C PROJECT REORT: *SNAKE +*

Contents

Abstract

1) Problem Statement

Compile the classic game--Snake using C language, and add some new ideas to the game.

2) Goals

Finish the basic functions: creating food, pressing arrow keys to control the snake, eating food to get more scores, optimizing the player experience,etc.

Make multiple modes based on the basic functions: double mode, infinite mode, moving-obstacle mode.

1. Introduction

Snake is a classic game dating back to 1976. It is simple, easy to operate and entertaining. Because it well meets the requirements of the player, since it has been published, *Snake* has become more and more popular around the world. Our group member are all fans of this game, so we decide to create our own *Snake* using C language.

In the game, player controls a snake to eat food. By pressing arrow keys, player can change direction that the snake is heading. Eating food means getting more scores, meanwhile the snake will grow longer, which add to the difficulty of the game. And the snake shouldn’t hit the wall or knocks its head into its body. There's only one life, one mistake means game over.

Only finish the basic functions is far from achieving our goals. To make our game more creative and entertaining, we want to develop multiple modes. In double mode, two players control their own snake separately and compete for higher score. In moving-obstacle mode, when player get a certain number of more score, an obstacle will be created. The obstacles will move automatically to make the game a little more challenging. In infinite mode, after player breaking the rules, the game will continue

and the snake will show up in the middle of the map.

2. Analysis

1) Single Mode

(1) Basic functions

Snake has developed many versions since it 1976, not matter how it changes, there are several functions that are always the most basic.

Create food: When a food has been eaten, a new food should be created. Food should be created randomly within the map. And it shouldn’t fall on the snake. If food falls on the snake head,score will plus 1 automatically; if food falls on the snake body, it will soon be covered and no new food will be created.

Get direction: Player can press arrow keys to control the movement of the snake. Input of other keys and opposite direction are invalid, which should be ignored.

Move snake: Snake should change direction according to the input, while it should also move along automatically when there is no input.

When a food has been eaten, the snake will grow longer, which makes it more difficult for the player to avoid hitting the snake itself.

Alive or not: Snake shouldn’t hit the wall or knocks its head into its body. A function which can end the game when player breaks a rule is necessary.

(2) Extended function

To optimize players' gameplay experience and make our game more humanized, we designed some extended functions.

Pause and continue: Player can press space key to pause at any time during the game, and press space key again to continue.

Speed up and slow down: Player can adjust the moving speed of the snake to their most comfortable degree by pressing F1(speed up) and F2(slow down).

Save&load: If player have to stop playing in the middle of the game while they

still want to continue it next time, they can press ESC to exit. Then all the data will be recorded in a file, including position of food, position of snake, score, speed and direction.When player open the game next time, they are able to choose whether to continue the unfinished game or start a new game, if they choose to continue, load in all the data,

Save the highest score.

Add cover and welcome page.

Play background music.

Player can choose whether to play again or exit when game is over.

Rules:

1.Press arrow keys to control the movement of the snake.

2.Press space key to pause or continue.

3.Press F1 to speed up and F2 to slow down.

4.Press ESC to save data and exit.

5.Eat food to get more score, and the snake will grow longer after eating a food.

2) Double Mode

Double mode allows two players to player together and compete for higher score.

To control the movement of the snake, one player uses arrow keys, the other use a/w/s/d keys. The essential difference between single mode and double mode is that, in single mode, we receive input from the keyboard and directly change the direction that the snaking is heading in, while in double mode input of two players need to be received at the mean time. So when getting input from the keyboard, firstly, we should judge the current input is from which player, then change the corresponding direction. Otherwise the situation will become extremely confused.

To avoid crash between two snakes, two different food should be created so players can chase for their own food. More comparison is needed in the creating food part to make sure that two food won't coincide or fall on the snakes.etc.

Another problem is how to judge who is winner. Setting a time limit is a good choice, the one who gets a higher score in the given time wins. But when someone

break the rules, the game will end earlier. So to be fair, the one who breaks the rules first should be punished, we think subtracting 5 points is appropriate.

Naturally, the the rules should also be changed. In single mode, snake shouldn't knock its head into it's body or hit the wall. In double mode, they shouldn't do so as well. Besides, one snake shouldn't knocks the other, and the one who knocks the other will be seen as the rule-breaker (Two snakes crashing head-to-head is a special case, which we think no one should be punished).

Considering that there are two players, we canceled the following function: pause and continue, speed up and slow down, save and load.

Rules：

1.Press the arrow keys to control the movement of green snake,eat red food

2.Press the a/w/s/d keys to control the movement of blue snake,eat yellow food

3.Eat your opponent's food , score minus 1

4.If you hit the wall/eat yourself/knock your opponent, the game will be ended and your score will minus 5

5.The one who gets a higher score in x minutes is the winner

3) Moving-obstacle Mode

In single mode, not only the walls are obstacles but the snake body can also be seen as a moving obstacle. In the moving-obstacle mode, every time the player get a certain amount of more score, a obstacle shaped like a little block will be created. It shouldn't coincide with food or snake as well.

To make the game more challenging, obstacles will move at random direction automatically. But as it moving around, it shouldn't go out of the walls or cover the food.

Though moving obstacles add fun to the game, if the level of difficulty is too high, player may soon loss interest. So it would be better that the moving speed of obstacles is a quarter of the snake.

As for the rules, when the snake hit an obstacle, or any part of the snake body is knocked by an obstacle, score will minus 1 and snake will become shorter. But hitting

an obstacle won't lead to game over, which make the game more entertaining.

Rules:

1.Press arrow keys to control the movement of the snake.

2.Press space key to pause and continue.

3.Press F1 to speed up and F2 to slow down.

4.Press ESC to exit and save.

5.An obstacle will be created every time you get x more points.

6.If you meet an obstacle, score will minus 1 and the snake will become shorter.

4) Infinite Mode

In the single mode, there is a function to judge if player break the rules and end the game. However, in infinite mode, the game will never come to an end. So first we should consider changing the alive-or-not judgement. But when snake hit the wall or eat itself, just no ending the game is not enough, because in those conditions, the snake will run out of the wall or the snake body will be damaged. To solve this problem, we should set a new position for the dead snake. The position should be set randomly but not too close to the wall to give the players enough time to react. As the snake will be set at another place, the previous snake body should be covered.

Another problem is that if the snake hit the wall, the wall will be damaged and need to be repaired.

To optimize player experience, the process above should be smooth and comfortable. Though it's Infinite mode, it's necessary to let the players know how many times the snake has dead. So we need a function to record it, which is to set the blood intuitively. If the player dies for a time, the blood will minus 1. If the blood turns to negative number, it will turn to full number at once to meet the player's expectations. Secondly, to add another interesting rules, we want to let the food has its own function, that when snake eats food, it will speed up or slow down randomly. We think this operation is acceptable for there have been a function for the players to adjust the speed freely. And setting this new function can improve the flexibility of this mode. In addition, save function is canceled for the equity of the game and it's no

sense to add save function to this mode.

Rules:

1.Press arrow keys to control the movement of the snake.

2.Press space key to pause or continue.

3.Press F1 to speed up and F2 to slow down.

4.Press ESC to exit.

5.Speed will change randomly after you ate a food.

3. Design

1) Kernel

(1) Linked list

As we all known, linked list is a linear data structure consisting of a collection of nodes which together represent a sequence. As the length of the snake is always changing, using a linked list to represent the snake is a good choice. The advantages are as follow:

Changes in length or position of the snake can be realized by inserting, removing or changing nodes in the list. There are some algorithms to implement these operations, mostly using a loop which can do list traversal. And these algorithms are clear and logical, each operation correspond to a loop block, which will make the program have a very clear structure.

Linked list and snake are very similar. The 'head' of a list is its first node, the 'tail' of a list is the last node. Each node represents a part of the snake. And they are both very flexible.

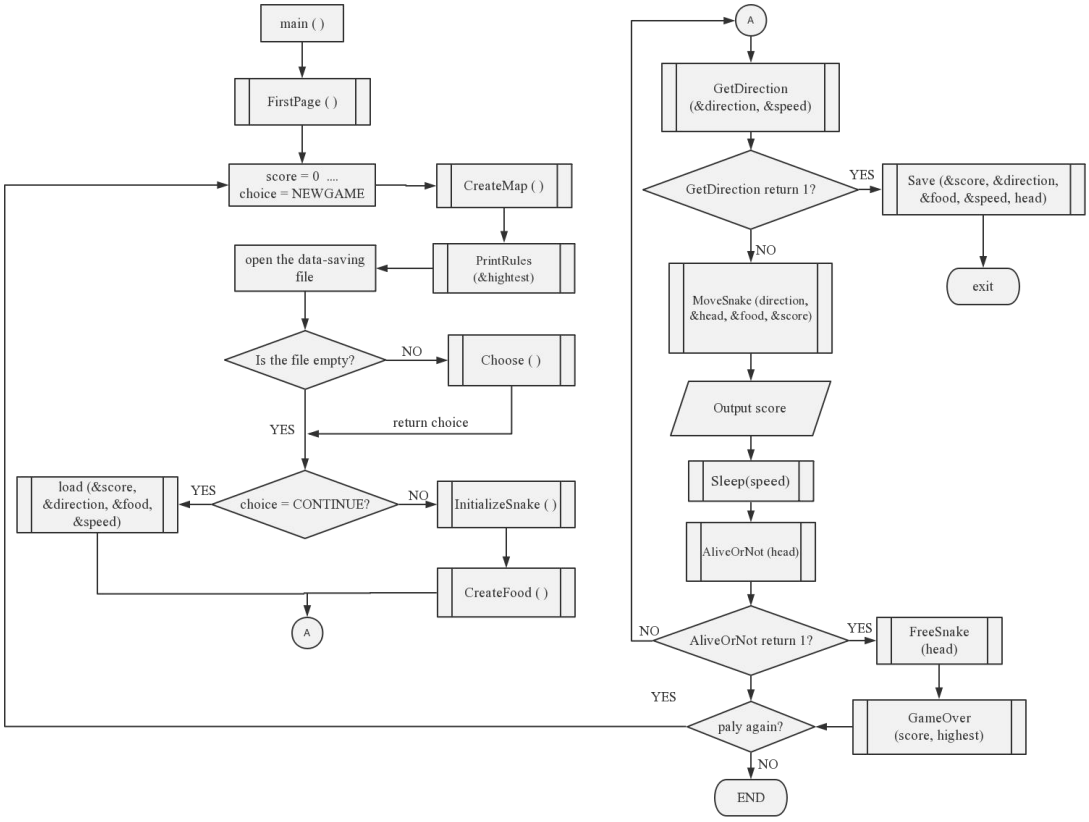
(2) Coordinate

To know where is the snake and what's happening in the game, we give each objects a coordinate to record their position in the map. The value part of each node is

a COORD variable. By comparing coordinates, we can judge if the snake eats a food or runs into the wall. Moving snake is to change the coordinate of each node.

2) Single Mode

(1)*main* function



When player open the game, firstly, the *FirstPage* function will give the player a warm welcome and show the rules. Then assign default value to some variables. *CreateMap* function will create a square representing the walls.

Then it comes to the save&load part, first check if the data-saving file is empty, if so,by default (variable *choice* has been assigned as NEWGAME), player will enter a new game, where a linked list will be created by the *InitizlizeSnake* function and food will be created as well. If the file is not empty, it means that the player exited in the middle last time. So the *Choose* function will ask the player whether to continue the unfinished game, if so, data will be loaded in, and a linked list will be created according to the position of the snake last time.

The infinite loop on the right is the most important part. The whole game process is all conduct by this loop.

In each round, firstly, the *GetDirection* function will check if there is any current and valid input from the keyboard, if so, *direction* will be changed, otherwise it will remain the same. *GetDirection* is the only function that can receive input from the player in the loop, which means it's the window between player and the program.

If input is ESC, this function will return 1, which will start the *Save* function to write coordinates of food and each node of the linked list, *speed*, *score* and *direction* in a file, then exit the game directly.

If input isn't ESC, then it goes to the *MoveSanke* function, in this function, the snake will move a step according to *direction*. And if there is no input before, the snake will move as the original direction, which will create a effect that the snake moves automatically. This function will also check whether snake has eaten a food and change score.

Sleep function will make the program suspend for a while, by changing its argument, we can change the speed of the snake. After several simple tests, we think use 250 as the default value of *speed* is suitable, that each round last about 250ms.

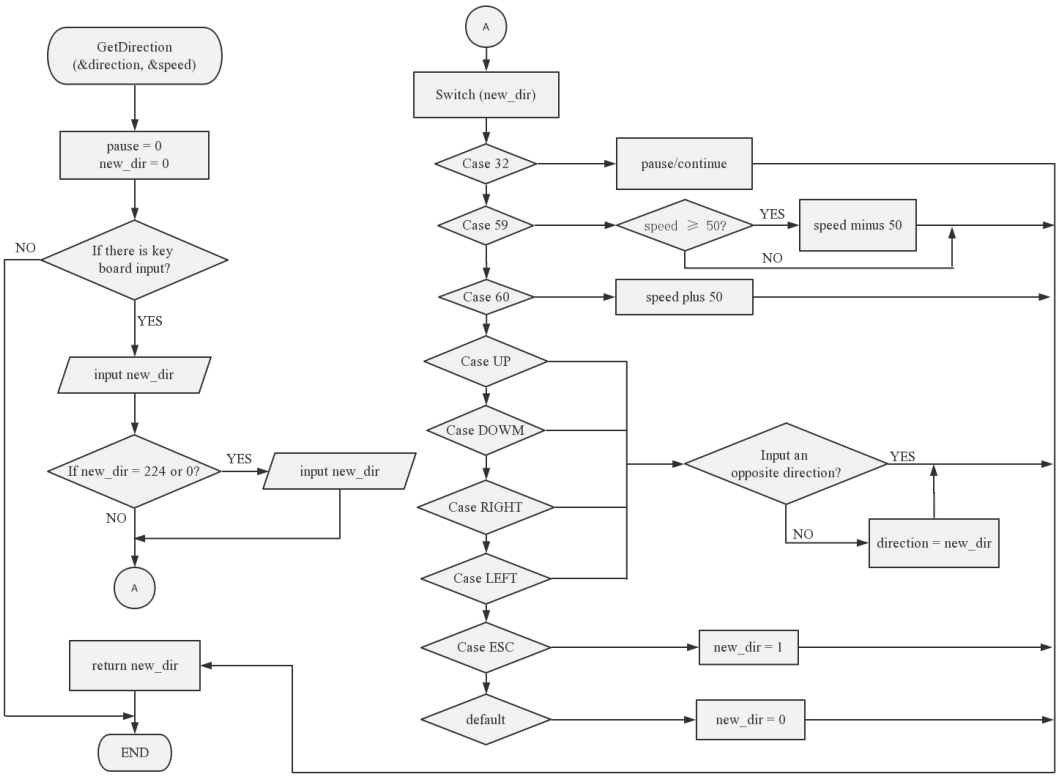
At last, the *AliveOrNot* function will judge if any rule was broken. If not, go to next round and repeat the same process, and the snake will move again and again. Otherwise jump out of the loop. The *GameOver* function will check if the current score is higher to save the highest score. And play can choose whether to play again.

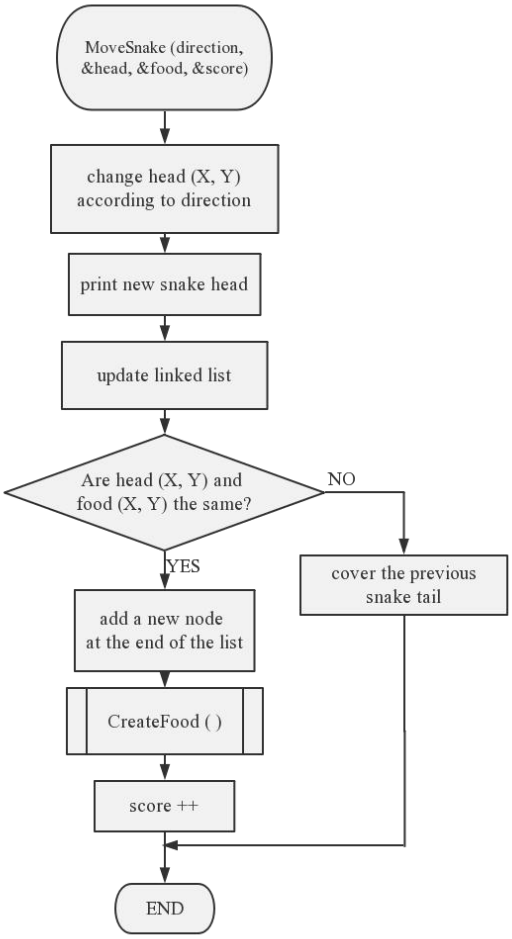
(2)*GetDirection*

First, check if there is current input from keyboard, if not, come to the end and *direction* will remain unchanged.

If so, receive the input. (By the way, pressing an arrow key or a function key will return 2 values, the first one is 224 or 0, the second one is the corresponding value.)

Then, according to the input, perform corresponding operations: Press space key to pause and continue. Press F1/F2 to speed up and slow down, that is to change the value of *speed* (the argument of sleep function). Press arrow keys to change direction,





but opposite direction is invalid. Press ESC, this function will return 1 to call another function which will save data later.

(3) *MoveSnake* function

First, change the coordinate of the head according to *direction*, then print new head at that position. Next, update the linked list, which will change the coordinate of each node to the one before it.

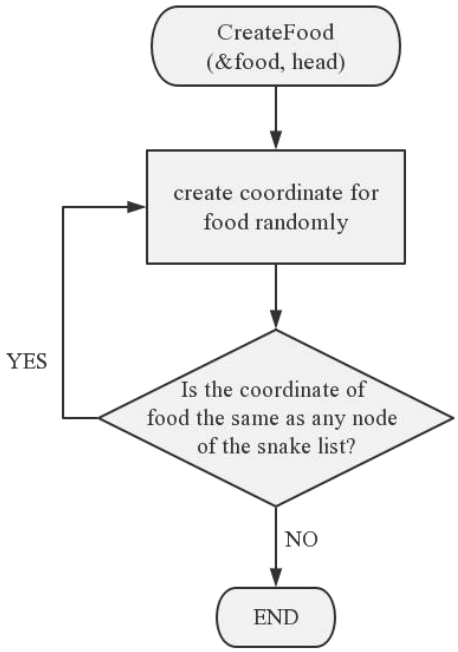
Then check if snake ate the food, that is to say, whether head(X.Y) and food(X.Y) are the same. If so, add a new node at the end of the list, create a new food and *score* plus 1. As a new head has been print before, it will looks like that the snake grows longer.

Otherwise the previous snake tail will be

covered, which will make a effect that the snake moves a step.

(4) *CreateFood* function

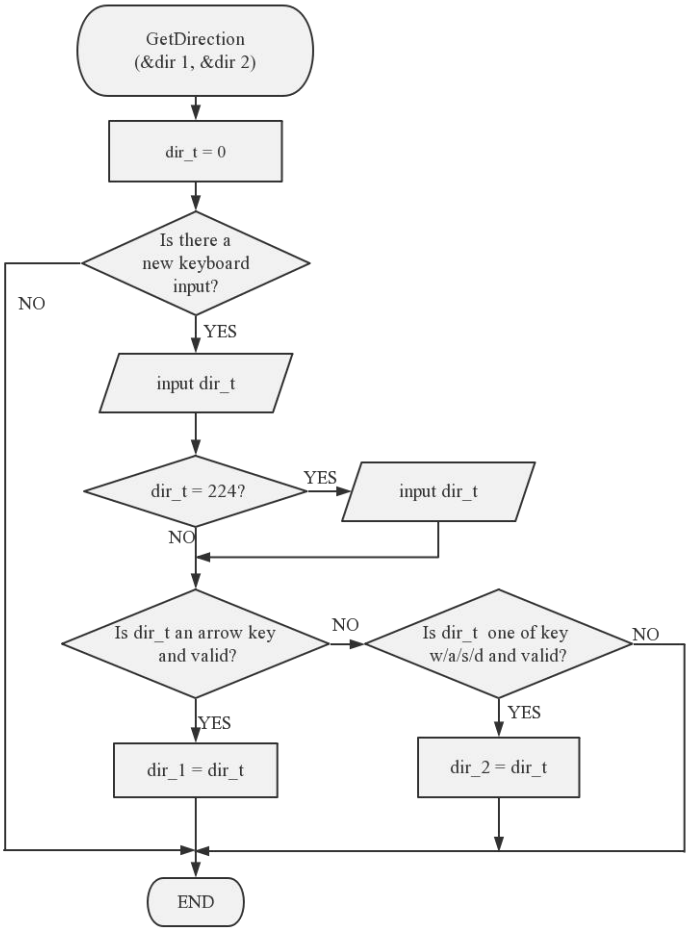
The coordinate of the walls are statics, when Generate random number, we can make sure that the food is created within the walls by some circulations. But the snake is moving, so after the food is created, we should check if it falls on the snake body, if so, create it again.



3) Double Mode

(1) *GetDirection* function

The key point is that each round in the loop only last about 250 millisecond, is a very short time. And in most cases, two players won't press keyboard too often. So we took advantage of the fast speed of computer and the slow reaction of human. In each round, it only captures one input (if there is one), and then change the corresponding direction while the other direction will remain unchanged. Later two snakes will be moved separately in other functions.

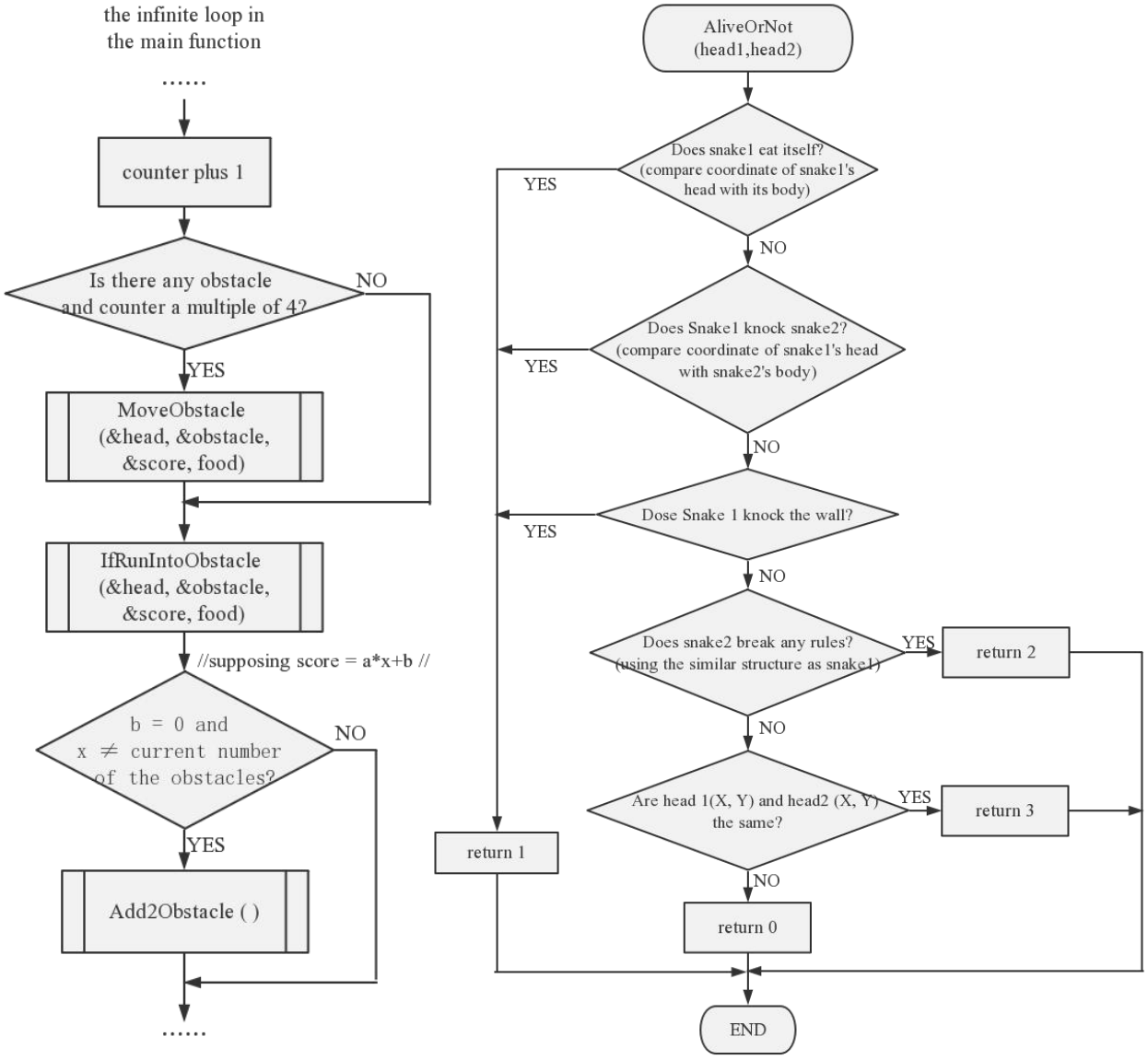


(2)*AliveOrNot* function

Another important part is the liability judgment.

Do a series of comparison for snake1 and snake2 to see if they break any rule. For example,if snake1 knocks snake2, it means the coordinate of snake1's head is the same as one part of snake2's body. If snake1 hitting the wall, it means the coordinate of the first node in the list is the same as the wall.

By returning different values, we can see who breaks the rules and later that player will be punished in the *GameOver* function, which means score minus 5.



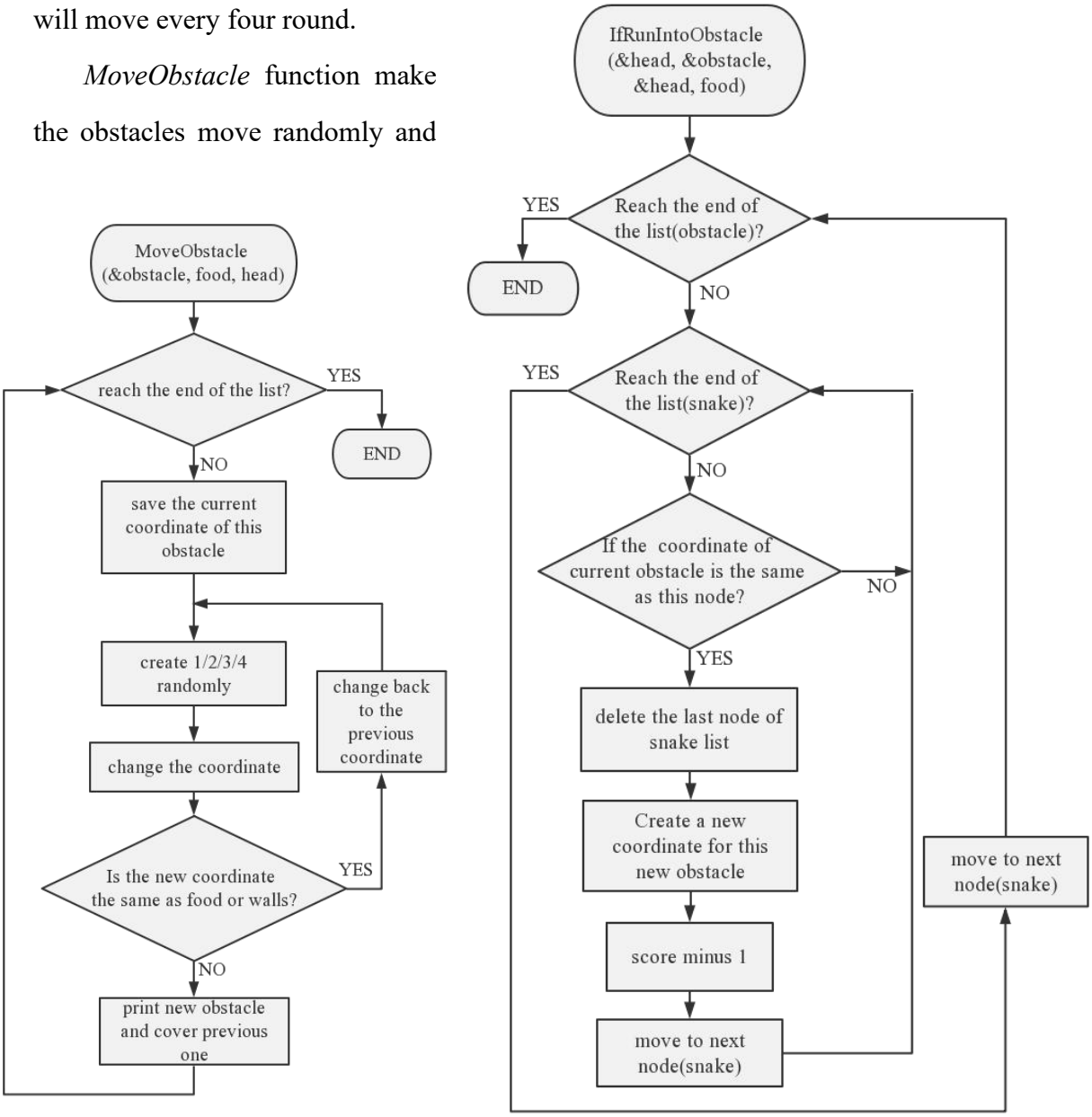
4) Moving-obstacle Mode

As we want to achieve the effect that every time the player gets a certain amount of more points a new obstacle will be created, the obstacles should also be a linked list.

(1) part of *main* function

This is the mainly changed part of the infinite loop in the moving-obstacle mode. We use the variable *counter to* make sure that the moving speed of obstacle is a quarter of the snake. Because in each round the snake will move a step, but obstacles will move every four round.

MoveObstacle function make the obstacles move randomly and



automatically.

Then the *IfRunIntoObstacle* function will check if the player meet an obstacle.

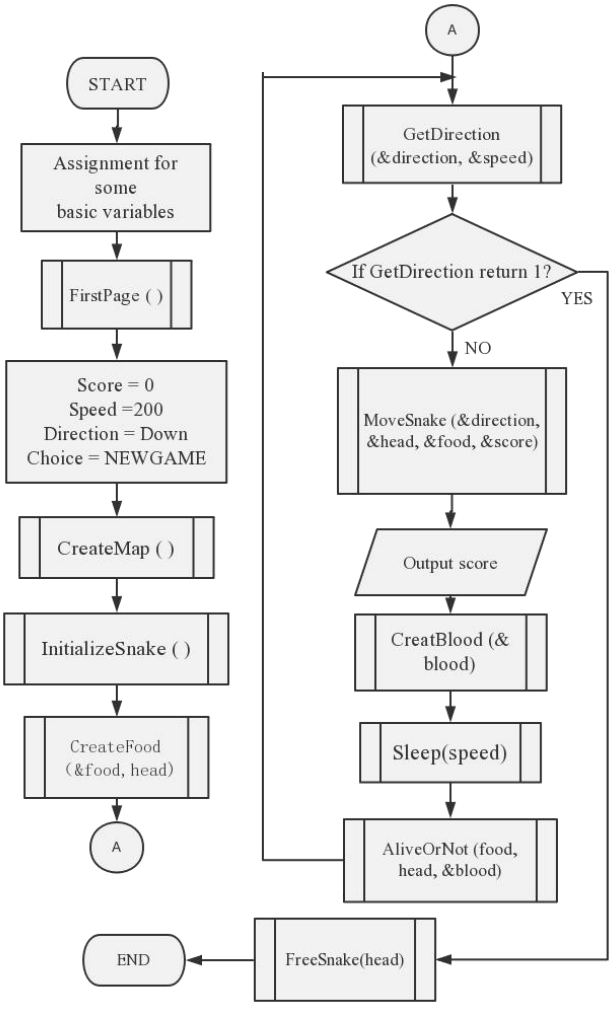
Supposing every time the player gets *a* more points a new obstacle will be created, and $score = ax + b$, if $b = 0$ and x doesn't equal to the current number of the obstacles, it means that the player has got *a* more points while a new obstacle hasn't been created, then the *Add2Obstacle* function will add a new node to the linked list which records all the obstacles.

(2) *MoveObstacle* function

First, save the coordinate of the current obstacle. Numbers of 1-4 will be created randomly to represent the direction. Then according to that change the coordinate. Next we check if the obstacle has covered the food or run out of the wall, if so,change the coordinate back to the previous one and create a direction again. If not, print an obstacle at the new position and cover the previous one to make the moving effect.

(3) *IfRunIntoObstacle* function

This function use a double loop to compare each node of the two lists to see if the snake runs into an obstacle.



5) Infinite Mode

(1)*main* function

In the *main* function of infinite mode. Here we only add one new function to set the blood in this mode, the *CreateBlood* Function. In the *GetDirection* Function, the snake still moves normally. But if the snake dies

it will come to this circulation,that means, the snake gets another life.

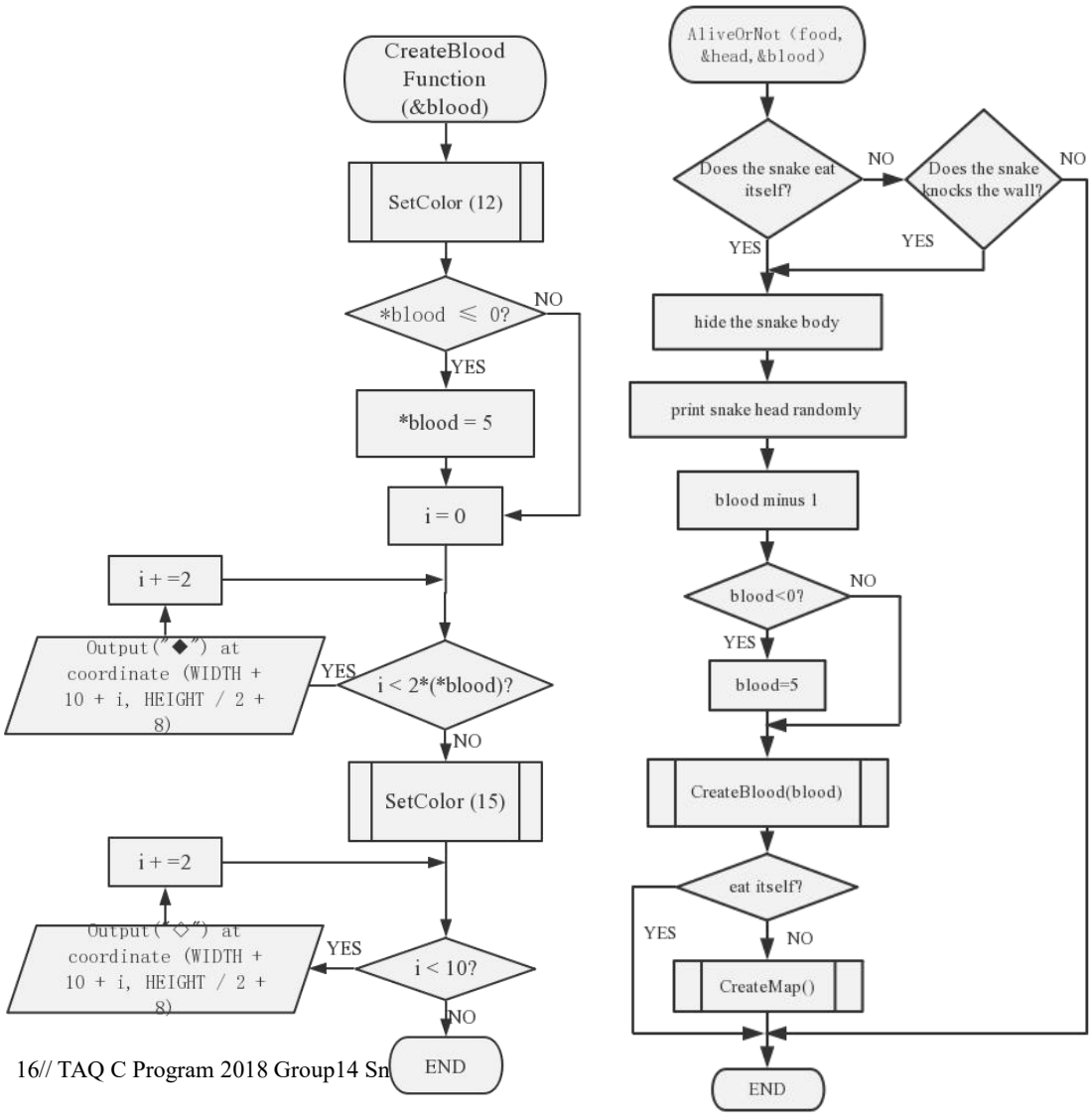
Second, we print blank patterns at the coordinate of the snake body to cover the useless snake body.

Third,we use the *CreateMap* Function in the Single Mode. When the snake head meet the wall, the previous snake body will be hided and the wall will be reprint at once, that means, it won't be eaten!!

Forth, we set the food with a little new function.The way to control the speed is similar with the way to set the function of F1, F2 in the Single Mode. One difference is to making it speed up and down randomly. Numbers of 1 or two will be created by *rand* function, if the result is 1, speed up, otherwise slow down.

(2) *CreateBlood* function

We use several simple circulations and create blood by setting the coordinate and



printing solid and hollow patterns.So if the snake dies, *blood* will minus 1 or become full value-5. Then, introduce this *CreateBlood* function according to how many times the snake has dead.

（3）*AliveOrNot* function

First, to solve the Analysis 1 problem, we use *rand* function to create new coordinate of its head.For the snake will never die, there is no need for this function to return a value. It will be same as the initial situation after its new printing.

6) Testing table

	Input	Expected Result	Real Result
Single Mode	Press right arrow key when moving upward.	Snake will turn right.	
	Press down arrow key when moving upward.	Snake will keep moving upward.	
	Press F1.	Snake will speed up.	
	Eat a food.	Score will plus 1 and snake will grow longer. A new food will show up.	
	Hitting the wall.	Game over.	
	Press ESC.	Exit directly.	
	Press two keys two quickly.	Error.	
Double Mode	Press ‘a’ key when snake2 moving upward.	Snake2 will turn left while snake1 will keep moving unaffectedly.	
	Snake1 knocks snake2.	Game over. Score of snake1 will minus 5.	
	One player keep pressing the	Error.	

	keyboard.		
Moving-obstacle Mode	Player gets a certain amount of more score.	A new obstacle will show up randomly.	
	Snake meet an obstacle.	Score will minus 1 and snake will become shorter.	
Infinite Mode	Hit the wall or eat itself	Blood minus 1 and the snake is newly created	
	Meet the food	The snake speeds up or down	
	Blood is 1 and it dies again	The same as first one except for turning into full blood	
	Press ESC	Exit the game.	

4. Implementation

We use VS2017 to write the code, it has updated some safe functions which may be different from other IDE.

1) Linked list and coordinate

```
typedef struct SNAKE {     //use a linked list to represent a snake
	COORD cor;
	struct SNAKE *next;
}snake;

typedef struct _COORD {     //definition of COORD in windows.h
	SHORT X;
	SHORT Y;
} COORD, *PCOORD;

void Pos(short x, short y)   //set position for the cursor
{
	COORD pos = { x,y };
	HANDLE hOutput;
	hOutput = GetStdHandle(STD_OUTPUT_HANDLE);
	SetConsoleCursorPosition(hOutput, pos);
}

int CompareCoordinate(COORD pt1, COORD pt2)   //compare two coordinates
{
	return (pt1.X == pt2.X&&pt1.Y == pt2.Y);
}
```

We use some windows API functions to set position of the cursor.

COORD is a structure that has been defined in windows.h.

CompareCoordinate function is a basic tool in the program.

2) Single Mode

(1) *main* function

```
int main()
{
	.....
	CreateMap();
	PrintRules(&highest);
	choice = NEWGAME;
	if ((err = fopen_s(&file, "memory_singlemode.txt", "rb")) == 0) {
		char ch = fgetc(file); fclose(file);
		if (ch != EOF) //if the file is not empty
			choice = Choose();//choose whether to continue the unfinished game
	}

	if (choice == CONTINUE)
		head = Load(&score, &direction, &food, &speed);
	else {
		head = InitializeSnake();
		CreateFood(&food, head);
	}

	for (;;) {
		if (GetDirection(&direction, &speed)== 1)
			Save(&score, &direction, &food, &speed, head);
		MoveSnake(direction, &head, &food, &score);
		Pos(WIDTH + 4, HEIGHT / 2 + 5);
		SetColor(15);
		printf("Score: %d", score);
		Sleep(speed);
		if (AliveOrNot(head))
			break;
	}

	FreeSnake(head);
	if (GameOver(score,highest)== NEWGAME)
		goto newgame;

	return 0;
}
```

If a file is empty, it only have one char which indicates the end of the file (EOF).

(2)*GetDirection* function

```
int GetDirection(int *direction, int *speed)//capture key presses from the user
{
	int pause = 0;
	int new_dir = 0;
	if (_kbhit()) {
```

```
            new_dir = _getch();
            if (new_dir == 224 || new_dir == 0)new_dir = _getch();//use _getch twice to capture
function keys
            switch (new_dir) {
            case 32:        //presse space key to pause
                  while (pause != 32)
                        pause = _getch(); break;
            case 59:        //presse F1 to speed up
                  if ((*speed) >= 50)
                        (*speed) -= 50; break;
            case 60:
                  (*speed) += 50; break;    //F2 slow down
            case UP:
            case DOWN:
            case RIGHT:
            case LEFT:
                  if (abs(new_dir - *direction) != 2 && abs(new_dir - *direction) != 8) {     //opposite
direction is invalid
                        *direction = new_dir; break;
                  }
            case ESC:
                  new_dir = 1; break;
            default:
                  new_dir = 0; break;
            }
      }
      return new_dir;
}
```

_kbhit checks if there is any current input from the keyboard. If the function returns a nonzero value, a keystroke is waiting in the buffer. Then use *_getch* to get the input. But arrow keys and function keys are a little bit special, it will return two values after being pressed. The first is 224 or 0, the second is the corresponding value. So *_getch* should be called twice.

(3) *MoveSnake* function

```
void MoveSnake(int direction, snake **head, COORD *food, int *score)     //move snake
{
      snake *p;
      COORD tail = (*head)->cor;          //save the previous coordinate of snake head
      COORD temp;

      switch (direction) {                                          //change the coordinate of snake head
      case UP:
            (*head)->cor.Y--; break;
      case DOWN:
            (*head)->cor.Y++; break;
      case RIGHT:
            (*head)->cor.X += 2; break;
      case LEFT:
            (*head)->cor.X -= 2; break;
      }
```

```
    Pos((*head)->cor.X, (*head)->cor.Y);                    //print new head
    SetColor(10);
    printf("○");
    Pos(tail.X, tail.Y);
    printf("●"); HideCursor();

    for (p = (*head)->next; p != NULL; p = p->next) {       //update the coordinates of snake body
        temp = p->cor;
        p->cor = tail;
        tail = temp;    //when the loop is over,tail is the coordinate of the previous snake tail
    }

    if (CompareCoordinate(*food, (*head)->cor)) {

        for (p = (*head)->next; p->next != NULL; p = p->next);    //find the end of the linked list
        p->next = (snake *)malloc(sizeof(snake));
        p->next->cor = tail;          //add a new node whose coordinate is the previous snake tail
        p->next->next = NULL;

        CreateFood(food, (*head));                    //create a new food
        (*score)++;
    }
    else {
        Pos(tail.X, tail.Y);//cover the previous snake tail
        SetColor(0);
        printf("●");
    }
}
```

Changing the linked list requires a double pointer. Pay attention that some variable is passed by value and some is passed by pointer.

(4)*CreateFood* function

```
void CreateFood(COORD *food, snake *head)
{
    snake *p = NULL;
    int in_snake = 0;
    srand((unsigned int)time(NULL));

    do {
        in_snake = 0;
        int x = (rand() % (WIDTH - 5)) + 3;          //make sure that food is in the map
        food->X = (x % 2) ? x + 1 : x;          //x-coordinate is even number
        food->Y = (rand() % (HEIGHT - 1)) + 1;
        for (p = head; p != NULL; p = p->next) {        //if food coincides with snake
            if (CompareCoordinate(p->cor, *food))
                in_snake = 1; break;
        }
    } while (in_snake);

    Pos(food->X, food->Y);
    SetColor(13);
    printf("★");
}
```

The reason why x-coordinate is even number is that '★' '■' '●' (representing

food, wall and snake) cover too unit in horizontal direction in the console window. And the default x-coordinate of food and walls are all even number. They must be consistent.

2) Double Mode

(1) *GetDirection* function

```
void GetDirection(int *dir1, int *dir2)
{
	int dir_t = 0;
	if (_kbhit()) {
		dir_t = _getch();
		if (dir_t == 224)	//
			dir_t = _getch();
	}
	//only arrow keys and 'awsd' are valid
	if ((dir_t == 72 || dir_t == 75 || dir_t == 77 || dir_t == 80) && abs(dir_t - *dir1) != 2 && abs(dir_t - *dir1) != 8) {
		*dir1 = dir_t;
	}
	else if ((dir_t == 100 || dir_t == 97 || dir_t == 115 || dir_t == 119) && abs(dir_t - *dir2) != 3 && abs(dir_t - *dir2) != 4) {
		*dir2 = dir_t;
	}
}
```

(2) *AliveOrNot* function

```
int AliveOrNot(snake *head1, snake *head2)
{
	snake *p;
	for (p = head1->next->next; p != NULL; p = p->next) {	//if eats itself
		if (CompareCoordinate(head1->cor, p->cor))
			return 1;
	}
	for (p = head2->next; p != NULL; p = p->next) {
		if (CompareCoordinate(head1->cor, p->cor))
			return 1;
	}
	//if run into wall
	if (head1->cor.X == 0 || head1->cor.X == WIDTH || head1->cor.Y == 0 || head1->cor.Y == HEIGHT)return 1;

	for (p = head2->next->next; p != NULL; p = p->next) {
		if (CompareCoordinate(head2->cor, p->cor))
			return 2;
	}
	for (p = head1->next; p != NULL; p = p->next) { //if knocks the other
		if (CompareCoordinate(head2->cor, p->cor))
			return 2;
	}
```

```
    if (head2->cor.X == 0 || head2->cor.X == WIDTH || head2->cor.Y == 0 || head2->cor.Y ==
HEIGHT)return 2;

    if (CompareCoordinate(head1->cor, head2->cor))return 3; //two snakes bump together head to
head
    return 0;
}
```

3) Moving-obstacle Mode

(1) the infinite loop in the *main* function

```
for (;;) {

        if(GetDirection(&direction, &speed)==1)
            Save(&score, &direction, &speed, &obstacle_number,head);
        MoveSnake(direction, &head, &food, &score);

        counter++;    //the speed of the obstacle is a quarter of the snake
        if(obstacle!=NULL&&counter%4==0)
            MoveObstacle(&obstacle, food, head);

        IfRunIntoObstacle(&head, &obstacle, &score,food);

        //if the player gets OBSTACL_COUNT more score ,create a new obstacle
        if (score%OBSTACL_COUNT == 0&&score/ OBSTACL_COUNT != obstacle_number) {
            obstacle_number++;
            Add2Obstacle(CreateObstacle(head, food, obstacle), &obstacle);

        if (AliveOrNot(head))
                break;
        }
```

OBSTACL_COUNT is a macro in our header file. Programmer can change it to change the difficulty level of the game.

(2) *MoveObstacle* function

```
void MoveObstacle(Obstacle **obstacle, COORD food, snake *head)
{
    Obstacle *r;
    int direction ;
    int recreat;
    COORD previous;
    srand((unsigned int)time(NULL));
    for (r = *obstacle; r != NULL; r = r->next) {
        previous = r->cor;
        do {
            direction = rand() % 4 + 1;
            switch (direction) {                              //change the coordinate of the obstacle
            case 1:                  // according to the direction which has been created randomly
                r->cor.Y--; break;
            case 2:
                r->cor.Y++; break;
            case 3:
```

```
                        r->cor.X += 2; break;
                case 4:
                        r->cor.X -= 2; break;
                }
                recreat = 0;    //obstacle can't cover food or bump the wall,if socreat direction again
                if (CompareCoordinate(food, r->cor) || r->cor.X == 0 || r->cor.X == WIDTH ||
r->cor.Y == 0 || r->cor.Y == HEIGHT) {
                        recreat = 1;
                        r->cor = previous;
                }
        } while (recreat);

        Pos(r->cor.X, r->cor.Y);                //print new obstacle
        SetColor(15);
        printf("■");
        Pos(previous.X, previous.Y);    //cover the previous obstacle
        SetColor(0);
        printf("■"); HideCursor();
    }
}
```

(3) *IfRunIntoObstacle* function

```
void IfRunIntoObstacle(snake **head, Obstacle **obstacle, int *score,COORD food)
{
    snake *p,*q,*pre;
    Obstacle *r,*l;
    for (r = *obstacle,l=NULL; r != NULL; l=r, r = r->next) {
                                //compare each obstacle with each node of the snake
        for (q = *head; q != NULL; q = q->next) {
            if (CompareCoordinate(r->cor, q->cor)) {
  //if you meet an obstacle,score minus 1 and delete the last node in the snake list
                (*score)--;
                for (p = (*head)->next, pre = NULL; p->next != NULL; pre = p, p = p->next);
                Pos(p->cor.X, p->cor.Y);
                SetColor(0);
                printf("●");
                pre->next = NULL;
                free(p);
                r->cor = CreateObstacle(*head, food, *obstacle);
            }
        }
    }
}
```

4) Infinite Mode

(1) *main* function

```
int main()
{
    .....
    blood = BLOOD;
    CreateMap();
    PrintRules();
    head = InitializeSnake();
```

```
	CreateFood(&food, head);
	pause = 0;
	while (pause != 32)//Press space to start the    game
		pause = _getch();

	for (;;) {
		if (GetDirection(&direction, &speed)== 1)
			break;
		MoveSnake(direction, &head,&food, &score, &speed);
		Pos(WIDTH + 4, HEIGHT / 2 + 5);
		SetColor(15);
		printf("Score: %d", score);
		Pos(WIDTH + 4, HEIGHT / 2 + 7);
		SetColor(12);
		printf("Blood: %d", blood);
		CreateBlood(&blood);
		Sleep(speed);
		AliveOrNot(food, head, &blood);
	}
	FreeSnake(head);
	return 0;
}
```

(2) *CreateBlood* function

```
void CreateBlood(int *blood)
{
	SetColor(12);
	int i;
	if (*blood <= 0) {
		*blood = BLOOD;
	}
	for (i = 0; i < *blood * 2; i += 2)
	{
		Pos(WIDTH + 10 + i, HEIGHT / 2 + 8); printf("◆");
	}
	SetColor(15);
	for (; i < 10; i += 2)
	{
		Pos(WIDTH + 10 + i, HEIGHT / 2 + 8); printf("◇");
	}

}
```

(3) AliveOrNot function

```
void AliveOrNot(COORD food, snake *head, int *blood)
{
	.....
	//If break the rules
	if (head->cor.X == 0 || head->cor.X == WIDTH || head->cor.Y == 0 || head->cor.Y == HEIGHT
|| eat_self)
	{
		//cover the snake body
		snake *hp = head;
		while (hp != NULL) {
			Pos(hp->cor.X, hp->cor.Y);
```

```
                printf(" ");
                hp = hp->next;
            }
            HideCursor();

            int x = (rand() % (WIDTH - 7)) + 3;
            head->cor.X = (x % 2) ? x + 1 : x;
            head->cor.Y = (rand() % (HEIGHT - 3)) + 1;

            if (--*blood < 0)
                *blood = 5;
            CreateBlood(blood);
            if (!eat_self) {
                CreateMap();
            }
        }
}
```

5. Testing and Debugging

	Input	Expected Result	Real Result
Single Mode	Press right arrow key when moving upward.	Snake will turn right.	√
	Press down arrow key when moving upward.	Snake will keep moving upward.	√
	Press F1.	Snake will speed up.	√
	Eat a food.	Score will plus 1 and snake will grow longer. A new food will show up.	√
	Hitting the wall.	Game over.	√
	Press ESC.	Exit directly.	√
	Press two keys two quickly.	Error.	Exit abnormally, open the game next time the player will be asked whether to continue.

Double Mode	Press ‘a’ key when snake2 moving upward.	Snake2 will turn left while snake1 will keep moving unaffectedly.	√
	Snake1 knocks snake2.	Game over. Score of snake1 will minus 5.	√
	One player keep pressing the keyboard.	Error.	The movement of both snakes will become out of control.
Moving-obstacle Mode	Player gets a certain amount of more score.	A new obstacle will show up randomly.	√
	Snake meet an obstacle.	Score will minus 1 and snake will become shorter.	√
Infinite Mode	Snake head meets food	Speed will change randomly.	√
	Run into wall or eat itself	Blood will minus 1 and snake will appear at a random position.	√
	Blood value falls to 1. Then repeat the last operation	Blood will be 5 and never game over.	√

// ‘√’ means the real result is the same as the expected result. //

(1)Single Mode

The most basic algorithm is the infinite loop in the main function. Each round last about 250 ms.

Why 250 ms? We all know the persistence of vision and most film has 24 frames per second. At first, we tired to use 50 as the argument of *Sleep* function. But in fact,

that is too fast for player to react. Actually, when player press F1 to speed up, *speed* (the argument of *Sleep* function) will minus 50, and the minimum value of *speed* is 50.

But this caused a little problem, that the snake is always twinkling while moving. Because the moving effect is achieved by printing the head and covering the tail continuously. Though 250 ms is suitable for player to react, it's too slow for human eyes.

As in one round, we only capture one input, so if the player press two key too quickly, it may cause error. By testing, we found that when the player press two opposite arrow keys too fast, the console window will be closed. And when open the game next time, it seem that the program thought the player want to save data, because the player will be asked whether to continue. We haven't figure out how to solve that problem.

When player press ESC, the console window will be closed immediately, which is a bit abrupt. Maybe we should set a function to ask whether to exit or go back to the game, like most games do.

(2) Double Mode

In double mode, we also only capture one input from the keyboard. So if one player deliberately keep pressing the keyboard, the other player may have no chance to get his/hers input received. The worst result is that the movement of both snakes become out of control. But in general, player won't change direction too often, and the game runs well.

(3) Moving-obstacle Mode

The obstacle moves according to direction (represented by 1-4) which is created by *rand* function. As the odds of four direction is almost the same, the obstacles only move with in a small area around where they have been created, which is a little bit boring. Maybe we can design an algorithm to make the obstacles move towards the snake.

(4) Infinite Mode

First, we find if the snake hits the wall in one direction, it will be newly created in the same direction. This makes us feel strange when testing the game. So we consider making the snake created in a different randomly direction might be better. Certainly, the *rand* Function must be used here.

Second, two ways to die will lead to newly creating. But we think if it eats itself, it's no need to create again. So under these circumstances, leave one effect that is making the *blood* minus 1 is OK.

Third, we think as the Infinite Mode, the food functions can be a lot. Adding new functions, such as double-score food, reducing-score food and so on can be taken into consideration to increase the flexibility of this mode.

6. Result&Conclusion

We finished the basic functions of the game Snake using C language, and based on that, we made multiple modes: double mode, infinite mode and moving-obstacle mode, each mode has its own new functions.

As we have mentioned in the Testing and Debugging section, but there are still some problem remain to be solved. For example, we want to combine four modes in one program, but that needs too much select statements, maybe we need some knowledge beyond C language. And there is still plenty of room for improvement. We want to learn further about shortest-path search algorithms, and write a program to which can make the snake search for food on its own, and improve the double mode into a man-machine competition.

But generally speaking, we think we achieved our goal successfully. What's more, during the process , we learn more about the superiority of the linked list. We get a deeper understanding of the pointer. And we know more about Windows API function. We also learn how to improve a project and learn about the importance of the comfortable game experience.