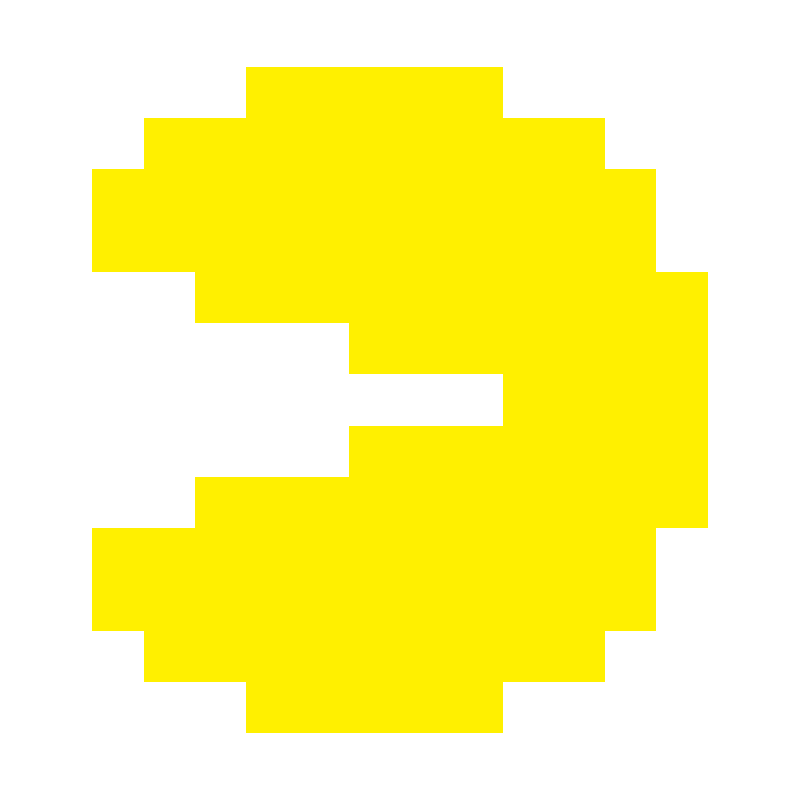
Hong Kong Taoist Association

Tang Hin Memorial Secondary School

Information and Communication Technology

School-Based Assessment



Pac-Man

Li Ho Yuen 6B 8

Contents

[Objective 3](#_Toc151927615)

[Introduction 3](#_Toc151927616)

[Workflow 4](#_Toc151927617)

[Design 4](#_Toc151927618)

[Programming language 4](#_Toc151927619)

[User interface 5](#_Toc151927620)

[Components--Map 6](#_Toc151927621)

[Player 8](#_Toc151927622)

[Ghosts 9](#_Toc151927623)

[Records 9](#_Toc151927624)

[Main Game 11](#_Toc151927625)

[Implementation 13](#_Toc151927626)

[Menu 13](#_Toc151927627)

[Game Example 14](#_Toc151927628)

[Map 19](#_Toc151927629)

# 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.

A video game with a game of pacman


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.

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.

## Workflow

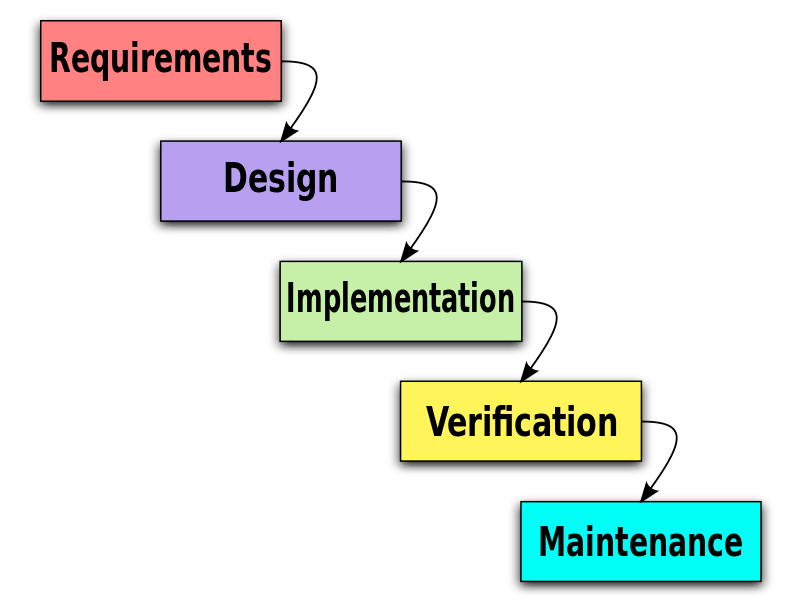
I use Waterfall Model to complete my program. The Waterfall model follows a structured approach with distinct phases, including requirements gathering, design, implementation, and others.

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.

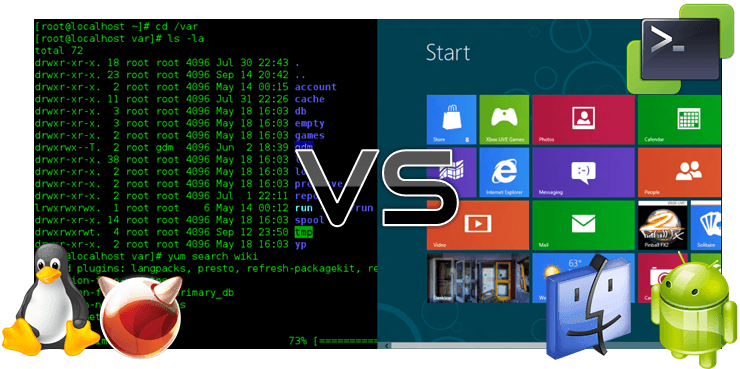
OOP is often more suitable for complex projects with multiple interacting components. It provides a structured and modular approach to handle complexity and offers better code organization and reusability. Procedural programming may be more appropriate for simpler projects with straightforward logic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

## Components--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:

A computer screen with text

Description automatically generatedA computer screen with text on it

Description automatically generated

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.

A screen shot of a computer code

Description automatically generated

The type class of the point

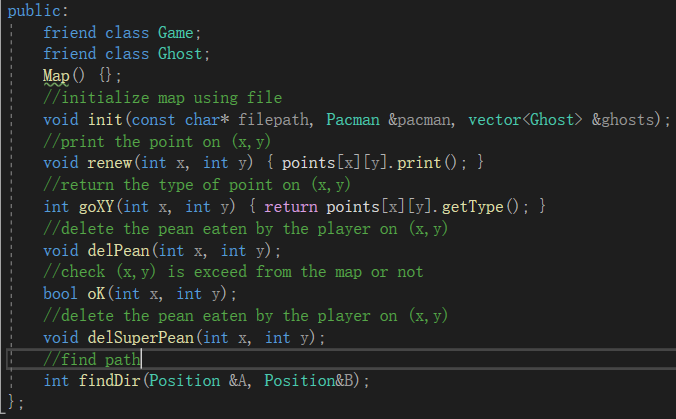
Thus, encapsulation can be used to package those classes into one class.

A screen shot of a computer code

Description automatically generated

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.

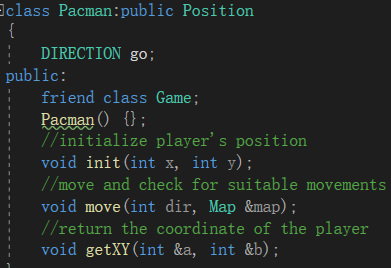
Thus, public classes should be defined for further reuse. For example, map class should be linked to ghost’s class, player’s class, main game’s class, etc.



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:



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.

Similar to 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:

A close-up of a computer error

Description automatically generated

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

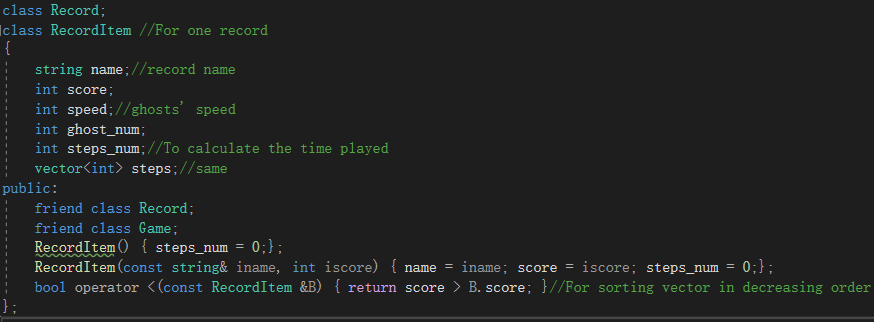
## Records

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.

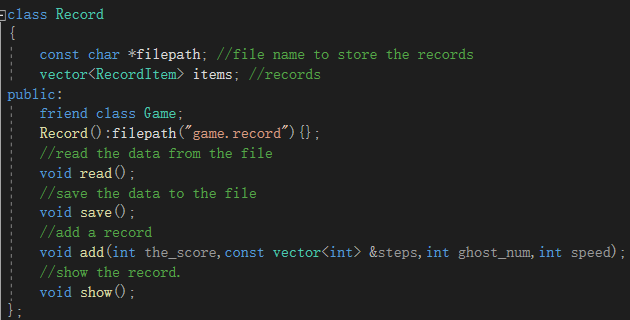
I have limited the number of records outputted to 10. Therefore, the size of the array to store the records won’t be too large. For a small amount of data, various sorting algorithms can be used, but some algorithms are particularly efficient for small data sets due to their simplicity and low computational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

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 allowing us to iterate over its elements using standard algorithms like std::find , std::sort, etc. This makes it convenient to perform common operations on vectors without manually managing indices or loops.

Note that although <vector> act 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:

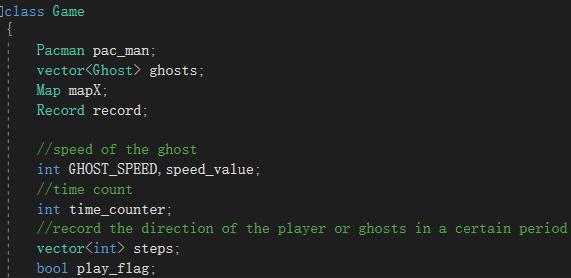


This is for one record. To encapsulate it, another class will be defined.



## 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.



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.

A screen shot of a computer program

Description automatically generated A screen shot of a computer program

Description automatically generated

The main flow of the program should be like this:

A diagram of a game

Description automatically generated

# Implementation

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

## Menu

A screen shot of a computer

Description automatically generated

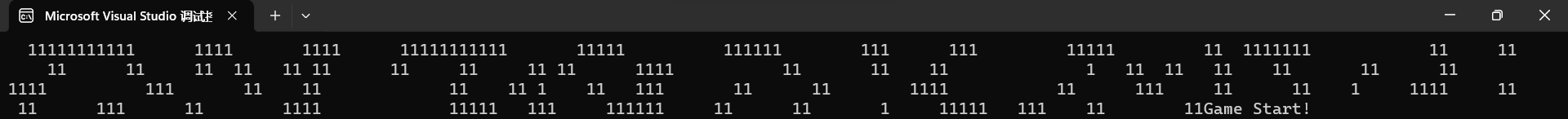
The program is run on the console. The reason why command line interface is preferred has been explained in “Design” part.

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

A computer screen with text

Description automatically generated

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.

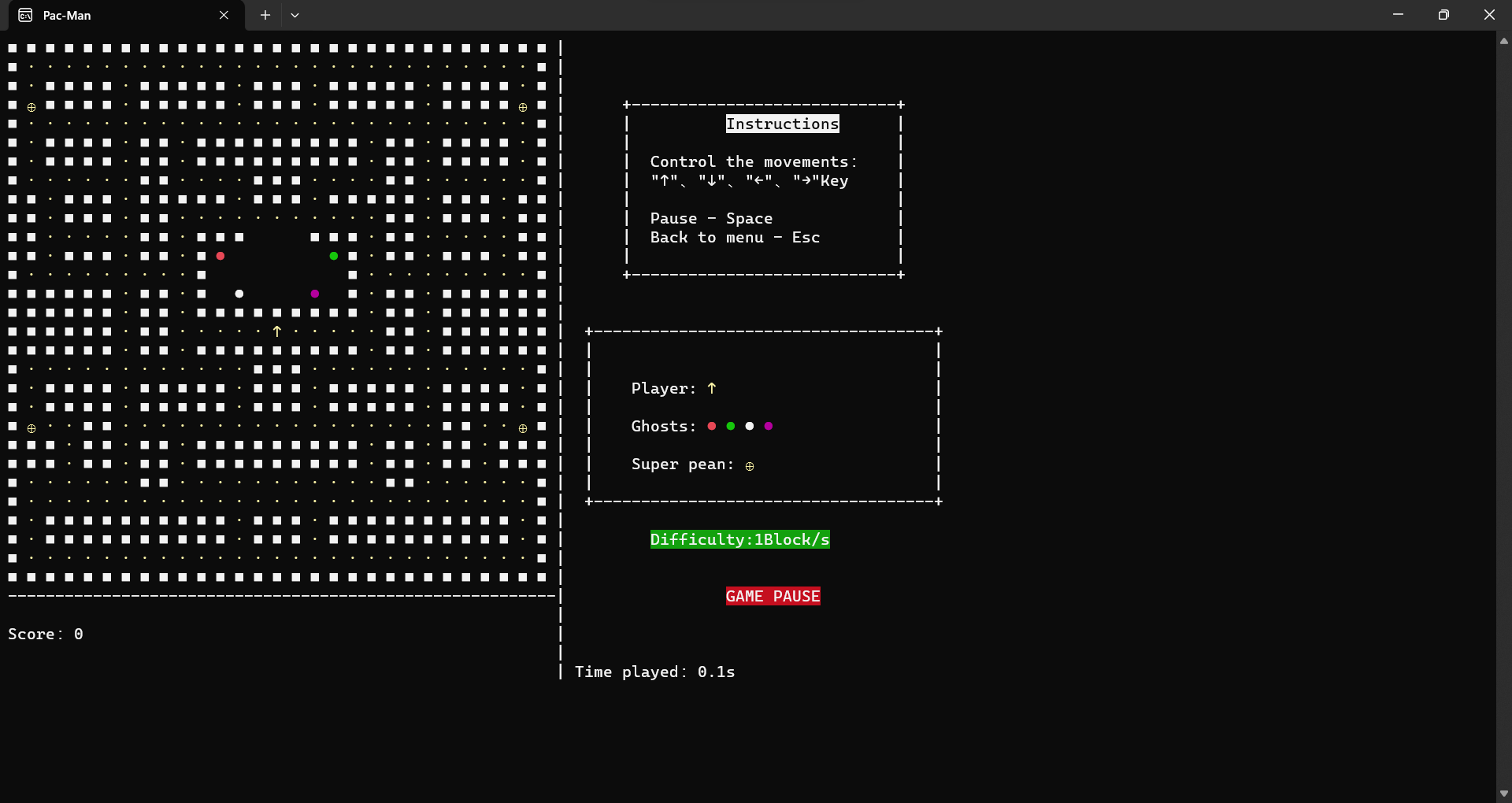
A screenshot of a computer

Description automatically generated

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)

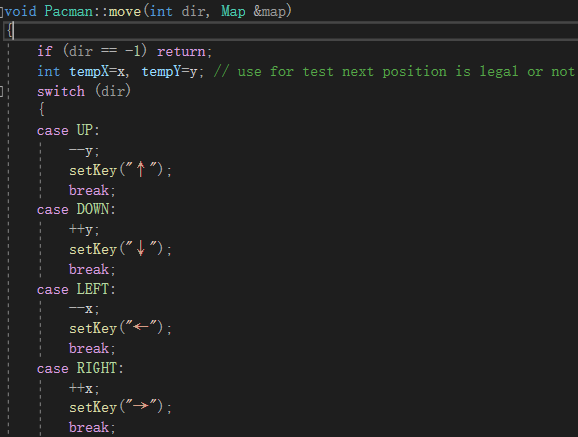
We can see that the layout of the map, instructions and the score, time, etc. are perfectly arranged by using “Gotoxy”. To show how the player is controlled, the instruction writes the key to control the player to go upward or downward etc. Also, the score represents the peaks eaten by the player. Time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

A screenshot of a computer

Description automatically generated

We can use arrow keys to control the player. By using ‘readkey’ 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.



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.

A screenshot of a computer

Description automatically generated

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.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

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 game over.

A screenshot of a computer

Description automatically generated

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.

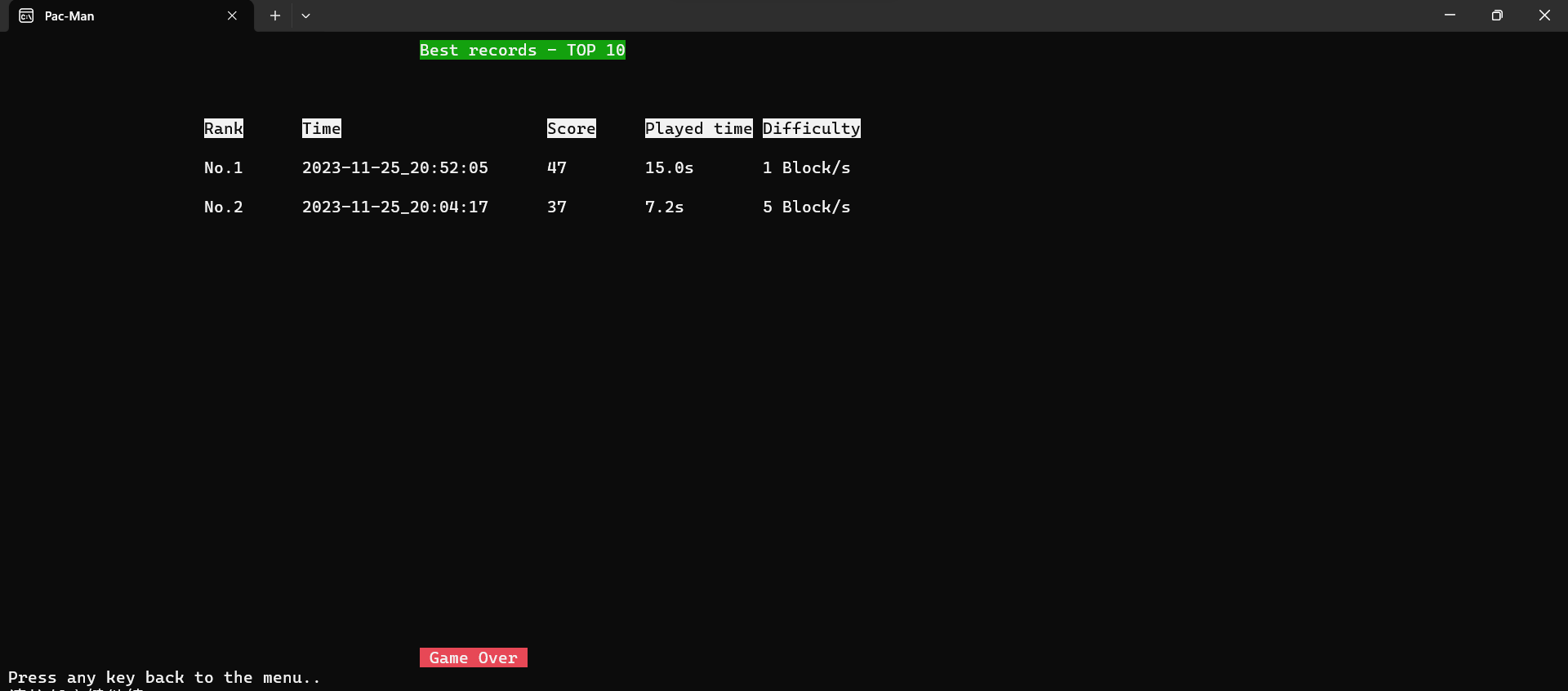
A screenshot of a computer

Description automatically generated

(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.

A screenshot of a computer

Description automatically generated

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.

A black and white square pattern

Description automatically generated

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.

--BIIIIIIIIIIIIG 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.

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

A screen shot of a computer program

Description automatically generated

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. A screenshot of a computer

Description automatically generated

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.

A computer screen shot of a program code

Description automatically generated

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 loop, the movement of the player should also be a loop. 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.

A screen shot of a computer program

Description automatically generatedFor 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.

A computer screen shot of a program code

Description automatically generated

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.

A black background with white text

Description automatically generated

, 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.

, where gameover() includes record adding and history showing. A black background with white text

Description automatically generated

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 loop and ghosts’ finding path loop are executing, and how to restart the game logic when the user end 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.

A computer screen shot of text

Description automatically generated

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

A computer screen with white text

Description automatically generated

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 loop. 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 loop.

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

A diagram of a game

Description automatically generated