

Pac-Man

**Table of Contents**

[Overview 1](#_Toc383769544)

[Game terms 1](#_Toc383769545)

[Before you start! 2](#_Toc383769546)

[Part 1 – Player 2](#_Toc383769547)

[Part 2 – Ghost 4](#_Toc383769548)

[Part 3 – Collision 6](#_Toc383769549)

[Testing 8](#_Toc383769550)

[Extra 8](#_Toc383769551)

# Overview

Make a simplified version of turn-based Pac-Man.

The Visual Studio solution and project have already been created for you. Open the file named "*Pacman.sln*". Within the project, there are 8 files: "*Console.h*", "*Console.cpp*", "*globals.h*", "*Player.h*", "*Player.cpp*", "*Ghost.h*", "*Ghost.cpp*" and "*main.cpp*" files.

# Game terms

* Good reference video of Pac-Man’s gameplay: <https://www.youtube.com/watch?v=uswzriFIf_k>
* Maze – collision that keeps the player within the level.
* Warps – the 2 spots on either side of the maze that allow you to warp to the opposite side.
* Player (Pac-Man) – PAC in globals.h – **<** symbol used to represent the location of the player in the maze.
* Ghosts – MGH in globals.h – smiley face character used to represent the enemies in the game. Pac-Man will lose a life if they collide (except for when Power Pellet is active, see below).
* Ghost cage and door – MGD in globals.h – spawning location of the ghosts and a one-way door that lets the ghosts leave the cage but prevents them and the player from entering.
* Dot – MDOT in globals.h – lowercase o’s that give the player 10 points if they are collected. Ghosts move right through them.
* Power Pellet – MPP in globals.h – digit 0 that gives the player 50 points when collected. Also allows the player to go on the offensive and attack the ghosts.

# Before you start!

* Look through the provided code, main.cpp and globals.h. Compile and run it. Make sure you understand all the code given (excepting the Look Nice section) including the example code.
* There are // TODO Part #: throughout the code. These will help guide you on where the code should be filled-in at.
* The maze skips every other X position; only odd indices contain anything important in the maze array. This is done to make it look nicer however it makes the code slightly more complex. When moving up and down you will only want to move 1 spot, but when moving left and right always move 2 spots at a time.
* When using the 2D maze array, remember it’s [row][col], which means it will be [position.y][position.x].
* Feel free to write the code within the parts in any order. For example you don’t need the player to enter their name or display the score in order to test Pac-Man’s movement and collision code.
* Skim through an entire Part before starting on it to get a sense of what you will be accomplishing each day. This can also help with the previous bullet point and guide you picking the easiest/most important things to work on first.
* **Disable the example code in main.cpp when you are ready to start.**

# Part 1 – Player

Player.h

* Write the class declaration for Player.
* Methods:
  + Overloaded constructor that takes in the maze, a Coord for the starting position and a string for the player’s name.
  + Destructor.
  + Move that doesn’t return anything and takes in the maze and a Coord for the movement.
  + DisplayHUD that doesn’t return anything or have any parameters.
  + A private Draw method that doesn’t return anything and takes in the maze.
  + A private ClearSpot method that doesn’t return anything and takes in the maze.
* Data members:
  + Coord for position.
  + Pointer to character for the player’s name.
  + Int for score.
  + Int for lives.

Player.cpp

* Constructor:
  + Assign the Coord data member to store that parameter.
  + For the name data member you will need to calculate the length of the parameter, then dynamically allocate the data member and finally copy the parameter into the data member.
  + Set score to 0 and lives to 3.
  + Call the Draw method.
* Destructor:
  + Deallocate any memory allocated by the class.
* Move:
  + The Coord parameter is the movement offset, so you will need to add it to your current position to get your new position. However **before** we do this we need to check that the new spot does not contain one of the line types.
  + To do this we need to look at the character in the maze at the desired position (don’t forget it’s maze[pos.y][pos.x]) and see if it’s 1 of the 7 line types used (look in globals.h). If so, return from this function as the move is invalid. Otherwise it’s safe to move.
  + The other thing we need to account for is collision with Dots and Power Pellets. If the new spot is a Dot, simply add 10 to the score. If it’s a Power Pellet add 50 to the score. During Part 2 we will need to add more code here to handle the collection of a Power Pellet.
  + Don’t do anything with MGH (ghosts) in this function as we want to be able to move to the same place as them for death/eating them which is handled in Part 3.
  + Call the ClearSpot method to erase Pac-Man’s graphics from the current location. ClearSpot must be called before updating our position data member (we want to clear the old spot not our new position).
  + Then apply the move Coord to our position using addition. Finally call the Draw method to display Pac-Man at the new position.
* Draw:
  + We need to claim our current spot in the maze array by assigning it to store PAC (globals.h).
  + Then we want to display PAC in yellow at the player’s current position (will need to use SetCursorPosition).
* ClearSpot:
  + Clear our current spot in the maze array by assigning it to store a space.
  + Then display a space at the player’s current position.
* DisplayHUD:
  + Move the cursor to x = 0, y = 31.
  + Display the player’s name, score and lives.

main.cpp

* Use input validation to ask for the player’s name. Make sure they can’t just press enter and have it accepted. Also be sure to support spaces in their name.
* Dynamically allocate the player using the overloaded constructor.
* Display the HUD by calling the player’s DisplayHUD method and then reset the console’s cursor to get it out of the way by calling the ResetCursor function defined in main.cpp.
* Write an infinite loop; this will keep the game running until you get a game over or choose to exit.
* Use input validation and the \_getch() function to get input from the player. Ensure they are only allowed to enter valid directions (w, a, s or d) or e for exit.
* If e is entered, exit the loop. Otherwise you will need to make a Coord structure that corresponds to their selected move. For example if they type in a ‘w’ you will need a Coord with x = 0 and y = -1. Don’t forget that the left and right movement will need to always be 2 instead of 1 to skip the empty spots.
* Call the player’s move method.
* Display the HUD and reset the cursor.
* After the infinite loop, deallocate the player object so we don’t leak any memory.

# Part 2 – Ghost

Ghost.h

* Write the class declaration for Ghost.
* Methods:
  + Overloaded constructor that takes in a ConsoleColor for the draw color of the ghost and a Coord for its starting position.
  + Move that doesn’t return anything and takes in the maze and an array of ghost pointers.
  + Draw that doesn’t return anything and doesn’t take anything in.
  + A private TryResetSpot method that doesn’t return anything and takes in the maze and an array of ghost pointers.
  + A private ClearSpot method that doesn’t return anything and takes in the maze.
* Data Members:
  + Coord for position.
  + ConsoleColor for the ghost’s draw color.
  + Int for direction to avoid moving in (prevents the ghost from ever turning around, otherwise they can end up jittering back and forth in a corner somewhere).
  + Bool for whether the ghost is inside the starting cage.

Ghost.cpp

* Constructor
  + Assign the Coord and ConsoleColor data members to store the parameters.
  + AvoidMove should be set to -1 or some other invalid direction as any move direction is fine at the start of the game.
  + InsideCage needs to be setup depending on the ConsoleColor of the ghost. All of them except for Red start inside the cage.
  + Finally call the Draw method.
* Move
  + This method is in charge of moving **this** ghost, **not** all of the ghosts. The only reason this method takes in the array of ghost pointers is to be able to pass it to the TryResetSpot method.
  + To start we need an infinite loop to ensure the ghost keeps trying to move until it finds a valid direction and doesn’t run into any collision.
  + **NOTE: This infinite loop can cause problems** if you have logic errors in your code. If the game seems to freeze-up at any point it is very likely it is stuck inside this infinite loop. Usually there is a reason it can’t successfully choose a direction to move - use debugging skills to figure out exactly what the problem is and annihilate it.
  + The ghost needs to randomly pick a direction to move in (keeping in mind that X’s move by 2 instead of 1). This is best accomplished using a 2nd loop since they will need to avoid moving in the opposite direction they last moved in (use AvoidMove data member for this).
  + This is further complicated by the fact that we want upwards movement to be weighted higher when inside the ghost cage. This really helps the ghosts get out of the cage when the game starts. If inside the cage, give them a 5 in 8 chance of moving upwards (not ignoring the AvoidMove data member though). If outside the cage, weight all directions evenly at a 1 in 4 chance.
  + We now need to handle the warps on either side of the map. If our position matches either of the 2 warps and the ghost is moving towards the side of the screen then we need to warp the ghost. The easiest way to handle this is by adjusting the desired move Coord. For example on the left side instead of moving X = -2, changing it to X = 52 will make it appear that the ghost warps to the right.
  + Next we need to check the movement + current position for collision with the walls. This should look very similar to what you wrote in Player’s Move. The difference is if inside the cage we will allow them to move into the MGD ghost door spot (globals.h), if outside the cage we won’t. Make sure if we hit collision we don’t just exit the function, instead we should still be inside the infinite loop to try picking a different direction.
  + No collision, so we need to actually move now. Set up the AvoidMove data member so that next time we can’t move the opposite direction of what we are about to move currently.
  + Call the TryResetSpot method.
  + Apply the movement to our current position.
  + Draw.
  + Finally break out of the infinite loop as we have successfully moved the ghost.
* Draw
  + Move the console’s cursor to the ghost’s position.
  + Display the ghost’s symbol (globals.h) using the ConsoleColor data member.
* TryResetSpot
  + We don’t want to clear the spot if another ghost is in the same position we are; they will handle clearing the spot when they leave. If you both attempt to handle it you will get some weird drawing results.
  + Loop through the ghosts array and find out if they are in the same spot as you. Keep in mind you are also in the array and that should **NOT** count as another ghost being in the same spot.
  + If there are no ghosts at your location, call the ClearSpot method.
* ClearSpot
  + Move the console’s cursor to the ghost’s position.
  + Display the item in the maze array at the ghost’s position (you may need to change the current color to match the item being displayed, i.e. White for a Dot).

Player.cpp

* Move
  + Needs to be updated to handle the warps on the sides of the map the same way the ghosts now handle them.

main.cpp

* Add the code to detect memory leaks.
* Seed random.
* Create ghosts.
  + Initialize an array of ghost pointers using NUM\_GHOSTS as the size, setting them all to nullptr.
  + Use a loop to dynamically allocate each ghost pointer in the ghosts array.
* Move ghosts.
  + Call the Move method of all the ghosts in the array.
* Free memory (also check for memory leaks now that we have a way to – use ‘e’ to exit the game and allow memory cleanup to happen for this to be accurate).

# Part 3 – Collision

Ghost.h

* Methods:
  + Kill that doesn’t return anything and takes in the maze.
  + Reset that doesn’t return anything and takes in the maze and a Coord for the position to reset to.
  + An inline GetPos method that returns the Coord data member.
* Data Members:
  + ConsoleColor for the scared color of the ghost (when power pellet is collected).

Ghost.cpp

* Constructor
  + Assign scared color to White.
* Draw
  + Needs a bool parameter added for whether the ghost is currently scared. Don’t forget to update this in the .h as well. May also need to update other methods to take in a scared bool as well to be able to pass to Draw. Not all of them need this update, since for example the constructor calls Draw but can simply pass false for scared since the ghosts are never scared initially.
  + If scared, we need to now use the scared color data member instead of our normal color to draw the ghost. We also want to flip the scared color between White and Blue every time we draw so that the ghost appears to be flashing.
* Kill
  + Call the ClearSpot method.
  + Reset our position to the center of the cage (X = 27, Y = 14).
  + Draw.
  + Change the InsideCage bool to true so that we can get back out of it.
  + Also clear the AvoidMove since any direction is valid again.
* Reset
  + Clear the AvoidMove data member so any direction is valid again.
  + Reset InsideCage based on our color (Red doesn’t start in the cage).
  + Call the ClearSpot method.
  + Finally set the position data member to the Coord parameter.

Player.h

* Methods:
  + Reset that doesn’t return anything and takes in the maze and a Coord for the position to reset to.
  + An inline Kill method that doesn’t return anything or take anything in, simply subtracts 1 from the lives data member.
  + An inline GetPos method that returns the Coord data member.
  + An inline GetLives method that returns the lives data member.
  + An inline GetPowerPellet method that returns the bool Power Pellet data member.
  + An inline GhostKilled method that doesn’t return anything or take anything in, simply adds 200 to the current score.
* Data Members:
  + Bool for whether player has a Power Pellet.
  + Int for keeping track of frames since player last collected Power Pellet.

Player.cpp

* Constructor
  + Assign the Power Pellet bool to false and the Power Pellet frames to 0.
* Move
  + We need to handle Power Pellet collection further now. Instead of just adding 50 to the score we also need to set the PowerPellet bool to true and the Frames to 25.
  + PowerPellet Frame’s data member needs to decrease by 1 at the beginning of the Move method. If it hits 0, turn off the PowerPellet bool. This should allow the player to have the PowerPellet for 25 moves and then turn off.
* Reset
  + Call ClearSpot.
  + Set the position data member to the Coord parameter.
  + Draw.

main.cpp

* CheckCollision function that returns a bool and takes in the maze, player pointer and the array of ghosts.
  + Loop through the ghosts and check if they are at the same position as the player.
  + If so and the Player has a PowerPellet:
    - Kill the ghost.
    - Call the player’s GhostKilled method to increase the score.
  + If so and the Player doesn’t have a PowerPellet:
    - Kill the player.
    - Return true – we don’t need to continue checking for collision.
  + After the loop, return false.
* After the player moves, call the CheckCollision function storing the return value into a local variable. The return value indicates whether the player has died.
* If the player is not dead, allow the Ghosts to move (Part 2) and call CheckCollision again, storing the return value into the local variable.
* If the player is dead from either CheckCollision call we need to reset the game.
  + If the player still has lives remaining (not a game over):
    - Reset the player.
    - Reset all the ghosts.
    - Draw all the ghosts (do not combine this with their reset).
  + If the player has no more lives (game over):
    - Call DisplayHUD so we can see the lives at 0.
    - Then display a Game Over message and break out of the game loop.

# Testing

* Don’t forget to test your game for all the different possibilities to ensure everything is working properly. Not seeding random can occasionally make this process easier.
* Try killing ghosts, getting killed by ghosts, getting the game over, using warps, ghosts using warps, ghosts sharing the same position in the maze and everything else you can think of.
* It is helpful to use debugging skills and breakpoints to test certain things. For example, you can make the ghosts move a specific direction to hit one of the warps instead of just waiting for it to happen.

# Extra

* Add a class for fruit that appears randomly in the maze and adds to the player’s score.
* AI has more complicated move logic to head toward the player.
* AI moves slower when player has a Power Pellet – every other time seems to work well. Make sure they still alternate colors every frame.
* Real-time gameplay – AI moves even if we don’t (will probably need to add some calls to Sleep or add timers to the AI to keep the speed of the game manageable).