Name : Mrugank Patil
Roll No: 22110158
B.Tech 2022
Computer Science and Engineering

Assignment 11

Lab Topic: Analyzing C#Console Games for Bugs

Overview

Focuses on debugging and analyzing open-source C# console games to identify and fix bugs.

Objectives

- Analyze and debug C# console game code.
- Use Visual Studio's debugging features: breakpoints, step-in, step-over, and step-out.
- Identify and fix bugs causing unexpected behavior or crashes.
- Understand how control flow and logic affect game behavior

Environment Setup

- Operating System:
    - Windows 10 or higher (64-bit)

- Tools & Versions:
    - Visual Studio 2022 (Community Edition)
    - .NET SDK (Latest stable version with C# 10 or later)
    - Git (Optional, for cloning GitHub repositories)

Tools & Software Installation

1. Install Visual Studio 2022
    - Download: https://visualstudio.microsoft.com
    - Workload: ".NET desktop development"
2. Install Git (Optional)
    - Download: https://git-scm.com
3. Clone the Games Repository
    - git clone https://github.com/dotnet/dotnet-console-games

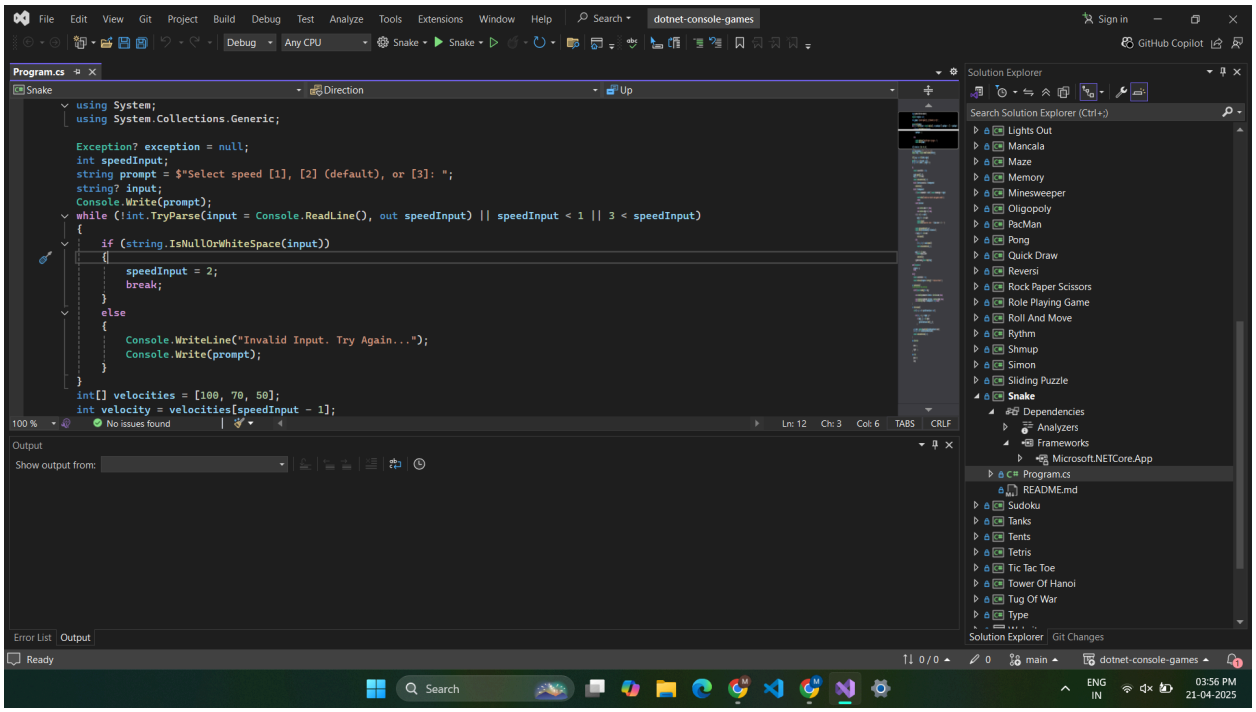
Task :

Snake Game

Task 1 :

Open the Project in Visual Studio and Launch Visual Studio Open Existing Project

- Click File → Open → Project/Solution
- Navigate to the chosen game's folder (snake) and select the .csproj file.



Task 2: Identify the Entry Point

- Even though the code has no Main() method explicitly, it's still a top-level program, which is allowed in C# 9.0+. That means this entire script is the Main() method behind the scenes.
- So the entry point is the first line: `Exception? exception = null;`

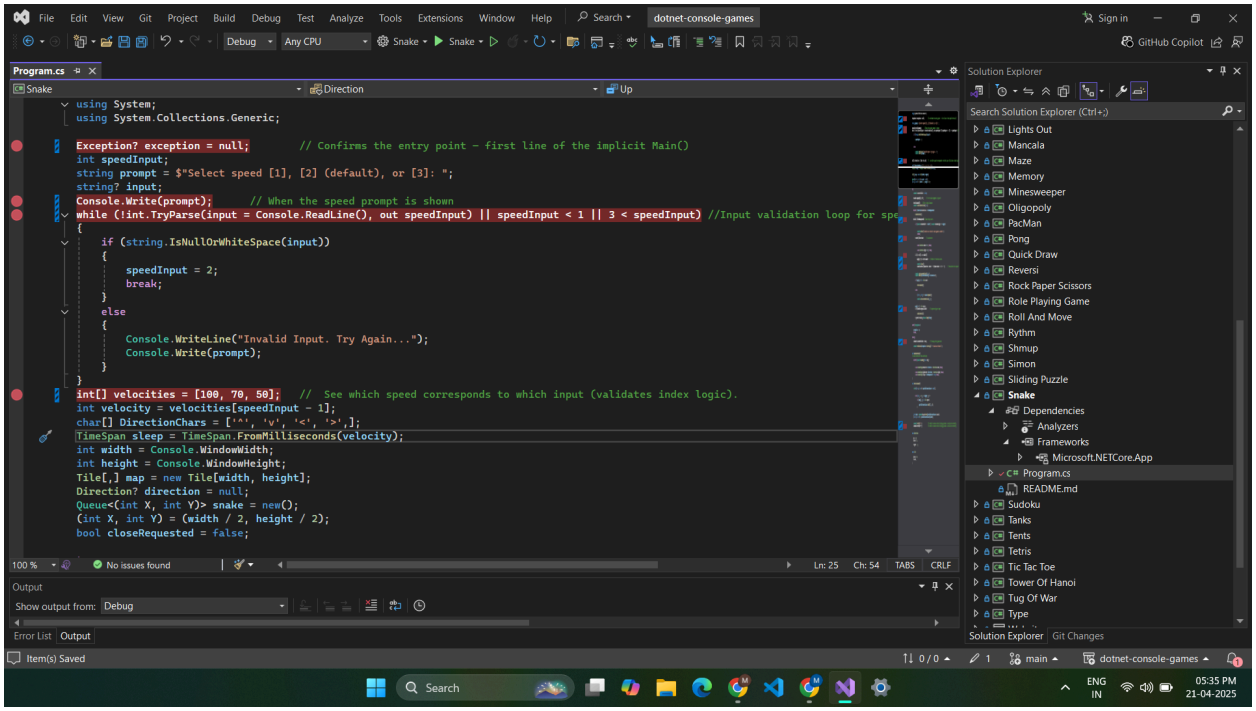
In C# 9.0 and later: Top-Level Statements are allowed. That means, instead of writing:

```
public static void Main()
{
  // code
}
```

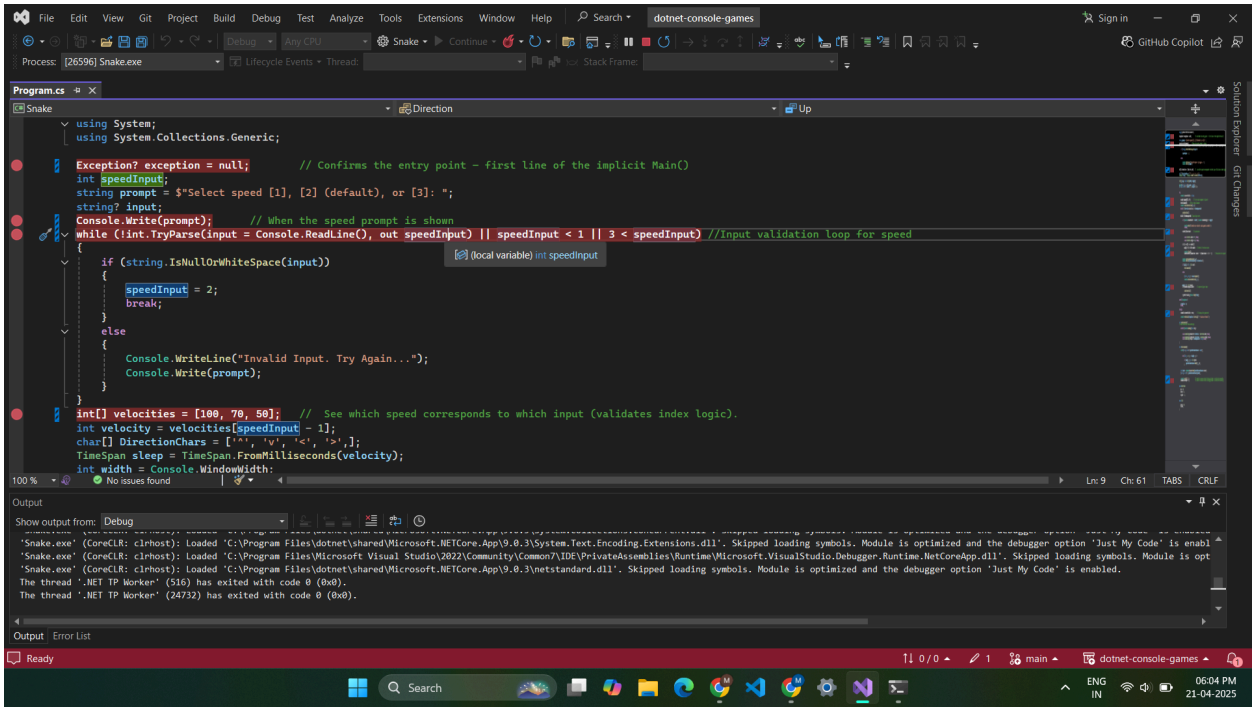
You can just start writing statements at the top of your file, outside of any class or method exactly like your Snake game code. (Exception? exception = null;) That is literally the first line of the top-level program, which means it's the first line executed when the program runs the effective entry point. Even though you don't see Main(), the compiler implicitly wraps all your top-level statements

Task 3: Insert Breakpoints + Run in Debug Mode

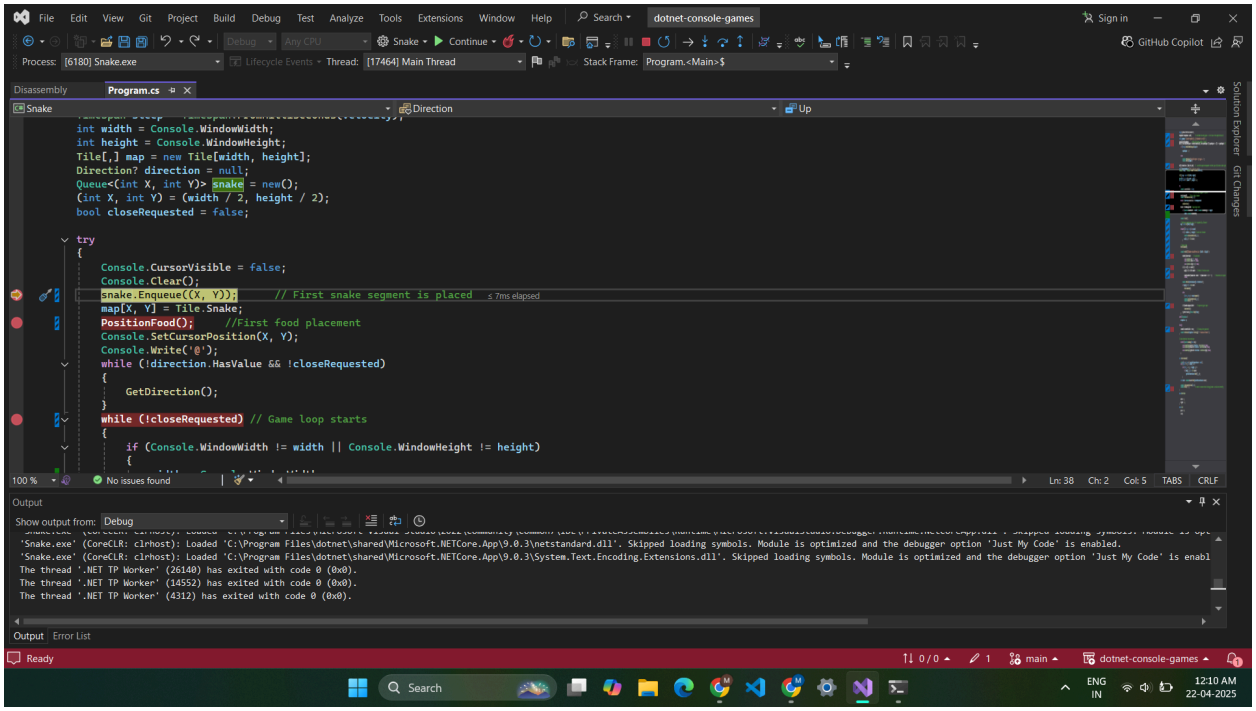
- In the code view (like Program.cs), click left of the line numbers to place red breakpoints.



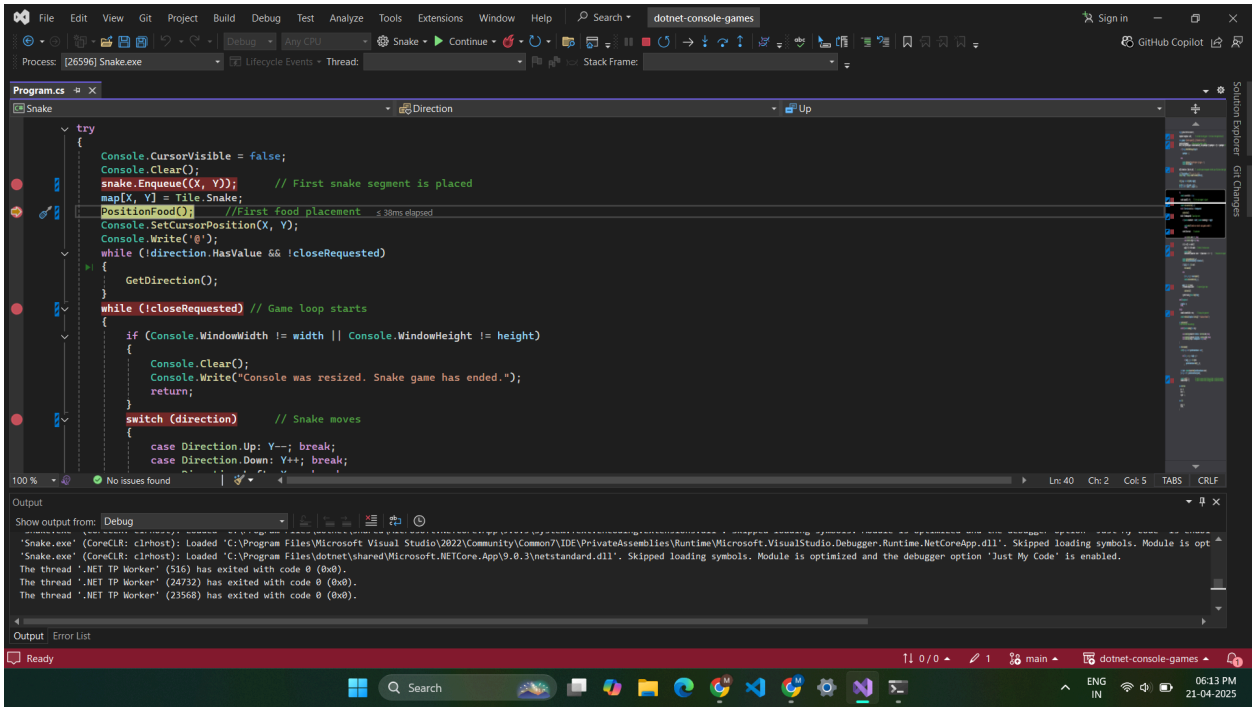
- F10 (Step Over): Execute next line without diving into method.



- F11 (Step Into): Dive into method.



- Shift+F11 (Step Out): Finish method and return to the caller.

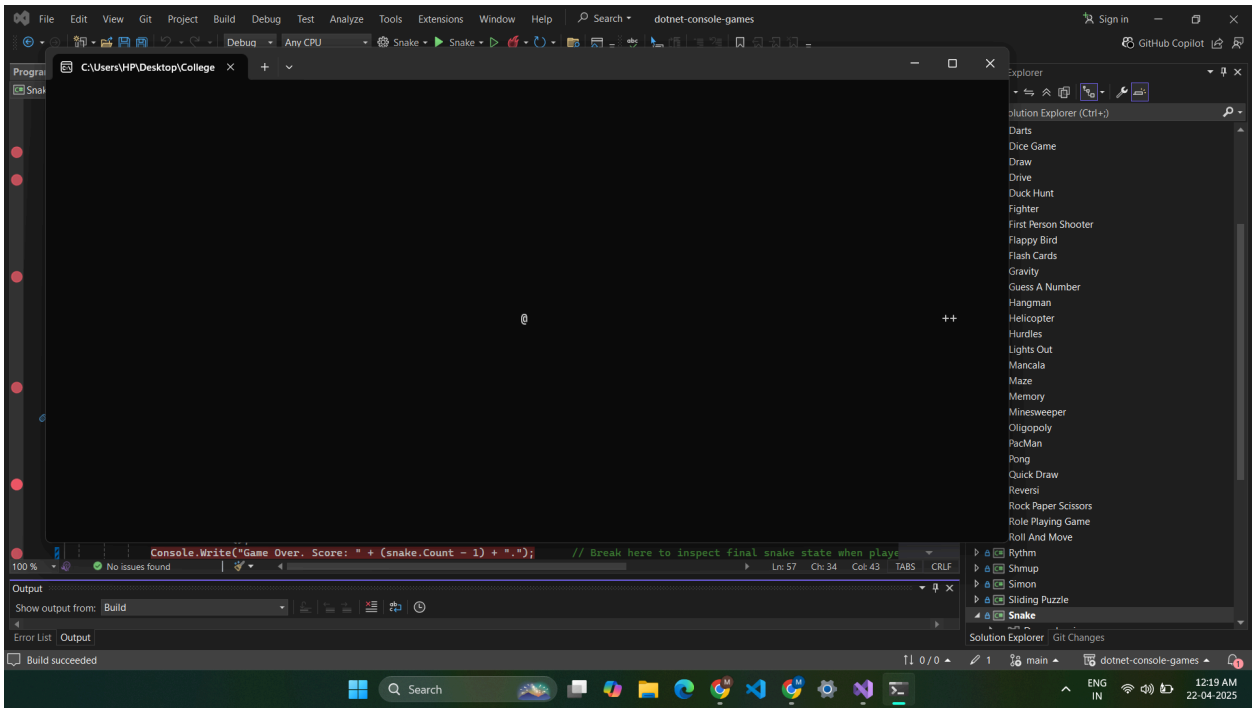


Task 4 : Hunt for a total of five bugs

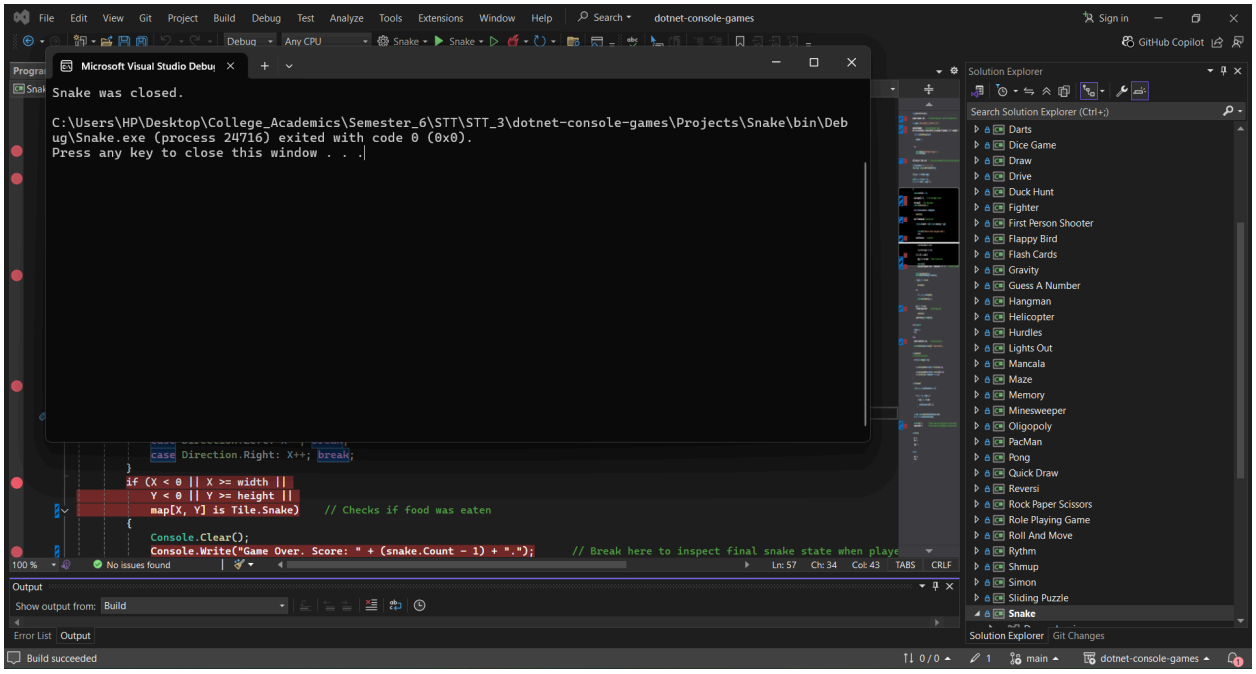
1. BUG 1: Window Resize ends game (Resizing the window ends the game)
    - Where in Code

```
while (!closeRequested) // Game loop starts
{
    if (Console.WindowWidth != width || Console.WindowHeight != height)
    {
        Console.Clear();
        Console.Write("Console was resized. Snake game has ended.");
        return;
    }
```

Before the fix and before resizing

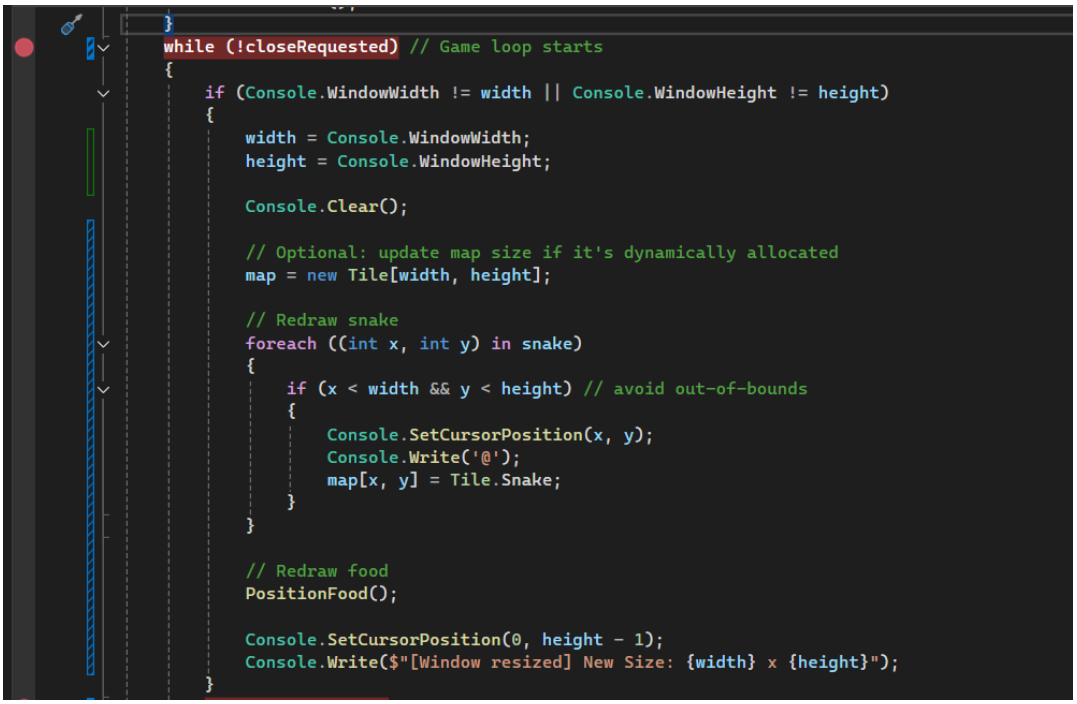


Before the fix and after resizing



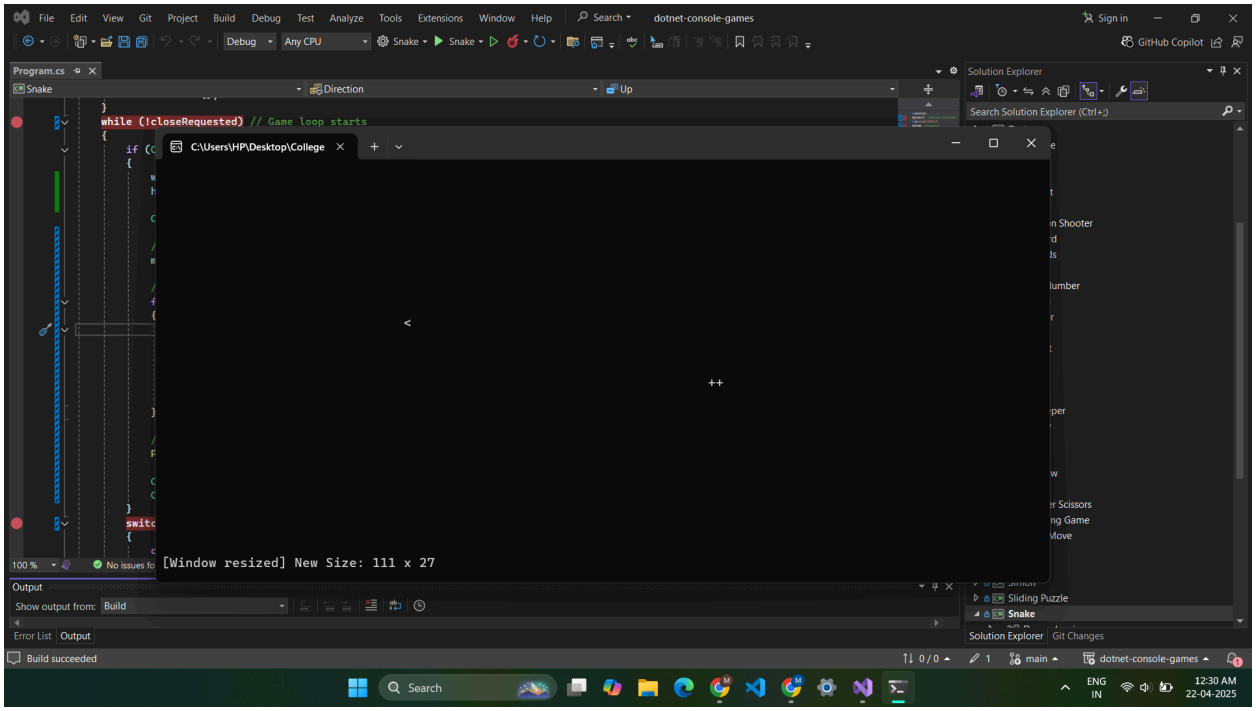
- Fix : Allow Dynamic Resizing
  Instead of ending the game, detect resize and adjust the game's internal width and height. Recompute map boundaries accordingly.

Updated code



```
}
while (!closeRequested) // Game loop starts
{
    if (Console.WindowWidth != width || Console.WindowHeight != height)
    {
        width = Console.WindowWidth;
        height = Console.WindowHeight;

        Console.Clear();

        // Optional: update map size if it's dynamically allocated
        map = new Tile[width, height];

        // Redraw snake
        foreach ((int x, int y) in snake)
        {
            if (x < width && y < height) // avoid out-of-bounds
            {
                Console.SetCursorPosition(x, y);
                Console.Write('@');
                map[x, y] = Tile.Snake;
            }
        }

        // Redraw food
        PositionFood();

        Console.SetCursorPosition(0, height - 1);
        Console.Write($"[Window resized] New Size: {width} x {height}");
    }
```

Running after the fix : Working fine



Fix Implemented: Instead of quitting when the console is resized, we update the internal width, height, and game map, and redraw the snake and food accordingly. This ensures a better and more dynamic user experience.

2. BUG 2 :

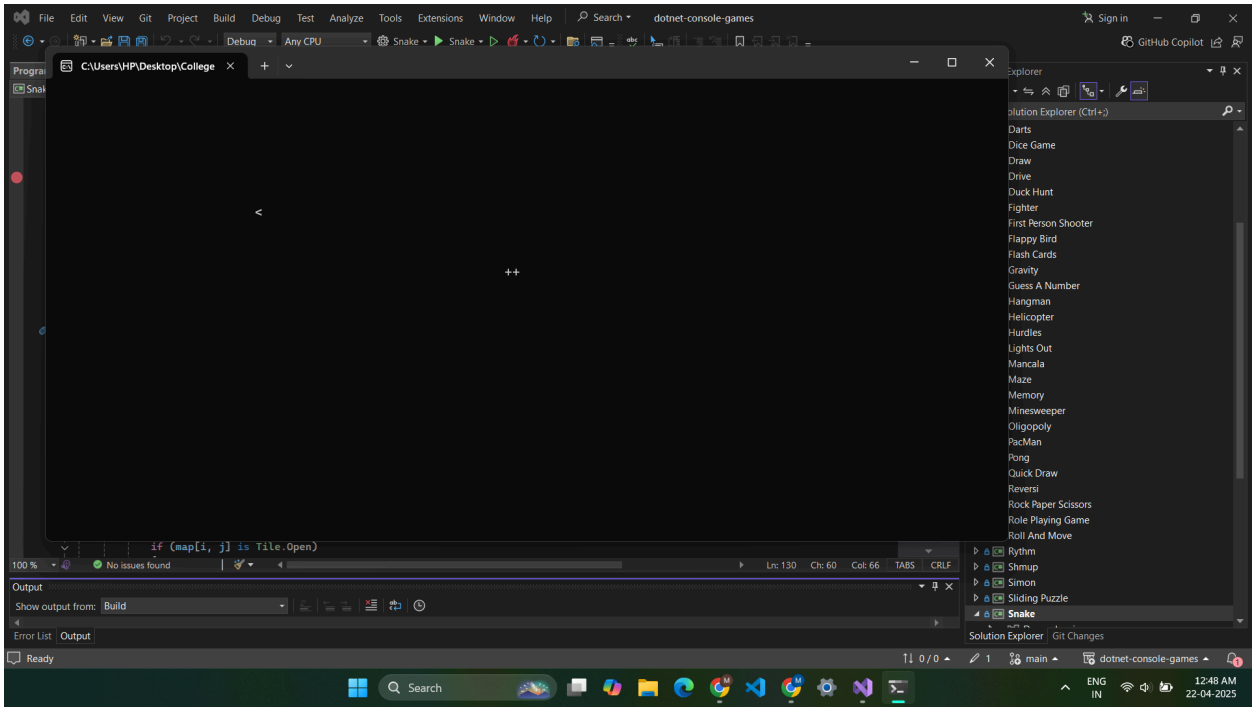
Injected Mutation: Change direction logic (from left to right and vice versa)

- Where in the code the error is located

```
void GetDirection()
// takes direction from arrow keys
{
    switch (Console.ReadKey(true).Key)
    {
        case ConsoleKey.UpArrow: direction = Direction.Up; break;
        case ConsoleKey.DownArrow: direction = Direction.Down; break;
        case ConsoleKey.LeftArrow: direction = Direction.Right; break;
        case ConsoleKey.RightArrow: direction = Direction.Left; break;
        case ConsoleKey.Escape: closeRequested = true; break;
    }
}
```

Now when the player presses the Left arrow, the snake will wrongly go Right. Snake moves in the wrong direction if pressed ←. That's our bug.

Before the fix : The snake is running in the different direction opposite to it



Updated code after the fix

```
2 references
void GetDirection()
// takes direction from arrow keys
{
    switch (Console.ReadKey(true).Key)
    {
        case ConsoleKey.UpArrow: direction = Direction.Up; break;
        case ConsoleKey.DownArrow: direction = Direction.Down; break;
        case ConsoleKey.LeftArrow: direction = Direction.Left; break;
        case ConsoleKey.RightArrow: direction = Direction.Right; break;
        case ConsoleKey.Escape: closeRequested = true; break;
    }
}
```

Fix : Corrected the direction between left and right to its proper directions.

- BUG 3:

  Injected Mutation: Change speed values

  Manipulating the speed of the Snake game to simulate a bug where the game becomes unplayably fast.

  - Where in the code the error is located

```
}
int[] velocities = [100, 7, 5];   //  See which speed corresponds to which input (validates index logic).
int velocity = velocities[speedInput - 1];
```

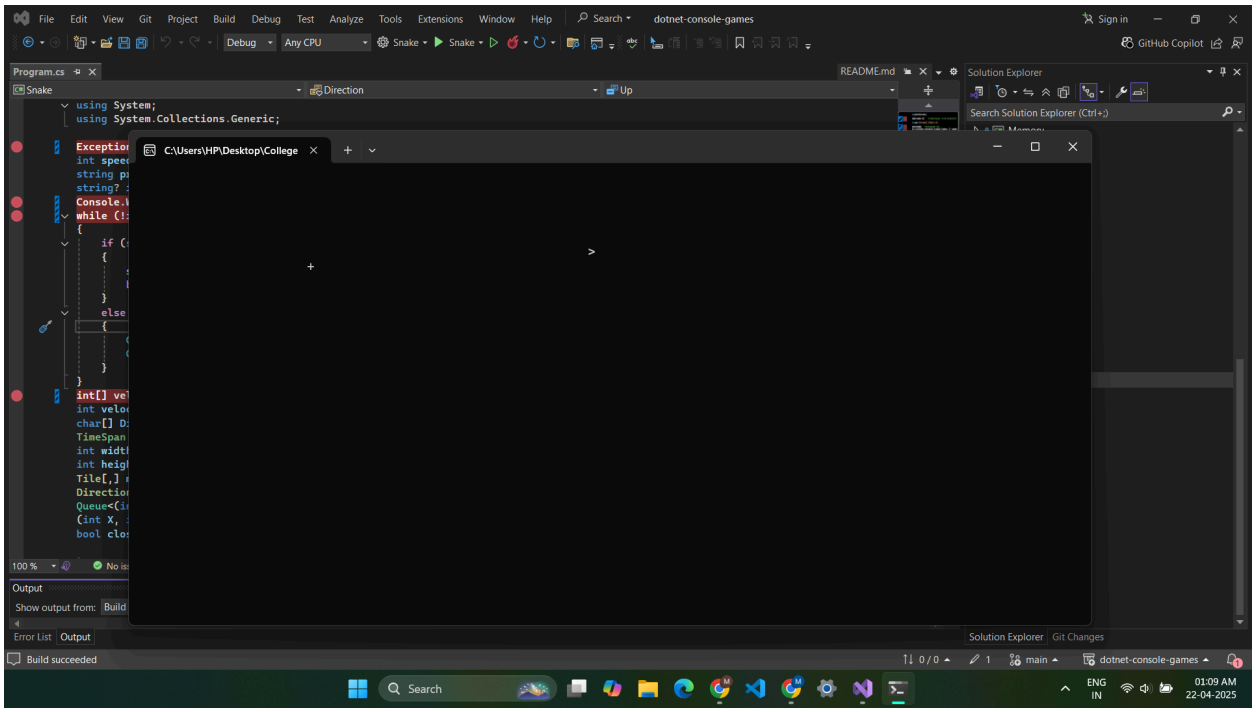
These values represent the sleep time in milliseconds (i.e., delay between snake movements) — lower values = faster game. We have changed it into [100, 7, 5]. This makes the snake almost lightning fast, making it nearly impossible to control — that's your mutation bug.

  - Updated code after the fix

```
}
int[] velocities = [100, 70, 50];   //  See which speed corresponds to which input (validates index logic).
int velocity = velocities[speedInput - 1];
```

Fixed the velocities to normal form as [100, 70, 50].

Game after the fix : Working Properly



- BUG 4 :

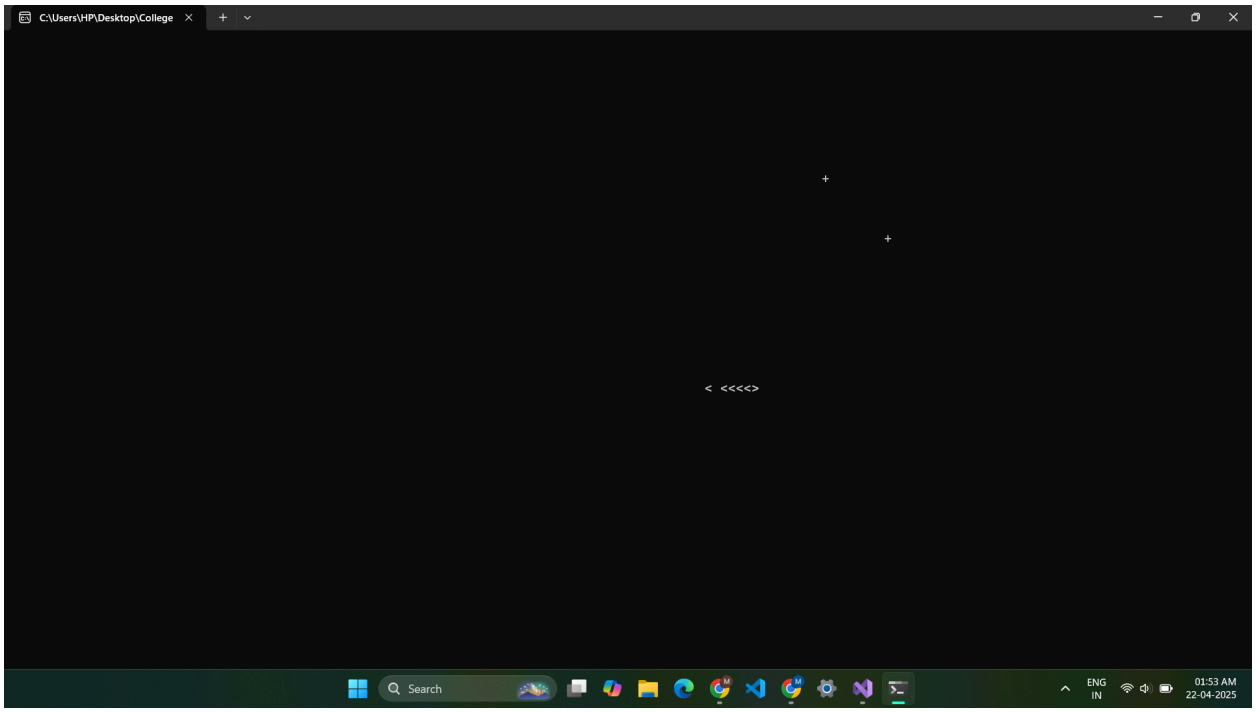
  Injected Mutation : Snake Eats Itself But Game Doesn't End

  - Where in the code where the error lies : Condition inside your game loop that checks if the snake collides with itself

```
        }
        if (X < 0 || X >= width ||
            Y < 0 || Y >= height)  // Checks if food was eaten
        {
```

There is no self-collision check in the above mutated code and that is why the game will continue running even if the snake eats itself also. Snake can move over itself and continue the game.

Game before the fix

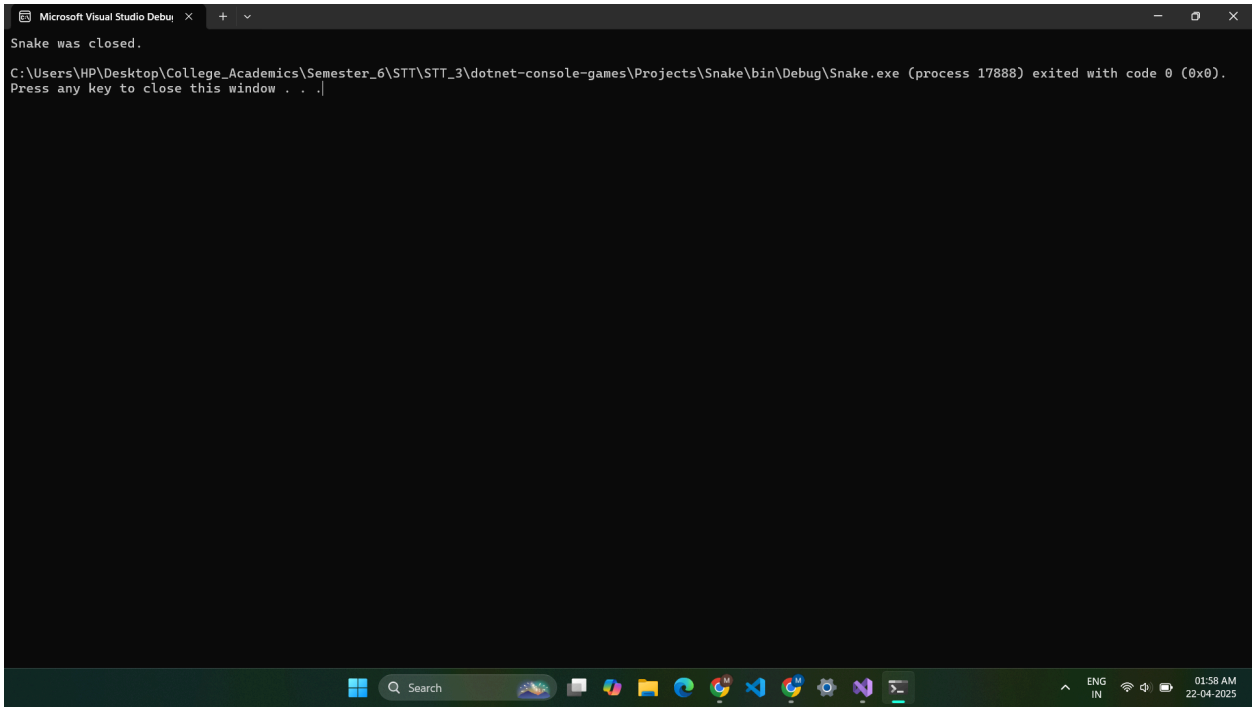


- Updated the code to fix it : Added the condition as map[X, Y] is Tile.Snake

```
        }
        if (X < 0 || X >= width ||
            Y < 0 || Y >= height ||
            map[X, Y] is Tile.Snake)  // Checks if food was eaten
        {
```

Now if the snake eats its own tale then the game gets over

Game after the fix : The game got closed as soon as the snake bites its own tale



Fix : Fixed the line as if (X < 0 || X >= width || Y < 0 || Y >= height || map[X, Y] is Tile.Snake) with adding map[X, Y] is Tile.Snake in the if condition. Now it the snake bites itself then the game will be over.

- BUG 5 :

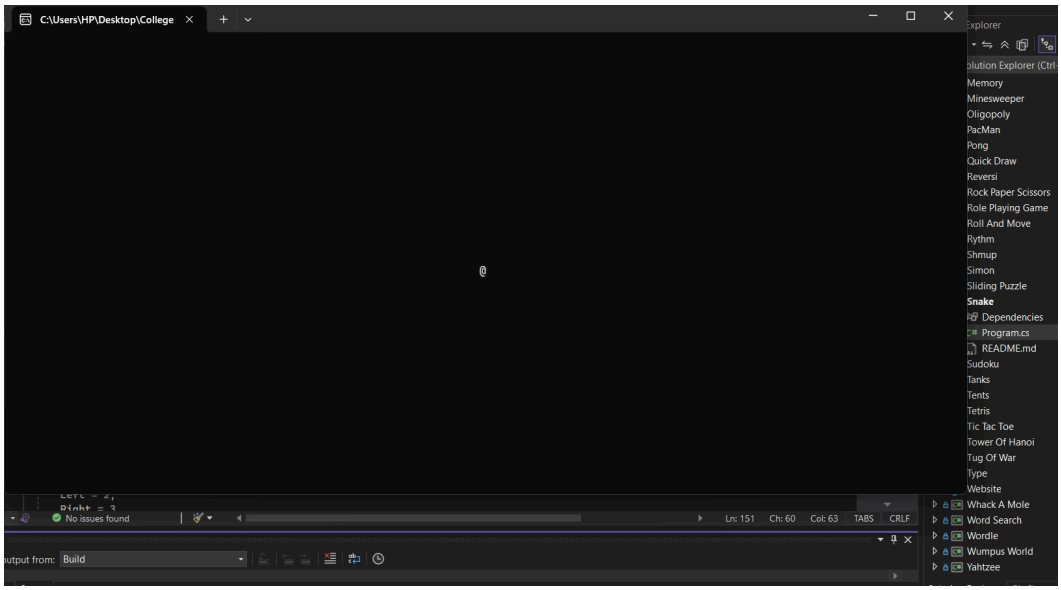
  Injected Mutation : Food Never Appears (Food never appears, game becomes unwinnable or hangs)

  - Where the error is there : Inside your PositionFood() function, where food is randomly placed, it's probably something like this:

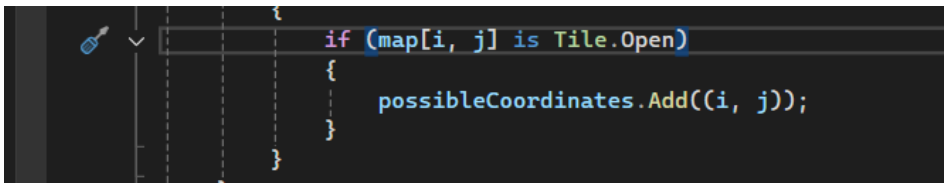
```
if (map[i, j] is Tile.Snake)
{
    possibleCoordinates.Add((i, j));
}
}
```

This will cause the possibleCoordinates list to only include positions where the snake already exists, which is never valid for placing food. As a result, the food either never appears, or the program crashes due to an empty list when it tries to pick a random index. The error is due to because instead of Tile.Snake there should be Tile.Open

Before the fix the game is with no food (“+” sign)



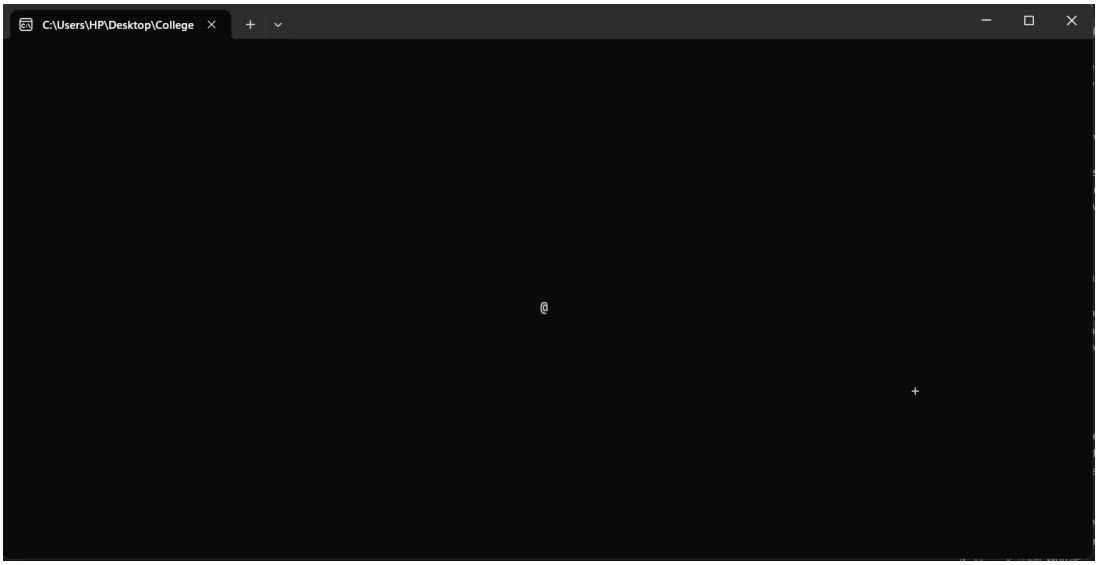
Updated code :



```
{
    if (map[i, j] is Tile.Open)
    {
        possibleCoordinates.Add((i, j));
    }
}
```

Converted the Tile.Snake to Tile.Open and now we are able to see the food (“+”).

Game after the Fix

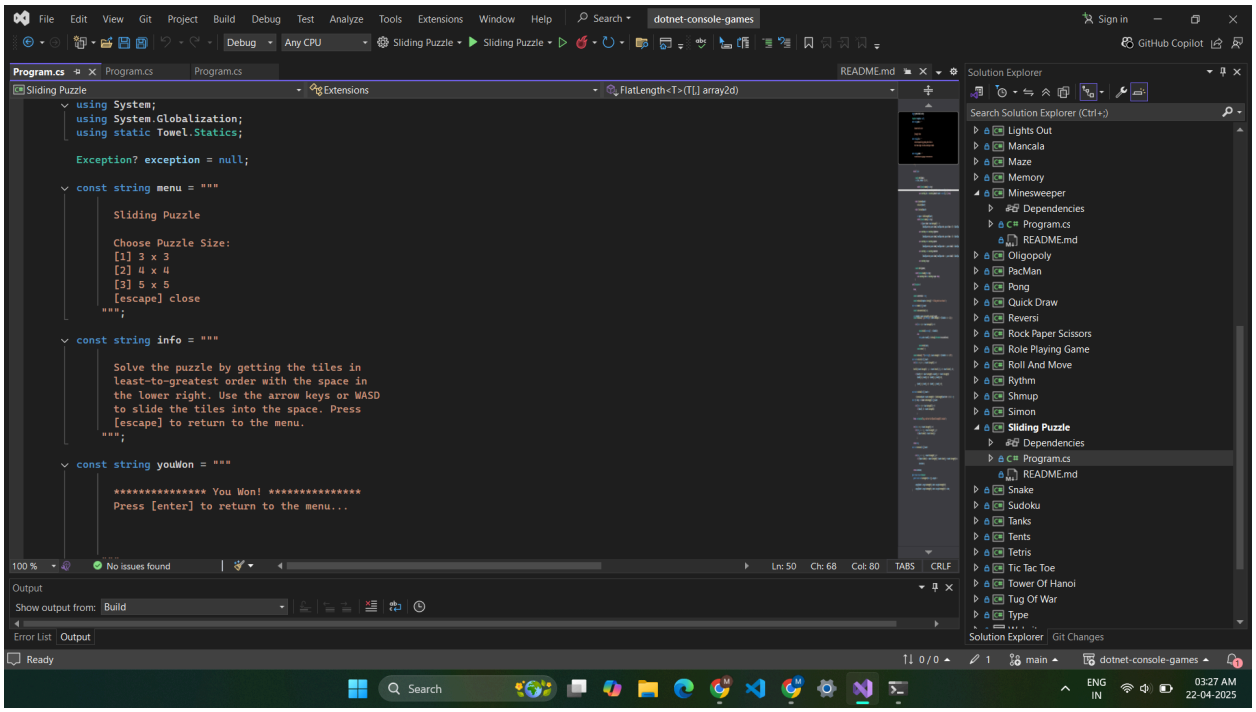


Sliding Puzzle

Task 1 :

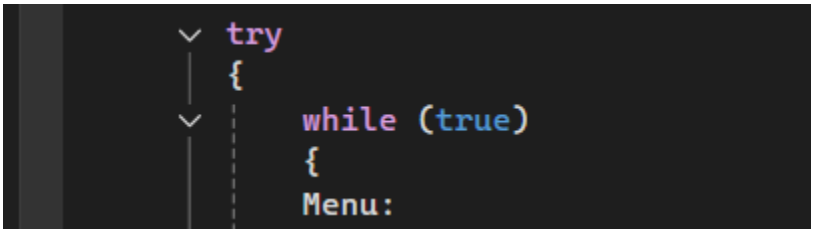
Open the Project in Visual Studio and Launch Visual Studio Open Existing Project

- Click File → Open → Project/Solution
- Navigate to the chosen game's folder (sliding puzzle) and select the .csproj file.



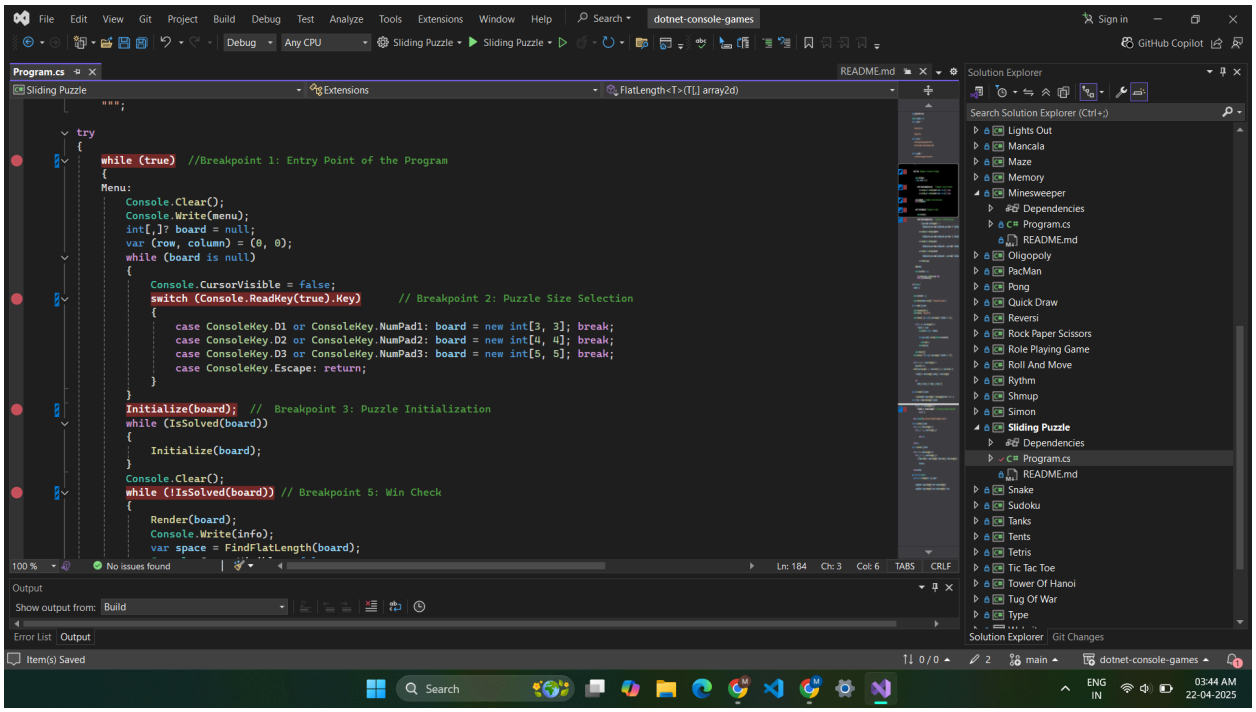
Task 2: Identify the Entry Point

The entry point of the given C# code is the try block inside the Main method, although the Main method itself isn't explicitly written here.



Task 3: Insert Breakpoints + Run in Debug Mode

- In the code view (like Program.cs), click left of the line numbers to place red breakpoints.



Task 4 : Hunt for a total of five bugs

- Bug 1: Incorrect FlatGet and FlatSet Logic
  - Location

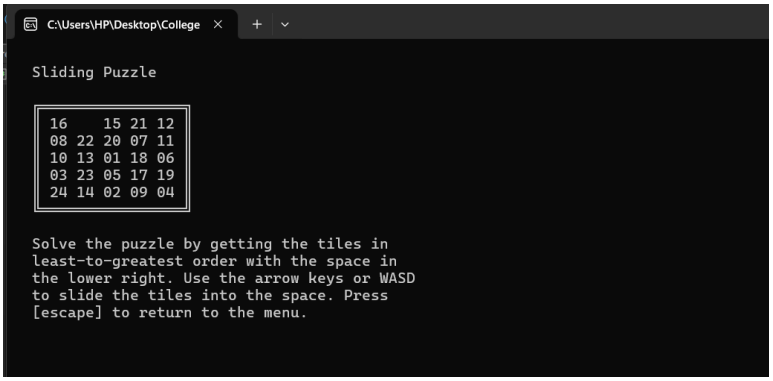
```
7 references
public static T FlatGet<T>(this T[,] array2d, int index) =>
    array2d[index / array2d.GetLength(0), index % array2d.GetLength(0)];
```

Issue :

Incorrect conversion from 1D index to 2D coordinates — index / array2d.GetLength(0) uses the row count instead of the correct column count.

Tiles are accessed incorrectly, causing unpredictable behavior in move logic and win checking.

Before the Fix

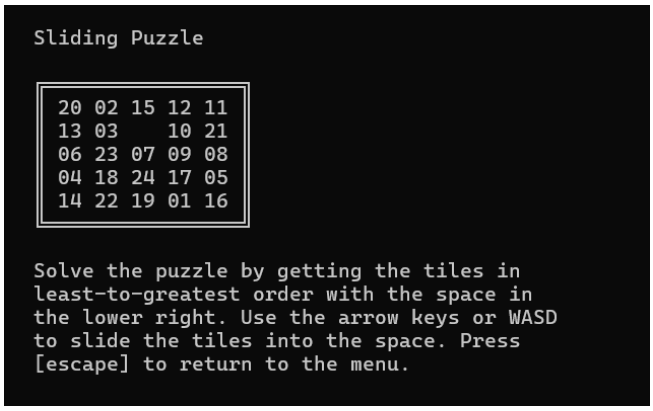


Corrected Code :

Use GetLength(1) (number of columns) instead

```
7 references
public static T FlatGet<T>(this T[,] array2d, int index) =>
    array2d[index / array2d.GetLength(1), index % array2d.GetLength(1)];
```

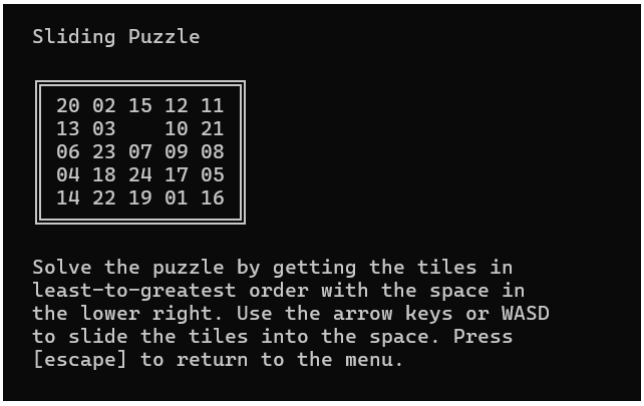
After the Fix



- Bug 2: IsSolved() – Incorrect Logic
  - IsSolved checks for inversion count rather than checking if tiles are in order.
  - Location : Below

```
2 references
static bool IsSolved(int[,] board)
{
    for (int i = 0; i < board.FlatLength(); i++)
    {
        for (int j = i + 1; j < board.FlatLength(); j++)
        {
            if (board.FlatGet(i) > board.FlatGet(j))
            {
                return false;
            }
        }
    }
    return true;
}
```

Before fixing the code



This fails when the board is not sorted but has no inversions, or vice versa.
Game thinks the board is solved when it's not (or never shows a win screen).

Corrected Code :

```
2 references
static bool IsSolved(int[,] board)
{
    for (int i = 0; i < board.FlatLength() - 1; i++)
    {
        if (board.FlatGet(i) != i + 1)
        {
            return false;
        }
    }
    return true;
}
```

- Bug 4: Inefficient Puzzle Initialization
    - This code will keep calling Initialize() over and over until the board is not solved — but that check should happen inside Initialize(), not outside. Otherwise, it can become inefficient and repetitive.
    - Location

```
Initialize(board);  //  Breakpoint 3: Puzzle Initialization
while (IsSolved(board))
{
    Initialize(board);
}
```

Issue:
The board may be repeatedly initialized with a solved configuration, leading to inefficient loops.
Performance degradation and potential infinite loop if logic is flawed.

Fix : Modify the Initialize(int[,] board) method to loop until the board is shuffled and unsolved. Replace the old Initialize method:

```
static void Initialize(int[,] board)
{
    do
    {
        for (int i = 0, k = 1; i < board.FlatLength(); i++)
        {
            board.FlatSet(i, k++);
        }

        Shuffle(0, board.FlatLength() - 1, i => board.FlatGet(i), (i, v) => board.FlatSet(i, v));

        if (!IsSolvable(board))
        {
            if (board[0, 0] != board.FlatLength() && board[0, 1] != board.FlatLength())
            {
                (board[0, 0], board[0, 1]) = (board[0, 1], board[0, 0]);
            }
            else
            {
                (board[1, 0], board[1, 1]) = (board[1, 1], board[1, 0]);
            }
        }
    } while (IsSolved(board));
}
```

Bug 5: Exception Thrown When Empty Tile Not Found

Current Code: In FindFlatLength():

```
        {
            if (board[r, c] == board.FlatLength())  // Find the Empty Space (Edge case bug-prone)
            {
```

Problem:

The empty tile is represented by 0, not FlatLength(). Looking for FlatLength() will never match, and causes the exception to be thrown: throw new Exception("bug. could not find (board.FlatLength()) in board");

Fix for Bug 5: Search for 0 instead

```
2 references
static (int Row, int Column) FindFlatLength(int[,] board)
{
    for (int r = 0; r < board.GetLength(0); r++)
    {
        for (int c = 0; c < board.GetLength(1); c++)
        {
            if (board[r, c] == 0)  // Find the Empty Space (Edge case bug-prone)
            {
                return (r, c);
            }
        }
    }
    throw new Exception("bug. could not find (board.FlatLength()) in board");
}
```

Challenges Faced

1. Handling Board Initialization
    - Initializing the sliding puzzle with a solvable and non-solved state was tricky.
    - Random shuffles often resulted in either an unsolvable board or a pre-solved one, leading to inefficiencies.

2. Bug Detection and Fixing
    - Some bugs like the one where the empty tile (value 0) wasn't correctly identified caused unexpected crashes.
    - Logical errors in conditions (like checking for FlatLength() instead of 0) required careful debugging.

3. Implementing Movement Logic
    - In both Snake and Sliding puzzles, accurately implementing tile movement, collision detection (in Snake), and tile swap logic (in Sliding Puzzle) involved handling edge cases like borders, corners, and invalid moves.

4. Ensuring Game Responsiveness
    - Ensuring real-time responsiveness of the Snake game (like snake speed, key press input delay, etc.) and maintaining a smooth visual flow was challenging using basic console operations.

5. Avoiding Infinite Loops
    - Especially in the sliding puzzle, failure to properly validate the initial state led to inefficient or infinite loops during board generation.

Observations

1. Importance of Solvability Checks
    - In sliding puzzles, it's not just enough to shuffle the board randomly; the board must be solvable, which requires calculating inversion counts and position of the empty tile.

2. Zero (0) as a Special Marker
    - The empty tile represented as 0 in the board matrix needs to be treated carefully to avoid breaking the puzzle logic or falsely detecting a solved board.

3. Separation of Concerns Improves Clarity
    - Breaking down logic into smaller functions like Shuffle(), IsSolved(), IsSolvable(), etc., made it easier to test and debug parts independently.

4. User Experience Matters
    - Small improvements like clear visual layout, proper timing, and input handling significantly improved the playability of the Snake game.

Conclusion

This project provided a hands-on understanding of game logic implementation, array manipulation, and real-time input handling. Through debugging and refining the Snake and Sliding Puzzle games, I learned the importance of:

- Writing modular, maintainable code,
- Carefully testing edge cases and validation logic,
- Ensuring both functional correctness and user experience.

Overall, the assignment served as a valuable exercise in combining algorithmic thinking with user-centric design, offering practical exposure to bug fixing and performance improvement in interactive programs.

Assignment 12

Lab Topic: Event-driven Programming for Windows Forms Apps. in C#

Overview

Grasping the event-driven paradigm and knowing how control flows in response to occurrence of events created by the user or by a particular condition of the program under development.

Objectives:

- Understand the event-driven paradigm in C#.
- Implement user-defined events and event handlers.
- Explore real-time event handling in console and GUI-based applications.
- Use Windows Forms controls and timer components.

Environment Setup:

- Operating System: Windows 10/11
- IDE: Visual Studio 2022 (Community Edition)
- Framework: .NET 6.0+ (or latest stable version)

Tools and Versions Used:

- Visual Studio: v17.7.4
- .NET SDK: 6.0 or above
- Programming Language: C#

Tasks and Step to Perform :

Part A

Step 1: Launch Visual Studio

1. Open Visual Studio 2022.
2. Click on "Create a new project".

Step 2: Create Console App Project

1. Select "Console App (.NET)".
2. Click Next.
3. Name the project, e.g., AlarmConsoleApp.
4. Choose location and solution name.
5. Click Create.

Step 3: Set Up the Main Program

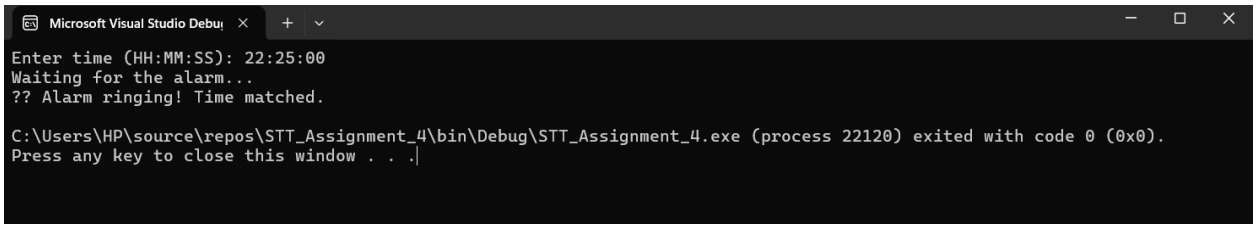
1. In Program.cs, delete the existing code.
2. Paste the following skeleton:

```
using System;
using System.Threading;

class AlarmSystem
{
    public delegate void AlarmEventHandler();
    public static event AlarmEventHandler raiseAlarm;

    static void Ring_alarm()
    {
        Console.WriteLine("Alarm ringing! Time matched.");
    }

    static void Main()
    {
        Console.Write("Enter time (HH:MM:SS): ");
        string input = Console.ReadLine();
        DateTime targetTime = DateTime.Parse(input);

        raiseAlarm += Ring_alarm;

        Console.WriteLine("Waiting for the alarm...");

        while (true)
        {
            if (DateTime.Now.ToString("HH:mm:ss") ==
targetTime.ToString("HH:mm:ss"))
            {
                raiseAlarm?.Invoke();
                break;
            }
            Thread.Sleep(1000);
        }
    }
}
```

Step 9: Run and Test the Application

1. Click the green Run button or press Ctrl + F5.
2. Enter time in HH:MM:SS format.
3. Wait until that time.
4. You should see:
   Alarm ringing! Time matched.

Output of the console (terminal)



PART 2

Step 1: Open Visual Studio

- Launch Visual Studio 2022.
- Click on "Create a new project".
- Choose "Windows Forms App (.NET)" and click Next.
- Name the project (AlarmClockWinForms) and click Create.

Step 2: Design the Form (Form1)

Steps to Add Form1 to Your Project

1. Right-click on the project name (AlarmClockWinForms) in the Solution Explorer.
2. Click Add > Windows Form...
3. In the dialog:
   - Name it Form1.cs
   - Click Add

After Adding:

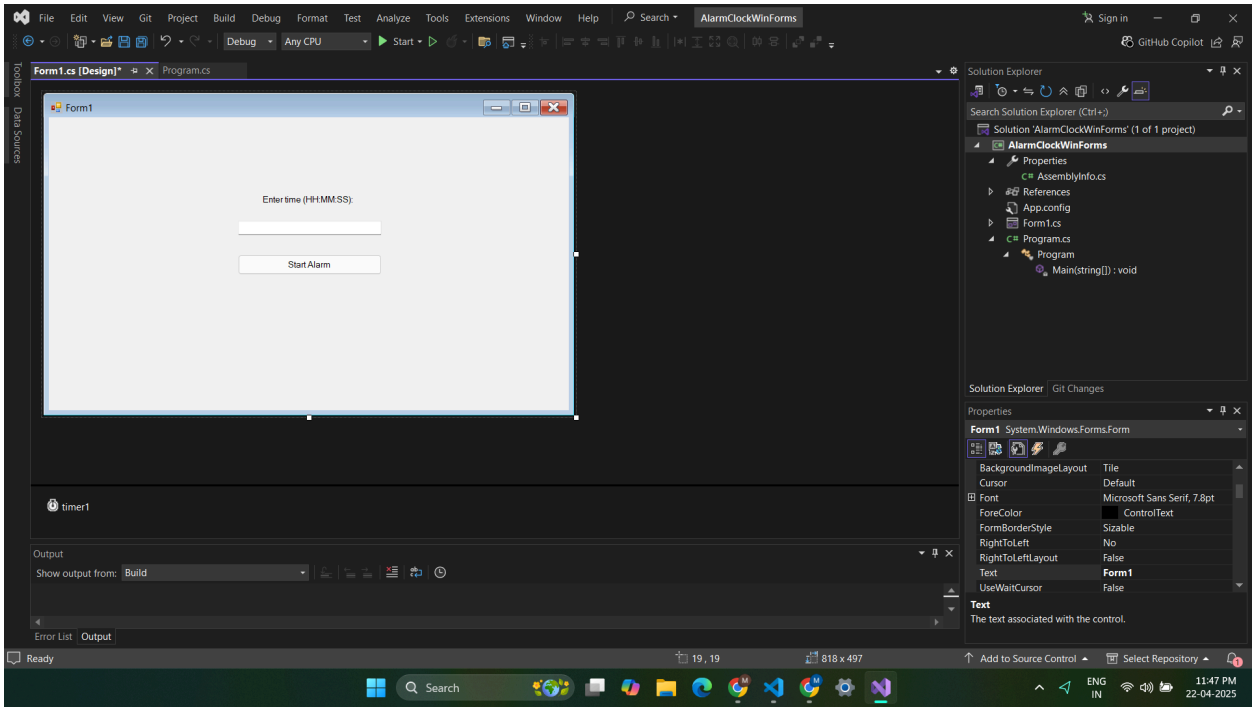
You'll see Form1.cs appear in Solution Explorer with a "Form1.cs [Design]" tab available when you double-click it.

Now you can:

- Drag and drop the controls from the Toolbox (Label, TextBox, Button, Timer)
- Set their names and properties as mentioned before.

In the Designer view (Form1.cs [Design]):

Control	Name	Properties/Settings
Label	lblTime	Text = "Enter time (HH:MM:SS):"
TextBox	txtTime	
Button	btnStart	Text = "Start Alarm"
Timer	timer1	Interval = 1000 (1 second)



Step 4: Implement Code in Form1.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace AlarmClockWinForms
{
    public partial class Form1 : Form
    {
        private DateTime targetTime; // Stores the alarm time
```

```
        private Random random = new Random(); // For background color
changes

        public Form1(){
            InitializeComponent();
        }

        private void btnStart_Click(object sender, EventArgs e)
        {
            // Try to parse time from the textbox input
            if (DateTime.TryParse(txtTime.Text, out targetTime))
            {
                // Set the full datetime with today's date
                targetTime = DateTime.Today.Add(targetTime.TimeOfDay);

                // Start the timer (tick every 1 second)
                timer1.Start();
            }
            else
            {
                MessageBox.Show("Invalid time format! Please enter in
HH:MM:SS format.", "Input Error",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            // Change the background color randomly every second
            this.BackColor = Color.FromArgb(
                random.Next(256),
                random.Next(256),
                random.Next(256)
            );

            // Check if current system time has reached the target
            if (DateTime.Now >= targetTime)
            {
                timer1.Stop(); // Stop changing colors
                MessageBox.Show("⏰ Time's up! Alarm triggered!", "Alarm",
```

```
                    MessageBoxButtons.OK, MessageBoxIcon.Information);
                }
            }
        }
}
```

Step 5: Wire Up Events

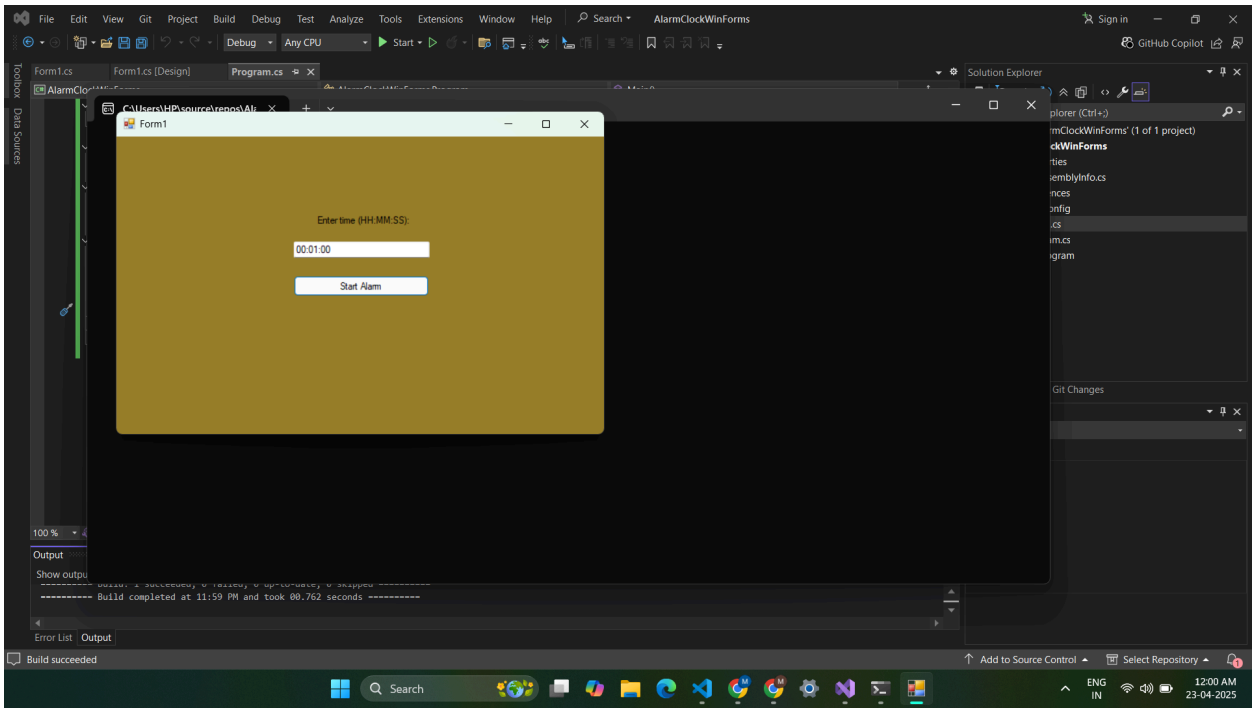
In Design View, select controls and go to Properties → Events (lightning icon):

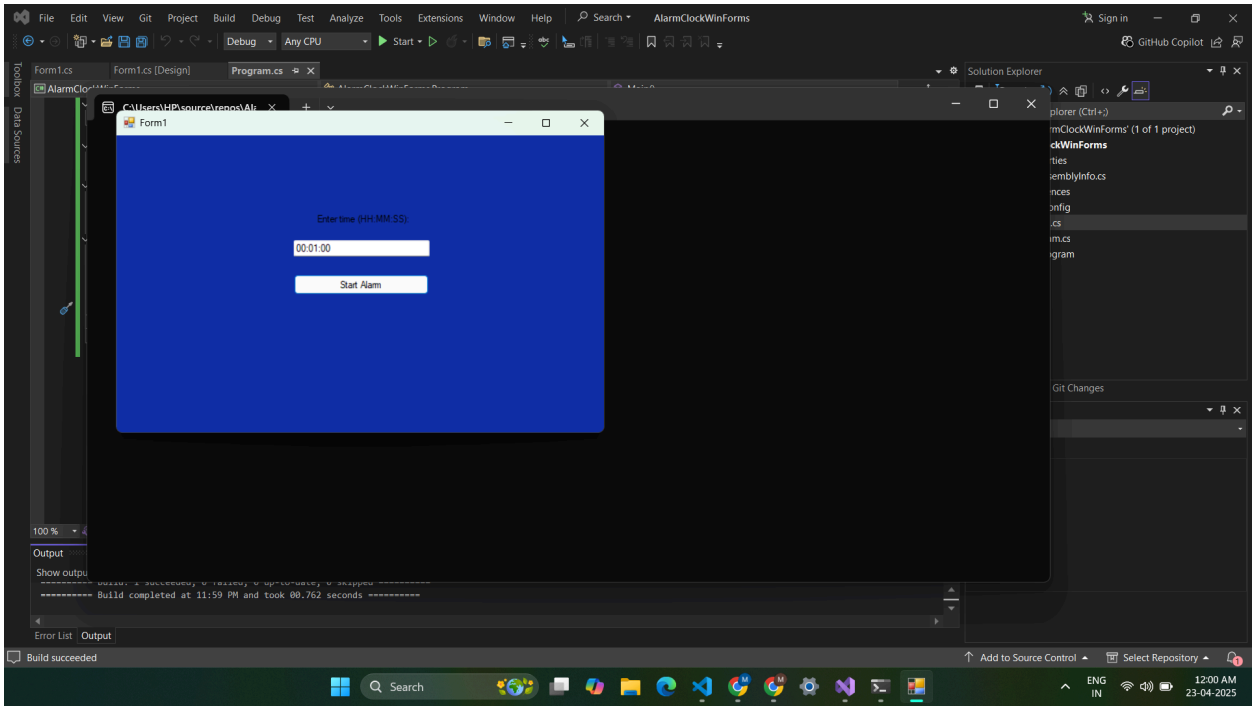
- Double-click btnStart to generate btnStart_Click.
- Select timer1 → Tick → assign timer1_Tick.

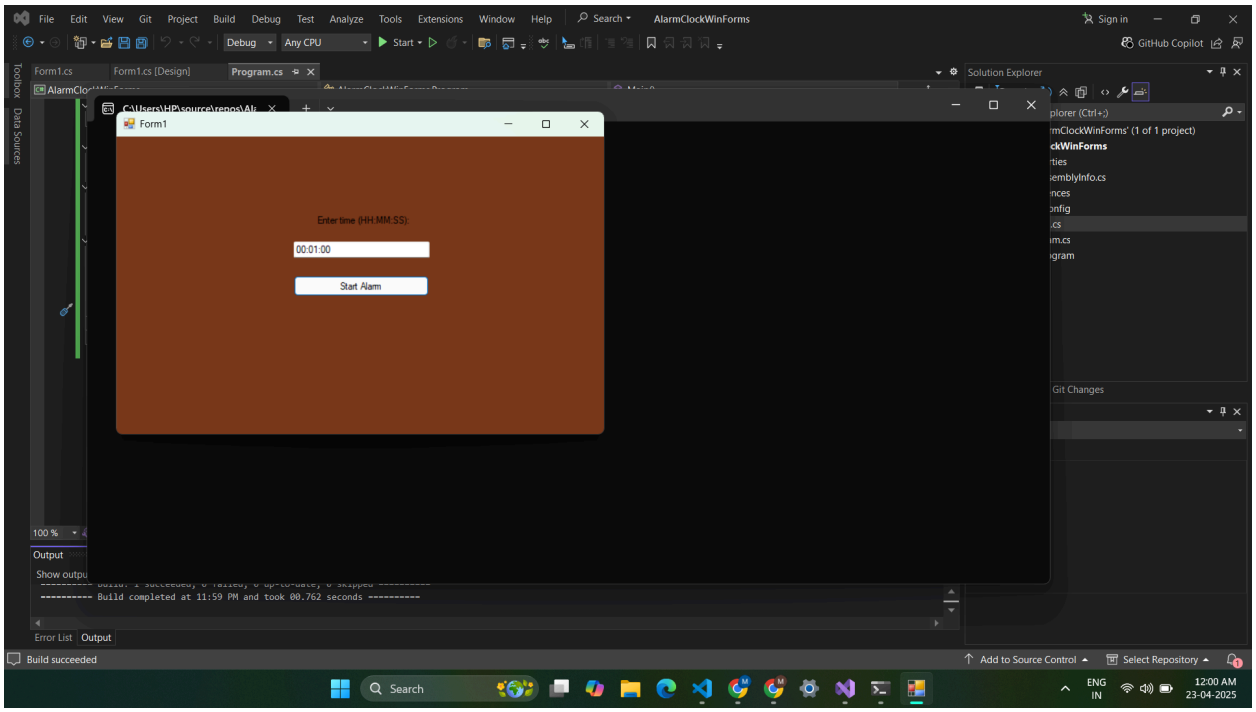
Step 6: Run the App

Click Start or press F5:

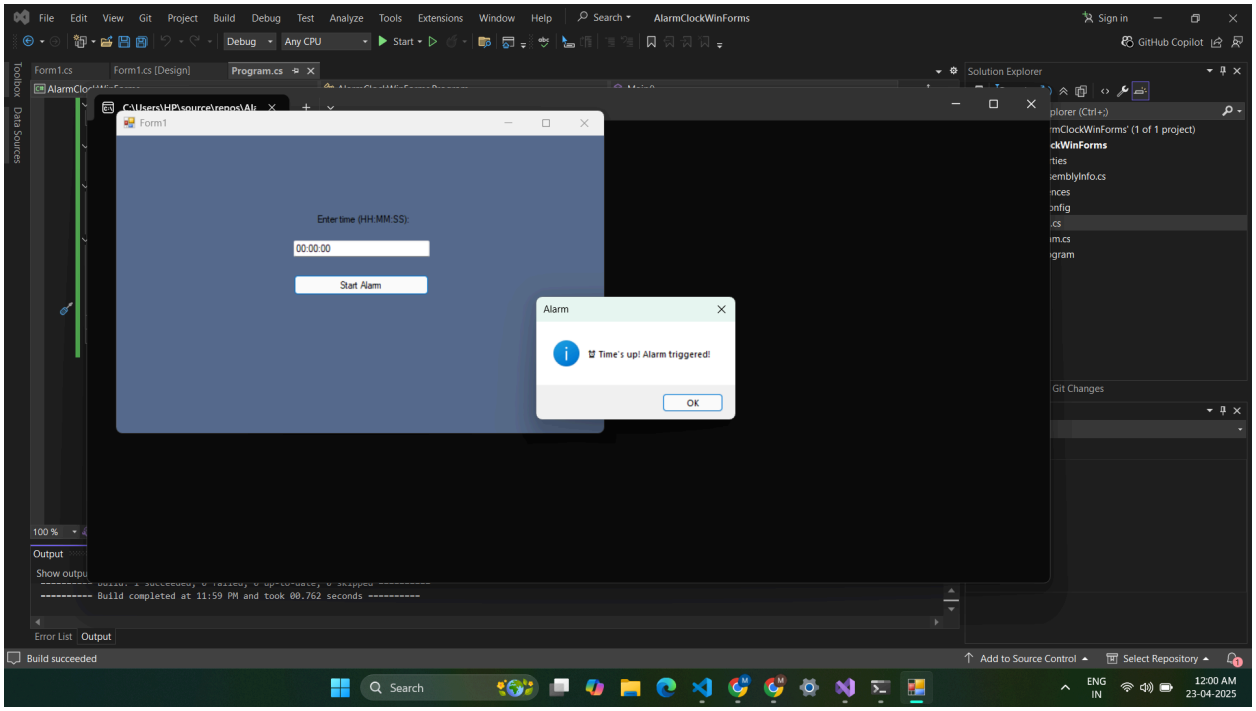
- Enter a valid future time in the textbox (e.g., 14:45:30).
- Click Start Alarm.
- The form's background will begin changing every second.
- When the system time reaches your set time, background stops, and a MessageBox appears.




File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
AlarmClockWinForms
Debug
Any CPU
Start
Form1.cs
Form1.cs [Design]
Program.cs
Solution Explorer
Form1
Enter time (HH:MM:SS):
00:01:00
Start Alarm
Output
Build completed at 11:59 PM and took 00.762 seconds
Error List
Output
Build succeeded
Add to Source Control
Select Repository
Search
ENG
IN
12:00 AM
23-04-2025


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
AlarmClockWinForms
Debug
Any CPU
Start
Form1.cs
Form1.cs [Design]
Program.cs
Solution Explorer
Form1
Enter time (HH:MM:SS):
00:01:00
Start Alarm
Output
Build completed at 11:59 PM and took 00.762 seconds
Error List
Output
Build succeeded
Add to Source Control
Select Repository
Search
ENG
IN
12:00 AM
23-04-2025

Times Up !!!!!



Output & Observations

- Real-time color-changing confirms the timer is working.
- Correct MessageBox confirms alarm logic.
- No delay/glitches in triggering events.

Comparison to Console App

Aspect	Console App	Windows Forms App
UI	Text-based	GUI-based
Input	Console.ReadLine	TextBox
Output	Console.WriteLine	Form background + MessageBox
Event	Custom event with delegate	Built-in Timer + Button Event
User Feedback	Less interactive	Visually engaging and dynamic

Lessons Learned

- Lesson: Importance of input validation and real-time UI updates.
- Takeaway: Windows Forms enhances event-driven logic with a more intuitive experience.

Summary

This lab helped in understanding:

- Core concepts of event-driven programming,
- Use of delegates and events in C# (Console),
- Timer controls and real-time visual updates in Windows Forms,
- Practical implementation of user-defined event triggers.