**贪食蛇案例**

# 1 案例介绍

## 1.1 贪食蛇游戏简介

贪吃蛇是一款经典的小游戏。玩家使用方向键操控一条长长的蛇不断吞下食物，同时蛇身随着吞下的食物不断变长，当蛇头撞到蛇身或墙壁时游戏结束。贪吃蛇最初为人们所知的是诺基亚手机附带的一个小游戏，它伴随着诺基亚手机走向世界。现在的贪吃蛇出现了许多衍生版本，并被移植到各种平台上。

## 1.2 项目截图



## 1.3游戏元素

游戏中元素分为：墙壁、蛇、食物以及蛇的可行区域和右侧的版本号和游戏玩法提示

**墙壁**：

\* 星号表示，代表一个区域范围，也就是蛇的可移动区域，蛇如果碰到墙壁视为死亡，也就是GameOver！

**蛇**：

分为蛇头 蛇身，蛇头用@符号表示，蛇身用 = 等号表示，当蛇吃到食物时候，蛇身+1，意味着身长度变成。贪食蛇可以通过不断地吃食物来增加自己的身体

**食物：**

#井号表示，蛇碰到食物会将食物吃掉

**可移动区域**

空格 表示，代表蛇可以移动的区域

**提示信息**

右侧展示，可以显示当前贪食蛇版本号、制作人员、游戏玩法等提示信息！

## 1.4 游戏规则

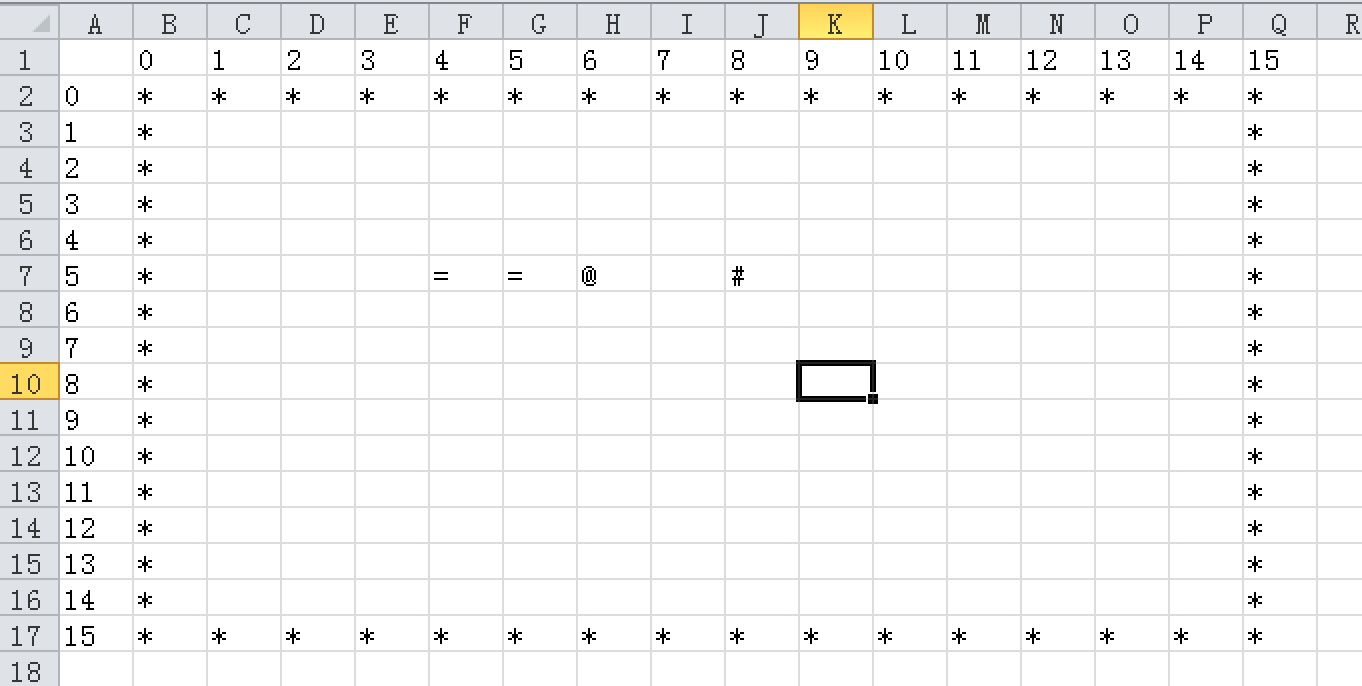
当运行起游戏时候，游戏画面是静止不动的，可以默认如上图中，蛇头朝右，游戏中设置了 w s a d 4个按键分别代表，上、下、左、右，也是用户比较常用的方向按键，当用户输入 w或者s或者d时候激活游戏，注意输入a不可以激活，因为蛇不可以180°转弯，因此**蛇的移动方向只可以一直向前或者90°旋转。**

当蛇吃掉食物时候，此时蛇会增加一个身段，另外食物需要重新随机的设置到屏幕上。

游戏结束方式有两种：一、蛇碰到墙壁视为死亡；二、蛇头碰到蛇身子，把自己吃掉也视为死亡。

# 2 实现思路

## 2.1 利用Excel模拟游戏



我们实现游戏，并非直接上代码，而是要先分清楚游戏的需求、玩法、和游戏中各个元素的作用以及游戏规则后才可以考虑动手

在没有任何成品的情况下，我们可以先通过画图或者电脑上PS或者有相关的游戏策划师，游戏美术人员提供的图片来分析游戏

在本案例中，我们可以先通过一个Excel表来分析游戏，因为恰巧Excel也是网格的，和我们的游戏和类似，其他游戏就需要根据不同的类型来用不同的图片模拟游戏了。

在Excel表中我们可以看到，模拟的游戏分为了横向和纵向，游戏内部可以看成一个二维数组，我们只是不断的去修改二维数组的内容，然后显示到屏幕上即可。

## 2.2 游戏搭建

很明显游戏分为两种状态，第一种就是静止态，也就是进入游戏时候的展示画面，在这个画面下，所有游戏元素都是静止不动的，二另一种就是激活游戏后开始移动游戏的状态

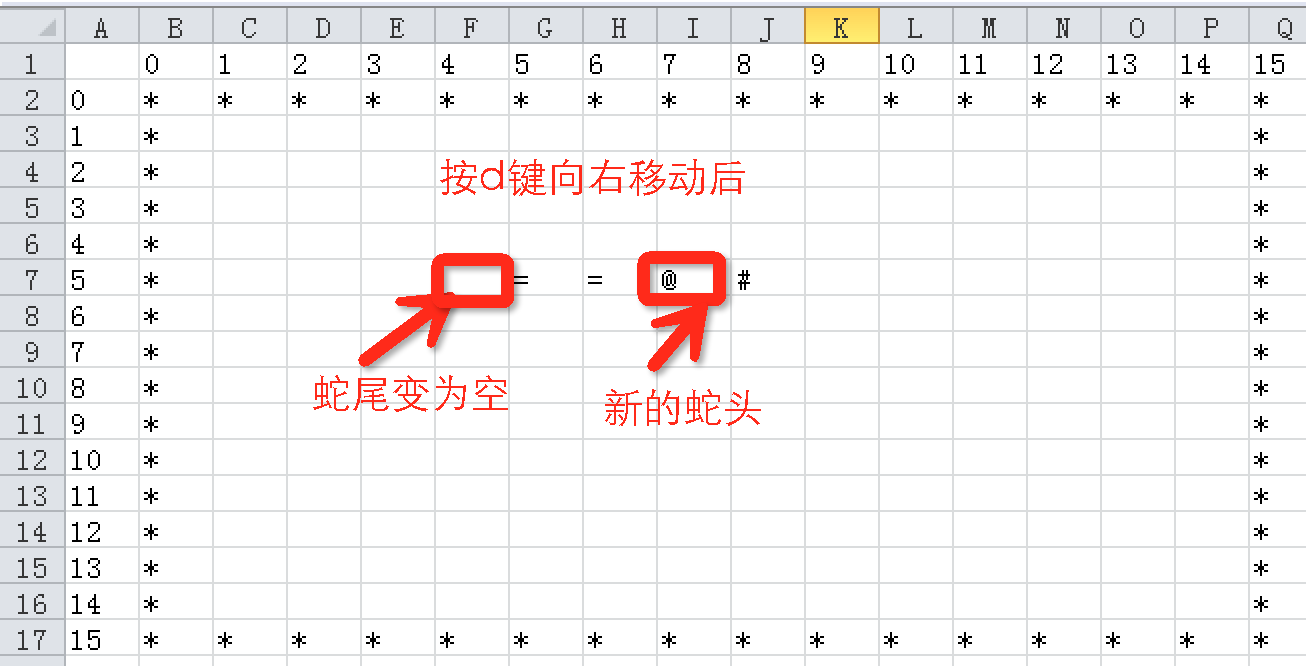
## 2.3 游戏移动

当激活游戏后，也分为两种，一种是死亡，这个是当蛇头碰到蛇身或者是碰到墙壁者两种死亡，这个我们暂时先不考虑，第二种是正常移动，那么我们先分析下正常移动

在正常移动时候，也分为两种状态

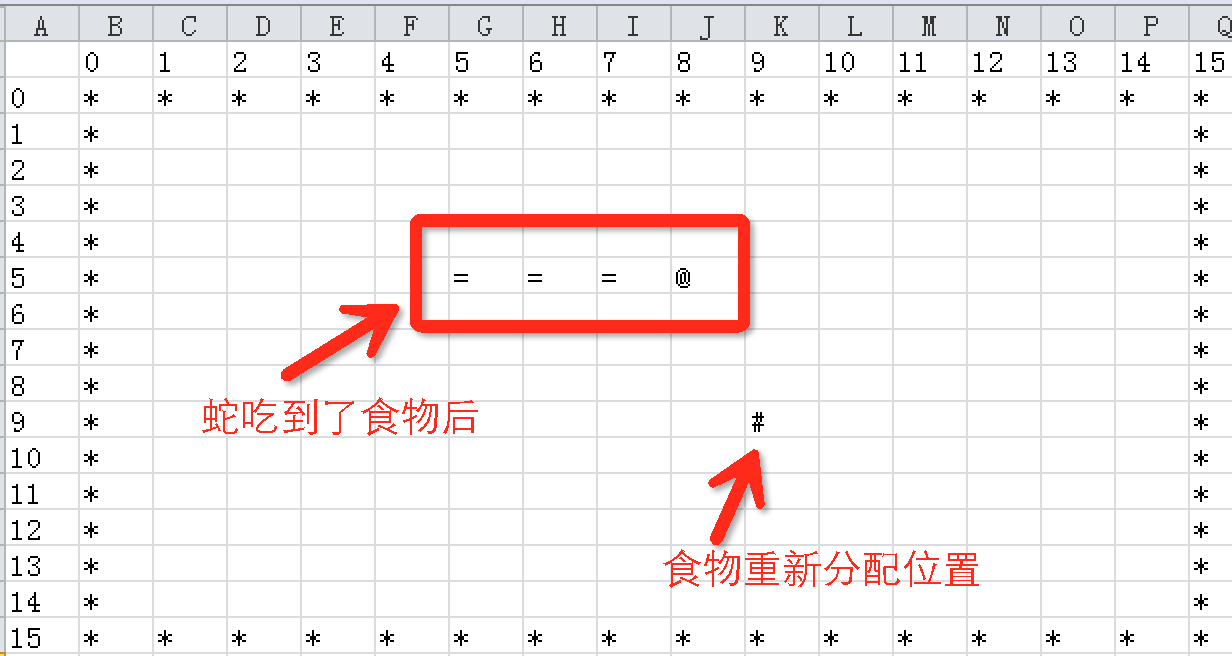
第一种：蛇没有吃的食物

这个时候，蛇只是单纯的移动，比如上述的蛇向前移动后，应该到对应的第五行第7列的位置，离食物很近了，但是还没有吃到，这时候，蛇应该更新蛇头的位置，并且将之前蛇尾巴的位置置为空格，也就是表示向前移动。



第二种：蛇吃到食物

如上图，这时候如果继续向右行走，蛇将吃到食物，当吃到食物时，当前的蛇头的位置应该为之前食物的位置 ，那么蛇尾由于吃到了食物，就还是在原有位置，然后食物再重新分配到一个其他的位置，这个位置不能是蛇、也不能是墙



# 3 具体实现

## 3.1 代码分析

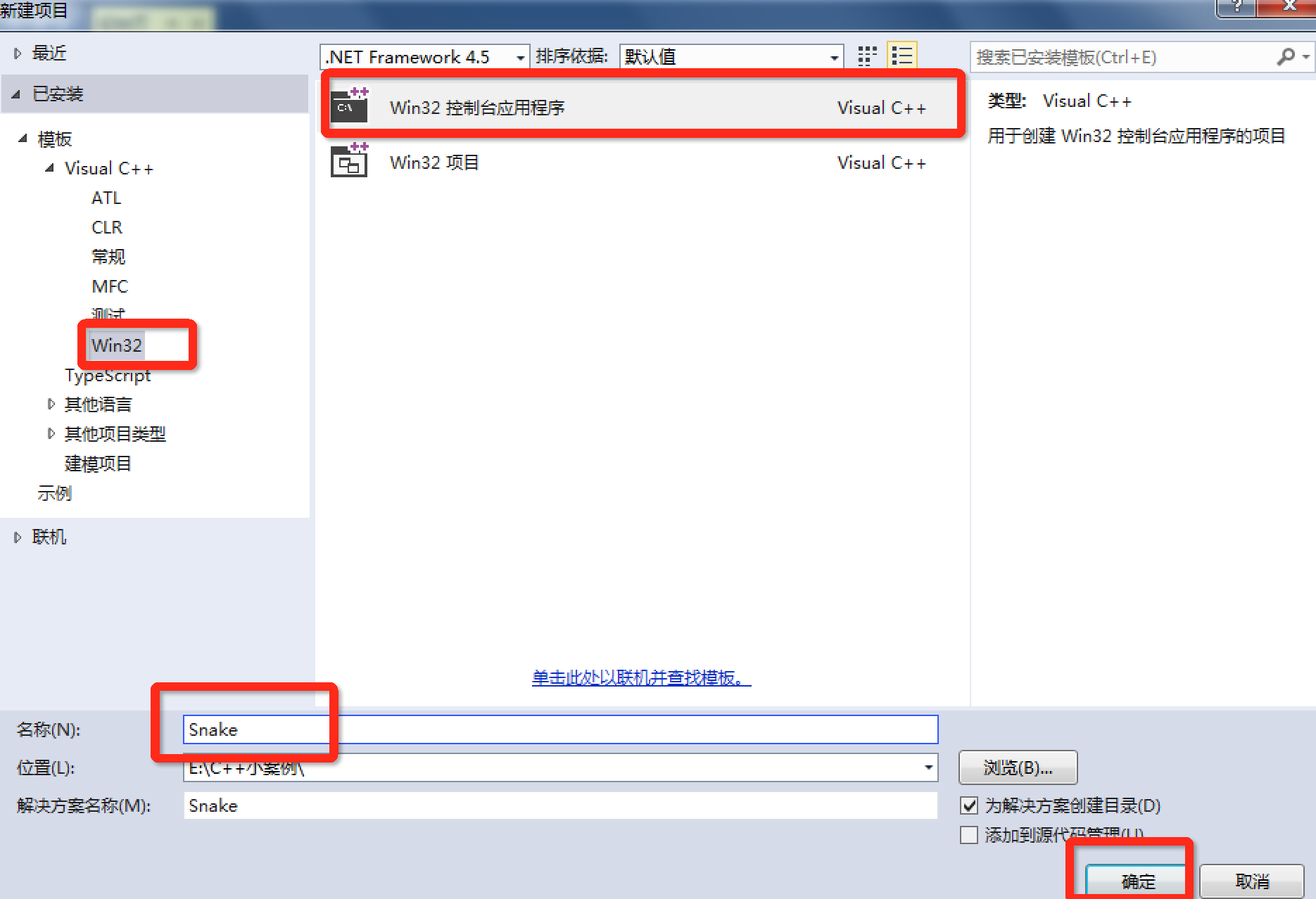
当模拟好了游戏玩法和分析透彻游戏后，再分析下利用代码该如何实现游戏，也就是分析如何敲代码，怎么敲，在上面的分析思路中，如果我们想利用C++的面向对象的思想来完成该项目，就应该把每个模块划分开，让各个功能之间相互依赖又分离实现。

那么总结一下我们应该具体来完成哪些模块

首先可以创建项目时候，有个程序入口，这个文件可以写为Game.cpp，游戏中的主要部分，在这个模块中主要用户接受用户的输入，调度其他模块来完成一个总指挥的作用！其次我们可以从最简单的模块入手，墙是最简单的，内部只要能维护一个二维的数组即可，然后是蛇模块，负责管理蛇的一切行为，最后还有个食物模块。

总结：游戏中模块分为：主的程序入口、墙、蛇、食物。

## 3.2 创建项目

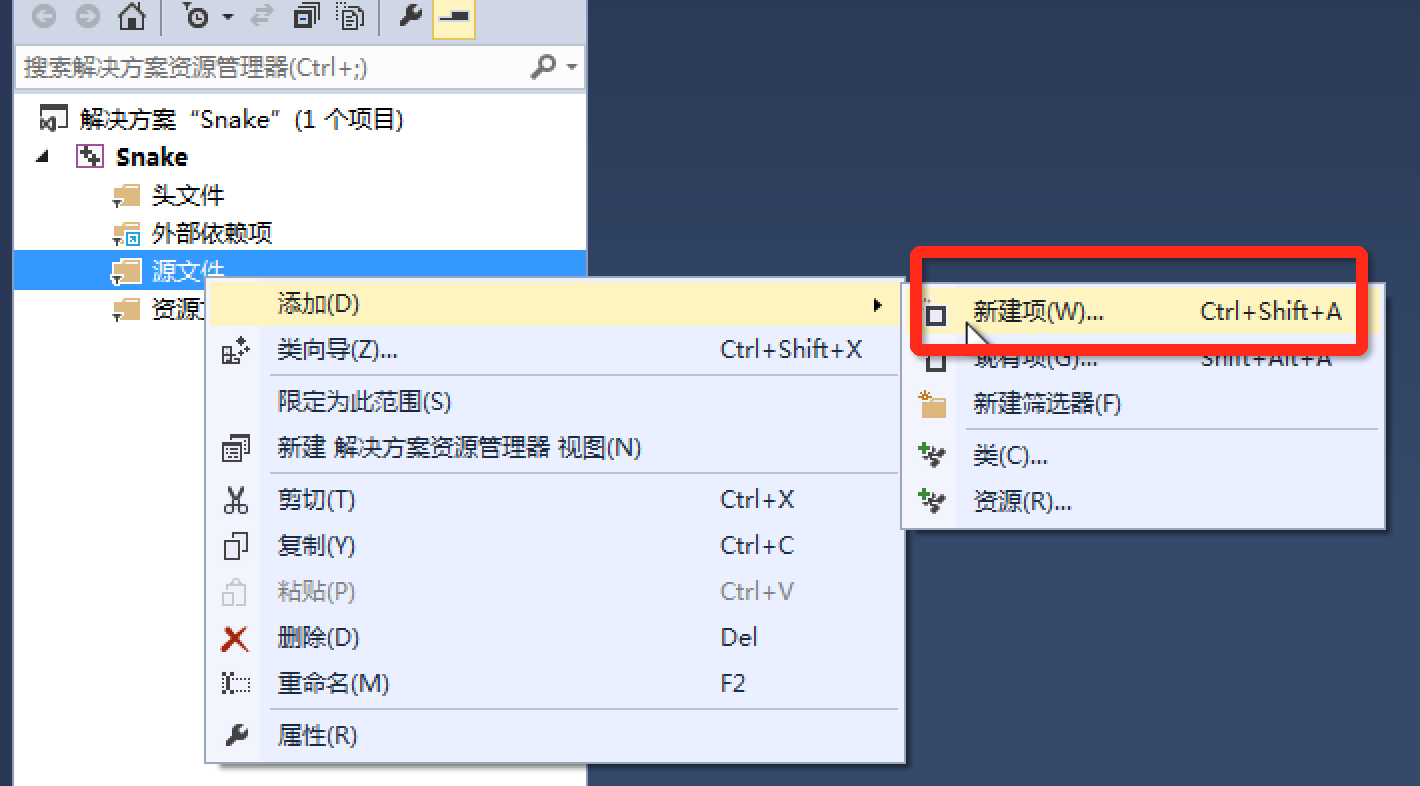
分析好了所有需求后开始实现游戏，第一步创建项目，打开VisualStudio，点击新建项目

创建控制台应用程序，名称为Snake，点击确定，之后点击下一步，创建一个空项目

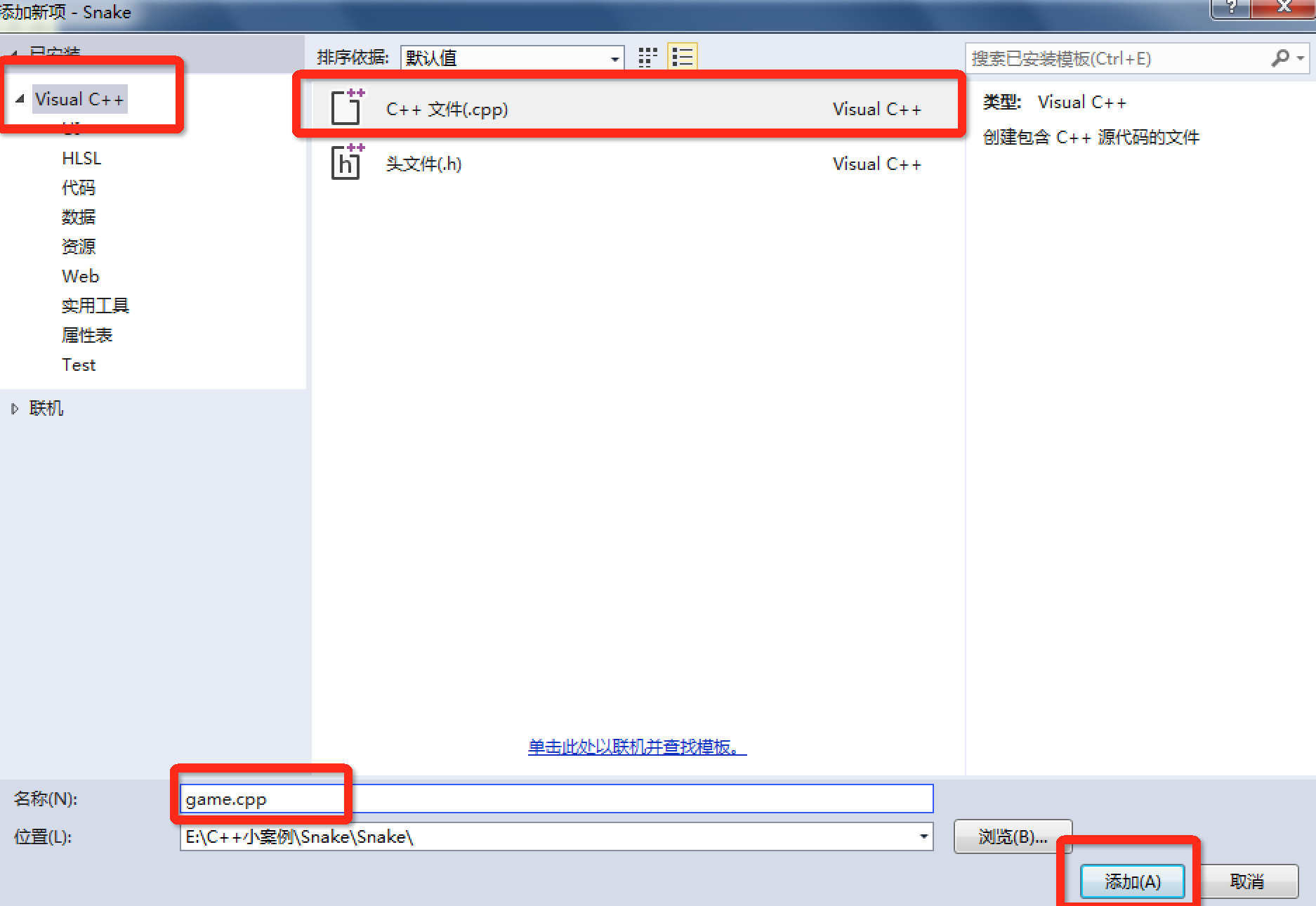


点击完成。

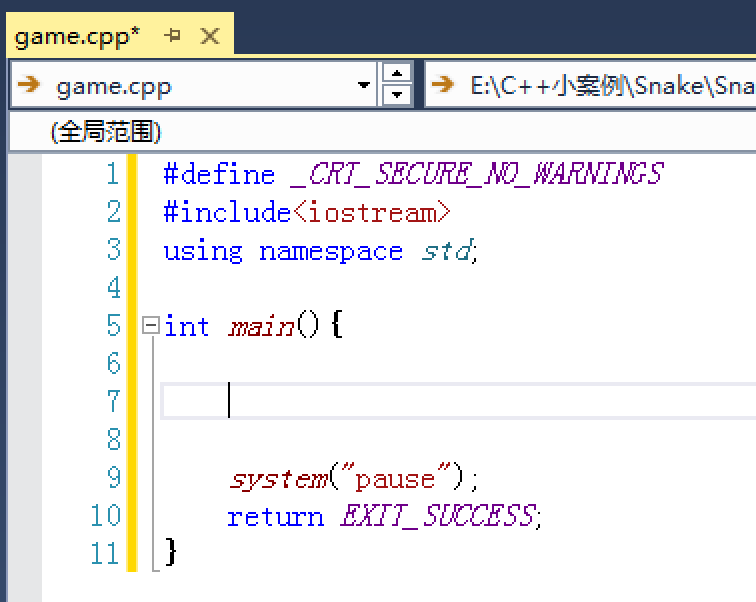
添加新建项



添加主模块 game.cpp



然后添加程序入口函数，也就是main函数如下图

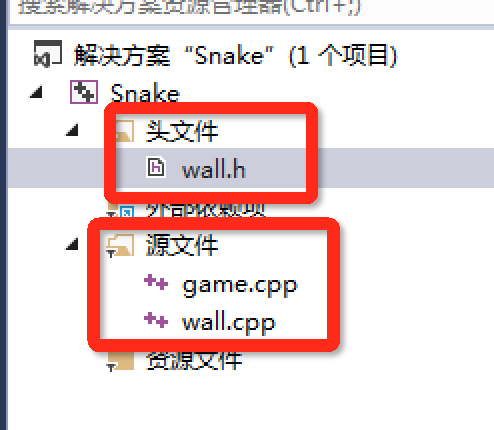


## 3.3 墙模块

下面我们开始制作游戏中墙模块的开发，首先经过分析，我们可以得出在墙模块中，我们需要维护一个二维数组，对整个游戏中的元素进行设置，所以我们可以声明一个二维数组：char gameArray[][],具体的行数和列数可以定义出一个枚举，比如本游戏中设置的是26行，26列，enum{ ROW = 26, COL = 26};

那么墙模块开发阶段，需要提供的主要接口是 初始化墙initWall, 以及打印墙，也就是将二维数组中的内容打印到控制台中，draw方法。当然对外还要提供出一个可以修改二维数组元素的方法以及根据索引获取二维数组元素的方法：getWall ，setWall；

### 3.3.1创建墙类 wall.h和wall.cpp



### 3.3.2 .h中声明墙模块中需要声明的成员方法和成员属性

#ifndef \_WALL\_HEAD

#define \_WALL\_HEAD

#include <iostream>

using namespace *std*;

class Wall

{

public:

enum

{

ROW = 26, //行数

COL = 26 //列数

};

Wall(); //构造

~Wall(); //析构

void initWall();//初始化数组

void draw(); //将数组画到控制台中

void setWall(int x, int y, char key); //根据索引设置二维数组中的元素

char getWall(int x, int y); //根据索引获取二维数组中的元素

private:

char gameArray[ROW][COL]; //维护的二维数组

};

#endif

### 3.3.3 .cpp中实现方法

Wall::Wall(){}

Wall::~Wall(){}

void Wall::initWall()

{

for (int i = 0; i < ROW;i++)

{

for (int j = 0; j < COL;j++)

{

//墙的条件

if (i == 0 || i == ROW - 1 || j == 0 || j == COL -1)

{

gameArray[i][j] = '\*';

}

else

{

gameArray[i][j] = ' ';

}

}

}

}

void Wall::draw()

{

for (int i = 0; i < ROW;i++)

{

for (int j = 0; j < COL;j++)

{

*cout* << gameArray[i][j] << ' '; //加一个空格间隙，打印出来比较好看

}

//版本信息以及其他提示信息

if (i == 5)

{

*cout* << " Snake Game V1.0" << " ";

}

if (i == 6)

{

*cout* << " Create by zt, 2016-09-10" << " ";

}

if (i == 7)

{

*cout* << " up : w" << " ";

}

if (i == 8)

{

*cout* << " down : s" << " ";

}

if (i == 9)

{

*cout* << " left : a" << " ";

}

if (i == 10)

{

*cout* << " right : d" << " ";

}

*cout* << *endl*; //打印换行

}

}

void Wall::setWall(int x, int y, char key)

{

gameArray[x][y] = key;

}

char Wall::getWall(int x, int y)

{

return gameArray[x][y];

}

### 3.3.4 game.cpp中引用并测试墙模块

main函数

int *main*(){

Wall wall; //墙对象

wall.initWall(); // 初始化墙

//测试

wall.setWall(5, 5, '@');

wall.draw(); //打印墙

*cout* << wall.getWall(0, 0) <<*endl*;

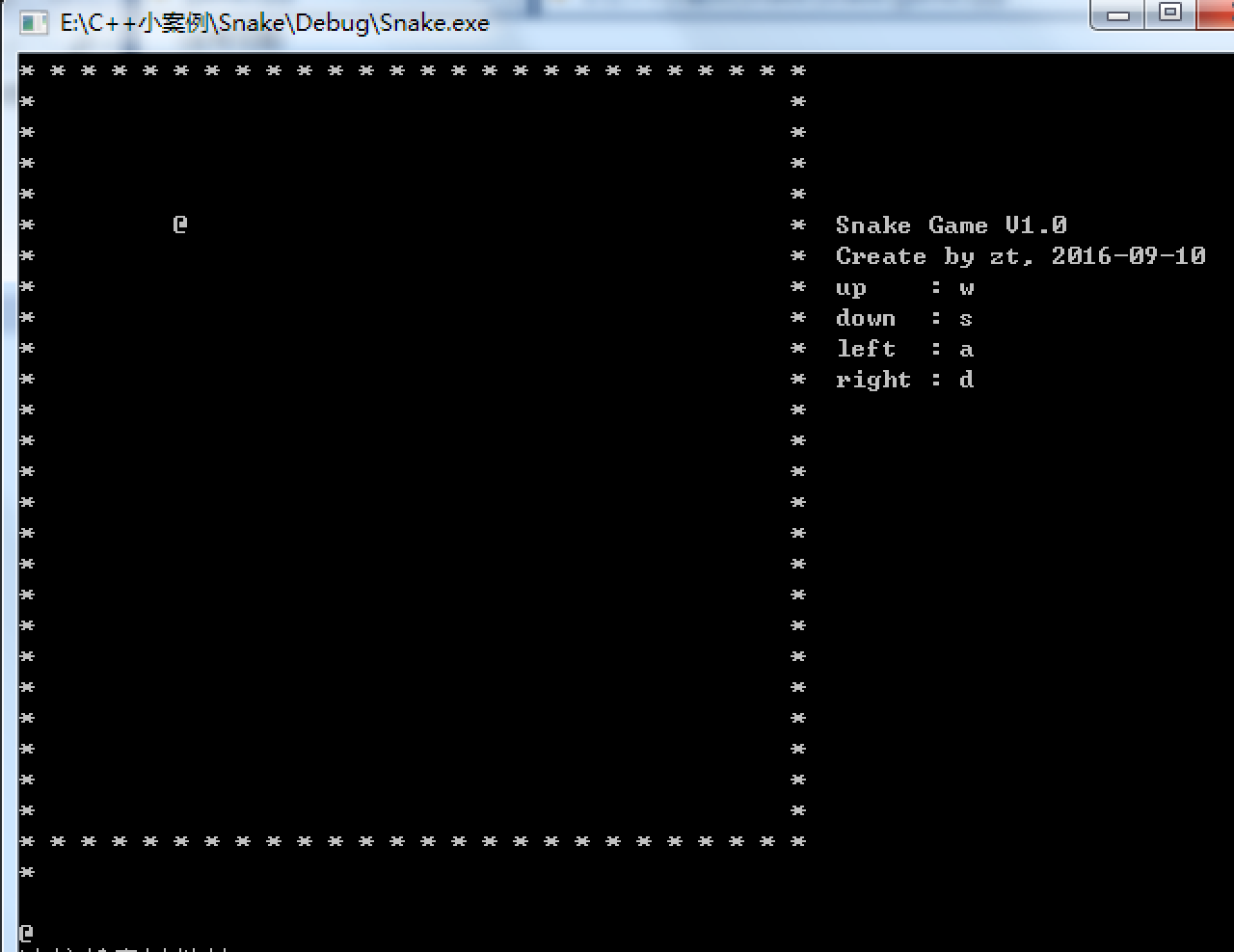
*cout* << wall.getWall(1, 1) << *endl*;

*cout* << wall.getWall(5, 5) << *endl*;

*system*("pause");

return *EXIT\_SUCCESS*;

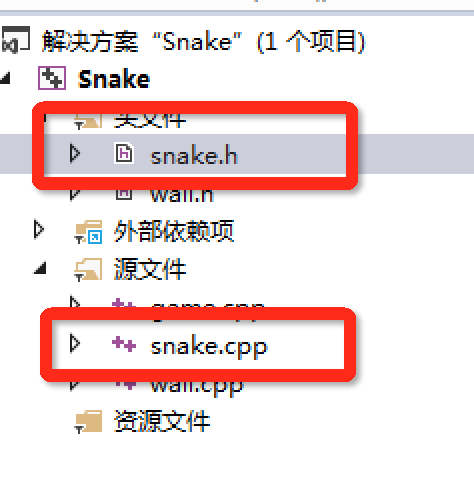
}



最后打印的 ‘\*’ ‘ ’ ‘@’是测试的对外接口，在开发中实时的测试十分重要

## 3.4 蛇模块

### 3.4.1 创建蛇类 snake.h 和snake.cpp



### 3.4.2 .h中声明成员方法和属性

首先分析下蛇模块中需要做的事情，蛇其实是一个链式结构，在本案例中，如果我们只学过C的链表，还未学习C++中的List容器，那么我们可以先从最基本的结构体来模拟蛇这个链表。

声明一个结构体 struct point ，每一个蛇的身段（节点）我们都称为一个point，在结构体中我们分为指针域（指向下一个节点）和数据域（保存具体数据）

其次，我们还需要提供一些方法将蛇初始化，initSnake;在初始化蛇中起始需要做两步操作，第一步销毁蛇所有节点，destoryPoint这一步是为了以后有可能会有重玩游戏的功能，所以我们初始化时候先清空所有蛇，第二步添加蛇节点 addPoint，这个方法就是给蛇增加蛇身的方法

当然我们还需要一个私有成员属性 Point \* pHead来记录蛇头，还一个私有成员保存墙的引用，因为我们需要将设置的内容保存到维护二维数组的墙中

#ifndef \_SNAKE\_HEAD

#define \_SNAKE\_HEAD

#include <iostream>

using namespace *std*;

#include "wall.h"

struct Point

{

//数据域

int x;

int y;

//指针域

Point \* next;

};

class Snake

{

public:

Snake(Wall &tmpWall);

~Snake();

// 初始化蛇

void initSnake();

// 销毁蛇

void destoryPoint();

// 添加蛇节点

void addPoint(int x, int y);

private:

//蛇头结点

Point \* pHead;

Wall & wall;

};

#endif

### 3.4.3 实现方法

这里我们利用初始化列表的方式，将私有成员属性墙进行赋值。

#include "snake.h"

Snake::Snake(Wall &tmpWall) :wall(tmpWall) //初始化列表获取墙对象

{

pHead = *NULL*; //构造时候头节点为空

}

Snake::~Snake()

{

}

void Snake::initSnake()

{

//销毁原来的节点

destoryPoint();

//初始化蛇,蛇头和2段蛇身

addPoint(5, 3);

addPoint(5, 4);

addPoint(5, 5);

}

void Snake::destoryPoint()

{

Point \* pCur = pHead;

while (pHead != *NULL*)

{

pCur = pHead->next;

delete pHead;

pHead = pCur;

}

}

void Snake::addPoint(int x,int y)

{

// 创建新节点

Point \* newP = new Point;

if (newP == *NULL*)

{

return;

}

// 新节点赋值

newP->x = x;

newP->y = y;

newP->next = *NULL*;

//修改原始的头为身子

if (pHead != *NULL*)

{

//设置当前的头结点为身子

wall.setWall(pHead->x, pHead->y, '=');

}

//新的节点添加到链表头部，不管有没有头结点，新节点都会是头节点

newP->next = pHead;

pHead = newP;

wall.setWall(pHead->x, pHead->y, '@');

}

### 3.4.4 测试蛇模块

在main函数中添加代码如下

int *main*(){

Wall wall; //墙对象

wall.initWall(); // 初始化墙

Snake snake(wall); //蛇对象

snake.initSnake(); //初始化蛇

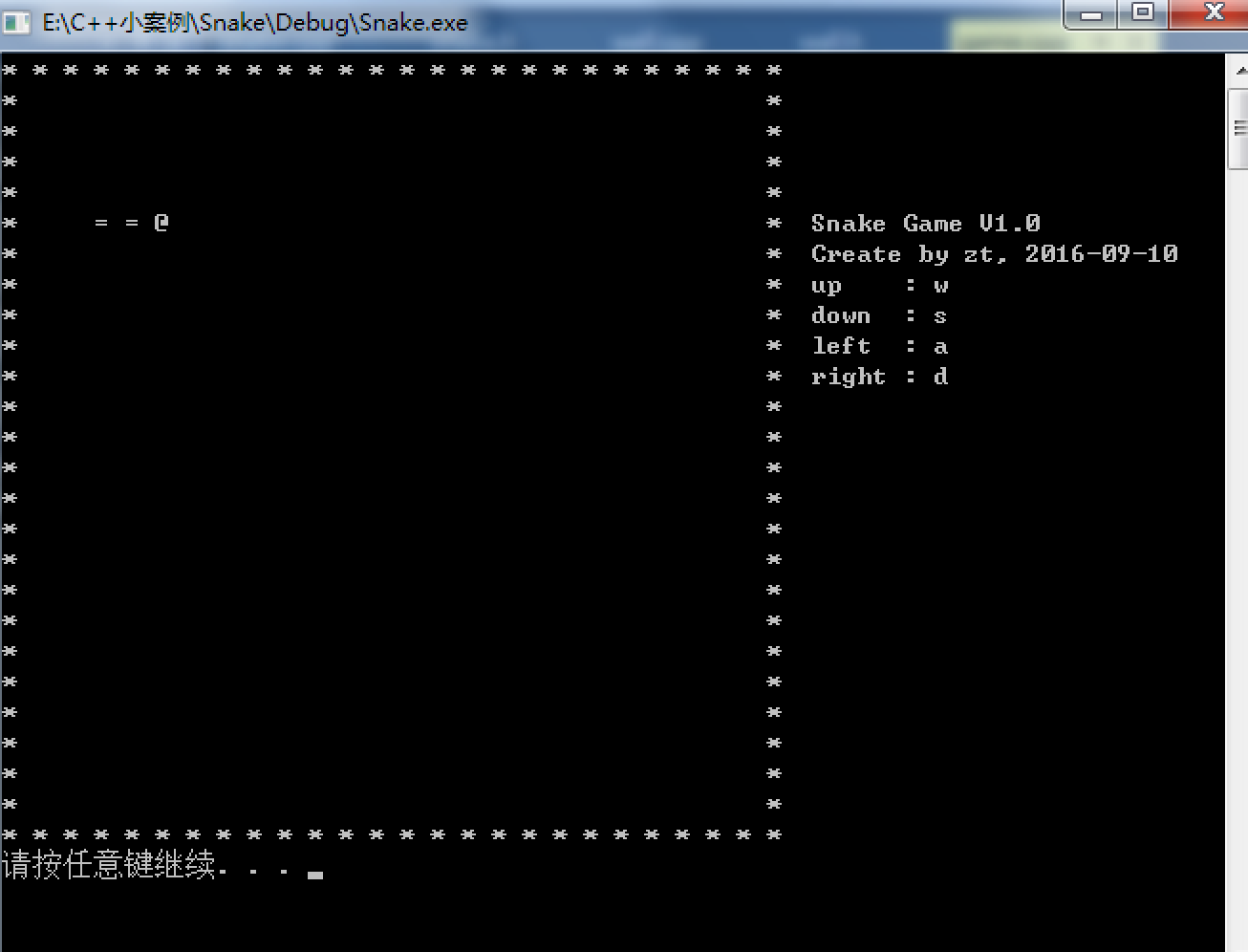
wall.draw(); //打印墙

*system*("pause");

return *EXIT\_SUCCESS*;

}

运行程序，最终可以看到一条为两个蛇身一个蛇头的蛇在屏幕中



## 3.5 食物模块

### 3.5.1 创建食物类 food.h 和 food.cpp



### 3.5.2 .h中声明食物模块中成员方法和属性

食物模块比较简单，无非是在屏幕中设置一个随机的位置做食物，但是这个位置有个要求就是不能在蛇上。

我们可以把设置食物的方法称为 setFood，当然我们可以记录这个随机的位置，称为FoodX、FoodY，由于需要设置到墙的二维数组里就需要引用墙的模块wall，因此.h中如下

#ifndef \_FOOD\_HEAD

#define \_FOOD\_HEAD

#include <iostream>

using namespace *std*;

#include "wall.h"

class Food

{

public:

Food(Wall & tmpWall);

void setFood(); //设置食物

private:

int foodX; //食物的X

int foodY; //食物的Y

Wall & wall; //墙模块引用

};

#endif

### 3.5.3 .cpp中实现方法

#include "food.h"

Food::Food(Wall & tmpWall) :wall(tmpWall)

{

}

void Food::setFood()

{

while (1)

{

//食物的随机位置

foodX = *rand*() % (wall.ROW - 2) + 1;

foodY = *rand*() % (wall.COL - 2) + 1;

//如果当前位置是可行区域，可以设置为食物并且退出循环，否则继续循环

if (wall.getWall(foodX, foodY) == ' ')

{

wall.setWall(foodX, foodY, '#');

break;

}

}

}

### 3.5.4 食物模块测试

int *main*(){

*srand*((unsigned int)*time*(*NULL*)); //随机数种子

Wall wall; //墙对象

wall.initWall(); // 初始化墙

Snake snake(wall); //蛇对象

snake.initSnake(); //初始化蛇

Food food(wall); //食物

food.setFood(); //设置食物

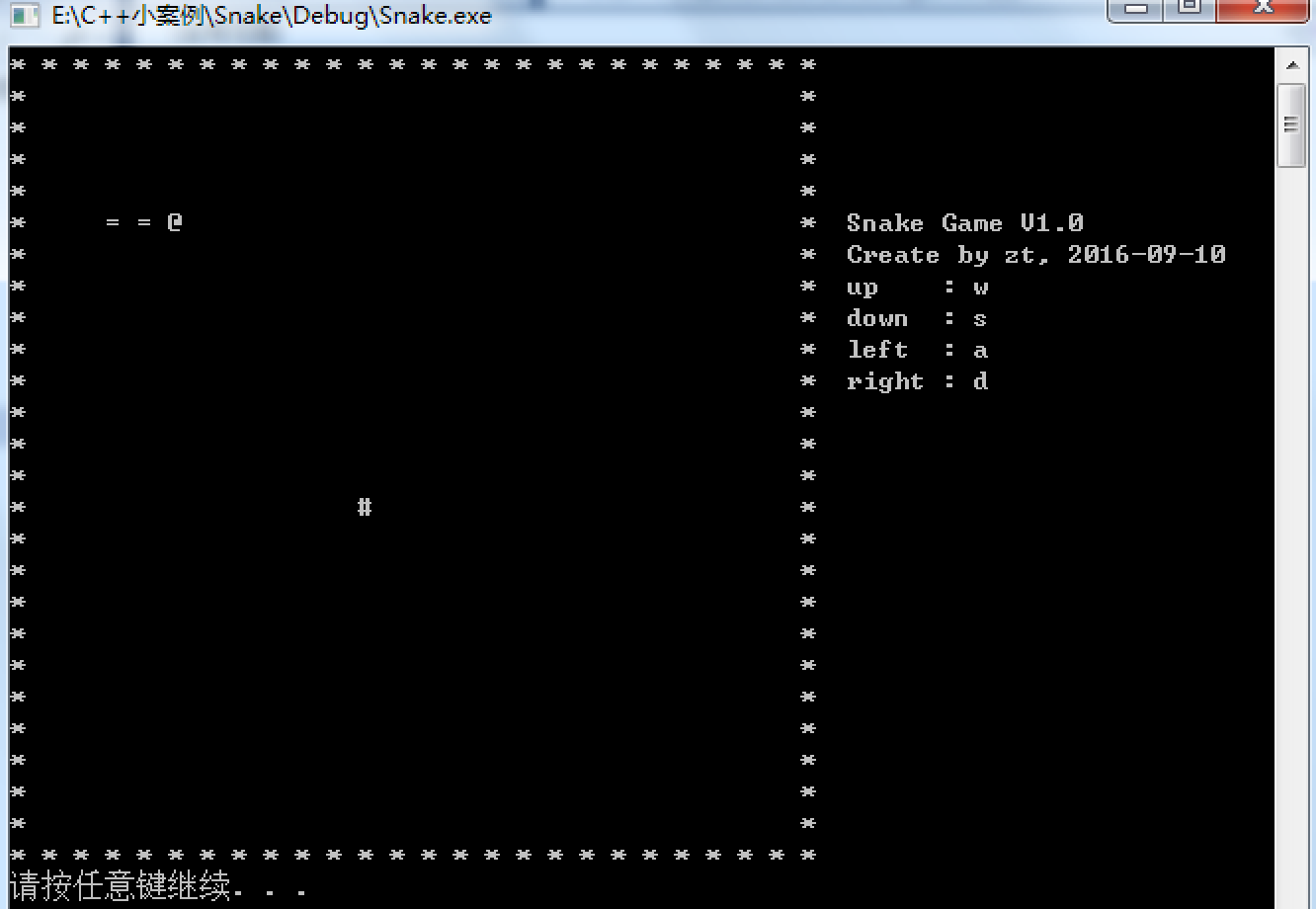
wall.draw(); //打印墙

*system*("pause");

return *EXIT\_SUCCESS*;

}

在main函数中引用食物模块，并且设置随机数种子，运行代码，可以看到食物呈现在屏幕中



## 3.6 主模块

### 3.6.1 分析

当我们完成了所有的元素的模块后，我们已经将游戏静止状态下的内容做好，下面就是让蛇运动起来，那么此时我们发现，蛇目前还没有移动能力，也就是对外没有提供一个移动的接口。

下面给snake蛇模块扩展出移动接口，move

移动时经过上面分析，分为两种，一种吃到食物，另一种正常移动，而正常移动时候需要我们把蛇的尾节点删除掉，这就需要我们再提出一个接口，删除尾节点，delPoint

### 3.6.2 蛇模块添加方法以及实现

snake.h中添加

代表上下左右的枚举

enum{ UP = 'w', DOWN = 's', LEFT = 'a', RIGHT = 'd' };

//移动蛇方法

bool move(char key);

//删除尾节点

void delPoint();

由于移动有成功移动，和失败移动，也就是GameOver时候，所有我们定义一个返回值为bool类型，参数为key，这个key就是用户输入的方向键，我们也可以定义出一个枚举，代表上下左右

.cpp中

bool Snake::move(char key)

{

int x = pHead->x;

int y = pHead->y;

switch (key)

{

case UP:

x--;

break;

case DOWN:

x++;

break;

case LEFT:

y--;

break;

case RIGHT:

y++;

break;

default:

return true;

}

if (wall.getWall(x, y) == '=' || wall.getWall(x, y) == '\*')

{

*cout* << "GameOver!" << *endl*;

return false;

}

if (wall.getWall(x, y) == '#')

{

addPoint(x, y);

//重新设置食物

food.setFood();

}

else

{

//正常移动

addPoint(x, y);

delPoint();

}

return true;

}

void Snake::delPoint()

{

//两个节点以上删除

if (pHead == *NULL* || pHead->next == *NULL*)

{

return;

}

//用两个临时节点，一个是前一个节点pre，一个是当前节点cur

Point \* pre = pHead;

Point \* cur = pHead->next;

while (cur->next != *NULL*) //指向的下一个不为空，就循环

{

pre = pre->next;

cur = pre->next;

}

//尾节点修改内容

wall.setWall(cur->x, cur->y, ' ');

delete cur;

cur = *NULL*;

pre->next = *NULL*;

}

delPoint，需要两个节点来寻找出尾节点，并且删除掉尾节点，将原有的倒数第二个节点的指向空，作为当前的尾节点；

move，需要判断出移动后的碰到的是否是墙、蛇、或者是食物，这几种可能性

### 3.6.3 测试新添加的蛇模块

我们在main函数中添加测试代码

int *main*(){

*srand*((unsigned int)*time*(*NULL*)); //随机数种子

Wall wall; //墙对象

wall.initWall(); // 初始化墙

Food food(wall); //食物

food.setFood(); //设置食物

Snake snake(wall,food); //蛇对象

snake.initSnake(); //初始化蛇

//测试

snake.move(snake.UP);

snake.move(snake.UP);

snake.move(snake.LEFT);

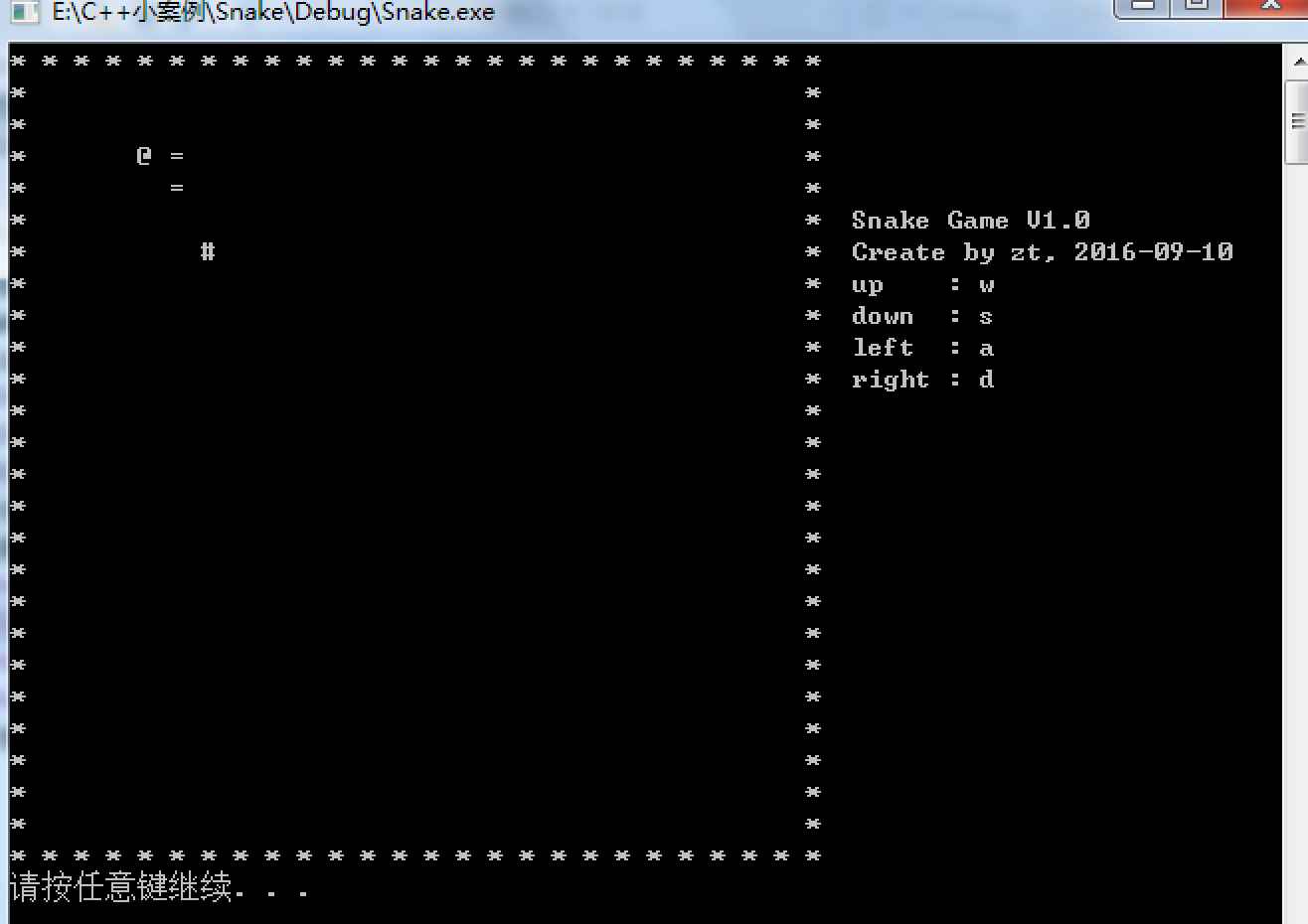
wall.draw(); //打印墙

*system*("pause");

return *EXIT\_SUCCESS*;

}

然后看没有测试代码和添加了测试代码后的效果，在这里我们写了3行测试代码，也就是将蛇进行了 上移、上移、左移，实现效果后可以看出蛇头已经朝左，证明移动方法成功



### 3.6.4 运行游戏，接受用户输入

接下来先将测试代码删除，然后我们准备接受用户的输入，根据用户输入来让蛇移动，首先，获取到用户的按键，然后判断是否是w、s、a、d，如果是再进行下一步操作

添加如下代码

char key = *\_getch*(); //conio头文件

if (key == snake.UP || key == snake.DOWN || key == snake.LEFT || key == snake.RIGHT)

{

//移动蛇

snake.*move*(key);

*system*("cls"); //清屏

wall.draw(); //重新绘制蛇

}

但是我们发现，蛇此时只能移动一次，再次按键就会退出程序，这时候我们需要通过循环来让蛇进行一直移动，而且我们还需要一个标示，这个标示表示蛇是否已经死亡，死亡的话我们就可以退出当前这个循环了，因此程序改为

//死亡标示

bool isDead = false;

while (isDead != true)

{

char key = *\_getch*();

if (key == snake.UP || key == snake.DOWN || key == snake.LEFT || key == snake.RIGHT)

{

//移动蛇

if (snake.move(key) == true)

{

*system*("cls"); //清屏

wall.draw(); //重新绘制蛇

}

}

}

运行代码，我们此时可以测试游戏了，只不过每走一步都需要玩家控制，也就是这个蛇只能按一次按键走一步，这并不是游戏应该的样子，正常游戏应该激活后，如果没有新的按键，就应该以之前的方向进行移动，因此我们也应该记录一下之前的方向，char preKey

再次修改代码

//死亡标示

bool isDead = false;

//上一次的运行方向

char preKey = *NULL*;

while (isDead != true)

{

char key = *\_getch*();

do

{

if (key == snake.UP || key == snake.DOWN || key == snake.LEFT || key == snake.RIGHT)

{

//移动蛇

if (snake.move(key) == true)

{

*system*("cls"); //清屏

wall.draw(); //重新绘制蛇

*Sleep*(300); //睡眠

}

}

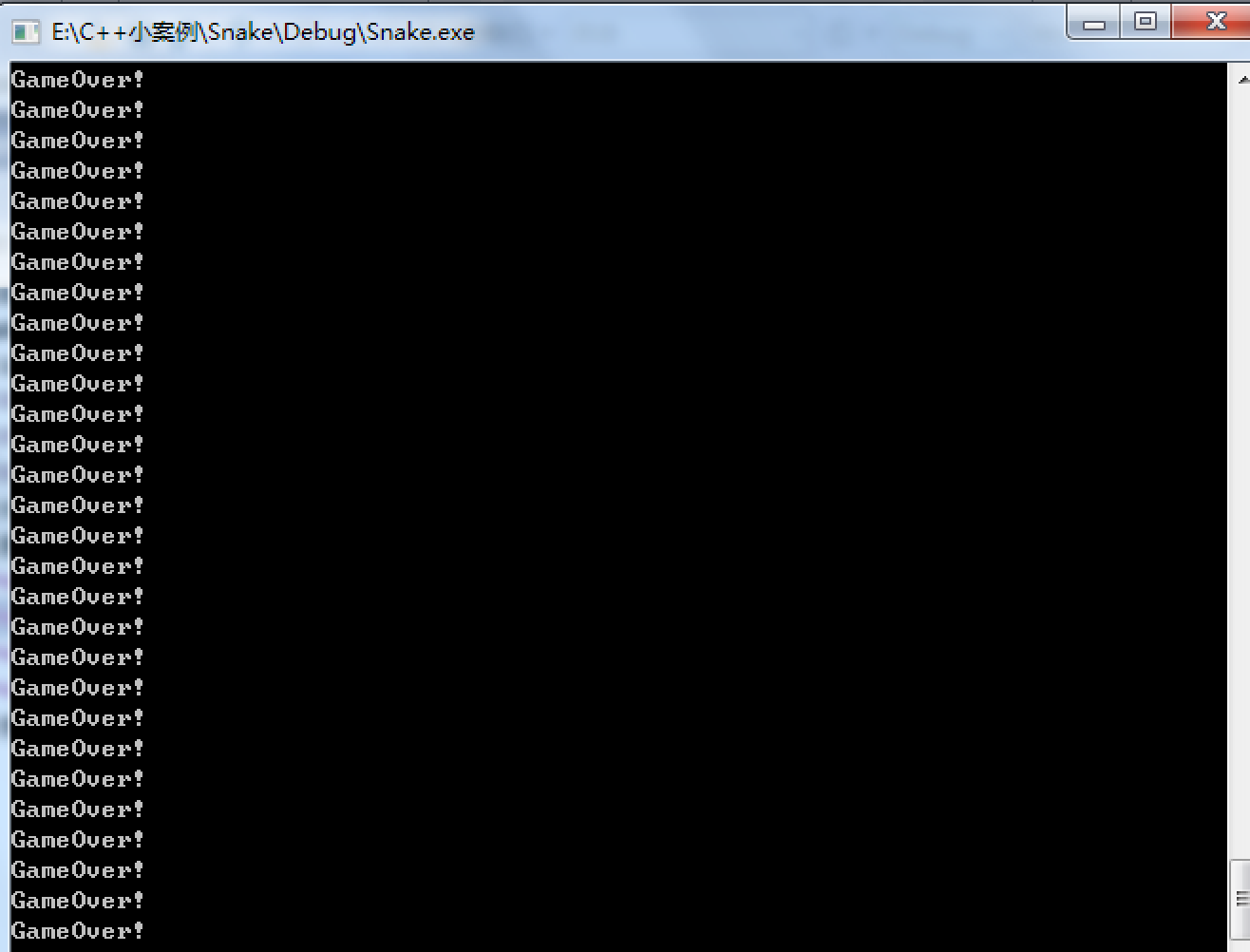
} while (!*\_kbhit*()); //当没有键盘输入时返回0

}

此时运行游戏，蛇就可以以一个方向进行移动了，我们还加了个睡眠时间控制蛇的移动速度。

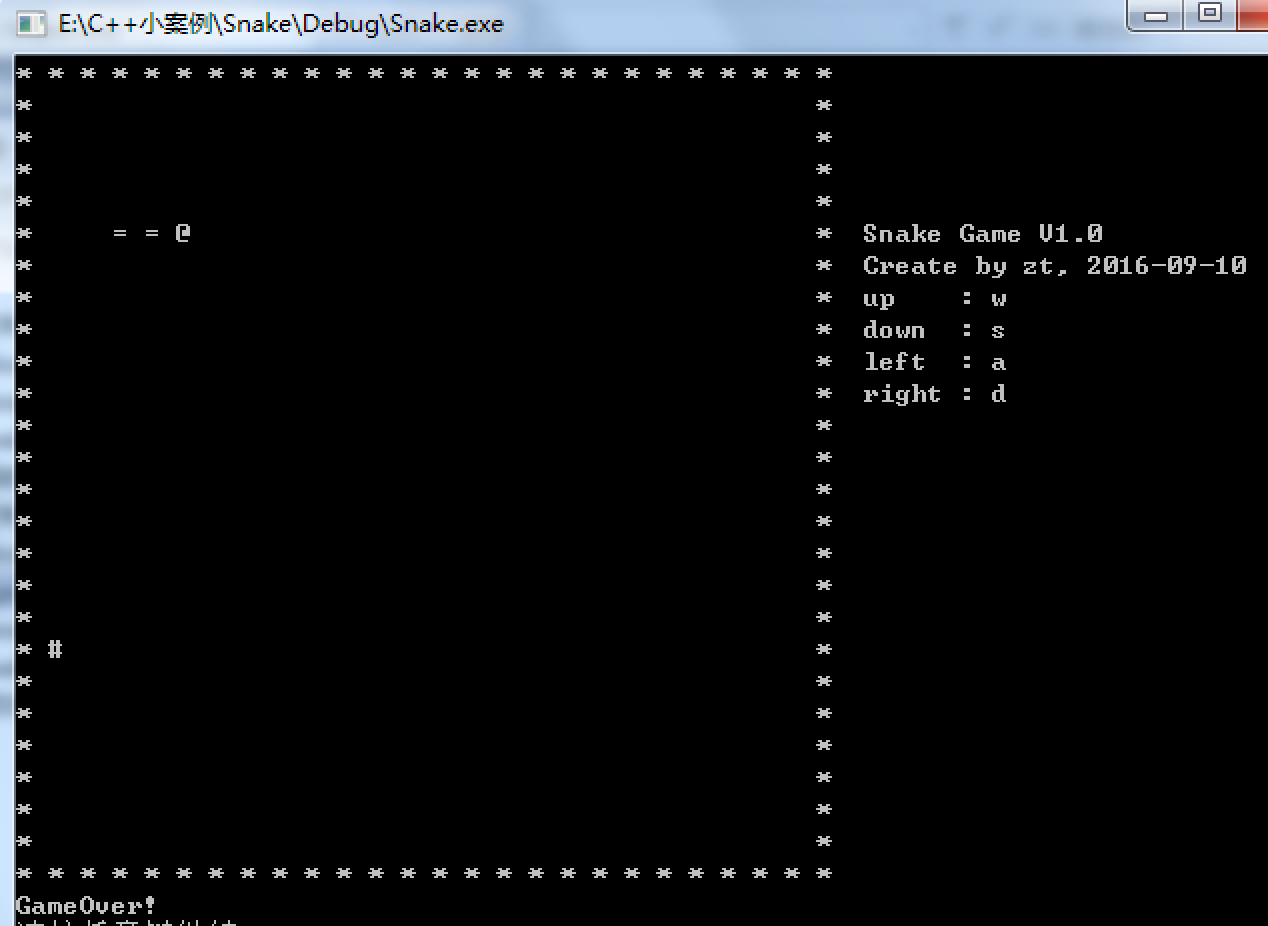
但是此时代码中存在许多bug，需要我们解决，首先我们的bug有这么几个

第一：目前我们只处理了移动正常的时候，蛇一旦死亡，就会一直显示GameOver，也就是没有处理死亡状态跳出循环，效果如图

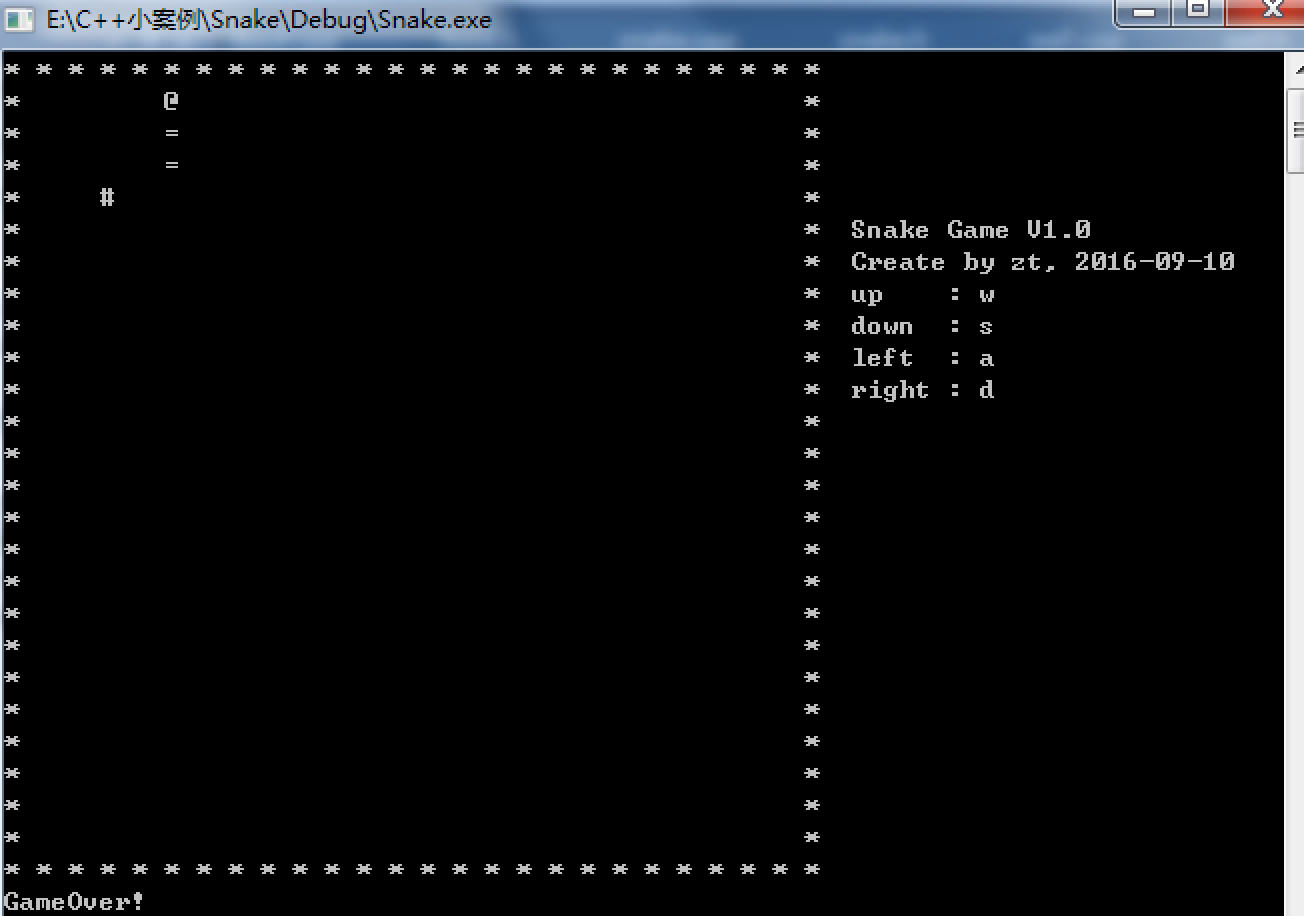


第二：我们没有处理反向移动，如何一上来我用a键激活游戏，那么游戏开始就会进入死亡，也就是上来就吃到了蛇身。但是游戏规则蛇不可以进行180°移动

效果如图，上来就GameOver



第三：蛇在死亡时候，只是死在了当前位置，并没有移动最后一步，这种显示并不友好，虽然我们自己可能看得出来是为什么死亡，但是玩家会觉得比较恶心，效果如图



其实已经碰到墙壁，但是没有走这步，就进入了GameOver状态。

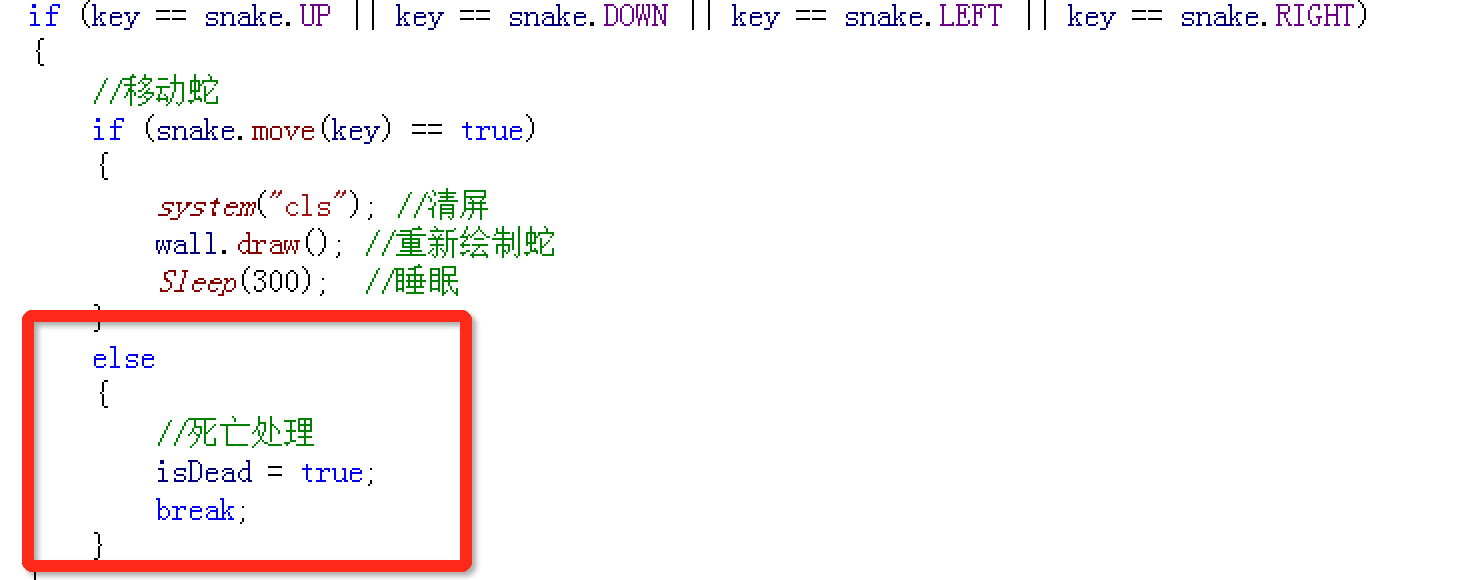
第四：当蛇有4个身段或更多时候，蛇如果碰到蛇尾，应该进行循环，而游戏中正常移动是先加节点，后删除尾节点，所以会导致循环蛇时候出现死亡，而正常的蛇其实是头尾同时移动，并不会直接死掉，所以我们应该判断一下蛇如果下一步到的位置是蛇尾，就不要进入死亡状态，bug效果



第五： 如果我们在游戏中，按了其他案件，也就是除了w a s d的按键，游戏就会停止。这也算个bug。

### 3.6.5 解决bug

#### 3.6.5.1处理死亡状态

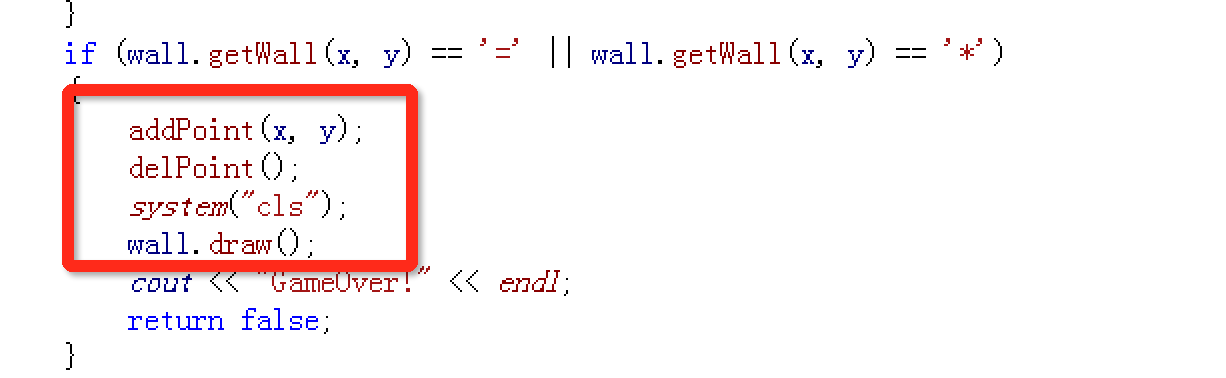


#### 3.6.5.2 180°移动

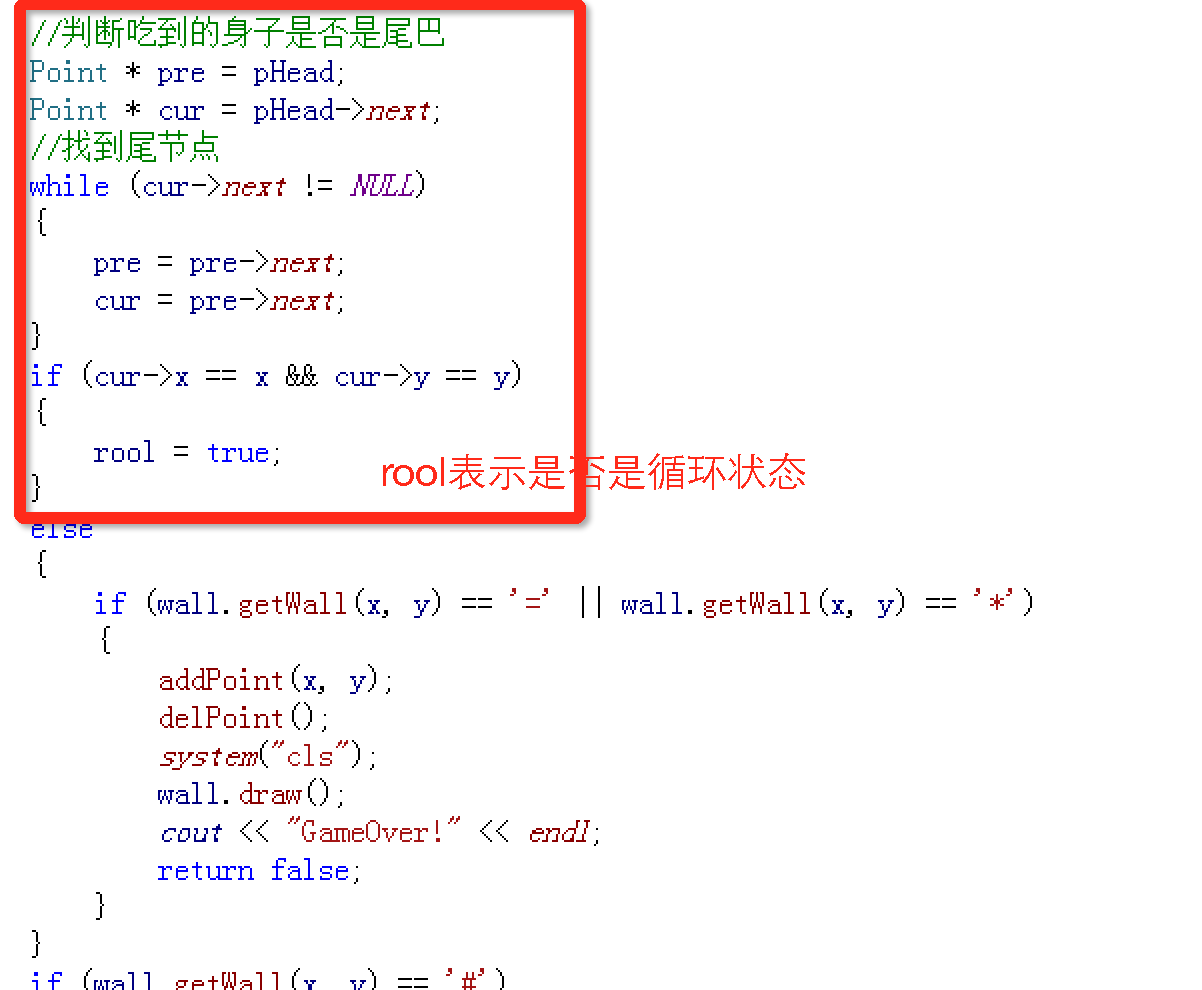


#### 3.6.5.3 显示蛇死亡的最后一步

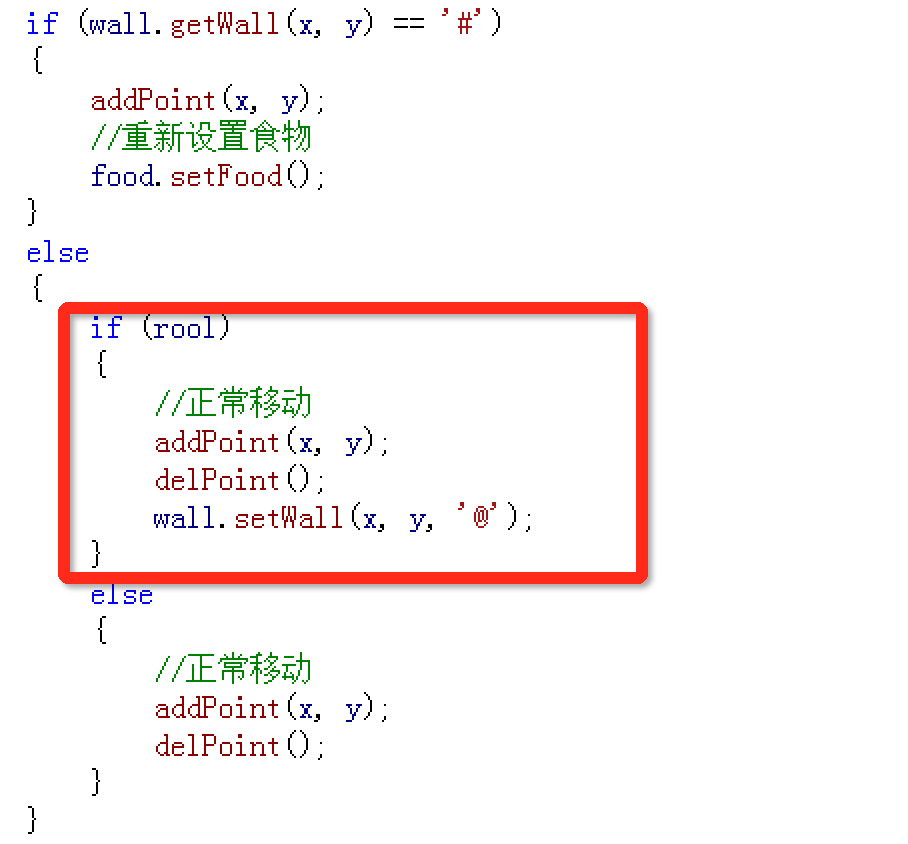
可以在蛇模块的move方法中添加如下代码



#### 3.6.5.4 解决循环问题



判断尾节点，是尾节点不进入死亡判断



如果是循环状态，还需要修改一下代码，否则显示会出现错误

#### 3.6.5.5 其他按键的屏蔽

if (key == snake.UP || key == snake.DOWN || key == snake.LEFT || key == snake.RIGHT)

之后我们可以加一个else

else

{

key = preKey;

}

这样当我们输入其他按键时候，就会保持上一个方向。

### 3.6.6 测试代码

解决所有问题后，玩游戏 测试代码。基本和贪食蛇一样。

# 5 游戏附加玩法

## 5.1 难度设定

我们可以根据蛇的长度不同，分别设定不同的游戏难度，比如我们可以分为3个难度，蛇长在50以内，50到100,100以上分别改变游戏的速度，根据这个需求。我们可以在蛇模块中加入新的API，获取蛇的长度：countList方法和 获取游戏睡眠时间 getSleepTime两个方法。

## 5.2 snake.h中 声明

//获取蛇移动时间

int getSleepTime();

//获取蛇的长度

int countList();

## 5.3 snake.cpp中实现

int Snake::getSleepTime()

{

int sTime = 0;

int size = countList();

if (size < 5)

{

sTime = 300;

}

else if (size >= 5 && size <= 100)

{

sTime = 200;

}

else

{

sTime = 100;

}

return sTime;

}

int Snake::countList()

{

int size = 0;

Point \* curP = pHead;

while (curP != *NULL*)

{

size++;

curP = curP->next;

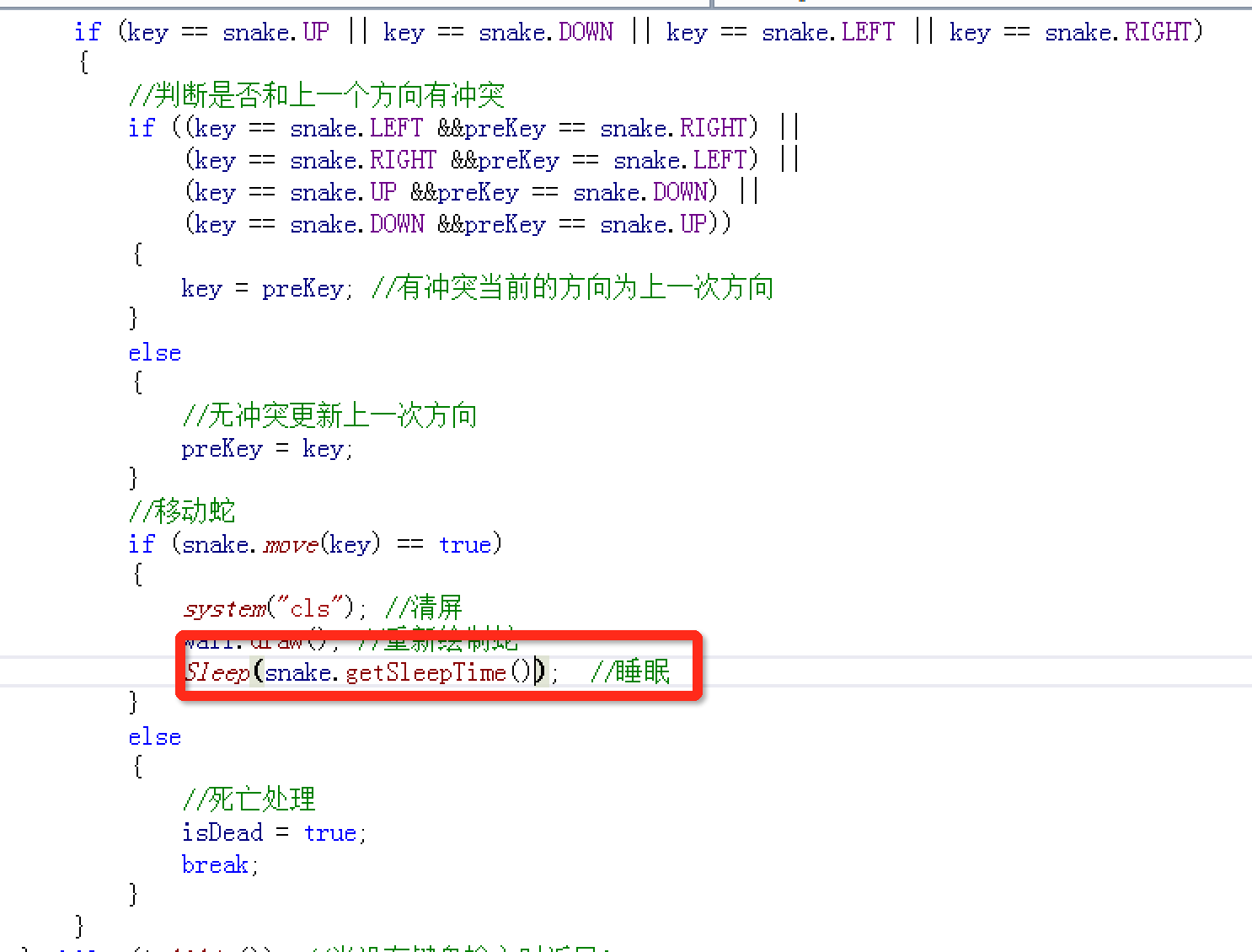
}

return size;

}

## **5.4 测试难度**

我们在测试时候可以将区间段设置小一些，这样方便测试，在game.cpp中，蛇移动时候代码修改为



# 6 分数模块

## 6.1 分数设定

游戏中通常会有些辅助的玩法来增加游戏乐趣，例如2048一开始也只是看谁攒的2的n次方多一些，后来游戏中也设定了分数，来比较谁的分数高些，同样我们也可以为贪食蛇来设定一个分数。

我们可以设定一个简单的分数，比如吃到一个食物，就加100分，这个分数实时在屏幕上显示着，直到GameOver时候，我们可以看到本局最终得分。

## 6.2 snake.h中声明

由于分数是依赖于蛇的长度，因此我们可以在蛇的模块中加入一个方法 getScore，来获取得分，主模块中调用方法，展示最后得分

snake.h中

//获取分数

int getScore();

## 6.3 snake.cpp中实现

int Snake::getScore()

{

int score = 0;

int size = countList(); //获取蛇长度

score = (size - 3) \* 100; //减去蛇初始长度

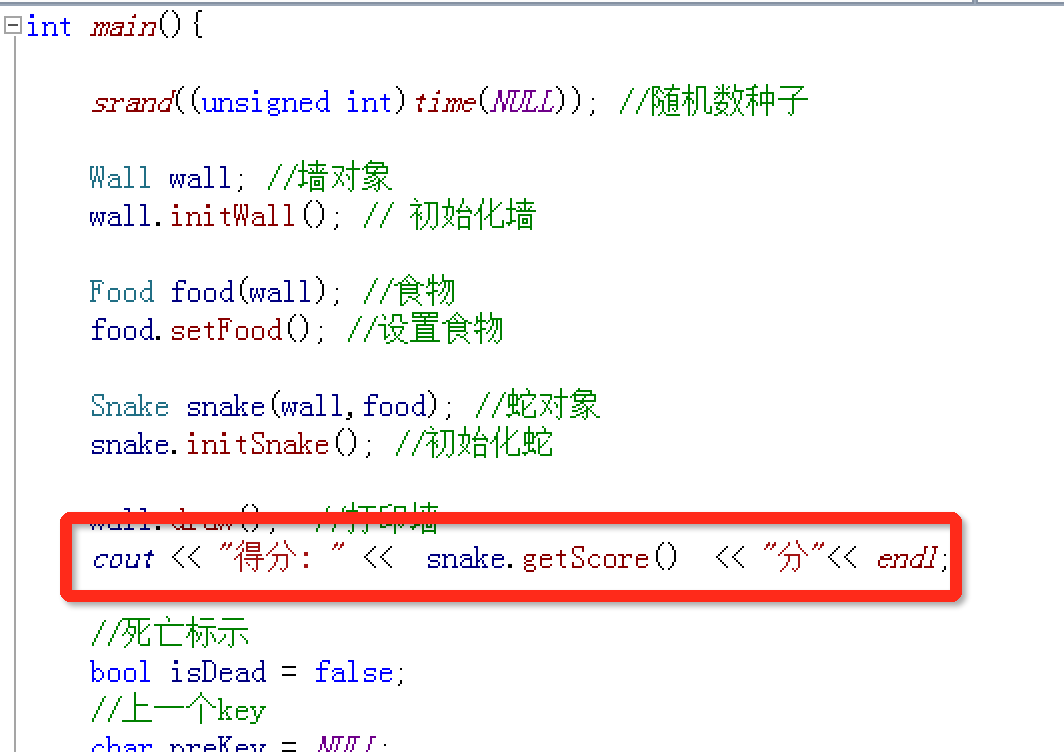
return score;

}

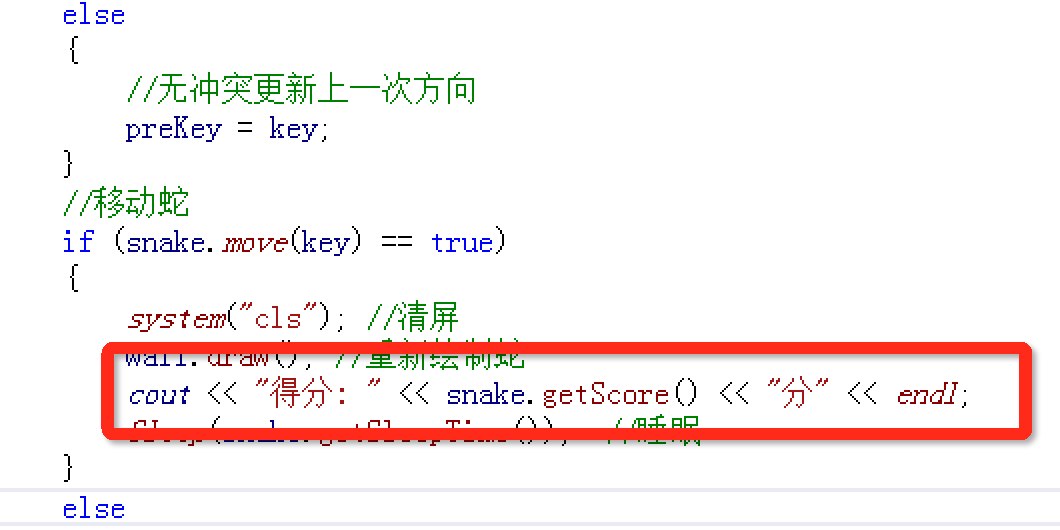
## 6.4 测试分数

分别在game主模块中初始位置、刷新移动位置，以及蛇在Move方法中判断游戏结束位置加入分数代码

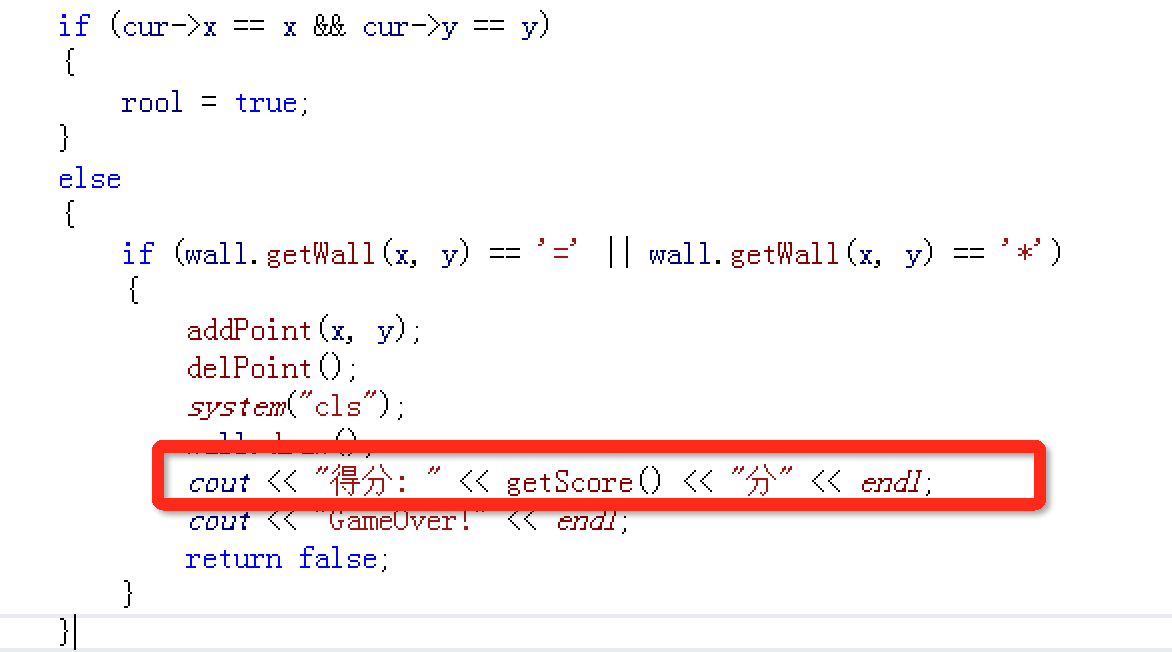
初始打印：



刷新移动：



GameOver：



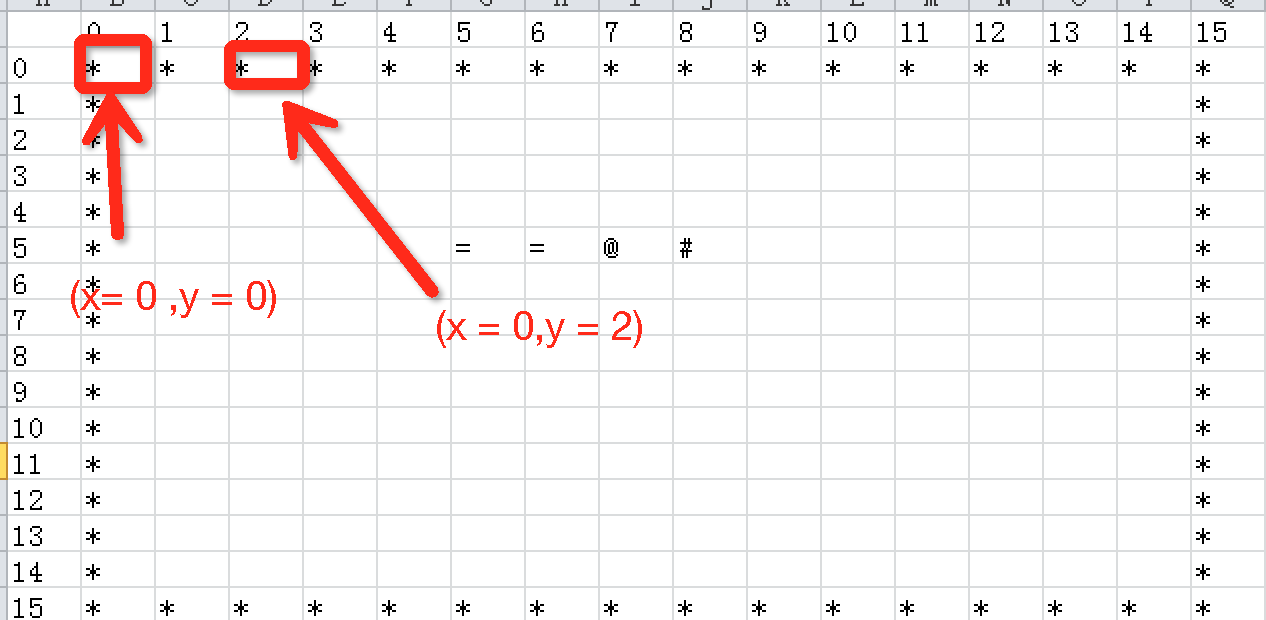
运行游戏，可以看到分数



# 7 优化游戏

我们现在做的游戏是依靠刷新进行重新绘制图片，但是如果不希望对整体刷新而实现同样的游戏效果，这就需要利用定位光标来修改游戏内容了

首先，要知道我们游戏中的二维数组在图形显示时，xy坐标如下



但是修改屏幕坐标点时候，屏幕坐标系是以向右为x正向，向下为y正向，这时候我们在通过修改光标实现游戏时候就应该对坐标系进行转换，而且我们在打印到屏幕时，横向每次都用空格进行了分割，因此还需要x方向\*2；



在snake中，先定义出修改光标的代码

void gotoxy(*HANDLE* hOut, int x, int y)

{

*COORD* pos;

pos.*X* = x; //横坐标

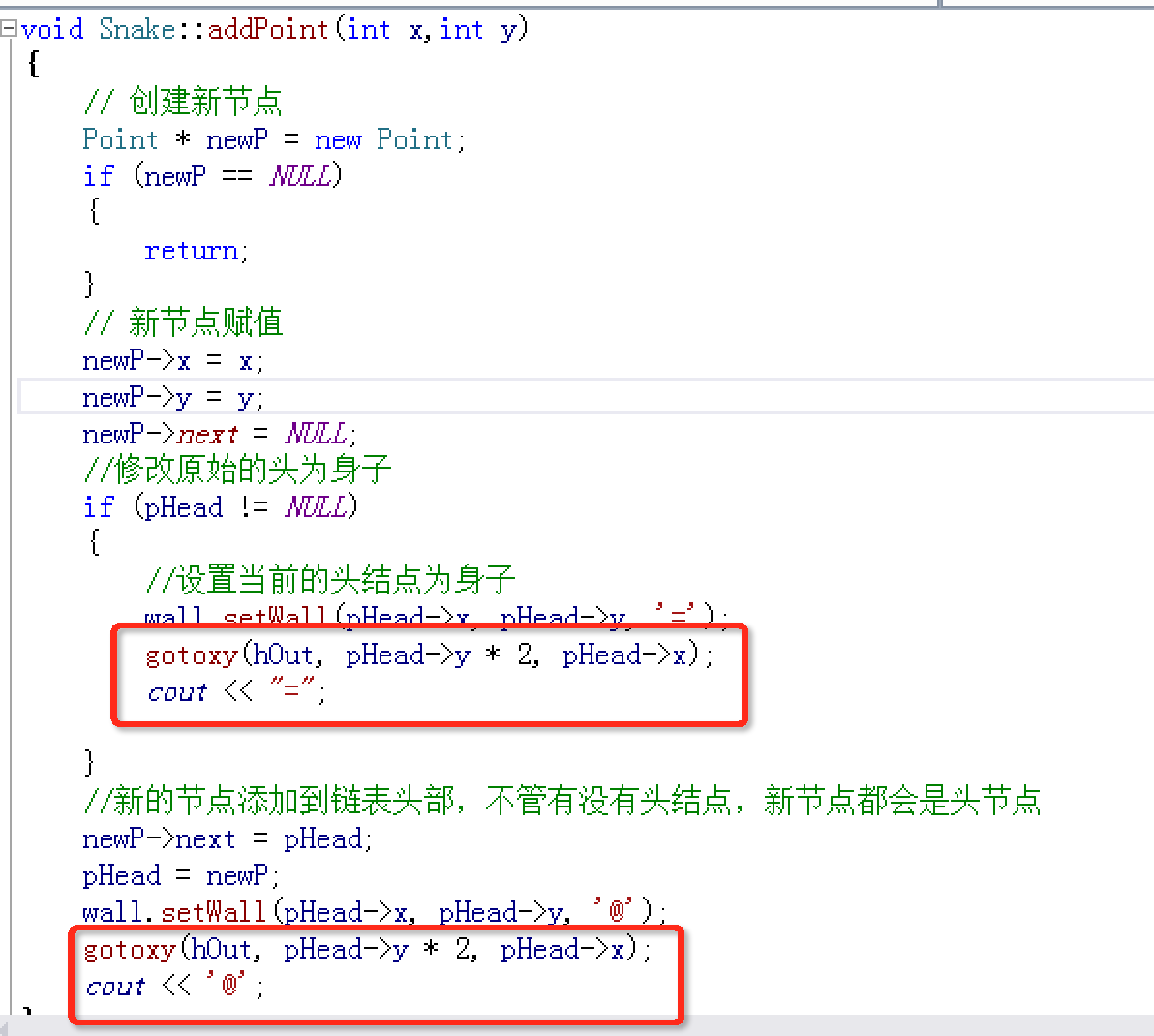
pos.*Y* = y; //纵坐标

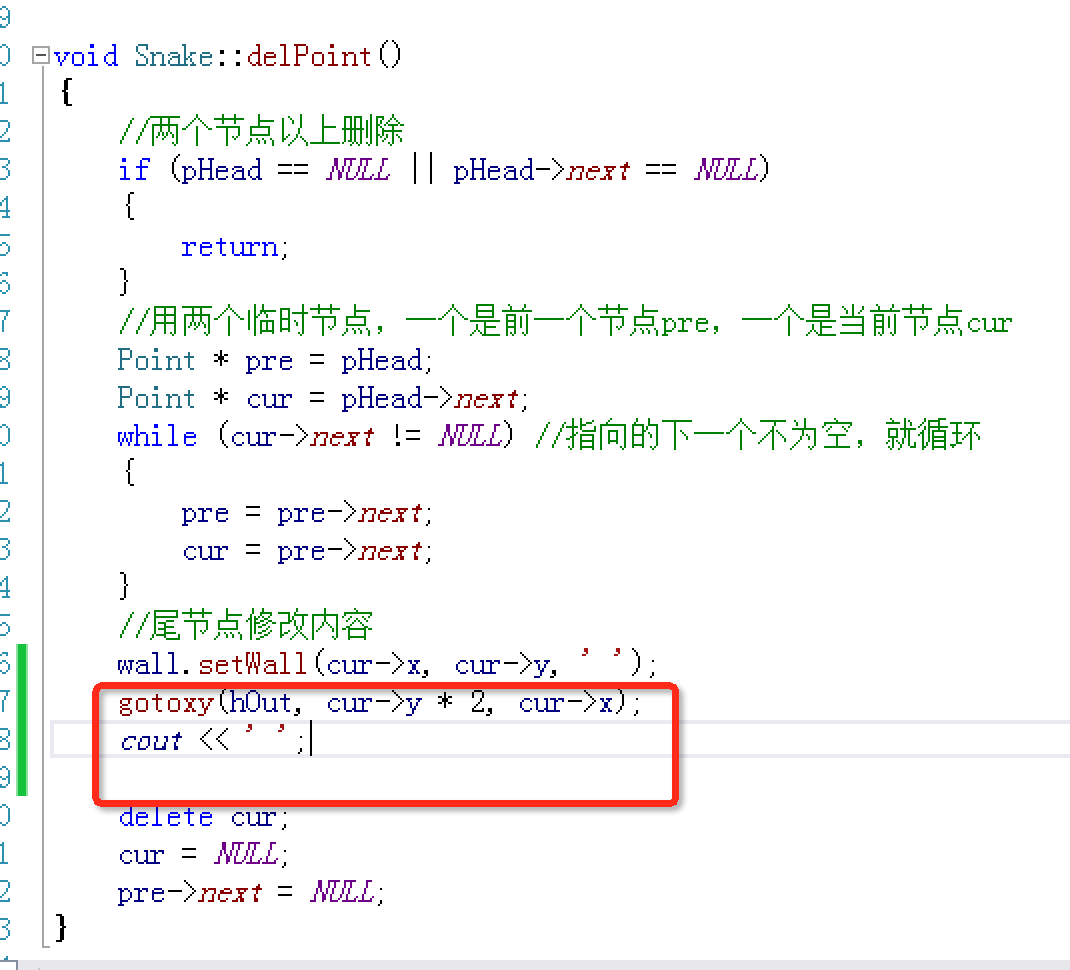
*SetConsoleCursorPosition*(hOut, pos);

}

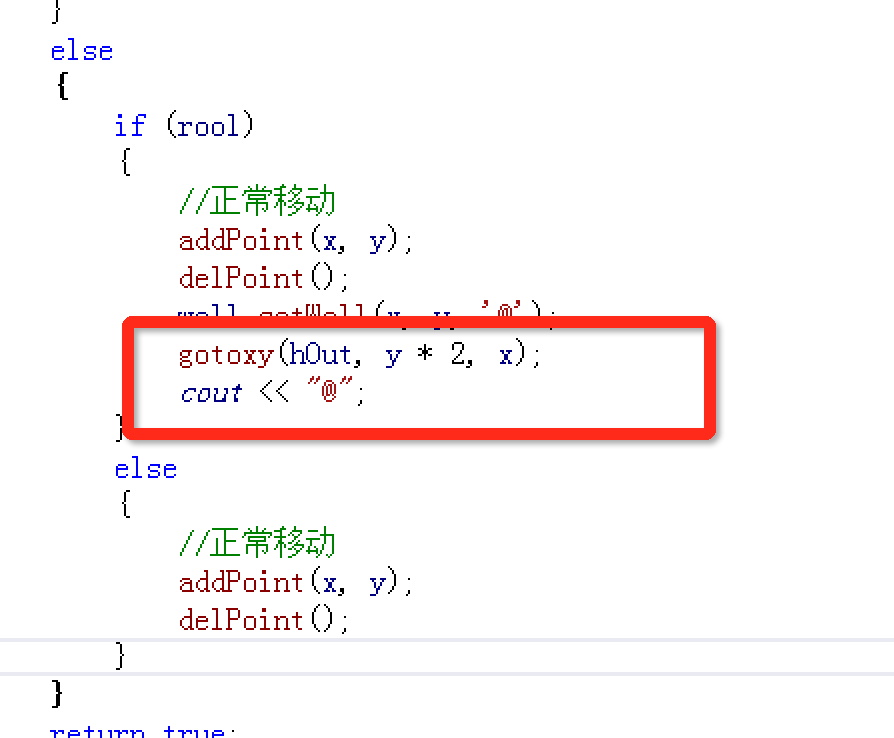
*HANDLE* hOut = *GetStdHandle*(*STD\_OUTPUT\_HANDLE*);//定义显示器句柄变量

修改snake模块中代码





move方法中

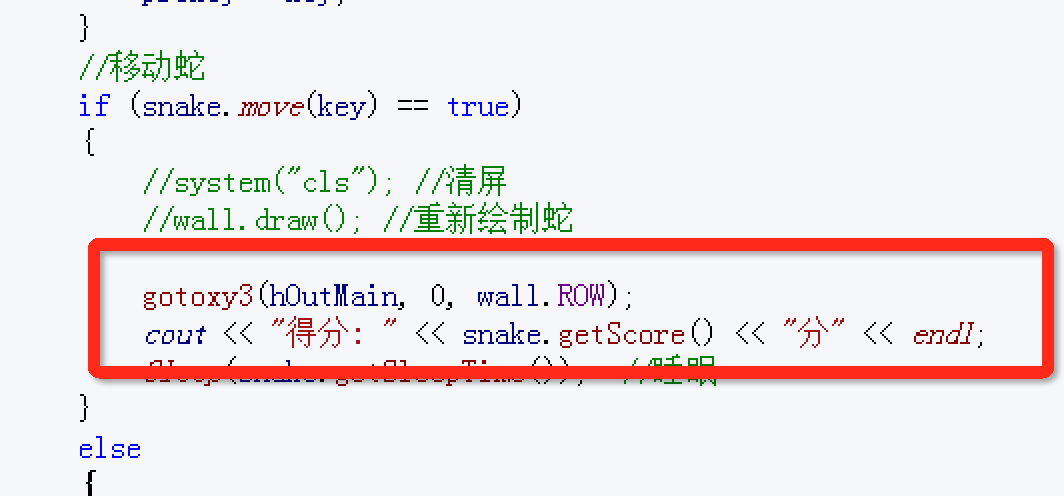


food模块中也需要同样操作



game.cpp中调整代码顺序



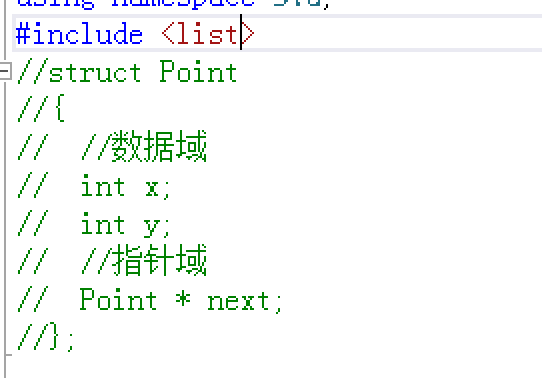


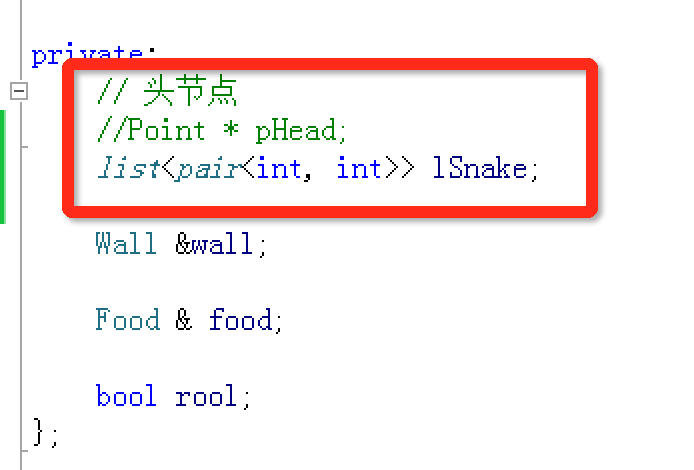
最后测试代码，可以不用刷新整个屏幕来进行游戏了！

# 8 利用STL实现游戏

之前我们用的是原生的结构体来模仿蛇的链式结构，当我们学过了List容器后，我们完全可以利用STL中的List来模拟蛇，当然我们需要修改代码，修改的地方如下：

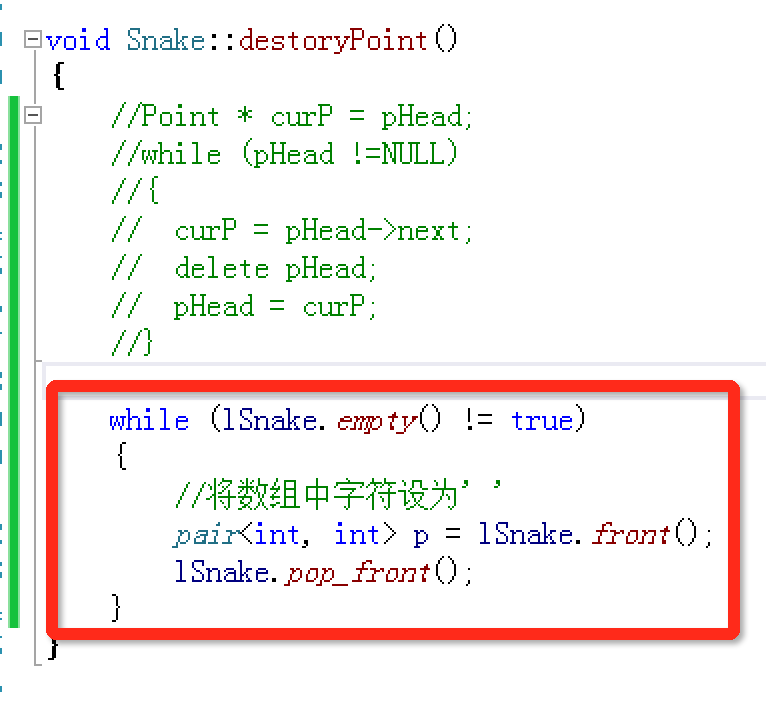
首先将snake中的Point的结构体注释，包括成员属性pHead



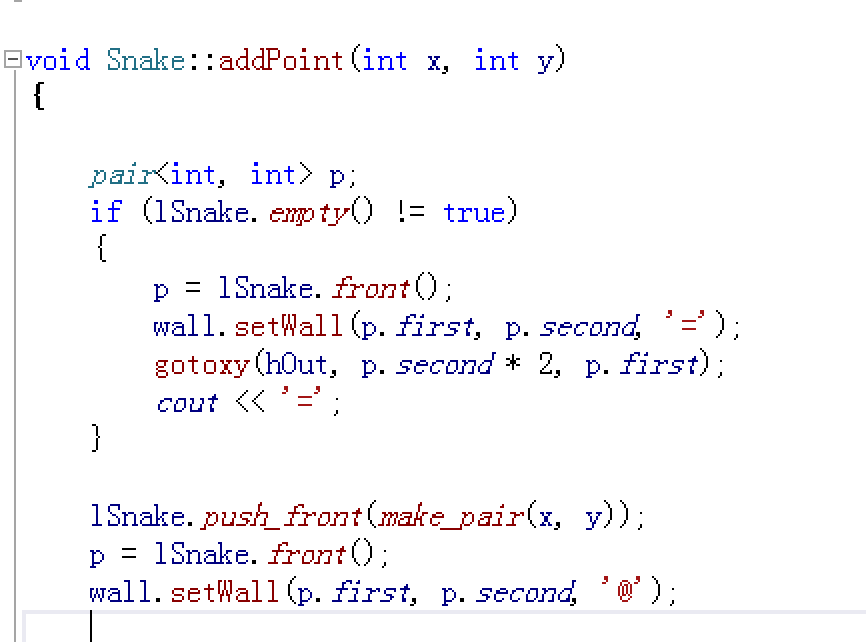


并且声明出一个list，容器中放入pair类型，也就是我们的数据域

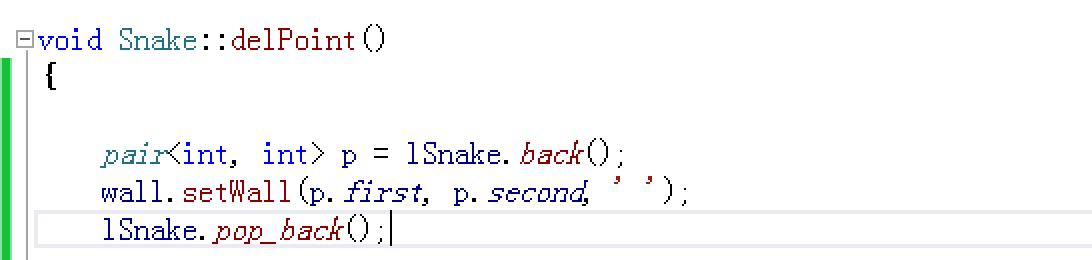
destroyPoint修改如下：



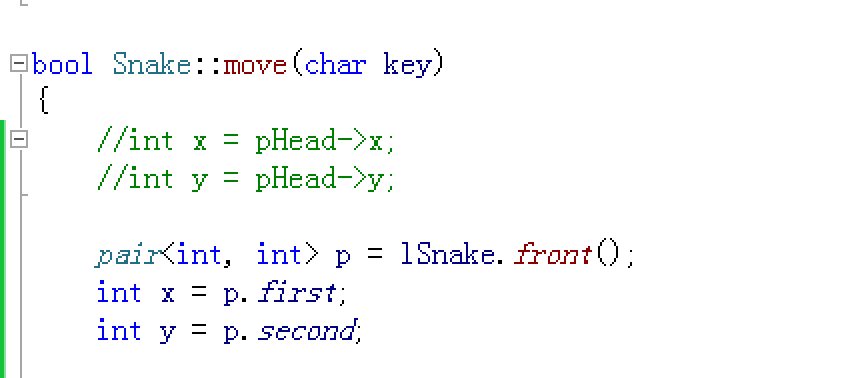
addPoint中

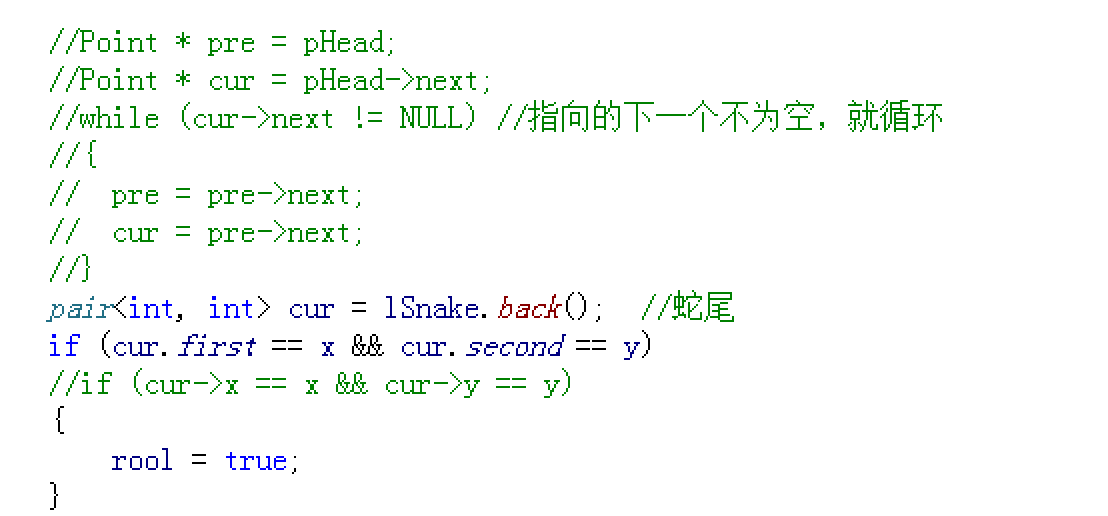


delPoint中：



Move中：





测试游戏，依然和原来效果一样，但是我们利用了系统提供的链表结构储存了蛇这个结构，代码会相对于简单很多。