务器部分，并发意味着服务器应当同时能接受并维护多个客户端传来的请求连接，并对其请求消息进行分析，进行数据库相关查询后，将查询结果返回客户端，这个过程中允许并发。

服务器的并发结构如下图所示：



Server使用map数据结构存储每一个来自Client的TCP连接，本程序中，最多可同时维持1000个客户端连接（使用信号与槽机制互不影响）。但应区分的是，有的连接虽然维持，但是不一定发来请求，对于这样的请求，若对于每一个来的连接都建立一个线程来做，这样会浪费相当多的系统资源。

为避免浪费过多的系统资源，本程序有两种机制来处理并发请求：

（1）只对那些向数据库的请求分配一个线程，对于维持的连接不必分配线程，同时查询线程执行完后自动退出，这意味着同一时刻可接受的数据库请求数是可分配的最大线程数，查询线程执行后立即退出意味着超出线程数量的请求被信号与槽机制放在等待队列中，待线程资源释放后，会重新为其分配线程资源。因此本程序最多同时维持1000个连接，同时执行最多最大线程数个查询任务。

（2）使用线程池预分配，避免为新来任务分配线程所带来的额外开销。线程过多会带来调度开销，进而影响缓存局部性和整体性能。而线程池维护着多个线程，等待着监督管理者分配可并发执行的任务。这避免了在处理短时间任务时创建与销毁线程的代价。线程池不仅能够保证内核的充分利用，还能防止过分调度。

1. 异常处理

本程序的异常处理分为两部分：

1. 游戏界面部分
2. 连接服务器部分

我们先描述游戏界面部分的异常处理机制，游戏界面的异常处理机制事实上是游戏逻辑的一部分，当玩家点击一张或多张牌在界面内拖动时，用户松开鼠标则认为用户要完成转移牌的操作。然而牌堆只能红黑相接且点数递减才能完成转移，缓存区可以放任何用户想要放置的牌，但是缓存区只有4个空位，一个空位不得放置多张牌。同时，目标区只有花色相同并且点数连续时，才可以放置入目标区。

对于不满足上面情况的牌转移，程序将会将其自动置回原位，而不是向用户报一个错误，这是参考Microsoft的实现方式。

接着我们来讨论游戏服务器部分的异常处理机制。

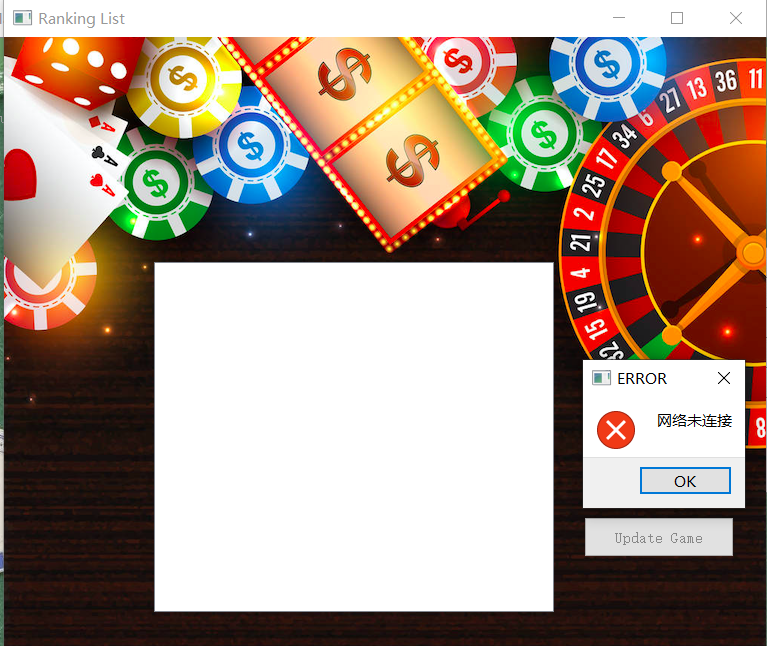
当出现下面几种情况时，我们需要做异常处理机制：

1. 当客户端使用TCP向服务器发送请求，但客户端所在机器网络未连接时。
2. 当客户端从外部文件读取服务器IP地址失败时。
3. 当客户端使用TCP向服务器发送请求，但连接并未建立成功时。
4. 服务器断开连接时。
5. 当客户端发送了错误的消息类型时。
6. 服务器数据库配置错误时。

下面我们详述各种情况下采取的异常处理机制。

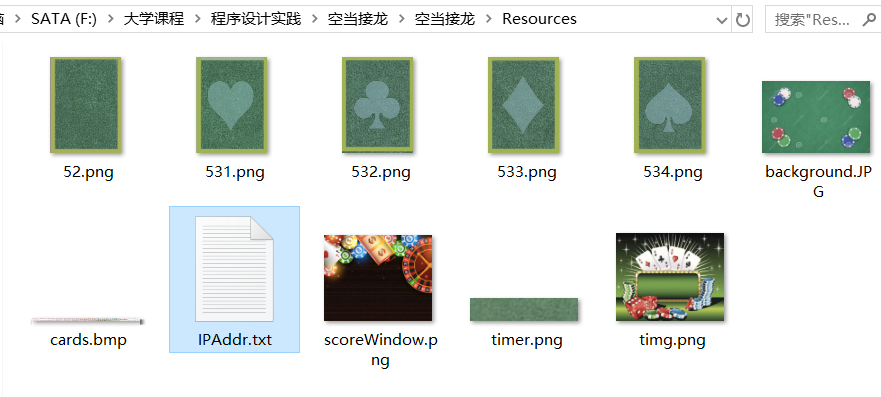
1. 当客户端使用TCP向服务器发送请求，但客户端所在机器网络未连接时

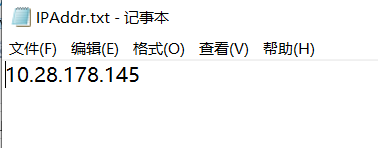
如下图，程序将会报出一个网络未连接的error。



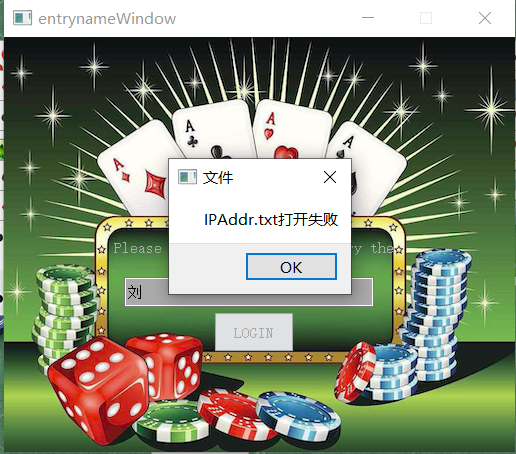
1. 当客户端从外部文件读取服务器IP地址失败时

如下图所示，服务器的IP地址记录在文件IPAddr.txt中，可能需要根据服务器所在的局域网不同修改具体的IP地址。

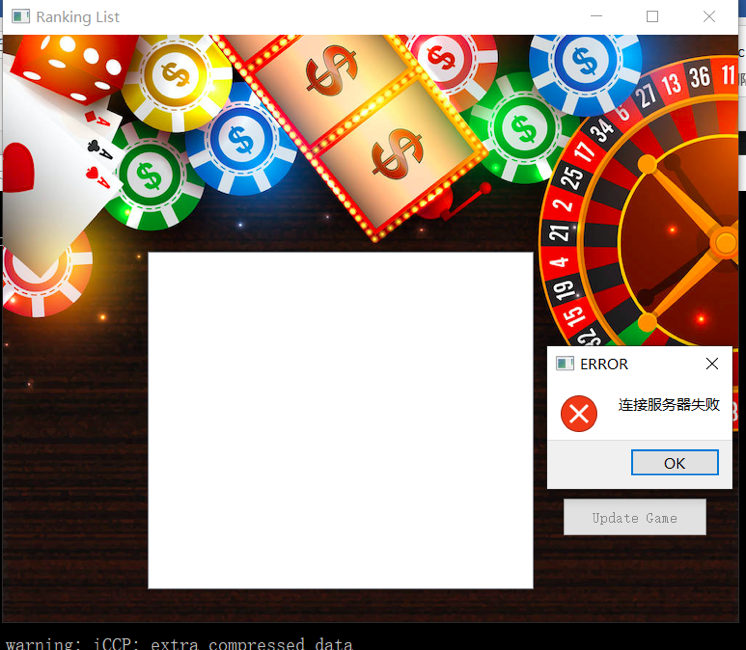




当文件打开失败时，会报出一个警告，此时需要用户检查该文件是否出现问题。



1. 当客户端使用TCP向服务器发送请求，但连接并未建立成功时



1. 服务器断开连接时



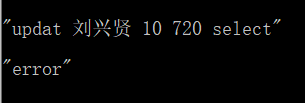
1. 当客户端发送了错误的消息类型时

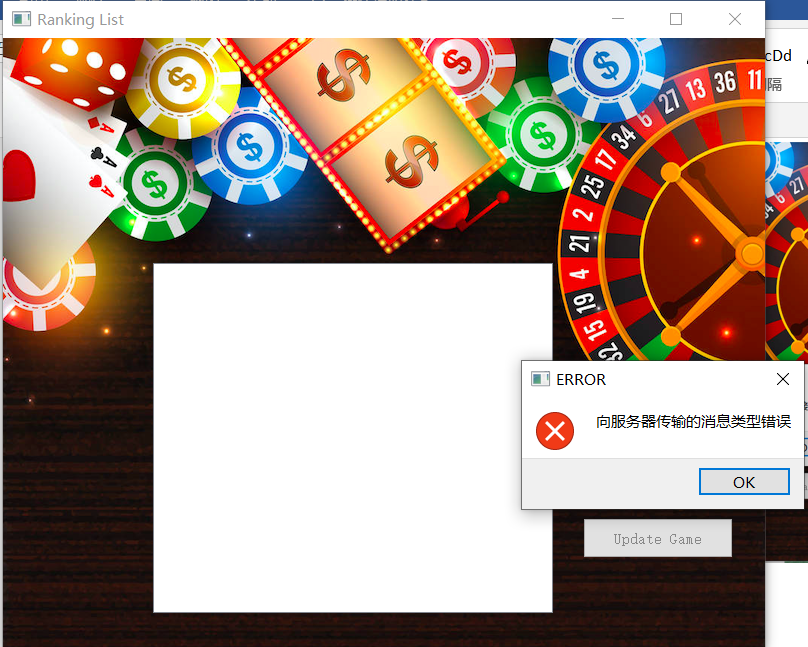
合法的消息类型如下：



这段话的含义是，名为刘的用户用720s完成了该局游戏，应在数据库中将其得分加10，并把排行榜返回。

当客户端发送了错误的消息类型时（如下图），程序将会报出一个错误。

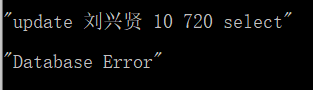


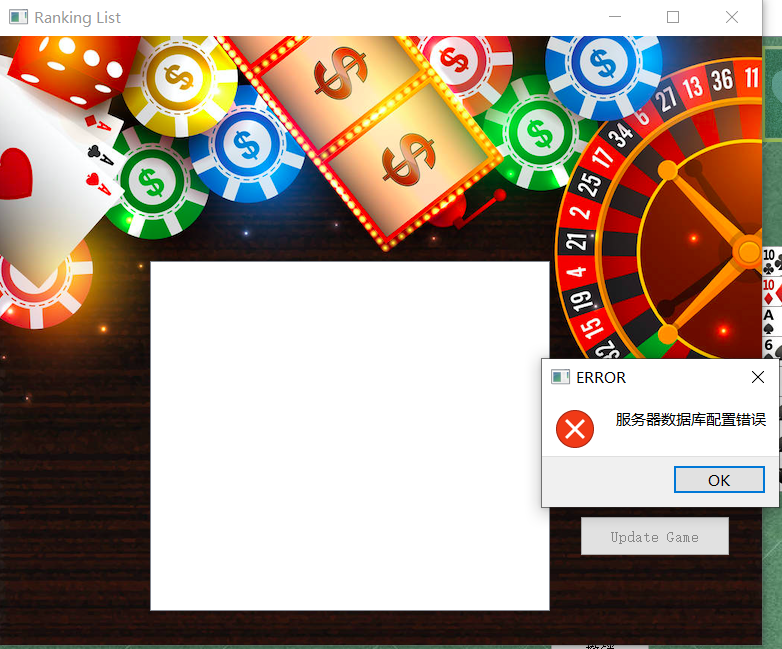


1. 服务器数据库配置错误时。

当服务器接收到客户端发来的请求消息，但由于服务器端未连接到数据库（可能配置错误或密码错误），导致查询结果不能正常返回，此时有必要通知客户端。

客户端收到一个消息：“Database Error”，然后显示错误窗口通知。





1. 用户使用手册

空当接龙用户使用手册

1. **产品介绍**

本产品借鉴自微软公司开发的FREECELL游戏，经开发人员重新修改界面、功能，旨在为每一个用户提供最优的游戏体验。

1. **产品功能**

**（1）、基本游戏功能：**

首先，介绍一下空当接龙这款游戏的游戏规则：

* **游戏目标**

在四个回收单元中各创建一叠牌，每叠 13 张，且花色相同。每叠牌必须按从小 (A) 到大 (K) 的顺序排列。

* **如何玩牌**

空当接龙仅使用一副牌玩，共 52 张，牌的正面朝上，排成八列。通过从这八列中移牌来创建四叠牌：

左上角是四个“可用单元”，移牌时可以在其中临时放牌。

右上角是四个“回收单元”，在其中构建获胜所需的牌叠。

如图：



* **具体玩法**

从每列底部拖牌，并按以下方式移动：

从列到可用单元。每个可用单元一次只能放一张牌。

从列到列（或从可用单元到列）。在列中必须按降序依次放牌，而且红黑花色交替。

从列到回收单元。每叠牌必须由同一花色组成，并从 A 开始。

每一列中只能移动最下面的牌，被压住的牌是不能直接移动的。左上的中转单元中的牌都可以移动 。

事先计划。深思熟虑地移动牌以建立长串，清除列，并将 A 移到回收单元中。

若有可能，尽早先移动 A。当小牌埋在列中时，最好提前移出并将其移到回收单元。

释放可用单元。可用单元越多，越好移牌。在移牌时尽量让可用单元保持为空。

清除整个列。在可能的情况下，移动某个列的每张牌，然后尽可能再以大牌开头填充该列，开头的牌不要低于 10。最好使用 K 开头。

**（2）、撤销功能：**

点击“撤销按钮”：每一次可以撤销自己上一步的操作，即可以重新执行上一步。甚至可以退回几步的操作，重新执行这些操作。

如图：



**（3）、计时器功能：**

玩家不仅要以完成游戏为目标，由于采用排行榜机制，所以玩家应该尽量快速的完成游戏，处于提醒玩家现阶段用时的目的，设计了计时器功能，在游戏界面的左上角存在计时器实时显示游戏时间，对玩家进行提醒。

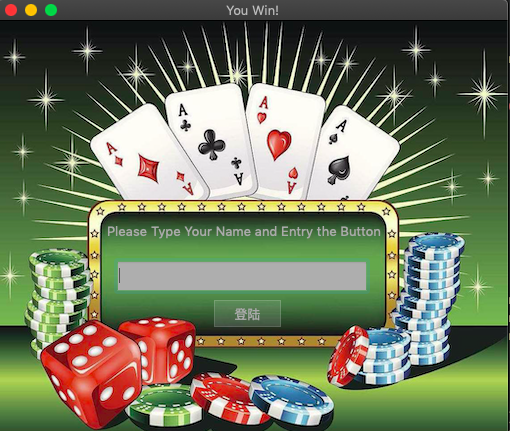
如图：



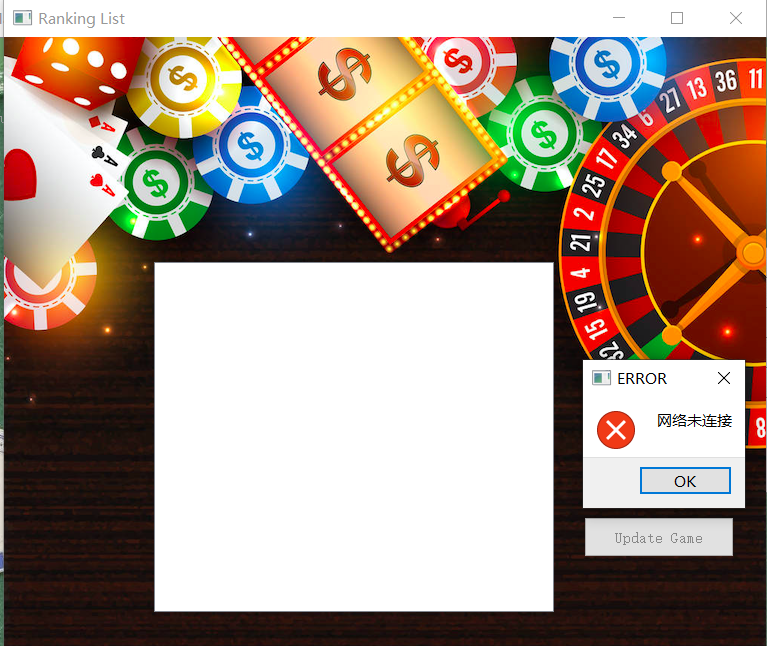
**（4）、排行榜功能：**

在每一局游戏结束之后，会有输入登录用户名的窗口，提示玩家输入自己的用户名：

如图：



点击LOGIN按钮，即可将玩家的信息传送至后台，对玩家的排名进行更新，弹出如下排行榜：



按下Update Game按钮，即可启动下一局游戏。

1. **产品使用说明 安装 使用**

本产品开放release版本供用户下载使用，后台服务器支持5000人同时在线，瞎子啊release版本，并且对服务器进行配置，即可使用。