Hong Kong Taoist Association
Tang Hin Memorial Secondary School
Information and Communication Technology
School-Based Assessment

Pac-Man

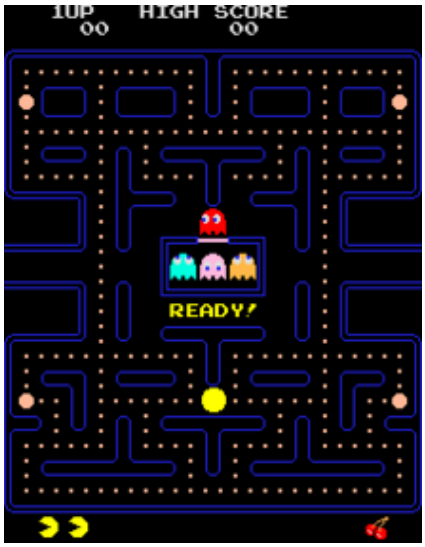
Li Ho Yuen 6B 8

Contents

Objective

Introduction

Pac-Man is an action maze chase video game; the player controls Pac-Man through an enclosed maze. The objective of the game is to eat all the dots placed in the maze while avoiding four colored ghosts—Blinky (red), Pinky (pink), Inky (cyan), and Clyde (orange)—who pursue Pac-Man. When Pac-Man eats all the dots, the player advances to the next level. Levels are indicated by fruit icons at the bottom of the screen.



The ghosts are in the center with Pac-Man below them. At bottom left is the player's life count, and at bottom right the level icon (in this case a cherry). At top is the player's score.

If Pac-Man is caught by a ghost, he loses a life; the game ends when all lives are lost. Each of the four ghosts has its own unique artificial intelligence (A.I.), or "personality": Blinky gives direct chase to Pac-Man; Pinky and Inky try to position themselves in front of Pac-Man, usually by cornering him; and Clyde switches between chasing Pac-Man and fleeing from him.

Placed near the four corners of the maze are large flashing "energizers" or "power pellets." When Pac-Man eats one, the ghosts turn blue with a dizzied expression and to reverse direction. Pac-Man can eat blue ghosts for bonus points; when a ghost is eaten, its eyes make their way back to the center box in the maze, where the ghost "regenerates" and resumes its normal activity. Eating multiple blue ghosts in succession increases their point value. After a certain amount of time, blue-colored ghosts flash white before turning back into their normal forms. Eating a certain number of dots in a level causes a bonus item—usually a fruit—to appear underneath the center box; the item can be eaten for bonus points.

Workflow

I use Waterfall Model to complete my program. The Waterfall model follows a structured approach with distinct phases, including requirements gathering, design, implementation, and others.

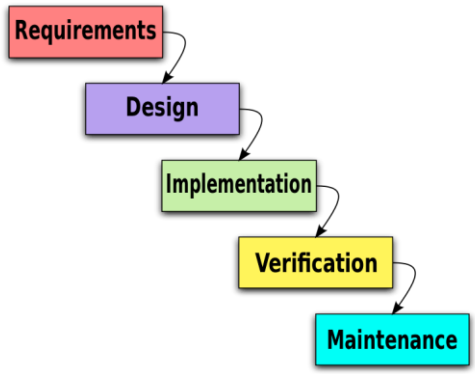


Fig1.2 The Waterfall Model

The Waterfall model is relatively simple to understand and implement, making it accessible for most people.

Design

Programming language

In the game Pac-Man, there are several key components that contribute to the gameplay experience:

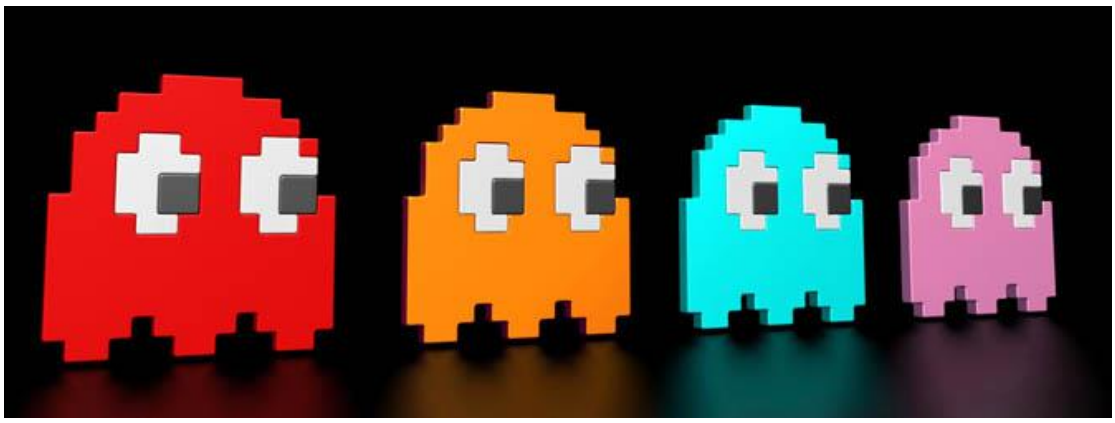
Pac-Man: The main character and player-controlled entity in the game. Pac-Man is a yellow, circular character with a voracious appetite for dots.

Maze: The playing area is represented by a maze consisting of a network of corridors and walls. The maze layout determines the paths that Pac-Man and the ghosts can traverse.

Dots: Scattered throughout the maze are small dots that Pac-Man must consume. Each dot adds points to the player's score and contributes to completing the level.

Power Pellets: Larger, flashing dots known as power pellets are strategically placed in the maze. When Pac-Man consumes a power pellet, the ghosts temporarily turn blue and become vulnerable. During this time, Pac-Man can chase and eat the ghosts for extra points.

Ghosts: The antagonistic entities in the game, there are typically four ghosts: Blinky (red), Pinky (pink), Inky (cyan), and Clyde (orange). The ghosts roam the maze, attempting to catch and eliminate Pac-Man. Each ghost has its distinct behavior and movement patterns, adding complexity and challenge to the gameplay.



Score and Lives: The game keeps track of the player's score, which increases with each dot, power pellet, ghost, or fruit collected. Players typically start with a set number of lives, representing the number of times Pac-Man can be caught by a ghost before the game ends.

These components work together to create Pac-Man. Therefore, I need to figure out whether procedural programming or object-oriented programming (OOP) is more suitable for this program.

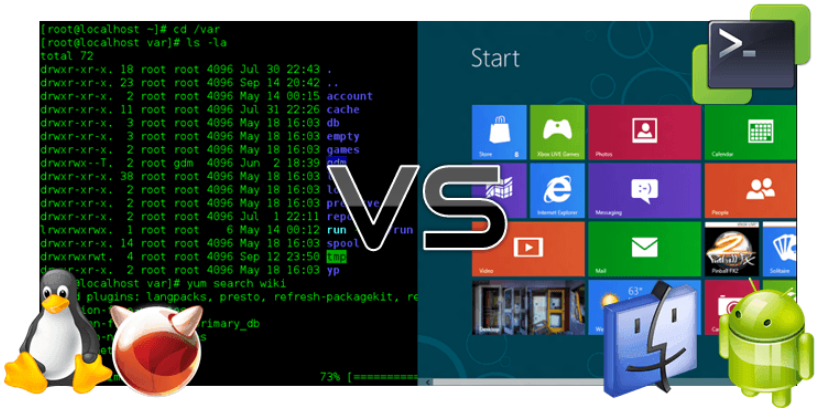
OOP is often more suitable for complex projects with multiple interacting components. It provides a structured and modular approach to handle complexity and offers better code organization and reusability. Procedural programming may be more appropriate for simpler projects with straightforward logic and data processing requirements.

Also, OOP can enhance code maintainability and scalability. By encapsulating data and methods within objects, OOP promotes modularity and information hiding, which makes it easier to update or modify specific components without affecting the entire system. This can be beneficial for long-term maintenance and future enhancements. Procedural programming may be more suitable for smaller projects or ones that are not expected to undergo significant changes over time.

Therefore, I chose C++ for my program. C++ is an object-oriented programming language, where C is a procedural programming language, it's easier to write program in C++ with some basic knowledge about C, other than choosing Python, Java, or other object-oriented programming languages to write the program.

User interface

I need to choose whether command line interface (CLI) or graphical user interface (GUI) is more suitable for the program.



Typical examples of CLI and GUI

Consider the complexity of the tasks or operations that the users need to perform. CLI interfaces are well-suited for complex and repetitive tasks, as they often provide more flexibility and scripting capabilities. If the tasks involve chaining multiple commands or require fine-grained control, a CLI might be more efficient. GUIs, on the other hand, are generally better for tasks that involve visual data manipulation, interactive operations, or require a more user-friendly and intuitive experience.

On the other hand, I should also consider the development effort and time required to implement each interface. Building a GUI often involves designing and developing visual elements, handling user input, and considering usability aspects. CLI interfaces, while they still require careful design, may have a simpler implementation as they primarily focus on text-based interactions.

Since Pac-Man doesn't require high visual effect, CLI is more suitable for this program.

Map

As I mentioned, there are several components that contribute to the gameplay experience. The first component is the maze. Without the maze, it's meaningless to put the Pac-Man and the ghosts in a blank. Therefore, I need to define the map. The map consists of space, wall, pean, and super peans. Since the program runs in command line, all the elements shown on the panel are strings. Therefore, the map can be seen as a 2-D array. (Two-dimensional array are arrays where the data element's position is referred to, by two indices. The name specifies two dimensions, that is, row and column.) Also, we can see each point on the panel as a structure. Then, we can define the following class:

```
enum DIRECTION { UP, DOWN, LEFT, RIGHT };

class Position
{
protected:
    string key;//The string of that point
    int x, y;  //The coordinate of that point
    int color;
```

```
 //Set the string
 void setKey(const string str);
 //Set the coordinate
 void setXY(int a, int b);
 //Print the string on particular coordinate
 void print();
 //Clear the string on particular coordinate
 void clear();
```

The class of the point on the panel

After defining the following class, it is easy to print the string on the coordinate on the panel. For example, peans, ghosts, player, etc.

With “Position” this class, Inheritance can be used to define the type of that point.

```
//Type of the point on the map, such as these:
enum MAP_POS_TYPE { SPACE, WALL, PEAN, SUPER_PEAN };
class MapPos: public Position
{
    int type;
public:
    //Set the type of that point
    void setType(int t);
    int getType() { return type; }
};
```

The type class of the point

Thus, encapsulation can be used to package those classes into one class.

```
class Map
{
    //Every point of the map, seen as different types of elements.
    MapPos points[MAP_SIZE][MAP_SIZE];
    //Player's score
    int scores;
    //Winning score
    int target_scores;
    //The freezing time of the ghosts after eating the super pean
    int freezeTime;
    void findPath(Position &A, Position&B);
```

Using encapsulation and inheritance can enhance code organization and modularity, making it easier to understand, maintain, and update the codebase. Encapsulation protects data from unintended modifications and inheritance promotes code reuse, reducing redundancy and promoting a more efficient development process. It also supports the principle of polymorphism, allowing objects of different subclasses to be treated uniformly, providing flexibility and extensibility to the codebase.

Thus, public classes should be defined for further reuse. For example, map class should be linked to ghost’s class, player’s class, main game’s class, etc.

```
public:
    friend class Game;
    friend class Ghost;
    Map() {};
    //initialize map using file
    void init(const char* filepath, Pacman &pacman, vector<Ghost> &ghosts);
    //print the point on (x,y)
    void renew(int x, int y) { points[x][y].print(); }
    //return the type of point on (x,y)
    int goXY(int x, int y) { return points[x][y].getType(); }
    //delete the pean eaten by the player on (x,y)
    void delPean(int x, int y);
    //check (x,y) is exceed from the map or not
    bool oK(int x, int y);
    //delete the pean eaten by the player on (x,y)
    void delSuperPean(int x, int y);
    //find path
    int findDir(Position &A, Position&B);
};
```

And that’s the basic header for the map of Pac-Man. For details, we will talk about it later in the implementation part. We should switch to other classes that also play an important role in the game.

Player

We can regard the player as a special point on the map too since we need to display it on the panel. The map class has done lots of things for this program. Therefore, we only need to define some basic work for the player:

```
class Pacman:public Position
{
    DIRECTION go;
public:
    friend class Game;
    Pacman() {};
    //initialize player's position
    void init(int x, int y);
    //move and check for suitable movements
    void move(int dir, Map &map);
    //return the coordinate of the player
    void getXY(int &a, int &b);
```

The direction is inherited from position class, where exists as Enum of up, down, left, and right.

Ghosts

Now we should give some challenges to the player.
Like the player's class, ghosts are also the points on the map and display it on the screen. Since map class has done lots of things for this program, we only need to define some basic work for the ghosts:

```
public:
        Ghost() {};

        Ghost(int x, int y); //initalise
        int move(Map &map,Pacman &pacman); //move and check for suitable movement
        int move(int dir,Map &map, Pacman &pacman);//same
        //check for the ghosts has crushed into player or not. If player get super pean, ghosts will back to home.
        bool hit(Pacman &pacman, Map &map);
};
```

A screenshot from mac (I don't know why I use mac to take the screenshot pls don't ask me

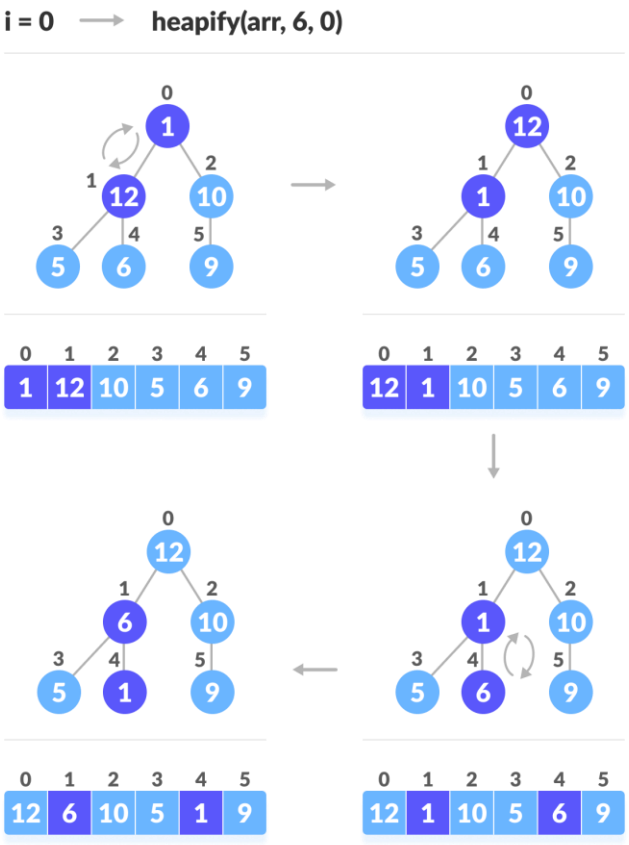
Records

We should record the score no matter if it's lost or won. Since it may have lots of records played by the user, we should only output some of them. Therefore, sorting should be used to show some of the best records. We should decide which sorting is more suitable for this class.

I have limited the number of records outputted to 10. Therefore, the size of the array to store the records won't be too large. For a small amount of data, various sorting algorithms can be used, but some algorithms are particularly efficient for small data sets due to their simplicity and low computational overhead.

<algorithm> library include lots of types of sorting. Including quicksort, heapsort, insertion sort, etc. The choice of quicksort, heapsort, or insertion sort is determined based on the characteristics and size of the range being sorted.

Here, I would like to talk about heapsort first.

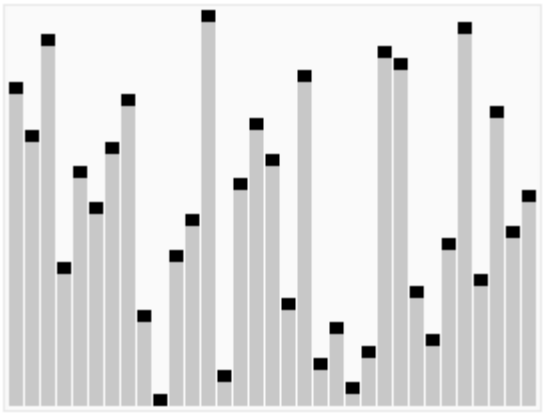


Just say something about it. Won't explain briefly. Binary tree is too difficult TAT

The first step in heapsort is to create a binary heap from the input array. This is done by starting with the given array and repeatedly applying a process called "heapify." Heapify ensures that the heap property is maintained, which means that for every node in the heap, the value of the node is greater than or equal to the values of its children (for a max heap). This process starts from the last non-leaf node and moves up to the root, ensuring that each subtree is a valid heap.

Then, once the heap is built, the maximum element (root of the heap) is extracted and placed at the end of the array. The heap is then updated by replacing the root with the last element in the heap and "sinking" this element down the heap to its correct position. This process is called "heapify-down" and is repeated until the heap is empty.

The extraction step is repeated until all elements have been extracted from the heap. Each time an element is extracted, it is placed at the end of the array in reverse order.



What?

After the extraction process is complete, the array will contain the sorted elements in ascending order. The sorted array is obtained by reversing the order of elements at the end of the array.

As we can see, Heapsort is efficient for large data sets. However, its performance is typically slower than quicksort for average cases. Therefore, the <algorithm> library won't choose these kinds of sorting since we're now handling light-weight data and they are time consuming when sorting these data. We can simply give determination to the library itself. Just call std::sort() is enough.

```
sort(items.begin(), items.end());
```

Why torment ourselves writing bubble sort something like that?

Also, I have decided to include <vector> library to store the records. Unlike traditional arrays, <vector> allows for dynamic resizing. We can easily add or remove elements from a vector without worrying about managing memory manually. And <vector> supports iterator-based traversal, which allows us to iterate over its elements using standard algorithms like std::find , etc. This makes it convenient to perform common operations on vectors without manually managing indices or loops.

Note that although <vector> acts like a stack which have “push” and “pop” operations, <vector> is not specifically a kind of stack. <vector> can work as a stack, but a stack cannot work as <vector> since we cannot insert or get an element at a random position in stack. Stacks take a container and only permit stack-like interactions with it. This effectively guarantees that all interactions with the container will obey LIFO (Last-In-First-Out): only the most recently added element in the container will be able to be accessed or removed. Therefore, <vector> should be used since we want to do things like iterate over elements or modify elements in arbitrary positions etc. So, we can define the record class as this:

```
class Record;
class RecordItem //For one record
{
    string name;//record name
    int score;
    int speed;//ghosts' speed
    int ghost_num;
    int steps_num;//To calculate the time played
    vector<int> steps;//same
public:
    friend class Record;
    friend class Game;
    RecordItem() { steps_num = 0;};
    RecordItem(const string& iname, int iscore) { name = iname; score = iscore; steps_num = 0;};
    bool operator <(const RecordItem &B) { return score > B.score; }//For sorting vector in decreasing order
};
```

This is for one record. To encapsulate it, another class will be defined.

```
class Record
{
    const char *filepath; //file name to store the records
    vector<RecordItem> items; //records
public:
    friend class Game;
    Record():filepath("game.record"){};
    //read the data from the file
    void read();
    //save the data to the file
    void save();
    //add a record
    void add(int the_score,const vector<int> &steps,int ghost_num,int speed);
    //show the record.
    void show();
};
```

Main Game

After finishing the design of those three modules, we can encapsulate those modules into a whole new structure for the game. The game class can simply call those functions or procedures provided by those modules.

```
class Game
{
    Pacman pac_man;
    vector<Ghost> ghosts;
    Map mapX;
    Record record;

    //speed of the ghost
    int GHOST_SPEED, speed_value;
    //time count
    int time_counter;
    //record the direction of the player or ghosts in a certain period
    vector<int> steps;
    bool play_flag;
```

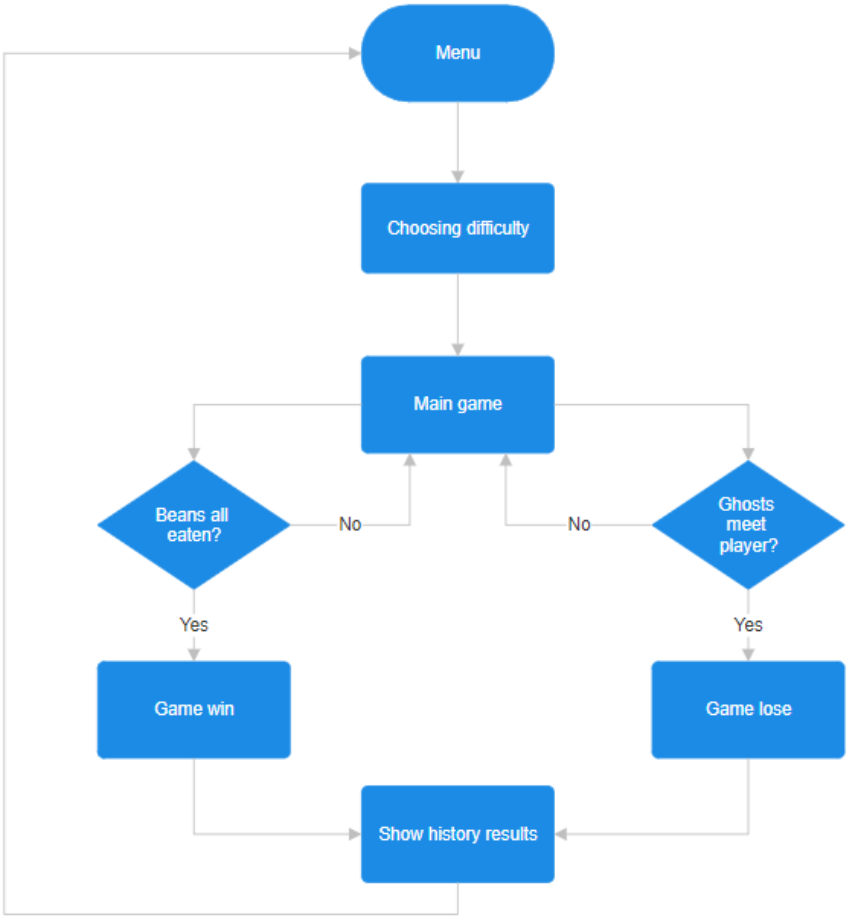
Also, the game class has public classes for the simple use in main.cpp file and has private classes for some functions that don't need to be used by other classes. For example, the layout UI for the instructions, game pause, etc.

Defining public classes and private classes in object-oriented programming is essential for establishing clear interfaces, promoting code reusability, facilitating inheritance and polymorphism, and achieving encapsulation. By keeping the classes private, their internal workings, data structures, and methods are not exposed to the outside world. This helps maintain data integrity, enhances security, and prevents direct manipulation of internal states. Private classes also enable us to modify or improve internal implementations without affecting external code that relies on those classes.

```
public:
    Game();
   //init
    void init();
    //start menu
    int start();
    //game difficulty
    bool settings();
    //test
    int show_record();
    bool loop();
    void play(int x);
```

```
private:
    //game pause
    void pause();
    bool game_over();
    bool game_win();
    //Score UI
    void infoUI();
    //Instruction UI
    void helpUI();
    //refresh the panel after pause
    void refresh();
```

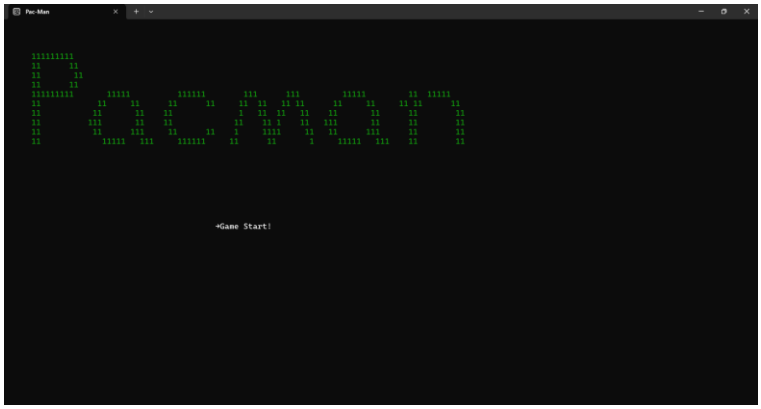
The main flow of the program should be like this:



Implementation

In this part, I'm going to show the execution of my design of the program.

Menu

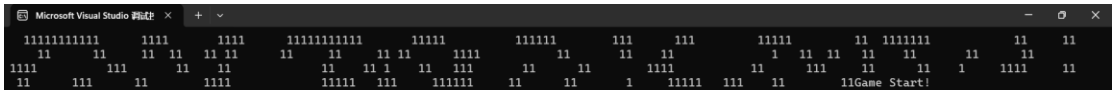


The program is run on the console. The reason why command line interface is preferred has been explained in "Design" part.

When we start the program, the title should be displayed. We can set the coordinate to output the string we want, simply:

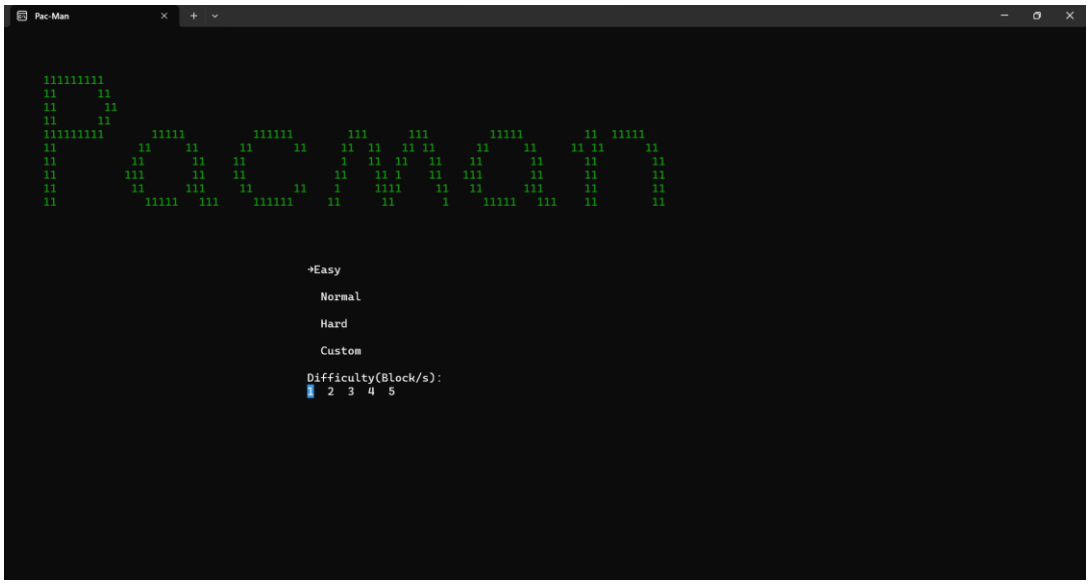
```
void Goto_XY(const int x, const int y)
{
    COORD position;
    position.X = x;
    position.Y = y;
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), position);
}
```

Therefore, we can make the layout become more user-friendly instead of simply arranging the layout in lines. Like:



What?

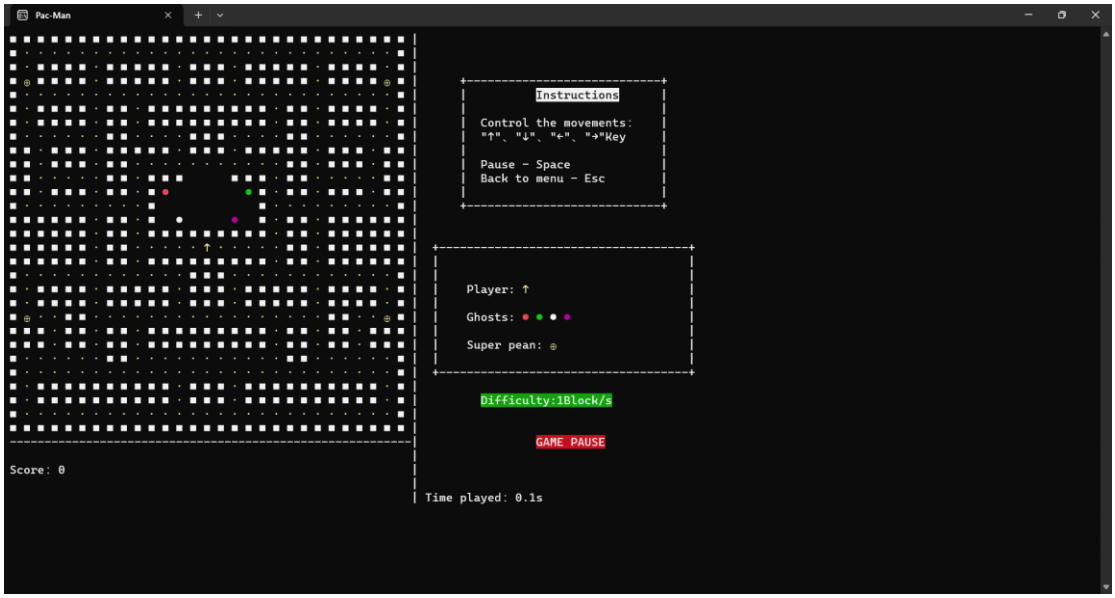
When “enter” is clicked, the menu will bring user to choose the difficulty for the game.



As we can see, there are 3 default difficulties for the user, which are shown as “Easy”, “Normal” and “Hard”. Each difficulty represents the speed of the ghost can move, shown as block per second (Block/s). 2-D arrays allow the ghost to move from one array to another. For the “Custom” difficulty, user can choose the speed of the ghosts from 1 to 5 block per second.

After we choose the difficulty, we will move to the main part of the program. We now first choose “Easy” for example.

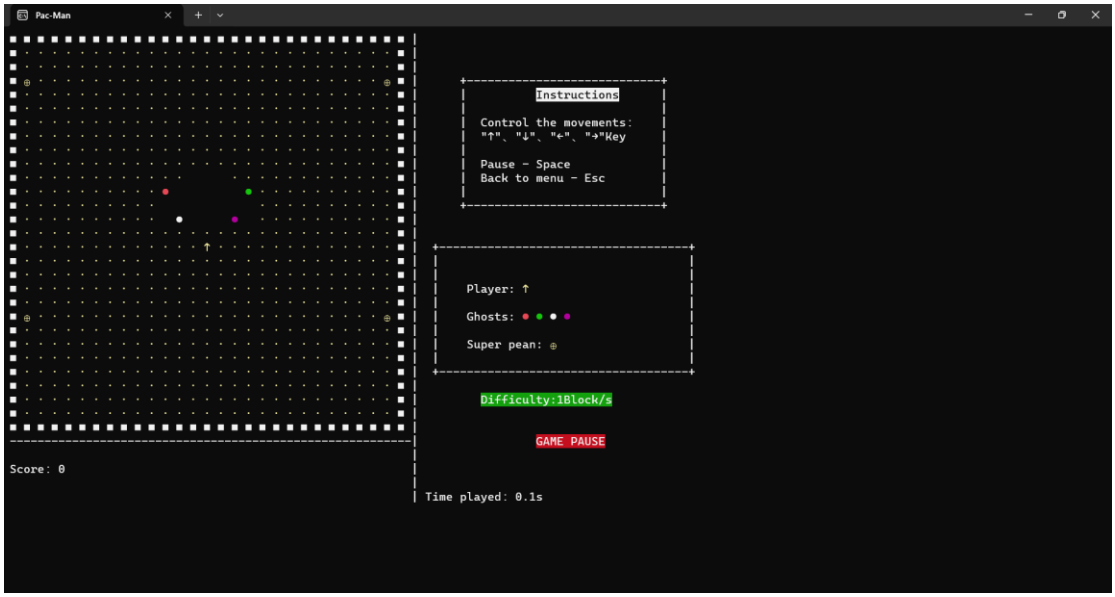
Game Example



A screenshot of the main game (For further explanation, all game screenshots will be taken from pausing)

We can see that the layout of the map, instructions and the score, time, etc. are perfectly arranged by using “Gotoxy”. To show how the player is controlled, the instruction writes the key to control the player to go upward or downward etc. Also, the score represents the peaks eaten by the player. Time represents the actual time played by the player corrected to nearest 0.1 second.

For simple explanation, we now use a simple map without walls instead of this complicated map.

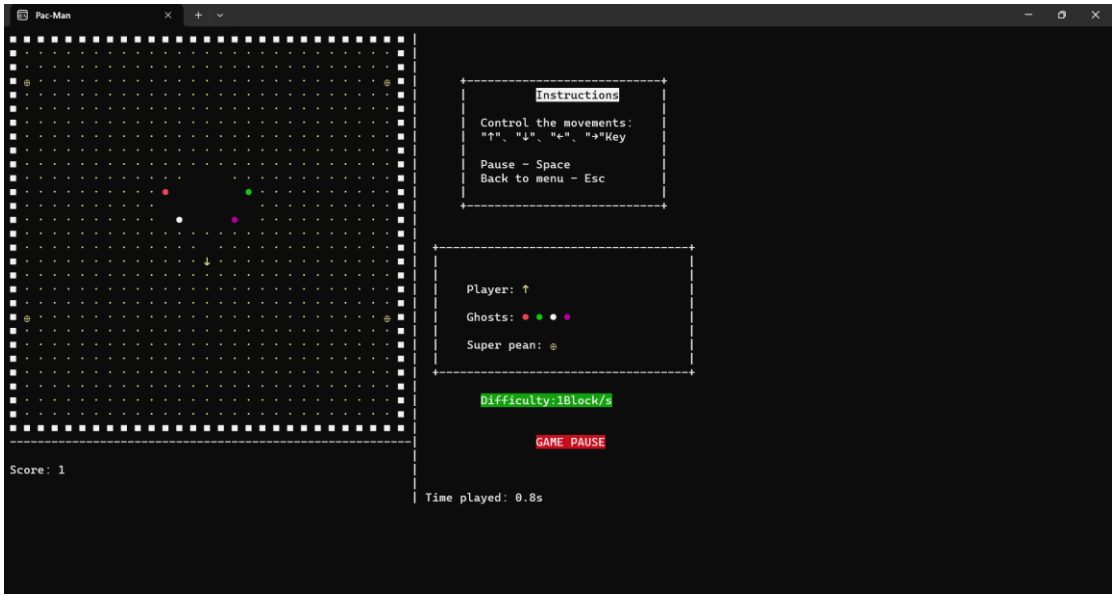


We can use arrow keys to control the player. By using ‘_getch()’ to modify the value of the player's array. Simple moves such as moving north, south, etc. can be done by adding or subtracting the value of x and y of the player's array.

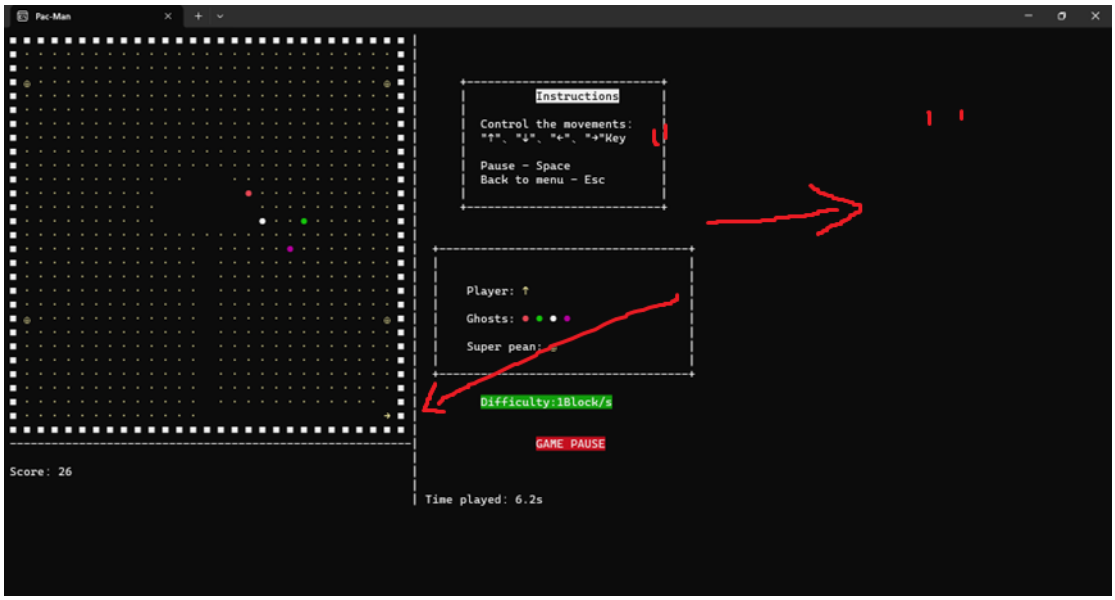
```
void Pacman::move(int dir, Map &map)
{
    if (dir == -1) return;
    int tempX=x, tempY=y; // use for test next position is legal or not
    switch (dir)
    {
    case UP:
        --y;
        setKey("↑");
        break;
    case DOWN:
        ++y;
        setKey("↓");
        break;
    case LEFT:
        --x;
        setKey("←");
        break;
    case RIGHT:
        ++x;
        setKey("→");
        break;
```

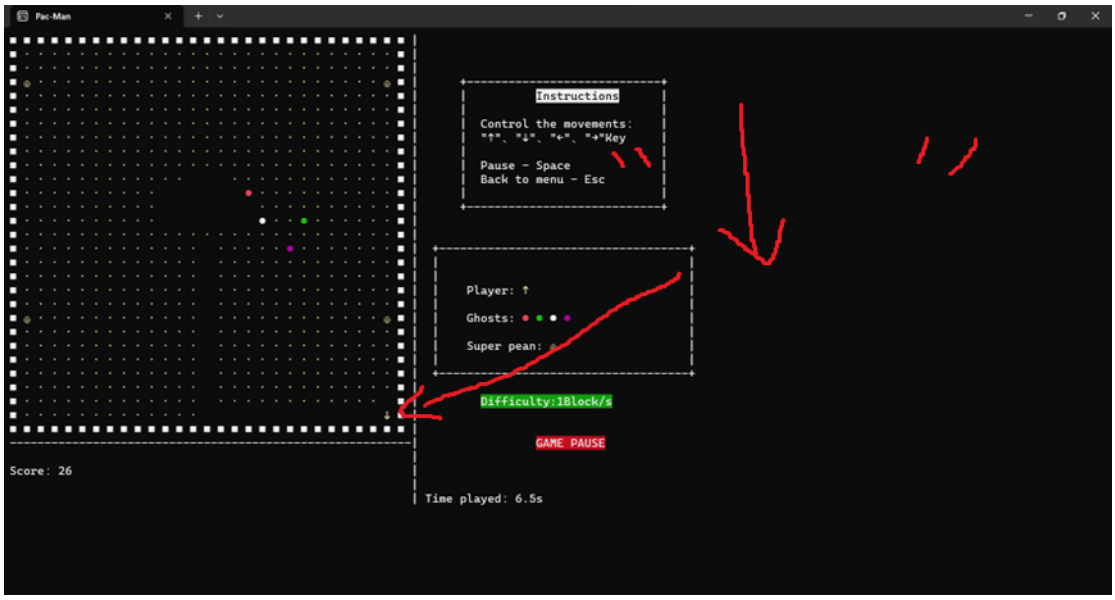
For northeast or etc. movement, it's not suitable in Pac-Man thus the 8 directions should be used in checking the ghost’s movement is legal or not since the player’s movement is limited as up, down, left, and right.

After we press the arrow button, the player will turn in such direction. And after checking, the player will move. For example, when ‘ ↓ ’ is pressed, the player will first turn the direction to downward then start the movement.



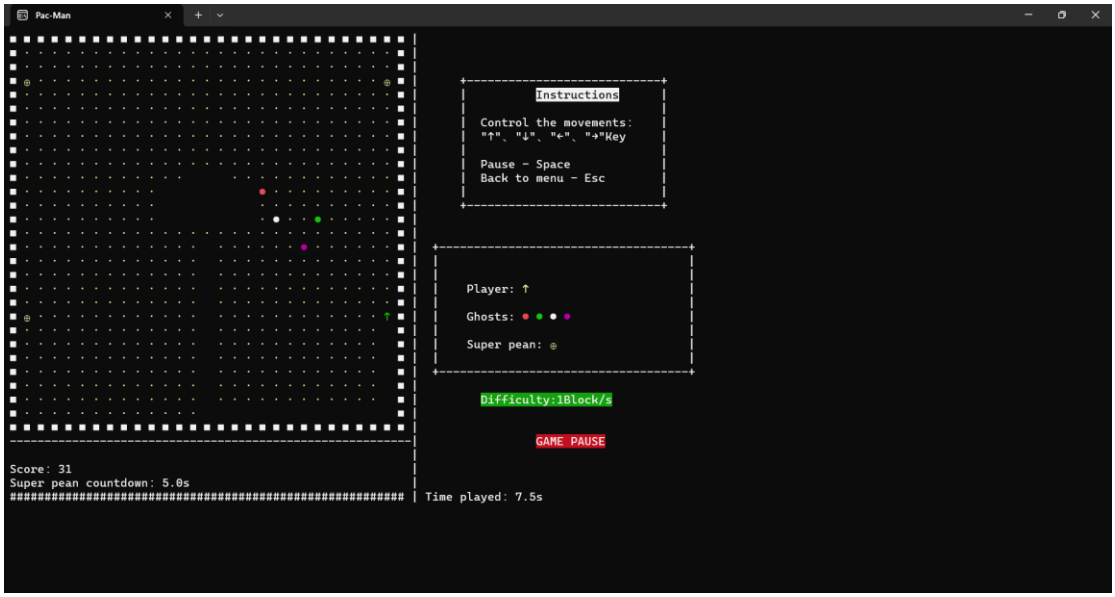
To prove the execution order, we can move to corner to test. Take the right-down corner as an example. When we try to move rightward or downward, the player won’t cross the border of the map, but the direction of the player will change.



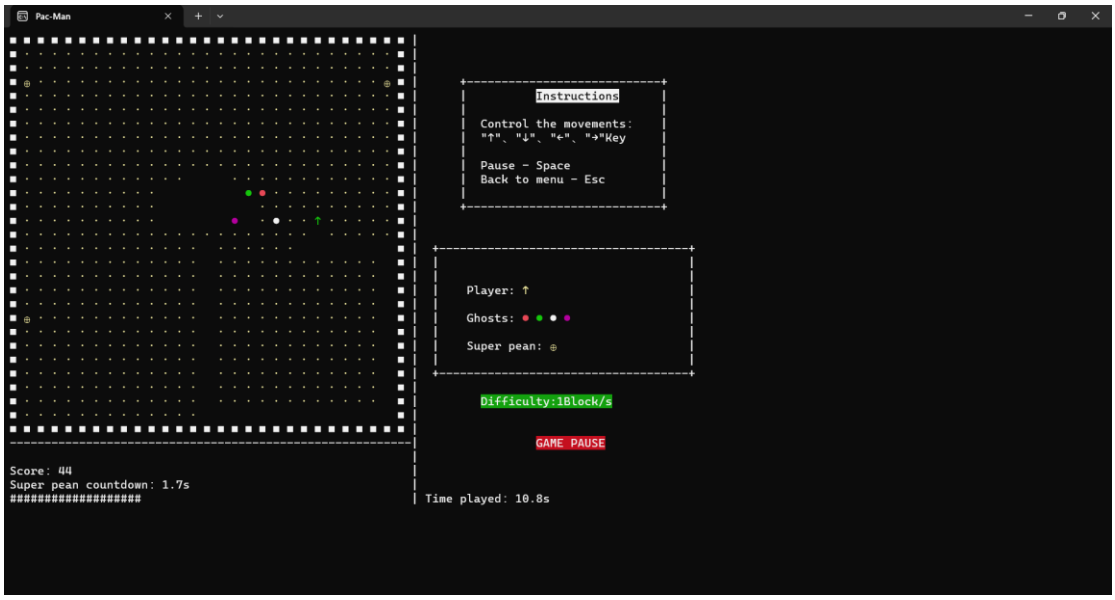


This can prevent some unexpected illegal movements. And we can see that the score will be synchronized with the peans in the map.

After we eat the super pean, the ghosts will freeze and stay at that position for a moment. During this period, when the player meets the ghosts, they will be sent back to their home located at the center of the map, instead of causing the game to be over.



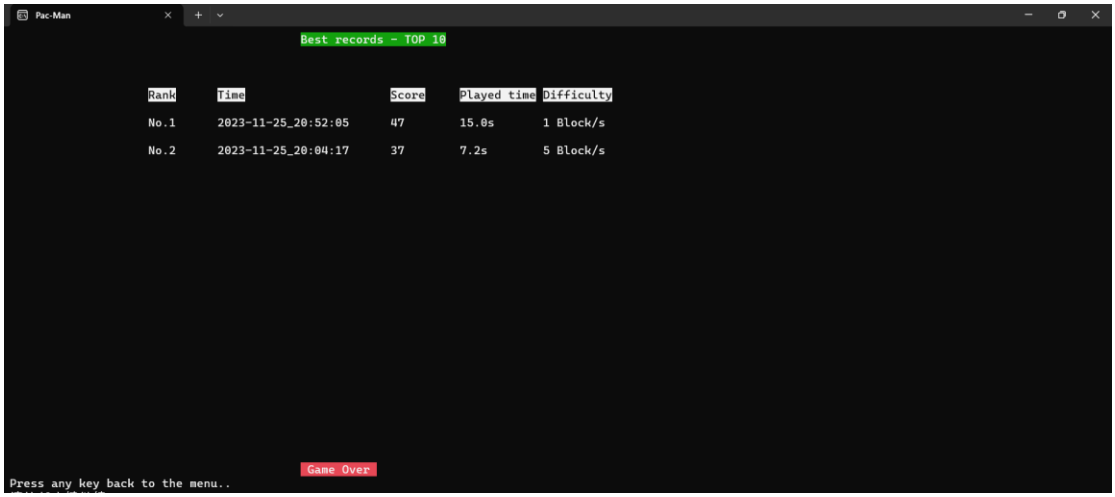
As we can see, there will be a countdown for the time of the super pean after we eat them. We will also change our color to show that we are now in super pean period.



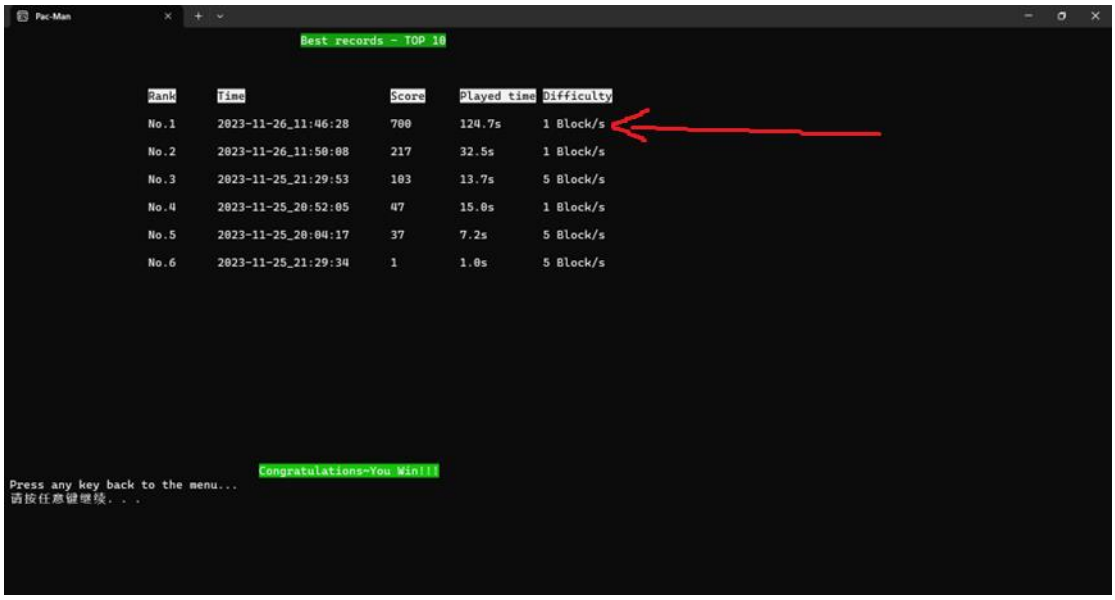
(There should be walls surrounded for their home, to simplify the explanation I deleted it)

Note that we will also get score when we eat the ghosts during the period.

If we meet the ghosts out of this period, the game will end and show the records.



To win the game, the player must eat all the peans on the map. After eating all the peans game will end and show the records.

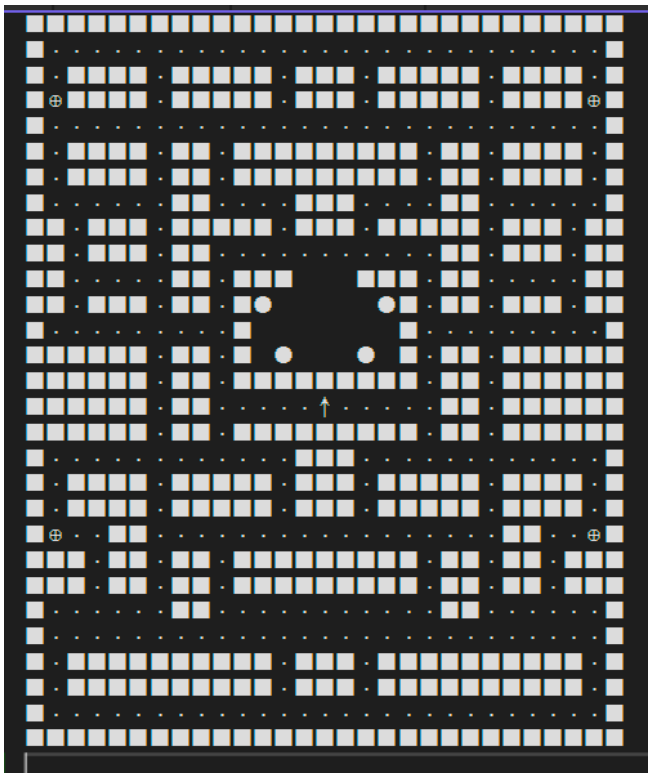


Still figuring out on how to show which record this the most updated…

That’s a simple example. We now focus on the map first.

Map

The map is designed as a 2-D array. Therefore, we only need to read the map from the text file, which can be customized by the user.



That’s the example of the map. We have defined the map as 29*29 size 2-D array, where:
--Square (■), as the wall of the map
--Dots (·), as peans. Also, as the score.
--BIIIIIIIIIIIG Dots (●), as ghosts.
--Dots with a cross (⊕), as super peans.
--Arrow (↑), as player.

So, the modification of map will become easier since the map is stored in text file. We can make our own map by changing those items. Note that we don’t want this game to become too hard, thus the number of ghosts is limited to at most 5.

Here, I would like to talk about including libraries in my program since most functions have been predefined in those libraries. Including libraries in object-oriented programming provides several advantages that enhance code development and efficiency.

First, Libraries offer pre-built components and functionalities that can significantly speed up development time. Instead of reinventing the wheel and implementing complex features from scratch, we can use the capabilities provided by libraries. This allows us to focus on the core logic of the program rather than spending time on lower-level details.

Also, utilizing libraries can boost productivity by reducing the amount of code we need to write and maintain. Libraries encapsulate complex functionality, so we can use them as black boxes without worrying about the internal implementation. This frees up our time and allows us to focus on the unique aspects of the program.

Moreover, Libraries often provide high-level abstractions that hide complex implementation details. This allows us to focus on using the library's functionality without worrying about the underlying implementation. Abstractions make code more readable, understandable, and easier to work with.

Back to the main idea. We can include <fstream> library if we want to perform file input or output operations. Therefore, we can initialize out map like this:

```
void Map::init(const char * filepath, Pacman & pacman, vector<Ghost> &ghosts)
{
    //init
    scores = 0;
    freezeTime = 0; //Superpean period
    target_scores = 0;  //winning score
    ghosts.clear();
    //Read in the map from the file.
    ifstream fin(filepath);
    if (!fin) {
        system("cls");
        cout << "Cannot found map file.\n\n" << endl;
        system("pause");
        exit(-1);
    }
```

As we can see, the game depends on the map file. Without the map file, the game will break and exit the program.

After the function finds the map file, it will start to read in the string stored in the text file. Of course, the function should also check if the map file is suitable for the program to run or not. Since we have defined the maximum size of the map is 29*29 2D-array, we need to check that the string stored in the map file hasn't exceeded 29.

```
char line[LINE_MAX]; //LINE_MAX = 200
int j = 0;
while (fin.getline(line, LINE_MAX)) {
    if (strlen(line) != MAP_SIZE * 2) {
        system("cls");
        cout << "Size of the map is not match. Please keep the size as 29*29. \n\n";
        system("pause");
        exit(-1);
    }
```

*We are now using some special symbols to represent the map which takes 2 spaces each. *2 is required.*

After ensuring that the size of the map is done, it's time to read in the file.

```
for (int i = 0; i < MAP_SIZE; i++) {
    char tempWord[3];
    tempWord[0] = line[i * 2];
    tempWord[1] = line[i * 2 + 1];
    tempWord[2] = '\0';
    string keyStr(tempWord);
    if (keyStr == "  ") {//Blank
        points[i][j].setType(0);
    }
    else if (keyStr == "■") {//Wall
        points[i][j].setType(1);
    }
    else if (keyStr == "·") {//Peans
        points[i][j].setType(2);
        target_scores++;
    }
    else if (keyStr == "⊕") {//Super peans
        points[i][j].setType(3);
    }
    else if (keyStr == "●") {//Ghosts
        points[i][j].setType(0);
        ghosts.emplace_back(Ghost(i, j));
        if (ghosts.size() > 5) {
            system("cls");
            cout << "Too many ghosts. That will be too hard. Please keep the number of ghosts below or equal to 5.\n\n";
            system("pause");
            exit(-1);
        }
        ghosts[ghosts.size() - 1].color = Ghost_Colors[ghosts.size() - 1];
    }
    else if (keyStr == "↑") {//player
        points[i][j].setType(0);
        pacman.init(i,j);
    }
```

We also limited the number of ghosts that exist on the map at the same time by the maximum number of 5.

That's the most direct way to read in the file into array. And since we're using the function provided by <istream> to read in the data, it will automatically append null character "\0" to the written sentence. Therefore, we need to check that it's finished the reading or not.

Also, we need to check if the row is exceeded from 29 or not. After checking it, the basic generation was completed.

Player

Since the main game is run in Boolean iteration, the movement of the player should also be a iteration. Here I use _kbhit() and _getch() functions provided by <conio.h> header file to detect the consistent input by the user. The movement of the player should be smooth to avoid being hit by the ghosts.

It's not possible to use scanf() and cin in the program since they are blocking input stream, meaning it will pause the program until the user enters input. This can be problematic for real-time games that require smooth and responsive input handling. Additionally, they may not be suitable for games with complex input requirements or those that need to handle multiple keys simultaneously.

For GetKeyState(), it’s difficult to control the time period detecting user input. If the period is short, many instructions will be executed; If the period is long, the function cannot even detect user’s input.

Thus, <conio.h> functions are used to detect user's input.

```
//Detect entered key
char ch;
if (_kbhit())
{
    ch = _getch();
    switch (ch)
    {
    case -32:
        ch = _getch();
        switch (ch)
        {
        case 72:
            pac_man.move(UP, map);
            dir_pacman = UP;
            break;
        case 80:
            pac_man.move(DOWN, map);
            dir_pacman = DOWN;
            break;
        case 75:
            pac_man.move(LEFT, map);
            dir_pacman = LEFT;
            break;
        case 77:
            pac_man.move(RIGHT, map);
            dir_pacman = RIGHT;
            break;
        default:
            break;
        }
        break;
    case 27://ESC
        return true;
    case ' ':
        pause();
        break;
    default:;
    }
```

For the move function, it’s used to display the direction change of the player. After changing, it needs to check if the movement of the player is legal or not. If not, return to the original position.

Also, that the peans or super peans should be checked. We have defined several types of things in map part. For example, if the player moves to the peans’ coordinate, score should be added, and the place should change into space.

Super peans will freeze the ghosts (Also same as adjust the freezing time variable to maximum and start to countdown) instead of adding score to the player. The place should also change into space.

After the player moved, the original position should change back to space.

For pausing, it has a higher priority to be executed. We need to detect that until the space is re-pressed the main game should not run.

```
void Game::pause()
{
    int bias_temp=23;
    Goto_XY(MAP_SIZE * 2 + 18, INFO_MARGIN_UP + bias_temp);
    if (!play_flag) {
        SetColor(PAUSE_COLOR); cout << "GAME PAUSE"; SetColor(WHITE_COLOR);
    }
    while (true) {
        char ch;
        if (_kbhit())
        {
            ch = _getch();
            switch (ch)
            {
            case 27://ESC
            case ' ':

                Goto_XY(MAP_SIZE * 2 + 18, INFO_MARGIN_UP + bias_temp);
                cout << "          ";
                return;
                break;
            default:;
            }
        }
        Sleep(10);
    }

}
```

And that's the most direct way ----- Nested loop. Trust me, it won't take up lots of resources.

Then we need to check win or lose.
We've done the counting on the peans on the map. Thus, it's easy for us to check if the player has reached the target or not.

```
if (map.scores == map.target_scores) {
    return game_win();
}
```

, where gamewin() includes record adding and history showing.

For game-over, we need to check if the ghost(s) has hit the player or not.

```
for (auto&ghost_i : ghosts) {
    if (ghost_i.hit(pac_man, map))
        return game_over();
}
```

, where gameover() includes record adding and history showing.

We've defined the ghost's group as vector since player can adjust the number of ghosts. Then auto is used.

Game logic

The layout of the main game has been shown in the upper part. We need to know how they work. For example, we need to know how to output the instructions while the player's movement iteration and ghosts' finding path iteration are executing, and how to restart the game logic when the user ends the pausing.

Since those user interfaces are not that important, we can just simply translate them into procedure to use. For the layout that needs to be printed in real-time like the map or the timer (We're now having time played counting and we want smooth movements), we can conclude them into another procedure for the main game looping.

```
void Game::refresh()
{
    system("cls");
    for (int i = 0; i < MAP_SIZE; i++) {
        for (int j = 0; j < MAP_SIZE; j++) {
            map.points[i][j].print();
        }
    }
    pac_man.print();
    for (auto &ghost_i : ghosts) {
        ghost_i.print();
    }
    infoUI();   //Timer, score, super pean countdown
    helpUI();   //Instructions
}
```

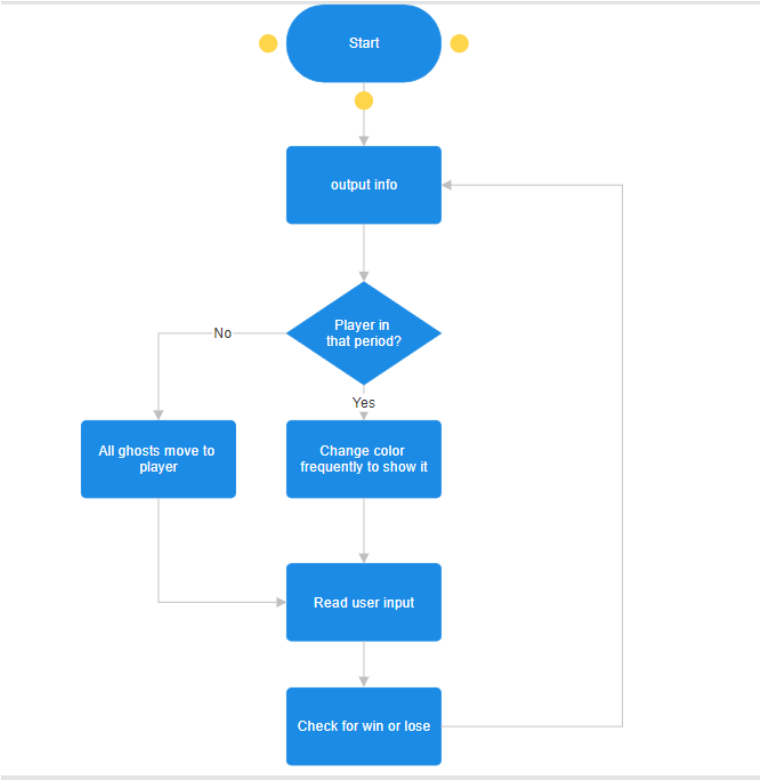
They are indisputably important since the main game iteration only stops for 50 milliseconds (ms). We don't want to have any layout problem.

```
bool Game::loop()
{
    refresh();
    // record the direction and movement, use for further dev.
    int dir_pacman = -1;
    int ghost_num = ghosts.size();
    int *dir_ghosts = new int[ghost_num];

    int speed_adapter = 0;
    while (true) {
        dir_pacman = -1;
        for (int i = 0; i < ghost_num; i++)
            dir_ghosts[i] = -1;
```

After the printing process is done, we can start our “chasing show” for the ghosts and player.

The game is based on looping. First, we should check if the player is in super pean period or not to connect to the next iteration. Then, we should check the instructions given by the user pressing the arrow keys to move the player. After it, we should check if the game has won or lost. Then we can print the information out and go on to the next iteration.

The flow of the iteration of the main game should be like this:



Note that check for 'ESC' button is included in the read use input state. If the user press esc, then the game will be shut down and back to the menu and re-choosing the difficulty.

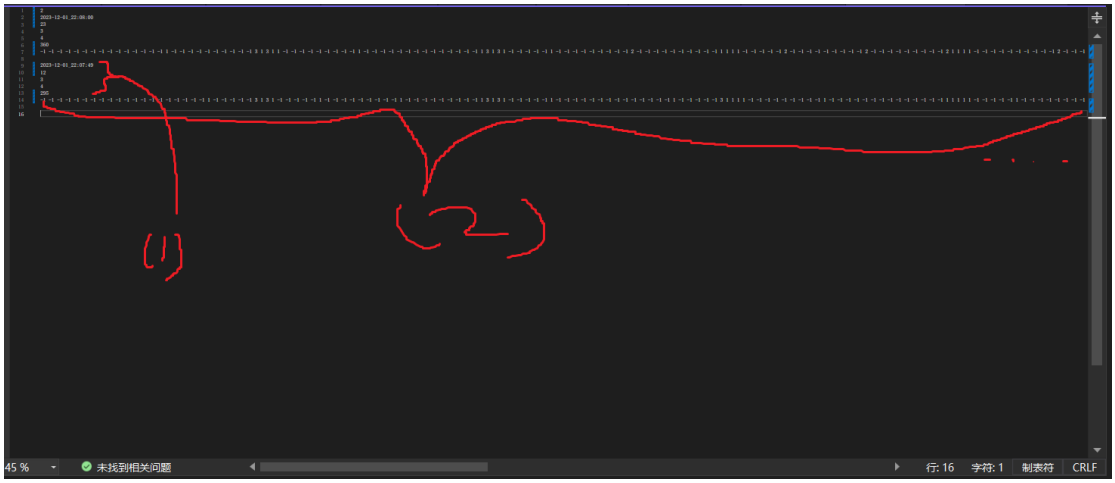
Record

Instead of defining a variable that accumulates time, I record every movement of the player and the ghosts for further development such as replay, etc. And we can use the movements recorded to calculate the time played.

```
//to calculate the time played
steps.push_back(dir_pacman);
for (int i = 0; i < ghost_num; i++)
    steps.push_back(dir_ghosts[i]);
```

In each iteration, the direction of the player and the ghosts will be recorded such as heading north, south, etc. (And that's why we need to define the direction as number, one is easier for us to manage the movement part, another one is to help us on further development since numbers are always easier to manage than characters)

One set of movements contains 5 numbers (normally with 4 ghosts), and since the iteration period is 50ms short, lots of sets of numbers will be generated.



(1): Containing the basic information of this record, including date, score, speed of the ghosts, number of ghosts.
(2): The movements record.

With these sets of data, we can easily find out what's happening on the movements of the ghosts or player if there is any problem.

Back to the main idea. If we want to modify the records, we always need to read in the file first.

```
void Record::read()
{
    ifstream fin(filepath);
    if (!fin) {
        return ;
    }
    items.clear();
    int num = 0;
    fin >> num;
    for (int i = 0; i < num; i++) {
        string name_str;
        int the_score;
        fin >> name_str >> the_score;
        items.push_back(RecordItem(name_str, the_score));
        int idx = items.size() - 1;
        fin>>items[idx].speed;
        fin>>items[idx].ghost_num;
        fin>>items[idx].steps_num;
        int max_i = items[idx].steps_num;
        for (int i = 0; i < max_i; i++) {
            int temp;
            fin >> temp;
            items[idx].steps.push_back(temp);
        }
    }
    fin.close();
}
```

Always do checking. (All variables have been defined in design part (p.12))

Simple reading data from the file and write in the array, where push_back() function is provided by <vector> and pushing the data into vector acting like a stack.

Now we have the in for the file, we should also have the out for the file.

```
void Record::save()
{
    ofstream fout(filepath);
    if (!fout) {
        system("cls");
        cout << "Cannot write the record file in this path. Please try another path to store the record.";
        exit(-1);
    }
    fout << items.size() << endl;
    for (auto& i_item : items) {
        fout << i_item.name << endl;
        fout << i_item.score << endl;
        fout << i_item.speed << endl;
        fout << i_item.ghost_num << endl;
        fout << i_item.steps_num << endl;
        for (int i = 0; i < i_item.steps_num; i++)
            fout << i_item.steps[i] << " ";
        fout << endl;
        fout << endl;
    }
    fout.close();
}
```

Always, always do checking.

Note that neither fin nor fout are Boolean itself, but they can be used in a conditional statement to check if the file stream is in a valid state for writing or reading data to the file.

With these two procedures' help, we can modify the records now.

```
void Record::add(int the_score, const vector<int> &steps, int ghost_num, int speed)
{
    read();
    time_t t = time(NULL); //Get the current time
    char time_string[64] = { 0 };
    strftime(time_string, sizeof(time_string) - 1, "%Y-%m-%d_%H:%M:%S", localtime(&t));
    items.push_back(RecordItem(time_string, the_score));
    int idx = items.size() - 1;
    items[idx].ghost_num = ghost_num;
    items[idx].speed = speed;
    items[idx].steps_num = steps.size();
    for (auto istep : steps) {
        items[idx].steps.push_back(istep);
    }
    sort(items.begin(), items.end());
    if (items.size() > 10) {
        items.pop_back();
    }
    save();
}
```

Should we do check?

Strftime function can have different format of time to represent. Here I choose the following format to represent the date:
Year//month//day//hour(24h) //minute//second

<vector> library dynamically adjusts its size to accommodate the number of elements it contains. It can grow or shrink as elements are added or removed, making it flexible for handling varying amounts of data. Therefore, we can simply call the variables instead of pre-defining the size of an array or character.

After sorting the elements stored in the vector, we need to pop the last element since we're now showing the best 10 records.

Then we need to show the records after the gameplay.

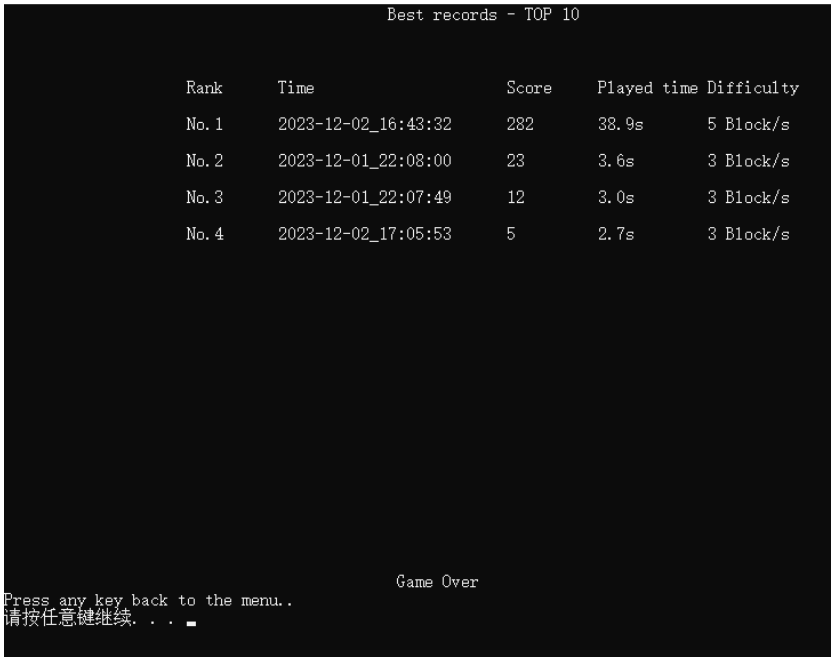
```
void Record::show()
{
    int left_margin_No = WINDOWS_SIZE_X / 2 - 30; //fixing layout
    int left_margin_name = left_margin_No + 10;
    int left_margin_score = left_margin_name + 25;
    int left_margin_time = left_margin_score + 10;
    int left_margin_speed = left_margin_time + 12;
    int up_margin = 4;

    read();
    SetColor(WHITE_COLOR);
    system("cls");
    Goto_XY(WINDOWS_SIZE_X / 2 - 8, 0);
    cout << "Best records - TOP 10";
    Goto_XY(left_margin_No, up_margin);
    cout << "Rank";
    Goto_XY(left_margin_name, up_margin);
    cout << "Time";
    Goto_XY(left_margin_score, up_margin);
    cout << "Score";
    Goto_XY(left_margin_time, up_margin);
    cout << "Played time";
    Goto_XY(left_margin_speed, up_margin);
    cout << "Difficulty";

    if (items.size() == 0) {
        Goto_XY(left_margin_No, up_margin + 2);
        cout << " (Records are empty...) ";
    }
    for (int i = 0; i < items.size(); i++) {
        Goto_XY(left_margin_No, up_margin + 2 + i * 2);
        cout << "No." << i + 1;
        Goto_XY(left_margin_name, up_margin + 2 + i * 2);
        cout << items[i].name;
        Goto_XY(left_margin_score, up_margin + 2 + i * 2);
        cout << items[i].score;
        Goto_XY(left_margin_time, up_margin + 2 + i * 2);
        printf("%.1fs", items[i].steps_num / (items[i].ghost_num + 1)*50.0 / 1000);
        Goto_XY(left_margin_speed, up_margin + 2 + i * 2);
        cout << items[i].speed<<" Block/s";
    }
```

Where is my checking?

Since read() function has helped us write in the data to the vector from the record file, we can simply using the value stored in vector to display the record.

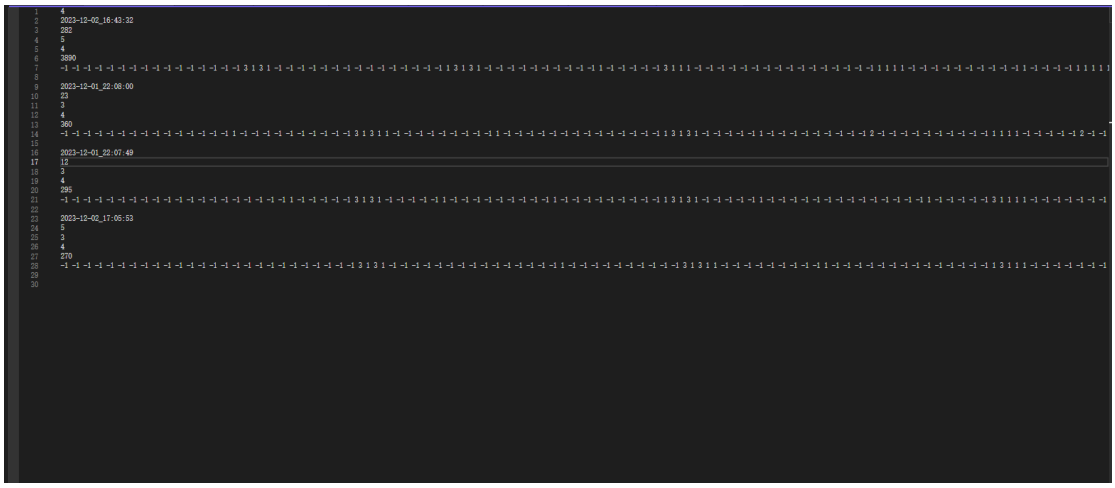
Note that time played can be calculated by counting how many steps walked by the player and ghosts.

I would like to give an example on how to calculate the time played.



Let’s take record 3 as an example since the time played is integer that easier for us to calculate.

From the record file, we can see the directions moved by the player and the ghosts.



First represent number of records stored. We export the third lengthy steps to count.

After a ‘Long time’ counting, 

. That’s the same as the upper line counted.

Divide it by 5 since there are 1 player and 4 ghosts in the example of the game looping, we can see that it completed 59 cycles of the game looping and each cycle consist 50ms. Multiply it by 50 and divide it by 1000 since it’s in milliseconds. We can get the final time played, which is 2.95s. After doing round up, it goes to 3.0s and displayed on the example.

That’s also true for other records.

Ghost

We need to figure out how ghosts find the player’s position and heading to it.

The pathfinding in Pac-Man is like a maze game which require you to find the shortest path to the exit.

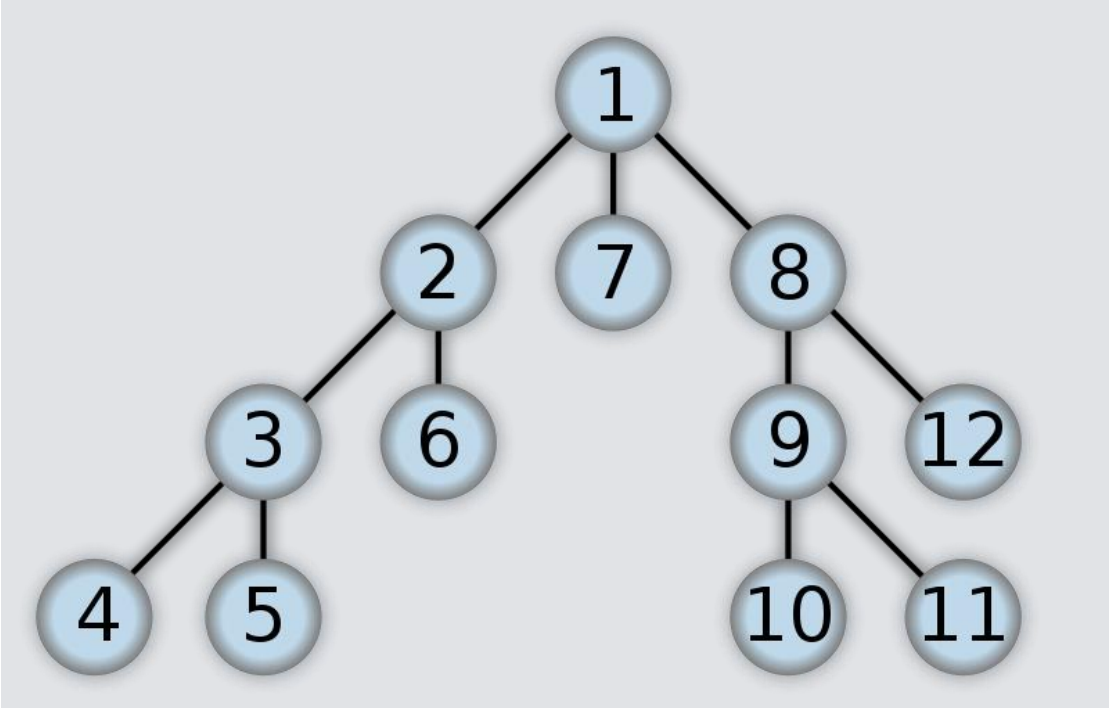
Finding the shortest path in a maze is a classic problem in computer science known as the "maze-solving problem." There are several algorithms we can use to solve this problem.

For example, the most popular algorithm nowadays is A* algorithm. It aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until the goal node is reached.

Also, there are other types of algorithms that is easier for beginner to learn like breadth-first search (Also known as BFS), depth-first search (Also known as DFS), and others. Those algorithms get its own characterises and its advantages or disadvantages.

For depth-first search, it explores the maze by going as deep as possible along each path before backtracking. It uses a stack data structure to keep track of cells to be explored. Initially, the source cell is pushed onto the stack. At each iteration, we pop a cell from the stack and mark it as visited. Then, we examine its neighbouring cells.

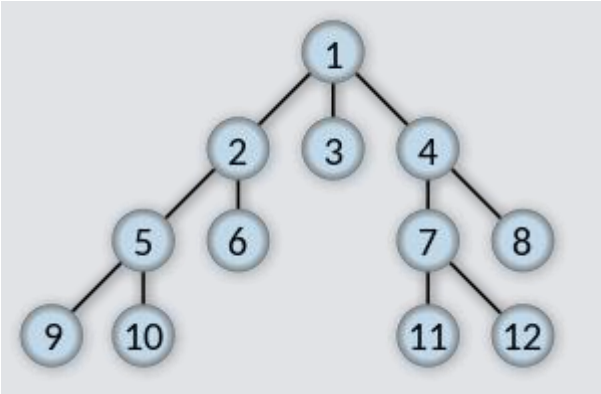
If a neighbouring cell is reachable and has not been visited, we push it onto the stack. This process continues until the destination cell is found or all reachable cells have been explored.



Note that Unlike breadth-first search, depth-first search does not consider the distance from the source, so it may not find the shortest path. Therefore, this algorithm is not preferred in my program since the ghosts should find the player in the shortest path.

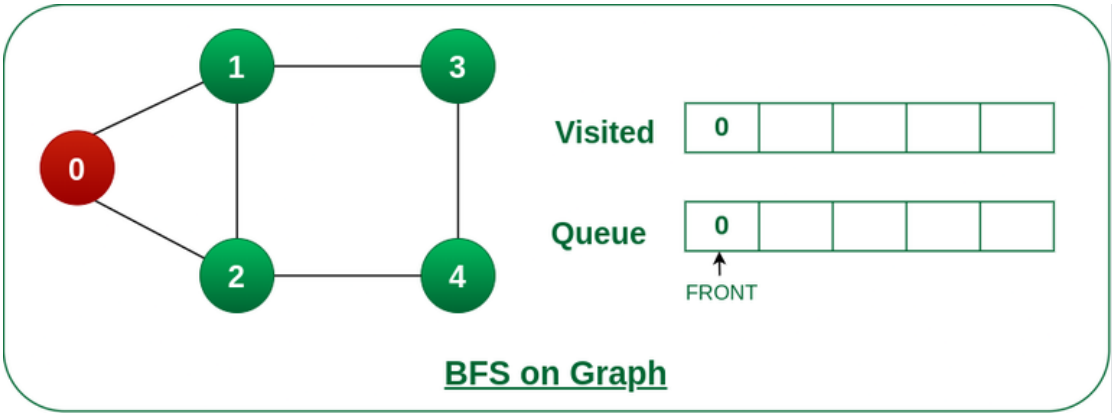
Overall, A* is the most efficient way to solve the maze problem. But for my programming level, I choose breadth-first search for my program since it's the easiest algorithm for the beginner.

Breadth-first search explores the vertices or nodes in a breadth ward motion, visiting all the nodes at the same level before moving on to the next level. The algorithm starts from a given source node and systematically explores its neighbours, then the neighbours of those neighbours, and so on. This exploration strategy ensures that breadth-first search visits nodes in the order of their distance from the source.

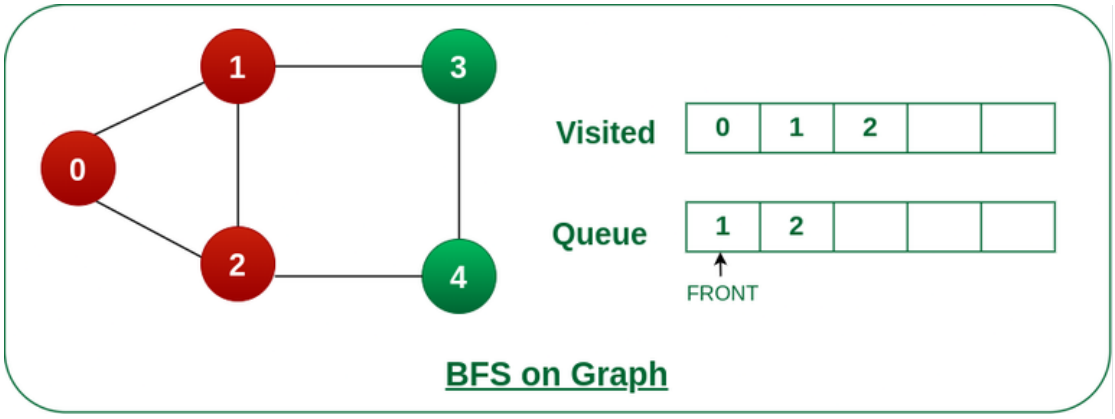


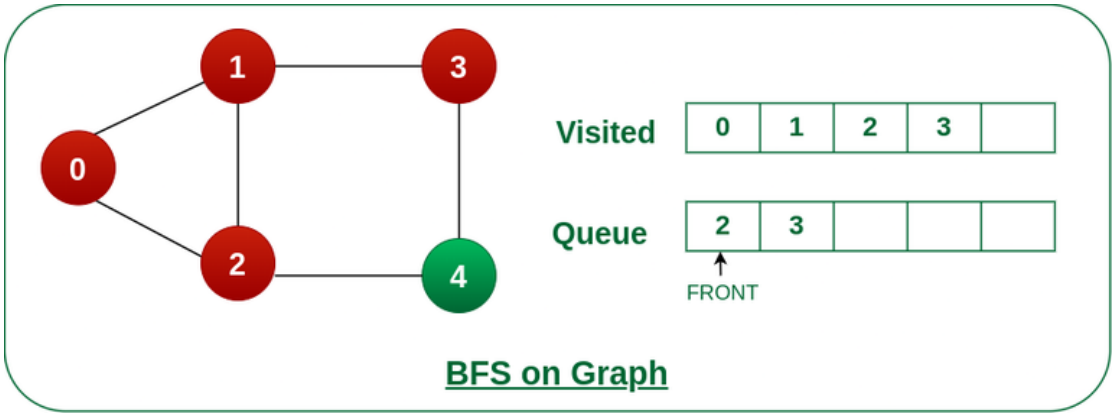
Visit node in breadth ward motion.

To keep track of the nodes to be explored, breadth-first search (BFS) utilizes a queue data structure. The source node is initially enqueued.

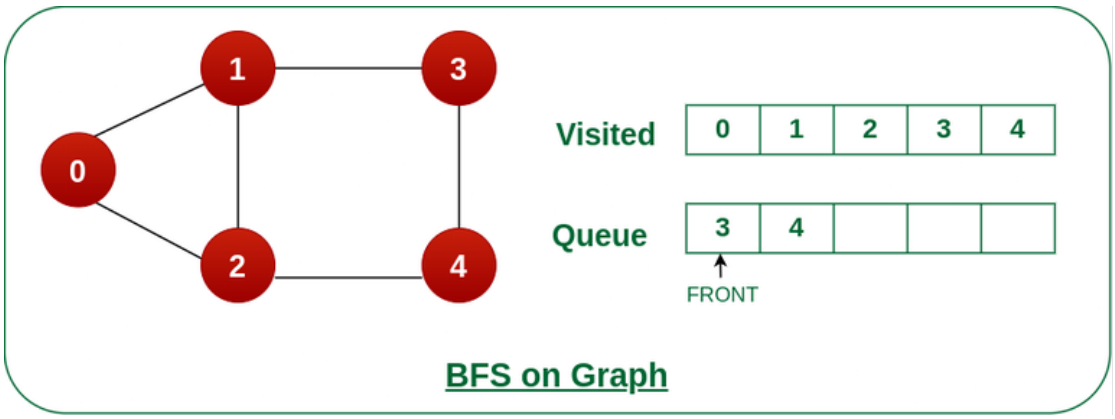


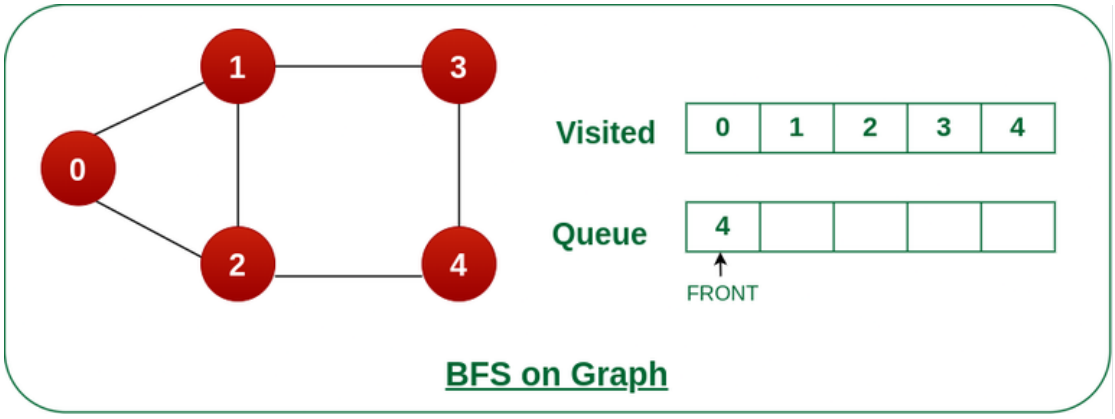
Then, we should dequeue a vertex from the front of the queue. Visit the dequeued vertex and process it as needed, also enqueue all the unvisited neighbors of the dequeued vertex.

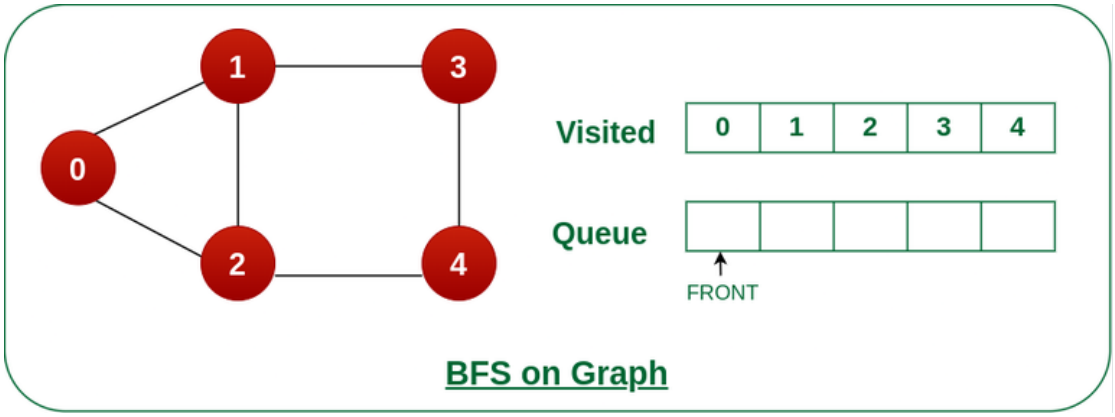




Note that we should mark each enqueued neighbor as visited. Repeat the above steps until the queue becomes empty.

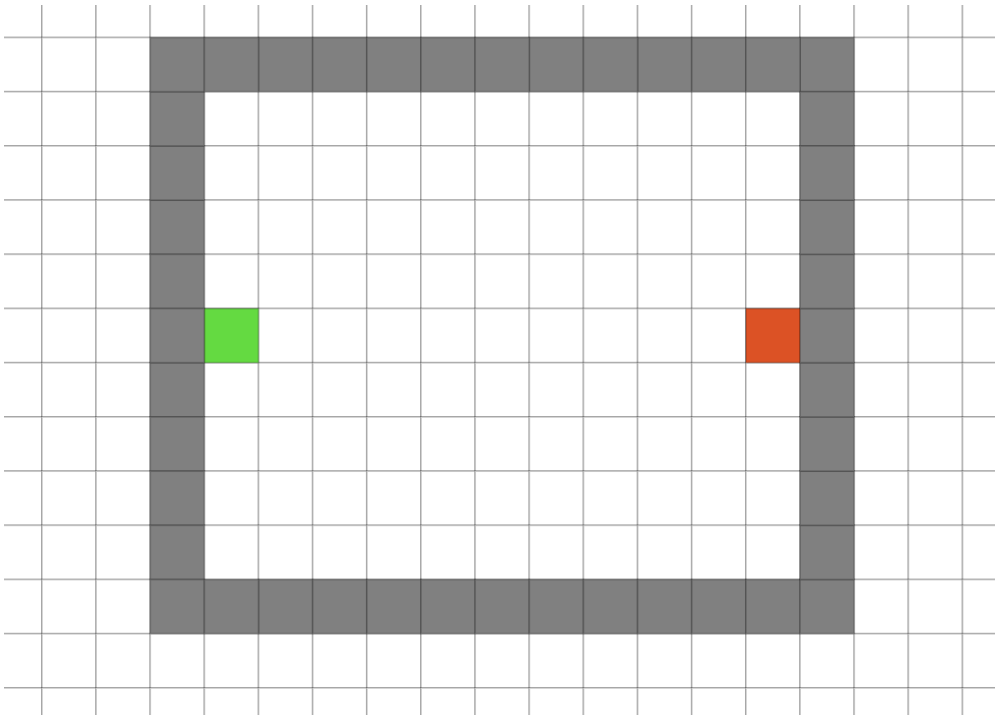






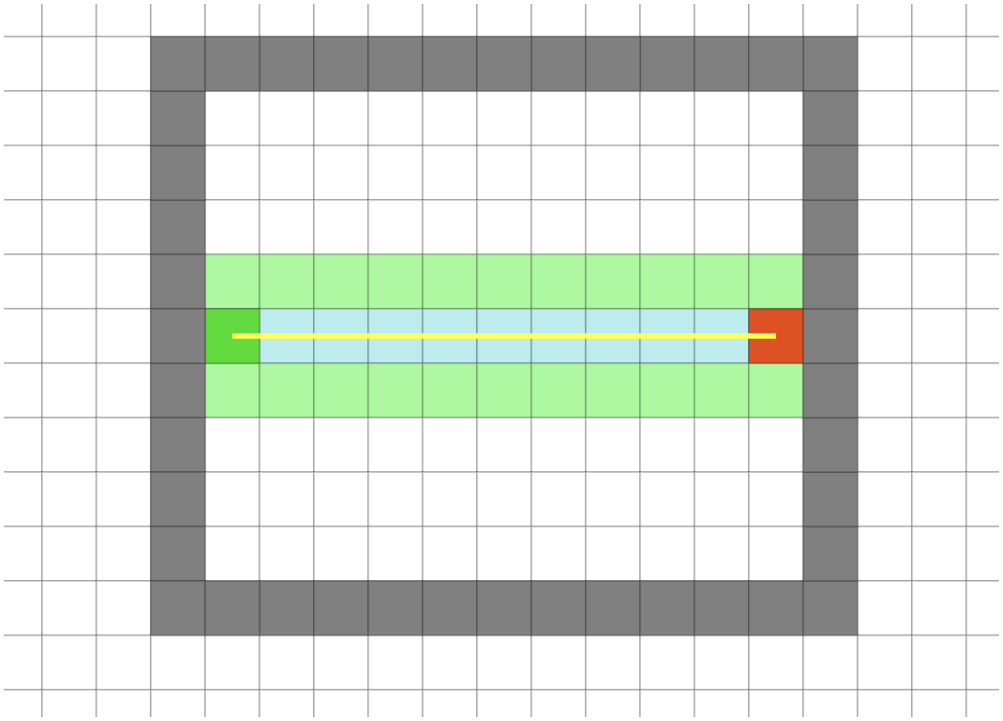
By following this process, BFS ensures that vertices are visited in the order of their distance from the source. It explores all the vertices at the current level before moving on to the vertices at the next level. This exploration strategy guarantees that the shortest path from the source to any reachable vertex is found.

We should have a more graphic example.

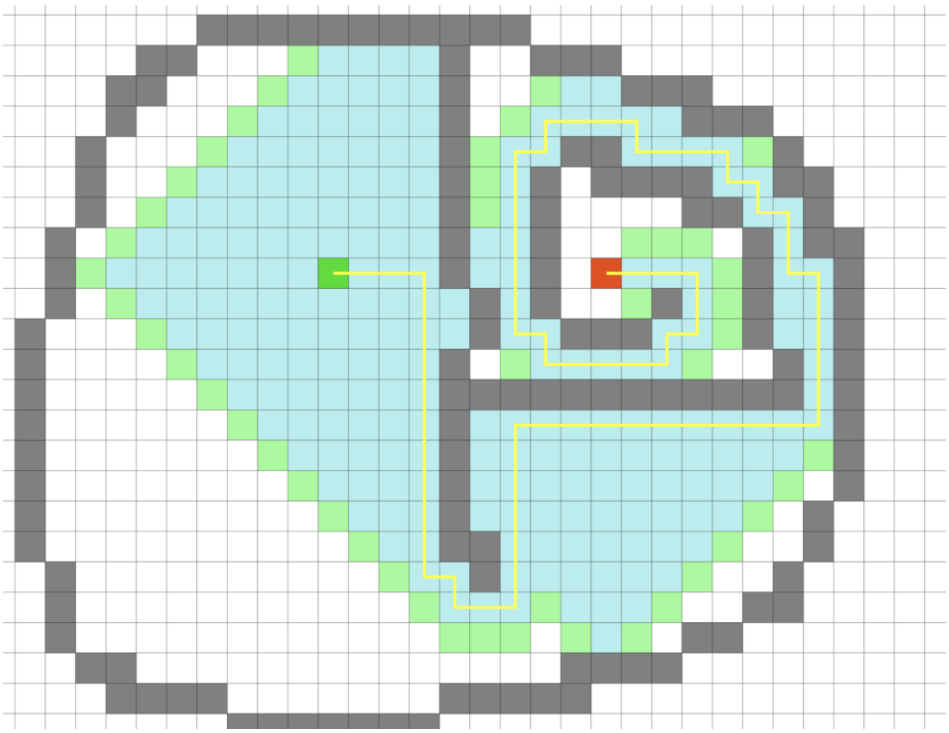


Here, we have a simple maze, where the green node is the start position, and the red node is the end position.

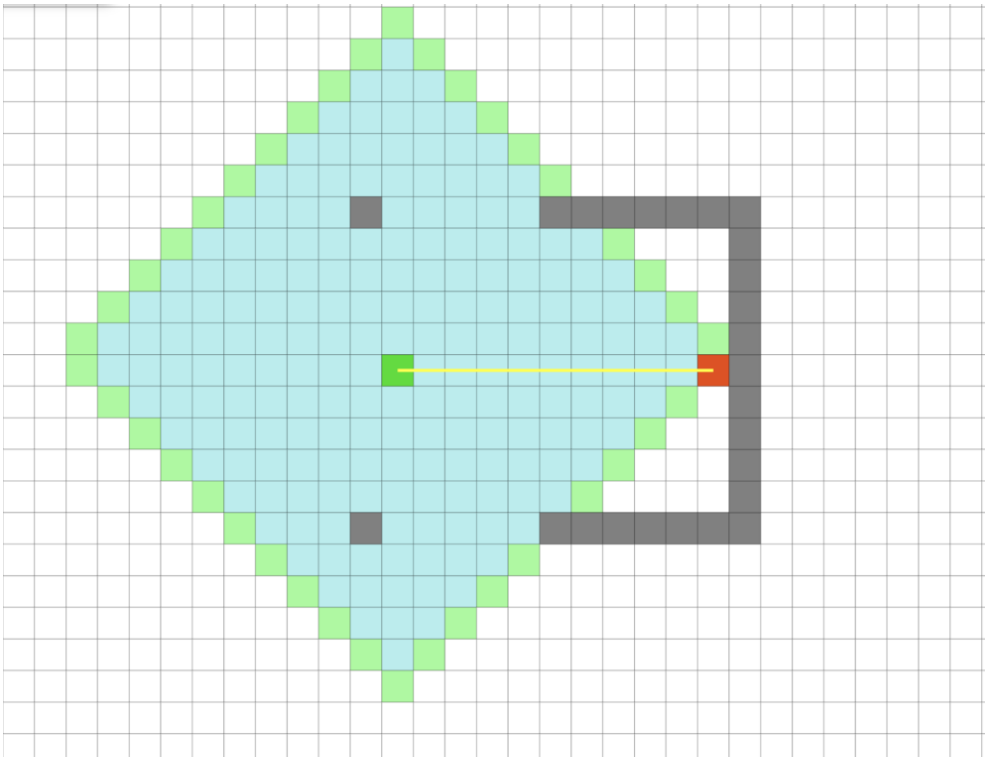
According to our theory, A* should have the most efficient way to find the path.



Also, according to our theory, depth-first search should explore the deepest node first, then back to another shallower node to keep searching for the end node.



Also, and also, according to our theory, BFS should explore all the neighbor nodes.



As we can see, BFS explores the graph layer by layer, starting from the source vertex and moving outward. It guarantees that the shortest path from the source to any reachable vertex is found. In this example, the shortest path from start position to any other vertex is found using BFS.

Now we should move to the programming part.

We have defined find path in design part (P.8). What we need to do is to put those theories into real code.

```
//To record the previous coordinate
int pre[MAP_SIZE*MAP_SIZE];
//Visited node
bool visited[MAP_SIZE][MAP_SIZE];
//Directions
int dx[4] = { 1, 0, -1, 0 };
int dy[4] = { 0, 1, 0, -1 };
void Map::findPath(Position & A, Position & B)
{
    for (int i = 0; i < MAP_SIZE; i++) {
        for (int j = 0; j < MAP_SIZE; j++) {
            visited[i][j] = false;
            pre[i*MAP_SIZE + j] = 0;
        }
    }
    PosXY st;
    st.x = A.x;
    st.y = A.y;
    queue<PosXY> myqueue;
    myqueue.push(st);
```

In here, I use 1-D array to store the previous coordinate. It's easier for us to calculate the previous coordinate instead of using a 2-D array.

1D array consumes less memory compared to a 2-D array. In a 2-D array, each element represents a specific row and column, which can lead to memory wastage if the grid is sparse or irregularly shaped. With a 1-D array, we can represent the grid using a linear sequence of elements, reducing memory requirements.

Also, managing a 1D array is often simpler compared to a 2-D array. We don't have to handle nested loops or deal with indexing in two dimensions. The div and mod operations allow us to convert a linear index into row and column coordinates, providing a straightforward mapping between the 1-D array and the 2-D grid.

For example, here we have a 5*5 grid.

1-D: (0) 2-D: (0,0)	(1) (0,1)	(2) (0,2)		
(5) (1,0)				
(10) (2,0)				
(15) (3,0)				
(20) (4,0)				

We can define the coordinates of the grid like this (although the default is also like this), then if we want to know a particular grid's coordinate like the right bottom, the coordinate in 2-D array is (4,4).

In 1-D array, we can simply calculate the coordinate by using the quotient and the remainder. Since the coordinate in 1-D array is (24), we can translate it into (4,4) by dividing the size of that grid (which is 5). The quotient is 4, and the remainder is 4.

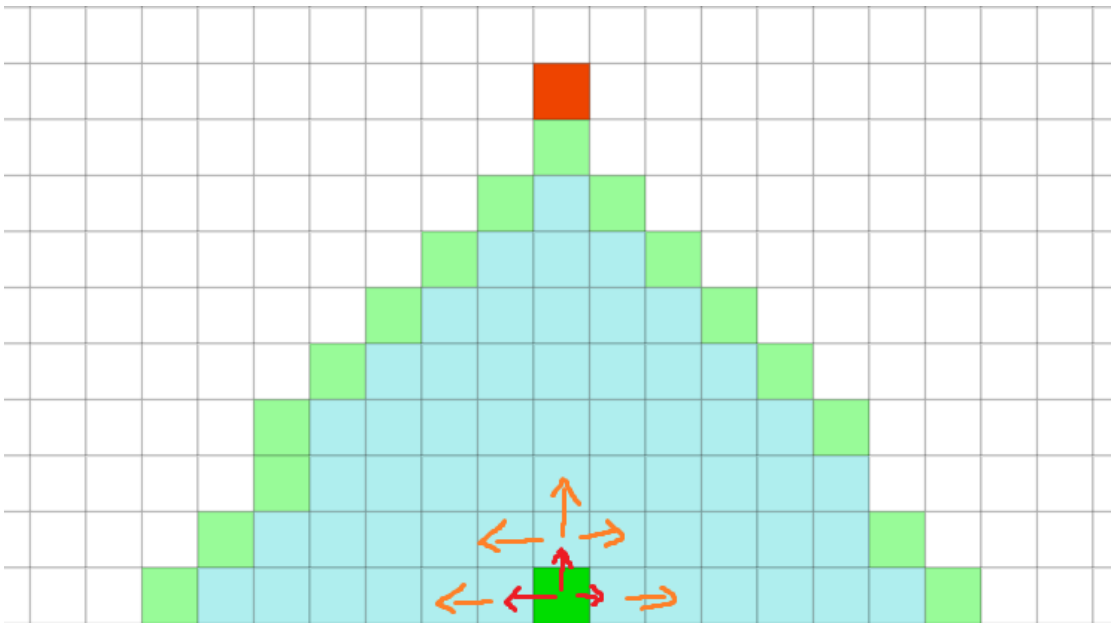
It can avoid the two 2-D arrays mix and cause some unexpected errors.

After we've done our preparation work, it's time to move to the iteration part.

```
while (!myqueue.empty()) {
    PosXY tempXY = myqueue.front();
    myqueue.pop();
    if (tempXY.x == B.x && tempXY.y == B.y) {
        break;
    }
    for (int i = 0; i < 4; i++) {
        int ix = tempXY.x + dx[i];
        int iy = tempXY.y + dy[i];
        if (ix >= 0 && ix < MAP_SIZE&&iy >= 0 && iy < MAP_SIZE && !visited[ix][iy] && points[ix][iy].getType() != WALL )
        {
            visited[ix][iy] = 1;
            PosXY next;
            next.x = ix;
            next.y = iy;
            pre[ix*MAP_SIZE + iy] = tempXY.x * MAP_SIZE + tempXY.y;
            myqueue.push(next);
        }
    }
```

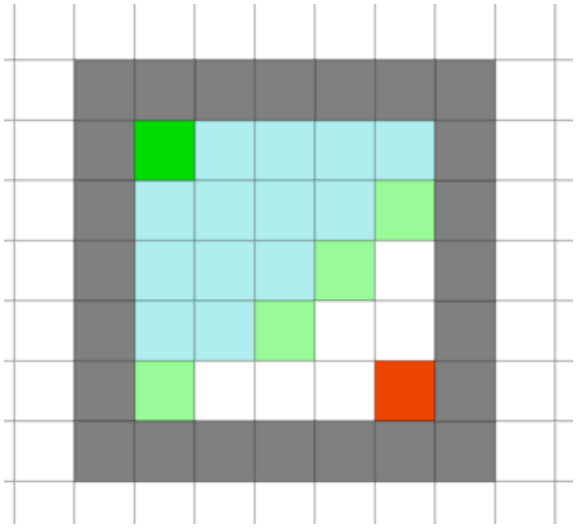
Just like the theory, we need to pop the first element in the queue each time. After the iteration finds the coordinate of the target, the iteration will immediately break.

For the nested loop, it's used to record the grids around the point which have been popped.

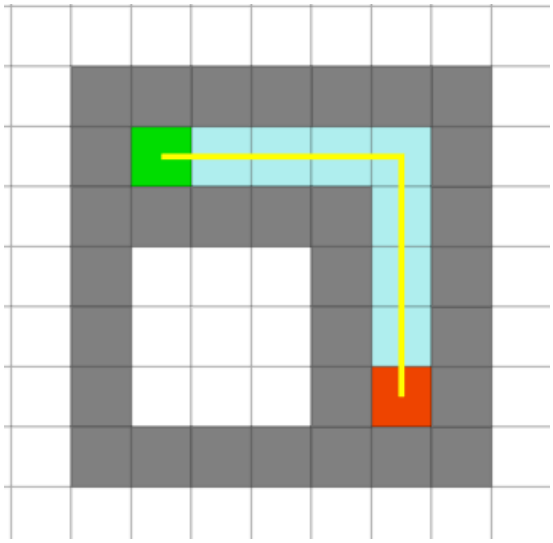


For example, the starting position will be recorded. The coordinate of it will also be recorded. Then, we can use dy and dx (which defined as {1,0, -1,0} and {0,1,0, -1}, and it's always true to calculate the upper or lower grid's coordinate. For example, the coordinate of the starting position is (y,x). [2-D array's x increases with columns, y increase with row.] When we want to find the coordinate of the upper grid of the starting position, we can know the coordinate by subtracting the y coordinate of the starting position, which is [y-1, x].) to record the grid around the starting position. We should also mark those coordinates as visited to avoid being pushed again into the queue. The iteration will keep spreading until the required grid is found.

After recording the grids, the previous coordinates should also be recorded to show where it comes from. Here, let's have another example on previous coordinates.



Let the coordinates of the starting position be (0,0). After executing the iteration, the path should be like this:



We use the coordinate that we have mentioned in the upper part. Then, the coordinate of the end position should be (4,4).

According to our postulate and the path shown, the coordinate of the path should be (in 2-D array): (0,0), (0,1), (0,2), (0,3), (0,4), (1,4), (2,4), (3,4), (4,4).

We have a 1-D array “previous” storing the previous coordinate of the path. Then, for the 1-D array, the value of it should be like this (int pre [25] ← 0~24):

	0*5+0=0	0*5+1=1	0*5+2=2	0*5+3=3					0*5+4=4
				1*5+4=9					2*5+4=14
				3*5+4=19					

All we need to do is calculate back the path by using the information we have, including end position’s coordinates, start position’s coordinates, previous coordinates set.

Following the example, we have the coordinates of the end position which are at (4,4).

```
int Map::findDir(Position & A, Position & B)
{
    findPath(A, B); // find the path, then record the previous coordinate in the array
    int endPoint = B.x*MAP_SIZE + B.y;
    while (true) {
        int prePoint = pre[endPoint];
        int ix = prePoint / MAP_SIZE;
        int iy = prePoint % MAP_SIZE;
        if (ix == A.x&&iy == A.y) {
            int dirx = endPoint / MAP_SIZE;
            int diry = endPoint % MAP_SIZE;
            if (dirx == ix && diry == iy - 1)
                return UP;
            else if (dirx == ix && diry == iy + 1)
                return DOWN;
            else if (dirx == ix - 1 && diry == iy)
                return LEFT;
            else if (dirx == ix + 1 && diry == iy)
                return RIGHT;
        }
        endPoint = prePoint;
    }
    return 0;
}
```

End point should be 4*5+4=24. In 'previous' array, the value in 24th store the previous coordinate, which is 3*5+4=19. Then, the previous point should be pre [24] = 3*5+4 = 19. It starts to check whether previous point converted to (x,y) coordinate match the start position or not by using div and mod.

Divide 19 by the map size (5), we get the quotient as 3, remainder as 4. That's also matches the path (3,4). If the coordinate of the previous coordinate does not match, it will become the end position and start the next iteration.

Since the end position's coordinate will change in the iteration until the coordinate matches the starting position's coordinate, the path will always be correct.

Then, after ix and iy (obtained from dividing the previous' coordinate) match the starting position, the function should point the direction for the movement towards the previous point. For example, starting position (0,0) should point to (0,1), and so on.

That's the find path algorithm.

Now we have the direction for the ghosts, what we have left is an easy job.

```
int Ghost::move(Map & map,Pacman &pacman)
{
    int res = -1;//return when no direction. Always do checking.
    int tempX = x, tempY = y;
    bool flag = true;
    while (flag) {
        int dir = map.findDir(*this, pacman); // bfs
        switch (dir)
        {
        case UP:
            go = UP;
            --y;
            break;
        case DOWN:
            go = DOWN;
            ++y;
            break;
        case LEFT:
            go = LEFT;
            --x;
            break;
        case RIGHT:
            go = RIGHT;
            ++x;
            break;
```

Note that "this" pointer is a special pointer that is automatically created within a non-static member function of a class. It points to the object for which the member function is called.

It can be used to distinguish member variables. When a member variable has the same name as a parameter or a local variable within a member function, using the "this" pointer can help differentiate between them. It allows us to access the member variable explicitly, ensuring that we are referring to the correct variable.

```
        case RIGHT:
            go = RIGHT;
            ++x;
            break;
        default:
            break;
        }
        if (!map.oK(x, y) || map.goXY(x, y) == WALL ) {
            x = tempX; y = tempY;
        }
        else {
            flag = false;
            res = dir;
            map.renew(tempX, tempY);
        }
        print();
    }
    return res;
}
```

Always, always, always do checking. You won't expect the program to crash, will you?

After the moving part of the ghosts is completed, we should also check whether the ghosts have collided with the player or not and thus give out the result of the game.

```
bool Ghost::hit(Pacman & pacman, Map &map)
{
    int a, b;
    pacman.getXY(a, b);
    if (a == x && b == y) {
        if (map.freezeTime > 0) {
            map.renew(x,y);
            x = init_x;
            y = init_y;
            print();
            pacman.print();
            return false;
        }
        else {
            return true;
        }
    }
    else
        return false;
}
```

For getting the coordinate of the player:

```
void Pacman::getXY(int & a, int & b)
{
    a = x;
    b = y;
    return ;
}
```

, where x and y have been defined in the initialize.

Note that if the ghosts have collision with the player during the super pean period, the ghosts should be eaten and teleported back to their home which is located at the center of the map.

And now what we have to do is to put these modules into the main game iteration.

```
while (true) {
    dir_pacman = -1;
    for (int i = 0; i < ghost_num; i++)
        dir_ghosts[i] = -1;
    //ghosts
    if (map.freezeTime == 0) {
        ++speed_adapter;
        if (speed_adapter == GHOST_SPEED) {
            speed_adapter = 0;
            //all ghosts move to player
            for (int i = 0; i < ghost_num; i++) {
                dir_ghosts[i]=ghosts[i].move(map, pac_man);
            }
        }
        pac_man.color = YELLOW_COLOR;
    }
    else { //Super pean period
        map.freezeTime--;
        PacmanColors_i = (PacmanColors_i + 1) % (sizeof(Pacman_Colors) / 4);
        pac_man.color = Pacman_Colors[PacmanColors_i];
        pac_man.print();
    }
```

Same as above. Become more familiar with those function.

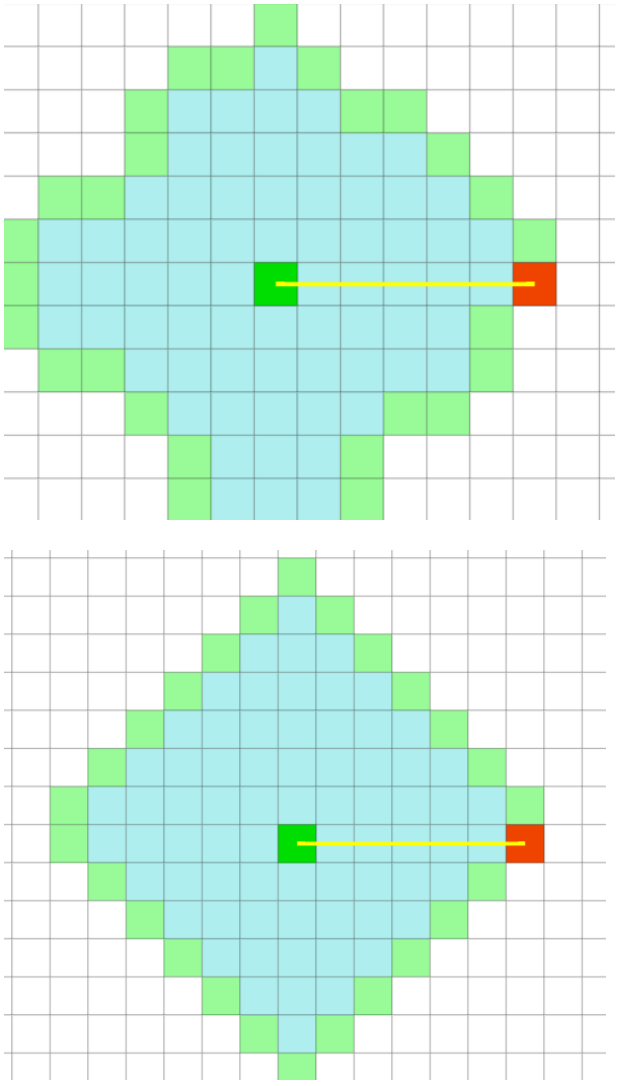
```
if (map.scores == map.target_scores) {
    return game_win();
}

for (auto&ghost_i : ghosts) {
    if (ghost_i.hit(pac_man,map))
        return game_over();
}
```

Conclusion

Here, we've finished all the parts of this game.

For the reason why I tried to record the movement of the ghosts and the player, is that I wanted to make a replay system since all the components are finished like direction, map, move, etc. This should be an easy job which manages these modules together. But what I discovered is that the game will easily crash during the replay and thus I give up this idea. If there is a better algorithm which consumes less memory, I may complete this function.

For the find path algorithm, there is another algorithm similar to breadth-first search called Dijkstra which may consume less memory. But it's too difficult to understand what the algorithm is doing and thus I choose breadth-first search.



Can you guess which is bfs and which is Dijkstra?

Reference

1. https://qiao.github.io/PathFinding.js/visual/ (bfs)
2. https://www.youtube.com/watch?v=KiCBXu4P-2Y (also bfs)
3. https://www.youtube.com/watch?v=Y37-gB83HKE (also bfs)