**《程序设计课程设计》实验报告**

**实验名称 《课程设计-外卖派单模拟系统》<版本号:1.0>**

**班 级 2018211318**

**组 号 （按照网站上各组对应的小组序号）**

**姓 名 凌国瀚 张北辰 张开元**

# 1．用户界面设计

## 1.1 开始界面

运行程序后首先出现的界面，如图1-1

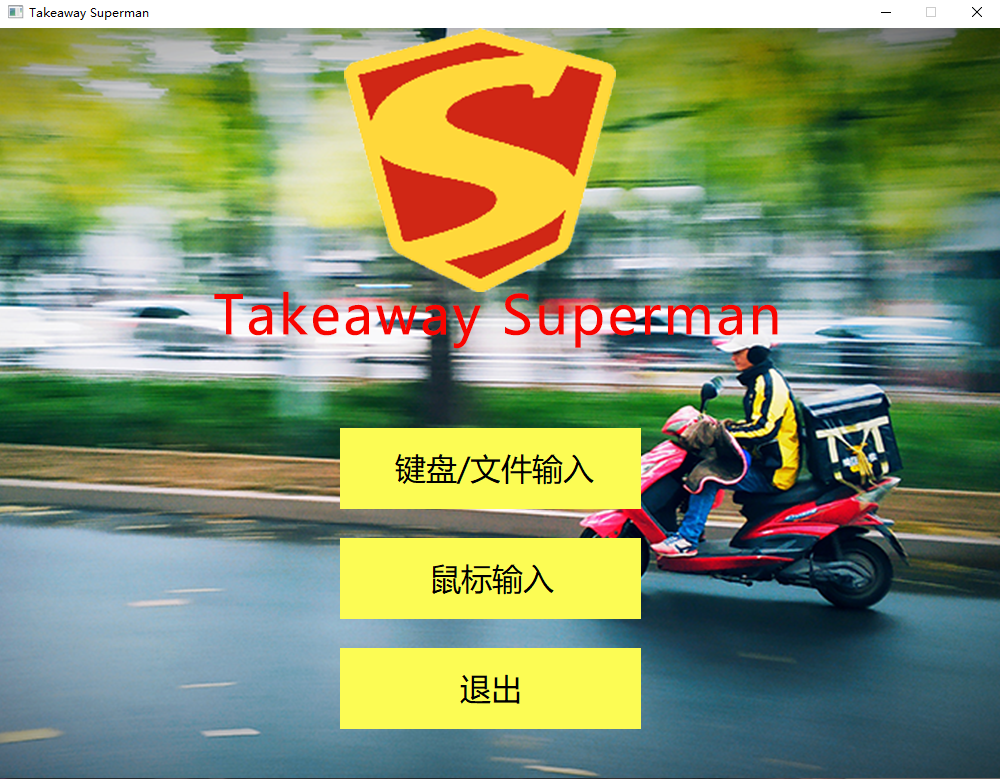


图1-1

该节目为开始界面，在logo与标题下方是三个按钮，按下对应按钮后，相应按钮会变成红色

按下第一个按钮（如图1-2），进入键盘/文件输入模式，检测doc文件夹内是否有标准文件名“sales.txt”,若有则自动进入订单强制文件输入方式。没有则在控制台中选择从键盘输入或是从鼠标输入（如图1-3）。（注：实际操作中，开始界面会被关闭）



图1-2



图1-3

按下第二个按钮（如图1-4），直接进入鼠标输入模式，运行图形界面鼠标输入

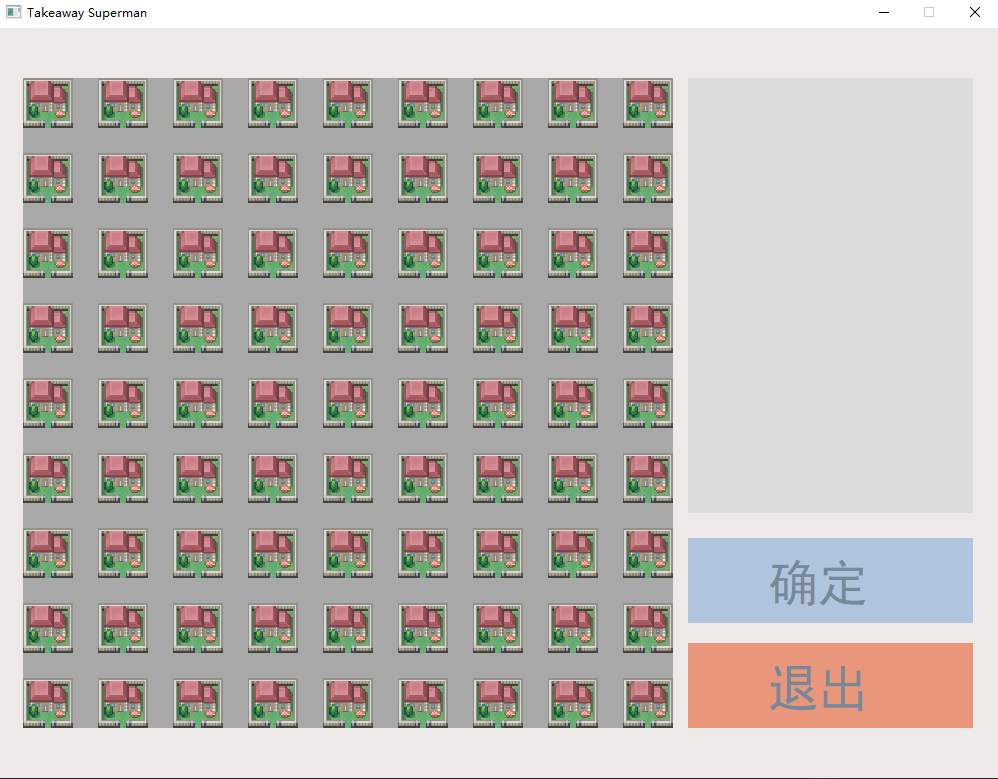


图1-3

按下第三个按钮，退出程序

## 1.2 图形（鼠标）操作界面

// TODO:

# 2. 高层数据结构设计

(包括：重要的数据常量定义、数据变量定义，即各模块要共享的数据类型和参数设计，相当于头文件内容，加文字描述)

## 2.1全局常量定义

const int TIME\_UNIT = 2; // 时间单位为TIME\_UNIT秒

const int INIT\_MONEY = 1000; // 起始资金

const int HIRE\_MONEY = 300; // 招募一个人需要的资金

const int DEAD\_TIME = 3; // 超过DEAD\_TIME时间单位后，被视为拒单

const int FINE\_FIRST\_TIME = 30; // 超时时间，第一次超时罚款

const int FINE\_MONEY = 50; // 第一次超时罚款金额

const int FINE\_SECOND\_TIME = 60; // 超时时间，第二次超时倒闭

const int MONEY\_GAIN\_ONE = 10; // 完成一单获得的钱

const int MAP\_X = 17; // 地图X大小

const int MAP\_Y = 17; // 地图Y大小

const int COMPANY\_X = 7; //公司所在X大小

const int COMPANY\_Y = 8; //公司所在Y大小

## 2.2 全局变量定义

/\*

为区分全局常量，全局变量命名如下所示

\*/

int CompanyMoney = INIT\_MONEY; // 当前公司有多少钱

int CompanyRiderCount = 0; // 公司骑手数量

int Time = -1; // 当前时间，即Time个时间单位

int CompanyOrderSum = 0; // 公司接单数

int CompanyOrderFinish = 0; // 公司完成单数

int CompanyOrderOverTime = 0; // 公司超时单数

OrderList \*AllOrderLog; // 全部订单记录(含有头结点)

OrderList \*Buffer; // 缓冲区(含有头结点)

RiderList \*AllRiderLog; // 全部骑手记录(含有头结点)

int Map[17][17]; // 输出用，0为道路，1为普通房屋，2为餐厅，3为宿舍，4为骑手

## 2.3 数据结构定义

/\*

全局结构体

注意：本项目所有链表都具有空头结点！！！

\*/

typedef struct \_Order // 订单

{

int id; // 订单ID

int begin\_time; // 起始时间戳

int warn\_time; // 超时时间戳

int end\_time; // 截止时间戳

int rest\_x, rest\_y; // 餐厅位置

int cust\_x, cust\_y; // 顾客位置

int status; // 订单状态 0未接单，1取餐，2送餐，3完成

} Order;

typedef struct \_OrderList // 订单链表

{

struct \_OrderList \*Pre\_order; // 上一个订单

struct \_OrderList \*Nxt\_order; // 下一个订单

Order \*Cur\_order; // 当前订单

} OrderList;

typedef struct \_Rider // 骑手

{

int id; // 骑手ID

int rider\_x, rider\_y; // 骑手当前位置

OrderList \*Bag; // 骑手背包订单

//Order \*Cur\_order; // 骑手当前订单

} Rider;

typedef struct \_RiderList // 骑手列表(便于遍历)

{

struct \_RiderList \*Pre\_rider; // 上一个骑手

struct \_RiderList \*Nxt\_rider; // 下一个骑手

Rider \*Cur\_rider; // 当前骑手

} RiderList;

# 3. 系统模块划分

## 3.1 系统模块结构图

本程序模块结构图如图3-1

模块主要分为三个部分，函数Functions，变量常量与头文件Global，输入输出IO

各个模块功能如下：

1. **模块名称：main.cpp**

**模块功能：**程序主函数与入口，有创建开始界面、运行文件/控制台输入主程序、调用鼠标输入主程序的功能

1. **Functions部分**
   1. **模块名称：algorithm.h**

**模块功能：**计算骑手到目的地的距离，寻找处理某个订单的最佳骑手，且提供方法将订单放入棋手背包。

* 1. **模块名称：creatList.h**

**模块功能：**建立空的头结点，即链表初始化。此文件为工具集。

**c) 模块名称：gameOver.h**

**模块功能：**破产与胜利结算，并释放程序的所有内存。

**d) 模块名称：listOperation.h**

**模块功能：**链表操作，方便其他模块对链表的删减与更改。

**e) 模块名称：orderFunctions.h**

**模块功能：**有关订单的函数集合，有完成判断订单超时，完成订单，派单等订单操作的功能。

**f) 模块名称：riderFunctions.h**

**模块功能：**有关骑手的函数集合，能够实现骑手招募，返回骑手具体信息，控制骑手移动等功能。

**g) 模块名称：pairtest.h**

**模块功能：**鼠标输入的图形界面的主程序，主要负责完成图形界面的各种操作。

**3. Global部分**

**a) 模块名称：consts.h**

**模块功能：**声明全局常量与全局变量。

**b) 模块名称：header.h**

**模块功能：**引用标准库与ege库的头文件，同时链接各个模块，使得main函数能统一调度。

**c) 模块名称：init.h**

**模块功能：**初始化地图二维数组与骑手的背包。

**d) 模块名称：structure.h**

**模块功能：**声明全局数据结构。

**e) 模块名称：unistd.h**

**模块功能：**该头文件在mingw64中可用，为了使其在Visual Studio中可用，自行封装了该头文件。

1. **IO部分**
   1. **模块名称：input.h**

**模块功能：**能够分别实现从文件输入与从控制台输入

* 1. **模块名称：output.h**

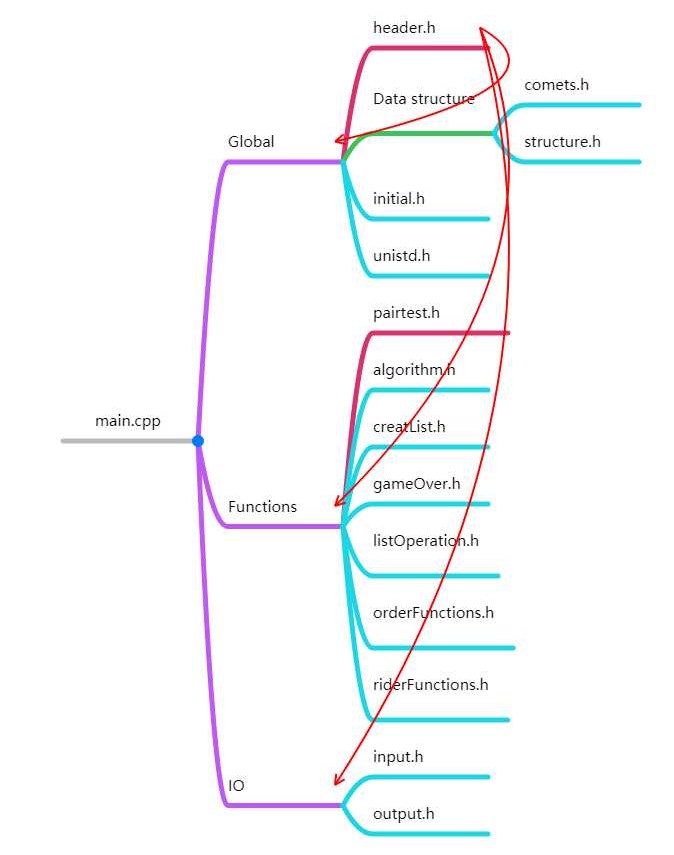
**模块功能：**能够分别实现字符画输出与文件输出

图3-1

## 3.2各模块函数说明

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **模块名称** | **函数原型** | **功能** | **参数** | **返回值** |
| **algorithm.h** | void assignOrder(Order \*AOrder, int RiderID); | 把订单放入对应骑手的背包 | Order类型订单，骑手ID | 无 |
| int virtualRun(int \*cur\_x, int \*cur\_y, int pos\_x, int pos\_y) | 统计模拟跑到目的地的步长，算法与riderMove同 | int \*cur\_x, int \*cur\_y:骑手当前坐标指针；  int pos\_x, int pos\_y:骑手目的地坐标 | 模拟步长的值 |
| int getDis(int a\_x, int a\_y, int b\_x, int b\_y); | 获取两点之间的曼哈顿距离 | int a\_x, int a\_y, int b\_x, int b\_y:两点的坐标 | 两点的曼哈顿距离 |
| **creatList.h** | OrderList \*creatOrderList(); | 初始化订单链表 | 无 | 返回订单链表的头结点 |
| RiderList \*creatRiderList(); | 初始化骑手链表 | 无 | 返回骑手链表的头结点 |
| **gameOver.h** | int isAllOrderFinished(); | 判断每个订单是否完成 | 无 | 完成返回1，未完成返回2 |
| void gameOver(int reason); | 破产结算 | 破产原因代码 | 无 |
| void gameSuccess(); | 胜利结算 | 无 | 无 |
| **listOperation.h** | void pop\_front\_order(OrderList\* List); | 传入链表，删除头部订单 | OrderList\* List: 订单链表 | 无 |
| void push\_back\_order(Order\* NewOrder, OrderList\* List); | 添加尾部订单 | Order\* NewOrder:要添加的订单  OrderList\* List:  要插入的订单链表 | 无 |
| void push\_back\_rider(Rider\* NewRider, RiderList\* List); | 骑手队列尾部添加骑手 | Rider\* NewRider:要添加的骑手  RiderList\* List:要插入的骑手链表 | 无 |
| void swap\_order(OrderList\* List1, OrderList\* List2); | 订单列表交换 | OrderList\* List1, OrderList\* List2:要交换的两个链表元素 | 无 |
| OrderList\* delete\_order(OrderList\* tOrder); | 删除订单，返回该订单的前一个订单 | OrderList\* tOrder:目标订单 | 返回该链表中前一个订单的指针 |
| **orderFunctions.h** | void isAnyOrderOverTime(); | 每次刷新时间点后判断是否有超时订单，若破产则结束游戏 | 无 | 无 |
| OrderList\* complishOrder(OrderList\* nowOrder) | 完成订单，在结单时调用 | OrderList\* nowOrder:待处理的订单，注意，为OrderList指针而非Order指针 | 若完成，返回上一个订单，未完成仍返回该订单 |
| int isComplishOrder(OrderList\* NowOrder, RiderList\* NowRider); | 判断订单是否结单 | nowOrder(OrderList \*)：目前遍历到的订单，OrderList指针而非Order指针；  NowRider(Rider \*)：目前的骑手，为RiderList指针 | 结单返回1，未结单返回2 |
| void arrangeNewOrder(); | 将新订单进行派单 | 无 | 无 |
| **riderFunctions.h** | int hireRider(); | 招募骑手,返回骑手的id，注意，前提条件是CompanyMoney>=400 | 无 | 招募的骑手的id |
| Order\* getRiderCurOrder(int rider\_id); | 输入骑手id返回骑手正在进行的Order的指针 | int rider\_id:骑手id | 骑手当前订单的指针 |
| OrderList\* getRiderCurBag(int rider\_id); | 输入骑手id返回骑手当前背包的OrderList指针 | int rider\_id:骑手id | 骑手当前背包的指针 |
| void getRiderCurPos(int rider\_id, int\* x, int\* y); | 返回骑手位置，注意传入的坐标为指针 | int ider\_id:骑手id  int \*x:骑手横坐标指针  int \*y:骑手纵坐标指针 | 无 |
| void riderMove(int rider\_id, int pos\_x, int pos\_y); | 输入骑手id和目的地（房子坐标），根据骑手当前位置，先左右，后上下，移动一个单位，默认已经判断是否停靠，不再重新检查 | int rider\_id:骑手id  int pos\_x:目的地横坐标  int pos\_y:目的地纵坐标 | 无 |
| void AllRiderMove(); | 所有骑手进行移动，作为主函数的一部分,目前的策略是派背包里的第一个单 | 无 | 无 |
| **pairtest.h** |  |  |  |  |
| **init.h** | void initMap(); | 初始化地图 | 无 | 无 |
| void initRiderBag(); | 初始化骑手背包，初始化结束时骑手背包都是未完成订单 | 无 | 无 |
| **input.h** | int inputFileOrder(); | 从文件输入 | 无 | 有输入文件返回1，无输入文件返回0 |
| void inputKeyOrder(); | 从键盘输入 | 无 | 无 |
| **output.h** | void printNearBy(Rider\* nowRider, FILE\* fPtr); | 做停靠判断并输出至文件 | Rider\* nowRider:目标骑手  FILE\* fPtr:输出文件 | 无 |
| void outputOnFile(); | 在文件输出 | 无 | 无 |
| void outputKey(); | 屏幕上输出文字 | 无 | 无 |
| void outputMap(); | 在屏幕上输出地图字符画 | 无 | 无 |
| **main.cpp** | void console(); | 文件/控制台输出主程序 | 无 | 无 |
| int startScreen(); | 开始界面 | 无 | 返回0为文件/控制台输入，返回1为鼠标输入，返回其他值退出程序 |
| int main(); | 程序入口 | 无 | 无实际意义 |

## 3.3 函数调用图示及说明

**调用说明：**

Main函数为程序的入口，进入main函数后，首先加载开始界面startScreen，接受用户鼠标数据，选择进入鼠标输入或文件/控制台输入。

若选择鼠标输入…

若选择文件/控制台输入，进入console函数，通过input模块，判断文件输入或是控制台输入。输入数据后进入主线程循环，循环退出条件为所有订单完成或无数据，循环首先通过isAllOrderFinished判断超时订单，接着依据当前现金通过hireRider雇佣棋手，然后通过initRiderBag处理订单更新背包，背包更新完成后，通过AllRiderMove移动骑手。程序还将通过arrangeNewOrder处理订单，循环末尾判断是否有完成订单并通过output模块执行相关输出。游戏正常结束时（即主循环退出时），通过gameSuccess进行结算。若游戏进程中发生意外或者游戏失败，则通过gameOver进行破产结算。

int main();

int startScreen();

void mainFunction();

void console();

input.h

isAllOrderFinished();

hireRider();

isAnyOrderOverTime();

AllRiderMove();

initRiderBag();

arrangeNewOrder();

isComplishOrder ();

output.h

gameOver();

gameSuccess();

# 4. 高层算法设计

## 4.1骑手移动

void riderMove()

{

if 骑手和目的地不在同一行

{

if在横向道路上

{

if在目的地右边

往左走

else if在目的地左边

往右走

}

else if在纵向道路上

{

if在目的地右边就要往左走

{

if目标在下或同行

{

骑手x++

骑手y--

}

else // 目标在上

{

骑手x--

骑手y--

}

}

else if在左边就要往右走

{

if目标在下或同行

{

骑手x++

骑手y++

}

else// 目标在上

{

骑手x--

骑手y++

}

}

}

else // 异常情况处理

{

gameOver(5);

}

}

else if 骑手和目的地不在同一列

{

if在纵向道路上

{

If目的地在下边

往上走

else if 目的地在上边

要往下走

}

else if在横向道路上

{

if在下边就要往上走

{

if目标在右或同列

{

骑手y++

骑手x--

}

else // 目标在左

{

骑手y--

骑手x--

}

}

if在上边就要往下走

{

if目标在右或同列

{

骑手y++

骑手x++

}

else // 目标在左

{

骑手y--

骑手x++

}

}

}

else // 异常情况处理

gameOver(5);

}

}

## 4.2 停靠判断并输出

void printNearBy()

{

int i;

const int dx[] = { -1, 1, 0, 0 };// x向量

const int dy[] = { 0, 0, 1, -1 };// y向量

int flag = 0;

int restFlag = 0, dustFlag = 0;

for (i = 0; i <= 3; i++)// 四个方向

{

while (遍历元素不为NULL)

{

if此位置房子作为餐厅有任务完成

{

flag = 1;

restFlag = 1;

}

if此位置房子作为宿舍有任务完成

{

flag = 1;

dustFlag = 1;

}

if (dustFlag && restFlag)

输出餐客坐标

else if (restFlag)

输出餐厅坐标

else if (dustFlag)

输出食客坐标

遍历链表

}

}

If无停靠信息

仅输出分号

}

## 4.3 找到某个订单的最佳派单手

void getBestRider()

{

while (遍历骑手)

{

通过getPath获得把包裹给此骑手的时长

if 判断是否为最快路径

{

是最快路径就暂定最佳骑手

}

遍历链表

}

通过assignOrder将订单加入骑手背包

}

**教师评语：**