Here’s a walkthrough of how this Win32 “Flappy Bird”–style game is structured and what each part does, so you’ll be ready to answer questions about it.

**1. Includes & Global Configurations**

#include <windows.h> // Core Win32 API

#include <vector> // Dynamic array of Pillar objects

#include <cstdlib> // rand(), srand()

#include <ctime> // time() for seeding rand()

#include <string> // std::string

#include <fstream> // File I/O for high‐score persistence

* **windows.h** gives you all the types and functions (e.g. HWND, HDC, CreateWindowEx, message loop, GDI drawing calls).
* The other headers are C++ STL and C library utilities for randomness, containers, strings, and reading/writing a file.

static constexpr int windowWidth = 1520;

static constexpr int windowHeight = 820;

static constexpr int spawnInterval = 60; // frames between new pillars

static constexpr int timerInterval = 12; // ms per frame (~83 FPS)

static constexpr int messageDisplayTime = 2000; // ms to show “You Lost”

static constexpr LPCSTR CLASS\_NAME = "FlappyClass";

static constexpr LPCSTR WINDOW\_TITLE = "Flappy Bird";

* These constexpr values configure the window size, how often to spawn pillars, frame timing, and the Win32 window class/name.

**2. Bird Class**

Responsible for tracking and drawing the player’s “bird.”

class Bird {

public:

Bird(int x, int y);

void update(); // apply gravity and move

void flap(); // give upward velocity

void draw(HDC hdc) const; // render as a circle

RECT getBounds() const; // for collision checks

private:

int x\_, y\_;

float velocity\_;

static constexpr int size\_ = 30;

static constexpr float gravity\_ = 0.4f;

};

* **Data**
  + x\_, y\_ are center coordinates.
  + velocity\_ is vertical speed (positive → falling).
* **update()**
  + Increases velocity\_ by gravity\_, then moves y\_ by that amount each frame.
* **flap()**
  + Sets velocity\_ to –10.0 so the bird jumps up.
* **draw()**
  + Creates a gold brush, draws a filled circle (Ellipse) centered at (x\_, y\_).
* **getBounds()**
  + Returns a RECT around the circle, used for collision detection.

**3. Pillar Class**

Handles the vertical barriers the bird must dodge.

class Pillar {

public:

Pillar(int x, int gapY);

void update(); // move left

void draw(HDC hdc) const; // draw two rectangles (top/bottom)

bool isOffScreen() const; // when to delete

RECT getTopRect() const;

RECT getBottomRect() const;

private:

int x\_, gapY\_;

static constexpr int width\_ = 50;

static constexpr int gap\_ = 150;

static constexpr int speed\_ = 6;

};

* **Creation**
  + You pass in the starting x (right side of window) and gapY (vertical center of the opening).
* **update()**
  + Subtracts speed\_ from x\_ to slide it left each frame.
* **draw()**
  + Draws two green rectangles:
    - **Top** from y=0 down to (gapY - gap\_/2)
    - **Bottom** from (gapY + gap\_/2) down to bottom of window.
* **isOffScreen()**
  + Returns true once the pillar’s right edge is past the left edge of the window (x\_ + width\_ < 0).

**4. Game Class**

Orchestrates everything: bird, pillars, input, scoring, collision, high‐score file I/O.

class Game {

public:

Game(); // seed RNG, load high score

void init(); // reset state

void update(); // main game logic per frame

void draw(HDC hdc); // render bird, pillars, text

void flap(); // on mouse click: either start or make bird jump

bool isGameOver() const;

private:

Bird bird\_;

std::vector<Pillar> pillars\_;

int score\_, highestPillarsCrossed\_;

bool gameOver\_;

int frameCount\_, pillarsCrossed\_;

bool showLostMessage\_, showHighScoreMessage\_;

int lostMessageTimer\_;

void loadHighestPillarsCrossed();

void saveHighestPillarsCrossed();

};

**Key Methods**

* **Constructor**
  + Seeds std::rand() and reads highestPillarsCrossed\_ from "highest\_pillars\_crossed.txt".
* **init()**
  + Clears pillars, resets the bird position, score counters, and hides all messages.
* **update()**
  + **If gameOver & showing “You Lost”**, advances a timer to hide the message after 2 s.
  + **Otherwise**:
    1. bird\_.update() (gravity + movement)
    2. Every spawnInterval frames, create a new Pillar at the right edge with random gap Y.
    3. Move every pillar; when the first pillar goes off‐screen, erase it and increment pillarsCrossed\_.
       - If pillarsCrossed\_ exceeds previous high, update highestPillarsCrossed\_ and flag high‐score message.
    4. **Collision checks**:
       - If bird’s rectangle goes above or below the window → game over.
       - If intersects any pillar’s top or bottom RECT → game over + show “You Lost.”
    5. On game over, write the new high score to file.
* **draw()**
  + Renders the bird, each pillar, then overlays:
    1. Current score (black text at top-left).
    2. Highest score.
    3. “You Lost!” in red if needed.
    4. “You Beat Your High Score!” in green if needed.
* **flap()**
  + If the game is already over, it calls init() to restart. Otherwise it calls bird\_.flap().

**5. Win32 Boilerplate & Game Loop**

**WndProc (Window Procedure)**

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wp, LPARAM lp) {

switch (msg) {

case WM\_CREATE:

SetTimer(hwnd, 1, timerInterval, nullptr);

break;

case WM\_TIMER:

game.update();

InvalidateRect(hwnd, nullptr, TRUE);

break;

case WM\_LBUTTONDOWN:

game.flap();

break;

case WM\_PAINT:

// call game.draw()

break;

case WM\_DESTROY:

KillTimer(hwnd, 1);

PostQuitMessage(0);

break;

default:

return DefWindowProc(hwnd, msg, wp, lp);

}

return 0;

}

* **WM\_CREATE**
  + Starts a Win32 timer that fires every timerInterval ms.
* **WM\_TIMER**
  + Each tick calls game.update() then forces a repaint (InvalidateRect).
* **WM\_LBUTTONDOWN**
  + Player clicks left mouse button: call game.flap().
* **WM\_PAINT**
  + Windows tells you to redraw: call game.draw(hdc).
* **WM\_DESTROY**
  + Clean up the timer and exit the message loop.

**WinMain (Entry Point)**

int WINAPI WinMain(...) {

// 1. Register window class with WndProc & background brush

RegisterClass(&wc);

// 2. CreateWindowEx with the specified CLASS\_NAME, WINDOW\_TITLE, sizes

HWND hwnd = CreateWindowEx(...);

ShowWindow(hwnd, nShow);

// 3. Standard message loop

while (GetMessage(&msg, nullptr, 0, 0)) {

TranslateMessage(&msg);

DispatchMessage(&msg);

}

return 0;

}

* **RegisterClass** binds your WndProc and class name.
* **CreateWindowEx** actually makes the window at windowWidth×windowHeight (plus frame decorations).
* **ShowWindow** makes it visible.
* The **message loop** pumps Windows messages (mouse clicks, paint requests, timer events) into your WndProc.

**6. Putting It All Together**

1. **Startup**
   * WinMain → register window, create it, start message loop.
   * Game constructor seeds RNG, loads high score.
2. **Game Loop** (driven by the Win32 timer)
   * **Every ~12 ms**: WM\_TIMER → game.update() → InvalidateRect → WM\_PAINT → game.draw().
   * **User Input**: clicking triggers WM\_LBUTTONDOWN → game.flap().
3. **Rendering**
   * Bird as a circle, pillars as rectangles, plus text overlays for score and messages.
4. **Game Over & Restart**
   * Collision sets gameOver\_ = true. Next click calls init(), resetting everything.
   * Highest‐score is saved/loaded from a simple text file.

**Possible Interview Questions**

* **Why use a Win32 timer instead of a game loop?**
  + Simplifies tying updates to Windows messages.
* **How does collision detection work?**
  + Using RECT overlap via IntersectRect.
* **Why store high score in a file?**
  + Persistence across runs without a database.
* **What happens when the window is resized?**
  + Window size is fixed (WS\_THICKFRAME removed), so resizing isn’t allowed.
* **How would you improve performance or structure?**
  + Double‐buffering to avoid flicker, or move GDI calls off the UI thread.

Feel free to drill into any specific method or design decision!

**Here’s a two-level pseudocode outline of the Flappy Bird Win32 game:**

**1. Simple, High-Level Pseudocode**

START PROGRAM

Initialize game window and graphics

Load high‐score from file

WHILE window is open DO

WAIT for timer tick (~12 ms)

PROCESS input events

IF left‐mouse clicked THEN

flap bird (or restart if game over)

UPDATE game state

IF game not over THEN

move bird under gravity

spawn new pillars at intervals

move existing pillars left

detect and handle collisions

update score and high‐score

DRAW everything on screen

draw bird

draw pillars

draw current score and high‐score

show “You Lost” or “New High Score” messages if needed

END WHILE

Save high‐score to file

END PROGRAM

**2. Detailed, Class-Based Pseudocode**

// --- BIRD CLASS ---

class Bird:

properties:

x, y // position

velocity // vertical speed

size = 30

gravity = 0.4

method update():

velocity += gravity

y += velocity

method flap():

velocity = -10

method getBounds():

return rectangle centered at (x, y) of width=size

method draw(hdc):

draw filled circle at (x, y) with diameter=size

// --- PILLAR CLASS ---

class Pillar:

properties:

x, gapY // horizontal position and center of gap

width = 50

gapHeight = 150

speed = 6

method update():

x -= speed

method isOffScreen():

return (x + width) < 0

method getTopRect():

return rectangle from top of window down to (gapY - gapHeight/2)

method getBottomRect():

return rectangle from (gapY + gapHeight/2) down to bottom

method draw(hdc):

draw two filled rectangles (top and bottom) at x

// --- GAME CLASS ---

class Game:

properties:

Bird bird

list<Pillar> pillars

int pillarsCrossed, highestCrossed

bool gameOver, showLostMsg, showHighScoreMsg

timers and counters for spawning and messages

constructor:

seed random

load highestCrossed from file

method init():

reset bird position and velocity

clear pillars list

reset scores, flags, timers

method update():

if gameOver:

advance lost‐message timer

return

bird.update()

if spawn timer reached:

create new Pillar at right edge with random gapY

reset spawn timer

for each pillar in pillars:

pillar.update()

if first pillar is off-screen:

remove it

pillarsCrossed++

if pillarsCrossed > highestCrossed:

highestCrossed = pillarsCrossed

showHighScoreMsg = true

check bird out-of-bounds → set gameOver, showLostMsg

for each pillar:

if bird.getBounds() intersects pillar top or bottom:

gameOver = true

showLostMsg = true

if gameOver:

save highestCrossed to file

method draw(hdc):

bird.draw(hdc)

for each pillar in pillars:

pillar.draw(hdc)

draw text “Score: pillarsCrossed” at top-left

draw text “Highest: highestCrossed” below it

if showLostMsg:

draw red “You Lost” in center

if showHighScoreMsg:

draw green “New High Score!” below center

method flap():

if gameOver:

init()

bird.flap()

// --- WIN32 SETUP & LOOP ---

function WndProc(hwnd, message, wParam, lParam):

switch message:

case WM\_CREATE:

set a repeating timer with interval = 12 ms

case WM\_TIMER:

game.update()

invalidate window to trigger repaint

case WM\_LBUTTONDOWN:

game.flap()

case WM\_PAINT:

begin paint → game.draw(hdc) → end paint

case WM\_DESTROY:

kill timer

post quit message

default:

call default window procedure

function WinMain():

define and register window class using WndProc

create window sized to windowWidth × windowHeight

show the window

WHILE GetMessage returns a message:

TranslateMessage

DispatchMessage

return exit code

This two-tiered breakdown should help you explain both the **big picture** (game loop, input → update → draw) and the **detailed structure** (how each class works and how Win32 ties it all together).

Let’s dive deeper into **why** each part of this code is structured the way it is, and **what purpose** each decision serves.

**1. Constants & Configuration**

static constexpr int windowWidth = 1520;

static constexpr int windowHeight = 820;

static constexpr int spawnInterval = 60; // frames between pillars

static constexpr int timerInterval = 12; // ms per frame (~83 FPS)

static constexpr int messageDisplayTime = 2000; // ms to show “You Lost”

* **constexpr** ensures these values are truly compile-time constants—no runtime overhead, easy to tweak in one place.
* **Window dimensions** are set so the play area is wide enough for reaction time but not so large that pillars zip by instantly.
* **spawnInterval** (in frames) decouples pillar frequency from real time; you can adjust difficulty by changing it without touching timing code.
* **timerInterval** (12 ms) yields roughly 83 updates per second—fast enough for smooth motion without overloading the CPU.
* **messageDisplayTime** centralizes how long feedback messages stay on screen.

**2. Bird Class: Encapsulation & Simplicity**

class Bird {

int x\_, y\_;

float velocity\_;

static constexpr float gravity\_ = 0.4f;

static constexpr int size\_ = 30;

…

};

* **Position & velocity** are private: no external code can accidentally warp the bird.
* **gravity\_ as a constant**: encapsulates the game’s “feel.” Tweak gravity in one place to make the bird heavier or lighter.
* **size\_ as a constant**: ensures consistency between drawing (circle diameter) and collision bounds.
* **update()**
  + Applies gravity first, then moves. This simple Euler integration gives believable, arcade-style falling.
* **flap()**
  + Instantly sets upward velocity. By separating input (flap) from physics (update), you avoid accidental double-application of forces.
* **getBounds()** returns an axis-aligned rectangle for collision checks—fast, simple, and good enough for a circle vs. rectangles.

**3. Pillar Class: Modular Obstacles**

class Pillar {

int x\_, gapY\_;

static constexpr int width\_ = 50, gap\_ = 150, speed\_ = 6;

…

};

* **x\_ & gapY\_** define pillar position and opening—no external code needs to know about width or gap, so these are private constants.
* **speed\_** lets you globally adjust how fast obstacles move toward the bird.
* **update()** moves the pillar by subtracting speed\_ from x\_, simulating the world scrolling left.
* **isOffScreen()** encapsulates the deletion condition. The Game class doesn’t guess coordinates; it asks the Pillar itself.
* **Separate getTopRect() / getBottomRect()** ensure collision logic and drawing use the same dimensions, eliminating duplication errors.

**4. Game Class: Central Control & State**

class Game {

Bird bird\_;

std::vector<Pillar> pillars\_;

int pillarsCrossed\_, highestPillarsCrossed\_;

bool gameOver\_, showLostMessage\_, showHighScoreMessage\_;

int frameCount\_, lostMessageTimer\_;

…

};

1. **Single responsibility**
   * **Loading/Saving High Score** lives here, not in WinMain or Bird.
   * **Spawning logic, collision, scoring, message timing** are all methods in Game, keeping global functions minimal.
2. **Random gap positions**
   * By seeding std::rand() once in the constructor, every run has unpredictability—but reproducible within a session.
3. **Frame counting**
   * frameCount\_ drives pillar spawning; this decouples pillar frequency from actual time, making tests easier.
4. **Message timers**
   * lostMessageTimer\_ tracks exactly how long to show “You Lost” so update logic remains clean.
5. **Collision detection**
   * Checking bird bounds vs. each pillar’s rect is simple and fast—no complex geometry needed.
6. **High-score logic**
   * When you cross a pillar, the Game class updates both the current count and the stored high score, then triggers a green “New High Score!” message.

**5. Win32 Integration & Game Loop**

LRESULT CALLBACK WndProc(...){

case WM\_CREATE: SetTimer(...); break;

case WM\_TIMER: game.update(); InvalidateRect(...); break;

case WM\_LBUTTONDOWN: game.flap(); break;

case WM\_PAINT: game.draw(hdc); break;

case WM\_DESTROY: KillTimer(...); PostQuitMessage(0); break;

}

* **Timer‐driven loop**
  + Using SetTimer / WM\_TIMER leverages Windows’ own message pump—simpler than a custom infinite loop.
* **InvalidateRect** after update tells Windows to repaint ASAP, ensuring draw calls happen on the UI thread in WM\_PAINT.
* **Event‐driven input**
  + WM\_LBUTTONDOWN directly maps to “flap” or “restart,” so you avoid polling mouse state.
* **Clean shutdown**
  + WM\_DESTROY kills the timer and signals the message loop to exit, preventing resource leaks.

**Why This Structure?**

* **Maintainability:** Classes isolate behavior (Bird vs. Pillar vs. Game). Changing pillar speed won’t accidentally affect bird physics.
* **Readability:** Each method does one thing—physics, drawing, collision—so you can point an interviewer right at the relevant code.
* **Performance:** Using simple integer math for positions/rectangles and letting GDI handle drawing keeps CPU usage modest.
* **Flexibility:** Tweak a few constants and you’ve instantly changed gravity, pillar spacing, or window size—no code restructuring.

By organizing the code this way, the author achieves a clear separation of concerns, easy tuning of gameplay parameters, and straightforward integration into the Win32 event loop. That’s exactly what you want when building a real-time arcade game in C++.