CS202

Lab 11: Analyzing C# Console Games for Bugs

Birudugadda Srivibhav 22110050

April 25, 2025

Introduction, Setup, and Tools

Overview

In this lab, I explored C# Console games using Visual Studio, focusing on understanding their control flow with the help of the Visual Studio Debugger. The main objective was to identify and fix bugs that could cause the game to crash.

Objectives

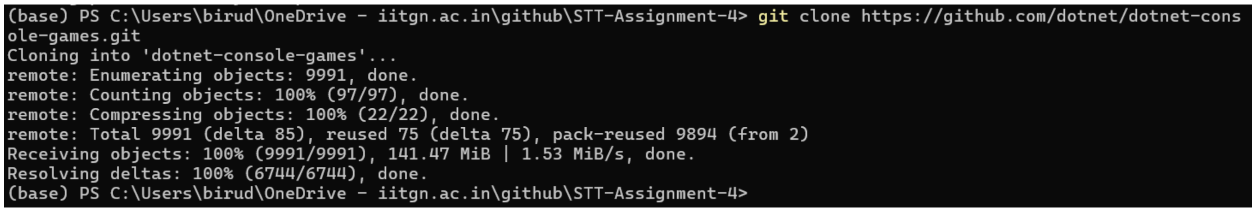
- Examine C# console game applications.
- Recognize the advantages of using the Visual Studio Debugger.
- Learn why, how, and when bugs arise in games, potentially causing crashes.

Environment Setup and Tools Used

- Operating System (Windows 11)
- Visual Studio 2022 (Community Edition), Visual Studio with .NET SDK
- C #

Methodology and Execution

Firstly, lets clone the dotnet-console-games repository:



```
(base) PS C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Assignment-4> git clone https://github.com/dotnet/dotnet-cons
ole-games.git
Cloning into 'dotnet-console-games'...
remote: Enumerating objects: 9991, done.
remote: Counting objects: 100% (97/97), done.
remote: Compressing objects: 100% (22/22), done.
remote: Total 9991 (delta 85), reused 75 (delta 75), pack-reused 9894 (from 2)
Receiving objects: 100% (9991/9991), 141.47 MiB | 1.53 MiB/s, done.
Resolving deltas: 100% (6744/6744), done.
(base) PS C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Assignment-4>
```

Now we will open the solution in the visual studio. Now we search for bugs in the games.

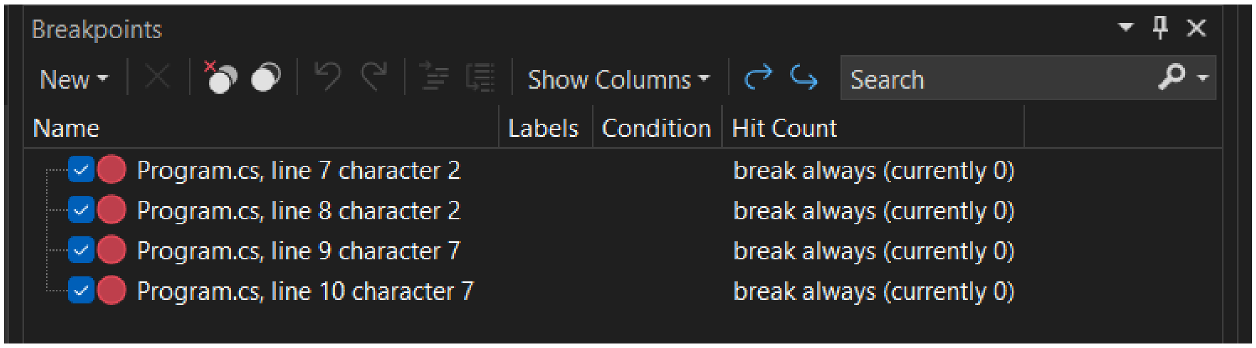
a) Bug-1

This bug was found in the **Guess a number** game. I inserted breakpoints at the following lines in the source code to trace the flow of execution:

- Line 7: bool valid = int.TryParse((Console.ReadLine() ?? "").Trim(), out int input);
- Line 8: if (!valid) Console.WriteLine("Invalid.");

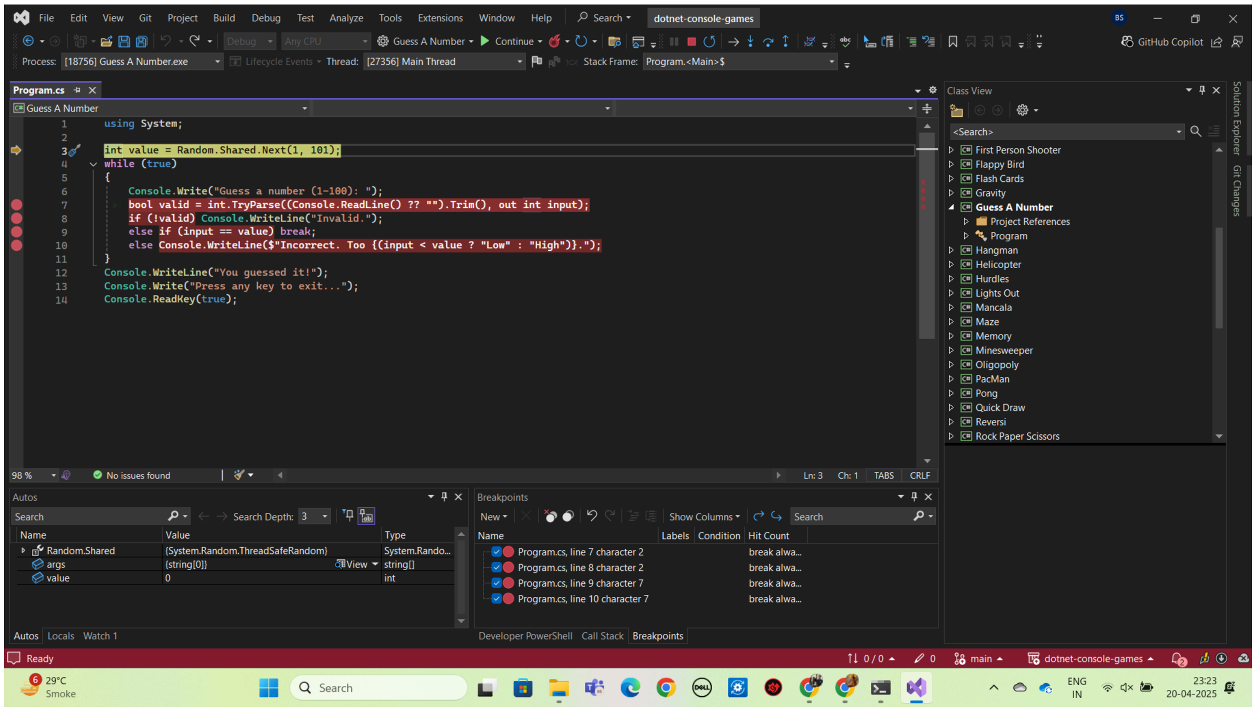
- Line 9: else if (input == value) break;
- Line 10: else Console.WriteLine(...)

Inserting breakpoints at these points pretty much cover what happens for each of the entered input. Now the bug is that **numbers like -1 or 0 are accepted and evaluated as guesses but these are not in range.**



Name	Labels	Condition	Hit Count
Program.cs, line 7 character 2			break always (currently 0)
Program.cs, line 8 character 2			break always (currently 0)
Program.cs, line 9 character 7			break always (currently 0)
Program.cs, line 10 character 7			break always (currently 0)

Now let us analyze the flow of the program when a negative number is taken as input.
Firstly, we start the execution in the Debug mode.

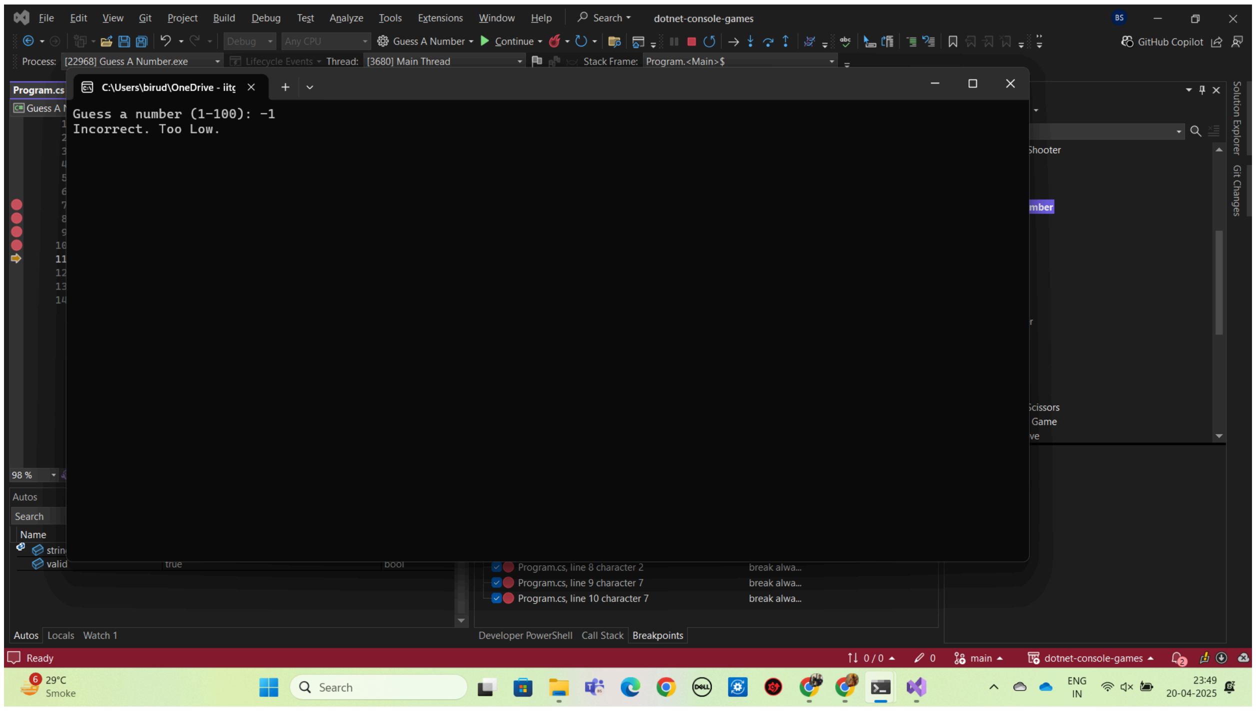


```
1   using System;
2
3   int value = Random.Shared.Next(1, 101);
4   while (true)
5   {
6       Console.Write("Guess a number (1-100): ");
7       bool valid = int.TryParse((Console.ReadLine() ?? "").Trim(), out int input);
8       if (!valid) Console.WriteLine("Invalid.");
9       else if (input == value) break;
10      else Console.WriteLine($"Incorrect. Too {(input < value ? "Low" : "High")}.");
11  }
12  Console.WriteLine("You guessed it!");
13  Console.Write("Press any key to exit...");
14  Console.ReadKey(true);
```

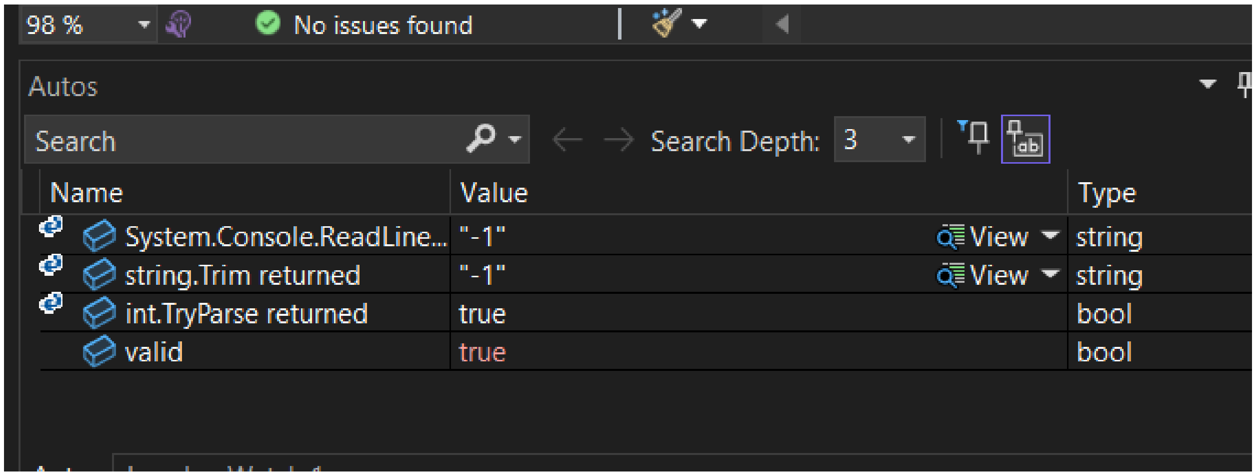
Name	Value	Type
Random.Shared	{System.Random.ThreadSafeRandom}	System.Rando...
args	{string[0]}	string[]
value	0	int

Name	Labels	Condition	Hit Count
Program.cs, line 7 character 2			break alwa...
Program.cs, line 8 character 2			break alwa...
Program.cs, line 9 character 7			break alwa...
Program.cs, line 10 character 7			break alwa...

When we reach line 7, the user input is asked, we type -1 in terminal as input.
But after entering the input, it goes to the next line of execution which is line 8 only because the **TryParse()** function returned true. This should not be happening. **TryParse()** only checks if input is a valid integer, not whether it's within range.



```
Guess a number (1-100): -1
Incorrect. Too Low.
```

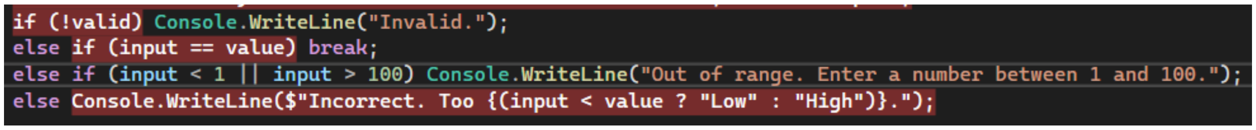


Autos

Name	Value	Type
System.Console.ReadLine...	"-1"	string
string.Trim returned	"-1"	string
int.TryParse returned	true	bool
valid	true	bool

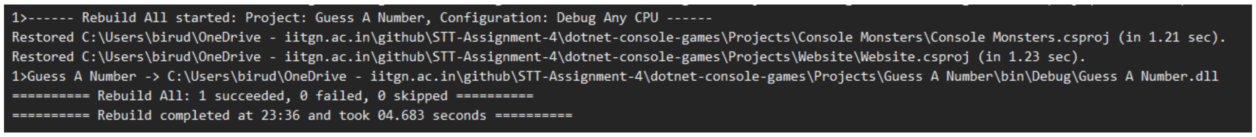
So now to correct this issue, we will enforce the code to check whether the user inputs numbers in the given range or not (which is 1-100 in this game).

We add this line to make it happen: **else if (input < 1 || input > 100) Console.WriteLine("Out of range. Enter a number between 1 and 100.");**



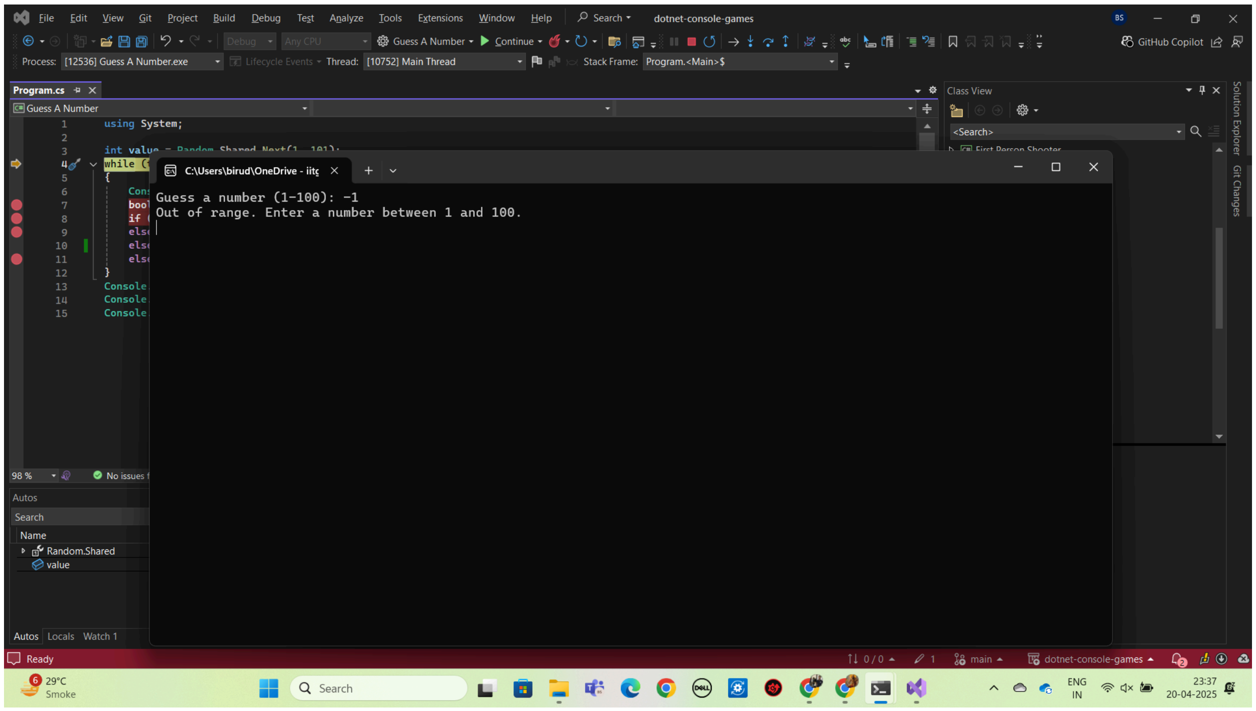
```
if (!valid) Console.WriteLine("Invalid.");
else if (input == value) break;
else if (input < 1 || input > 100) Console.WriteLine("Out of range. Enter a number between 1 and 100.");
else Console.WriteLine($"Incorrect. Too {(input < value ? "Low" : "High")}.");
```

Now the bug should be removed. We will rebuild this project to see if the bug has actually been corrected.



```
1>------ Rebuild All started: Project: Guess A Number, Configuration: Debug Any CPU ------
Restored C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Assignment-4\dotnet-console-games\Projects\Console Monsters\Console Monsters.csproj (in 1.21 sec).
Restored C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Assignment-4\dotnet-console-games\Projects\Website\Website.csproj (in 1.23 sec).
1>Guess A Number -> C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Assignment-4\dotnet-console-games\Projects\Guess A Number\bin\Debug\Guess A Number.dll
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
========== Rebuild completed at 23:36 and took 04.683 seconds ==========
```

Now rerun the program and enter a -ve number as input.



```
Guess a number (1-100): -1
Out of range. Enter a number between 1 and 100.
```

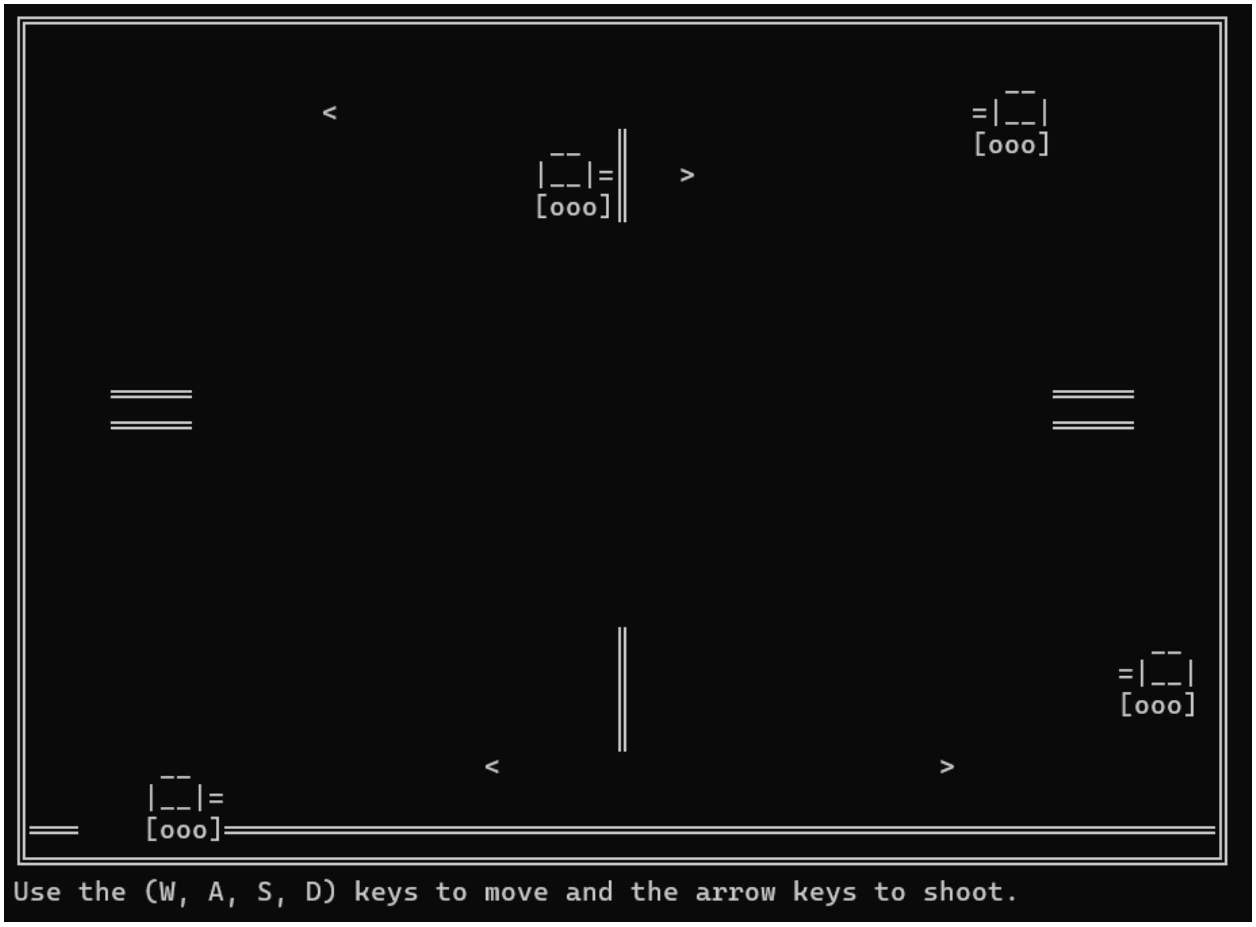
We can clearly see that this time, it asks to enter a number in the valid range.

b) Bug-2

This bug was found in the **Tanks** game. The bug occurs because when a tank is shooting, the bullet is spawned immediately next to the tank, but there's no collision check at the moment of spawning. If a tank is positioned right next to a wall and shoots, the bullet appears on the other side of the wall.

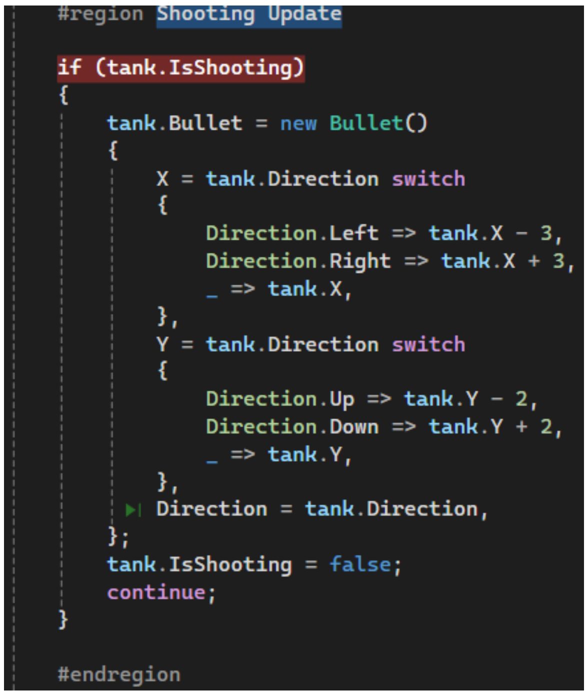
Breakpoints are created at the following location to detect the above situation:

- **Bullet Creation**: Place a breakpoint at the beginning of the tank shooting logic to observe the initial position where bullets are created.
- **Bullet Movement and Collision**: Place a breakpoint at the beginning of the bullet update logic to watch how bullets move and when collisions are detected.
- **Collision Check Function**: Place a breakpoint inside the `BulletCollisionCheck` method to see what conditions are being evaluated.



```
Use the (W, A, S, D) keys to move and the arrow keys to shoot.
```

I found that the issue is in the **Shooting Update** region of the code:



```csharp
#region Shooting Update

if (tank.IsShooting)
{
    tank.Bullet = new Bullet()
    {
        X = tank.Direction switch
        {
            Direction.Left => tank.X - 3,
            Direction.Right => tank.X + 3,
            _ => tank.X,
        },
        Y = tank.Direction switch
        {
            Direction.Up => tank.Y - 2,
            Direction.Down => tank.Y + 2,
            _ => tank.Y,
        },
        Direction = tank.Direction,
    };
    tank.IsShooting = false;
    continue;
}

#endregion
```

To solve the bug, we need to add a collision check immediately after creating the bullet. If the bullet is determined to be in a collision state at creation, we should not create it at all.

To implement this fix, we'll need to modify the `BulletCollisionCheck` method to be accessible from this section of the code. Currently, it's defined later in the code within the Bullet Updates section. We should move it to a location where it can be accessed from both sections.

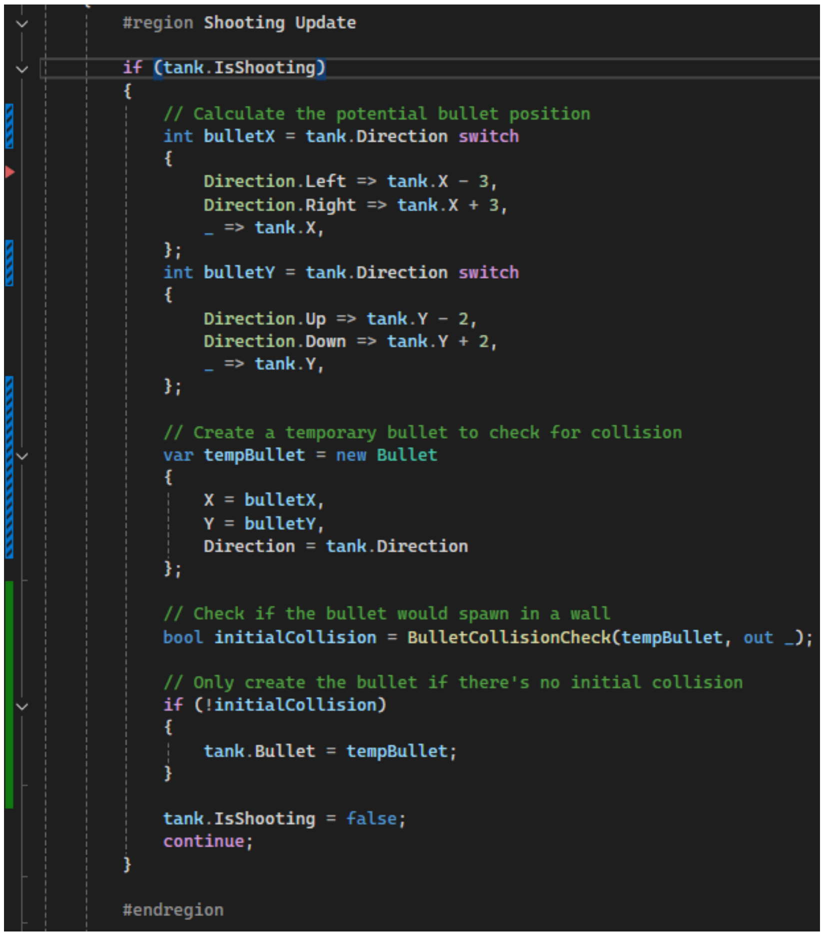
This solution ensures that bullets don't spawn inside walls, fixing the bug where bullets appear to go through walls when tanks are positioned next to them.

The fix works by:

1. Calculating the potential position of the bullet
2. Creating a temporary bullet object with that position
3. Checking if that position would result in a collision
4. Only creating the actual bullet if there's no collision

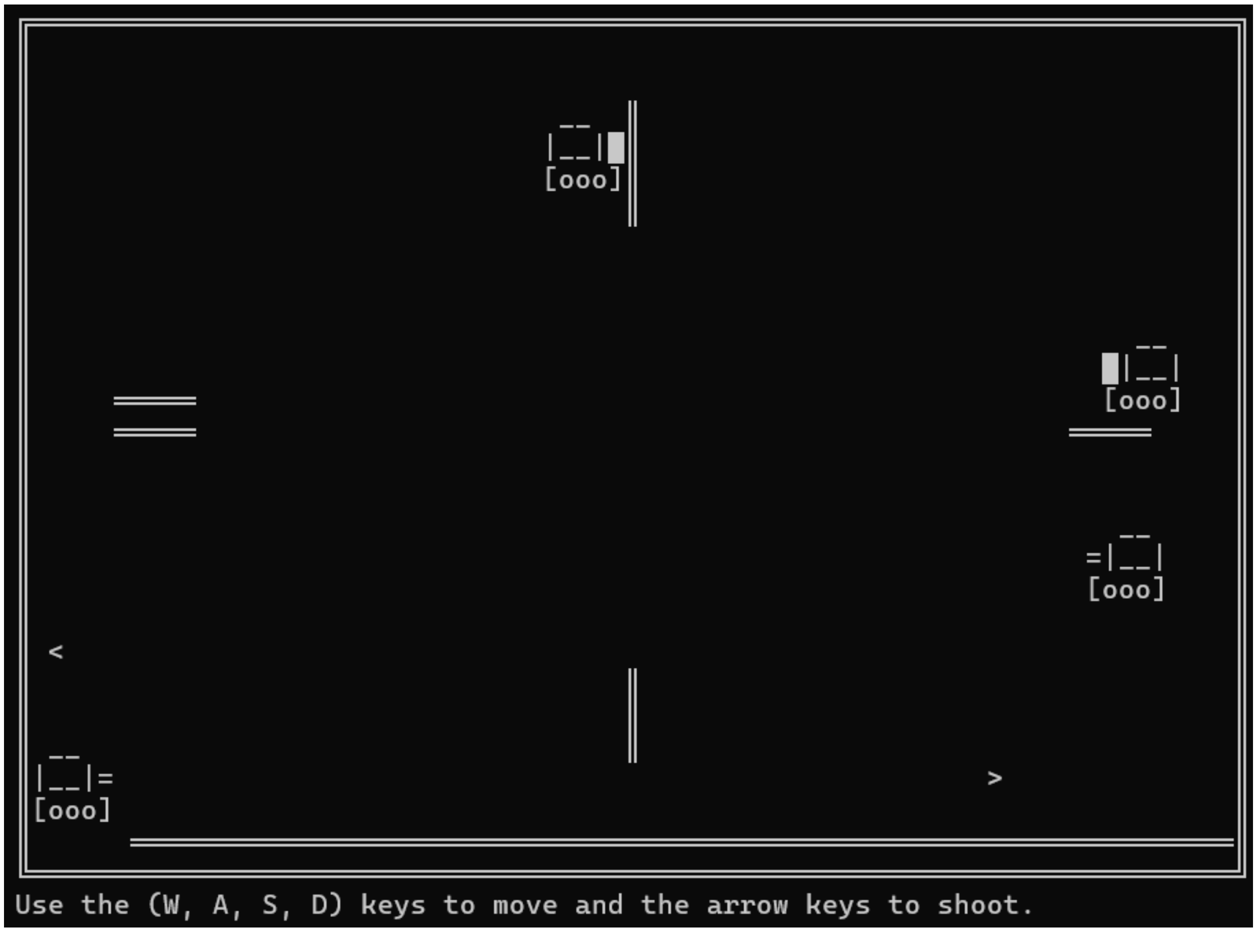
This way, when a tank is next to a wall and tries to shoot, nothing will happen because the bullet would spawn inside the wall, which is properly detected as a collision now.

The below is the fixed code:



```
#region Shooting Update

if (tank.IsShooting)
{
    // Calculate the potential bullet position
    int bulletX = tank.Direction switch
    {
        Direction.Left => tank.X - 3,
        Direction.Right => tank.X + 3,
        _ => tank.X,
    };
    int bulletY = tank.Direction switch
    {
        Direction.Up => tank.Y - 2,
        Direction.Down => tank.Y + 2,
        _ => tank.Y,
    };

    // Create a temporary bullet to check for collision
    var tempBullet = new Bullet
    {
        X = bulletX,
        Y = bulletY,
        Direction = tank.Direction
    };

    // Check if the bullet would spawn in a wall
    bool initialCollision = BulletCollisionCheck(tempBullet, out _);

    // Only create the bullet if there's no initial collision
    if (!initialCollision)
    {
        tank.Bullet = tempBullet;
    }

    tank.IsShooting = false;
    continue;
}

#endregion
```

We now rebuild the project and play the game again, the bug should now be fixed. The below screenshot proves it:



```
Use the (W, A, S, D) keys to move and the arrow keys to shoot.
```

Clearly the bullet collided with the wall this time and did not escape and hence the issue is fixed.

c) Bug-3

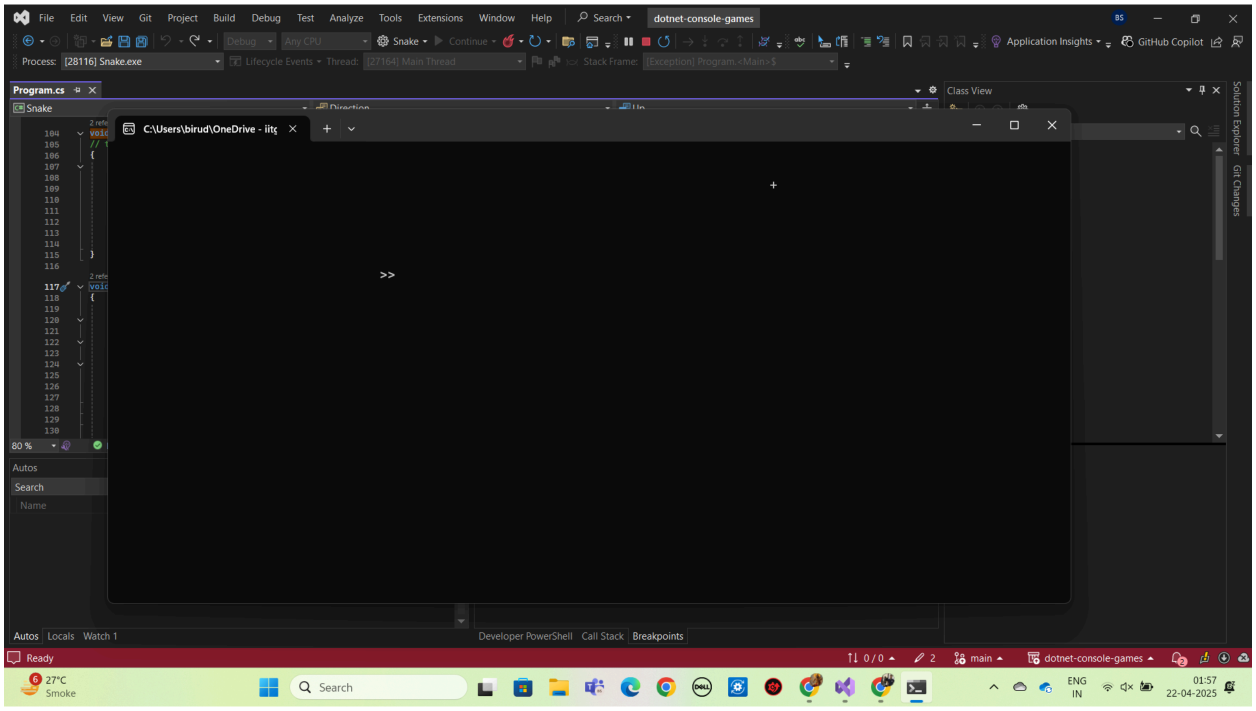
This bug was found in the **Snake** game. When the snake is moving in a particular direction (say, right), and the player quickly presses the **opposite direction** (left), the snake's head immediately turns around and moves **into its own body**, which results in a collision, even when the snake's length is just 2.

But logically, a real snake can't reverse instantly, it should be restricted from doing a 180° turn.

I placed breakpoints in **GetDirection()** function and the **Collision Check Block** to understand this issue more.

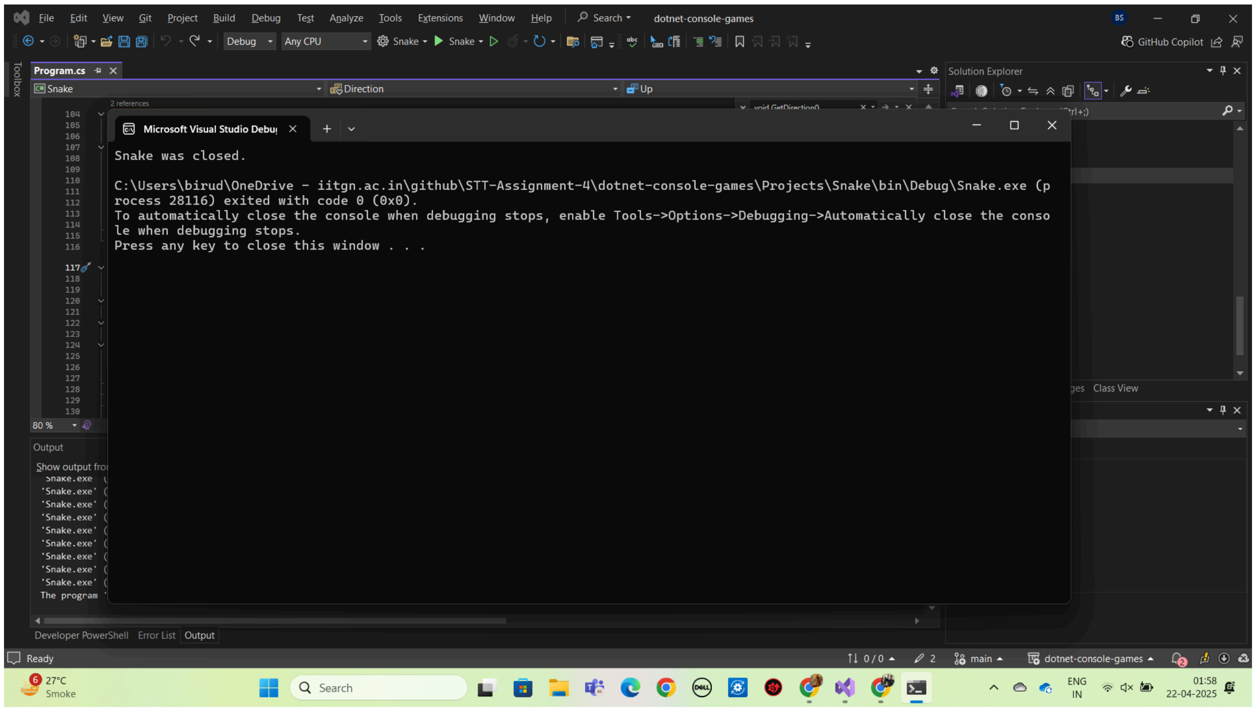
To fix this bug, We'll **ignore any direction input that tries to reverse the current direction** directly.

Inside the **GetDirection()** method, we add a check to ignore opposite direction inputs based on the current movement.



```
>>
```

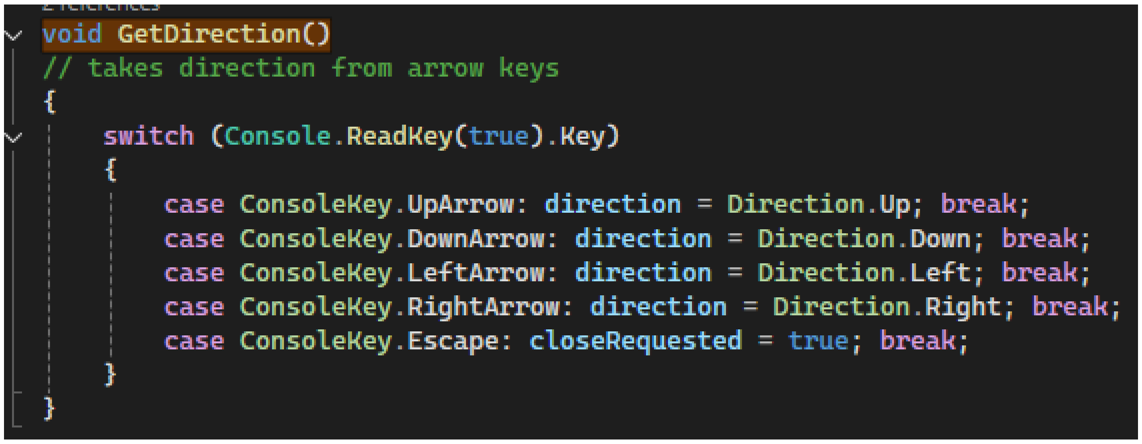
In the above image you can see that the snake is moving in right direction, as soon as i press the left arrow key, the game ends:



```
Snake was closed.

C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Assignment-4\dotnet-console-games\Projects\Snake\bin\Debug\Snake.exe (p
rocess 28116) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

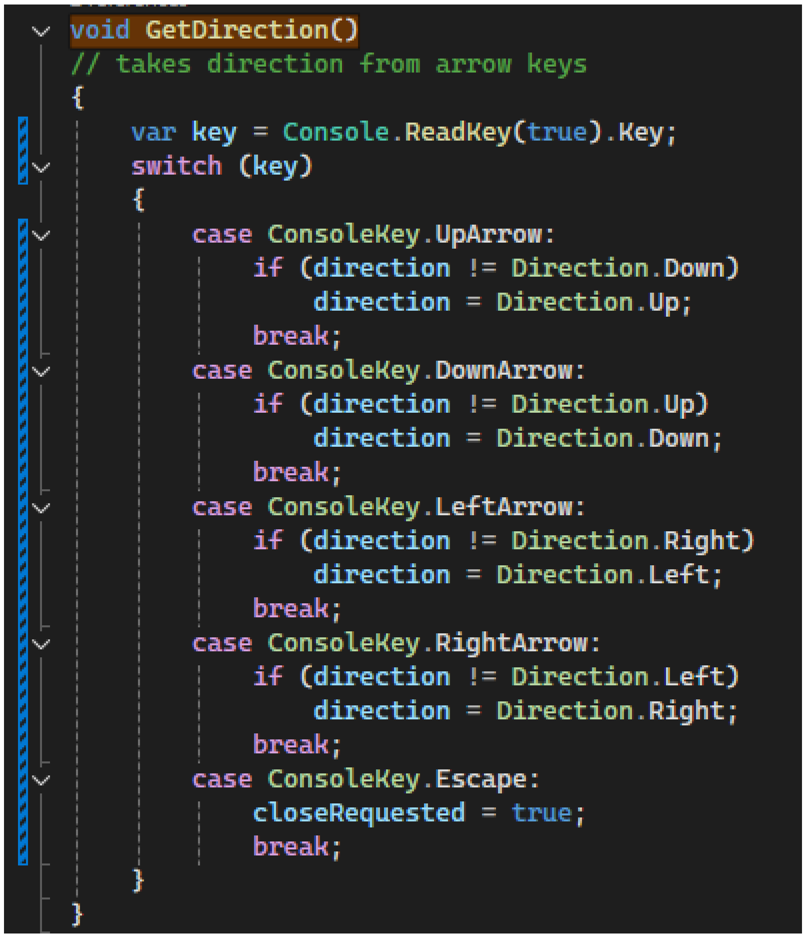
Now we will fix the above issue.

The below is the code snippet causing the issue where direction reversal is not checked:



```
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;
    }
}
```

Modified code:



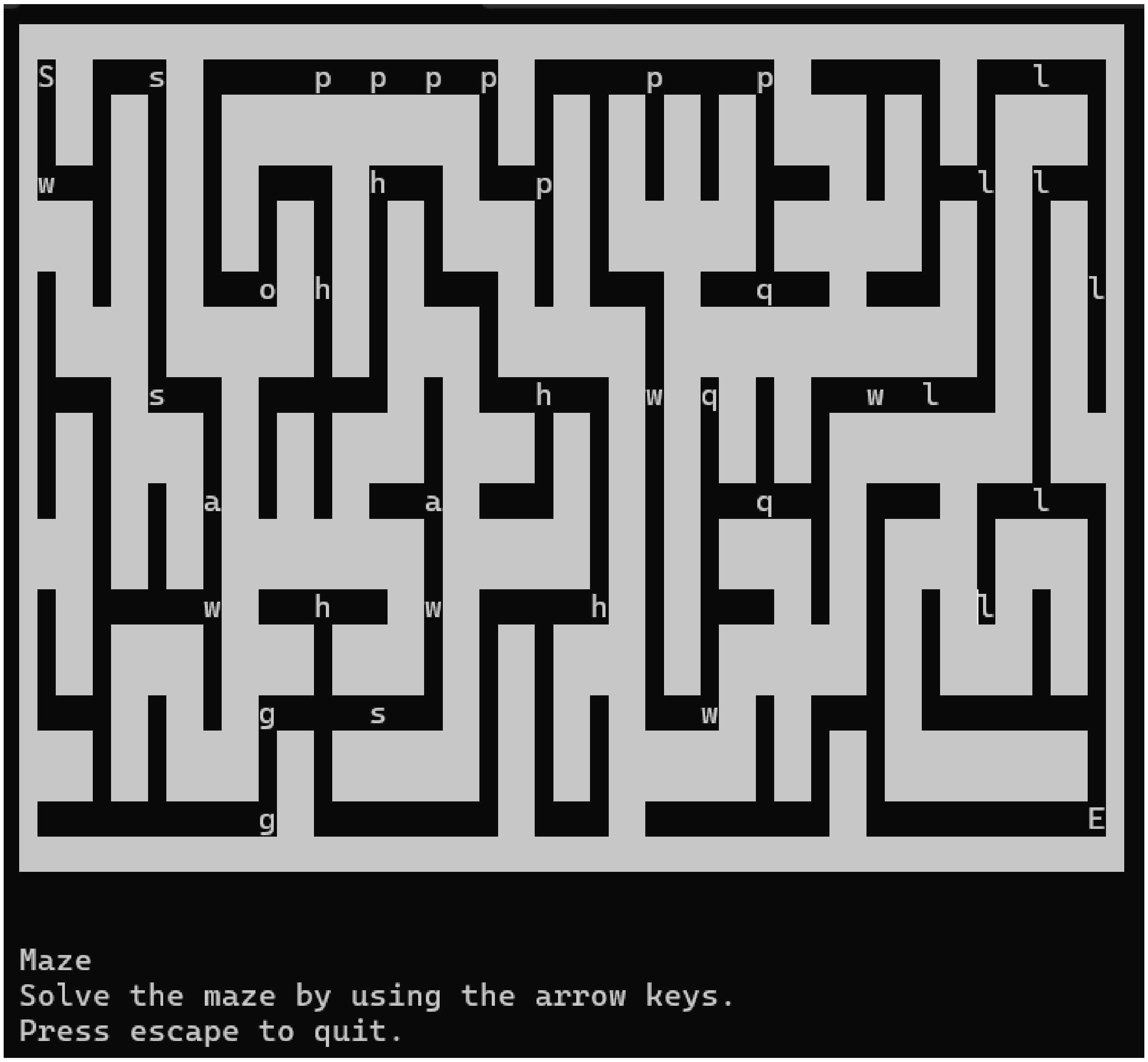
```
void GetDirection()
// takes direction from arrow keys
{
    var key = Console.ReadKey(true).Key;
    switch (key)
    {
        case ConsoleKey.UpArrow:
            if (direction != Direction.Down)
                direction = Direction.Up;
            break;
        case ConsoleKey.DownArrow:
 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

After modifying the code, the snake is not allowed to do a 180 degree turn.

d) Bug-4

This bug was found in the **Maze** game. In the original version of the maze game, player movement was restricted to arrow keys. However, a bug caused any other key presses (like alphabets, numbers, or special characters) to be printed directly onto the console screen. This disrupted the game interface and led to visual clutter, especially when players accidentally typed something other than the intended movement keys.

This bug was caused by **Console.ReadKey()** Method, there was no need of placing any breakpoints in this case.

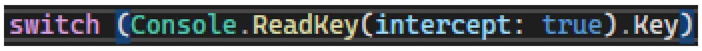


```
Maze
Solve the maze by using the arrow keys.
Press escape to quit.
```

Clearly you can see in the image my random key presses were also being printed in the screen making it clumsy.

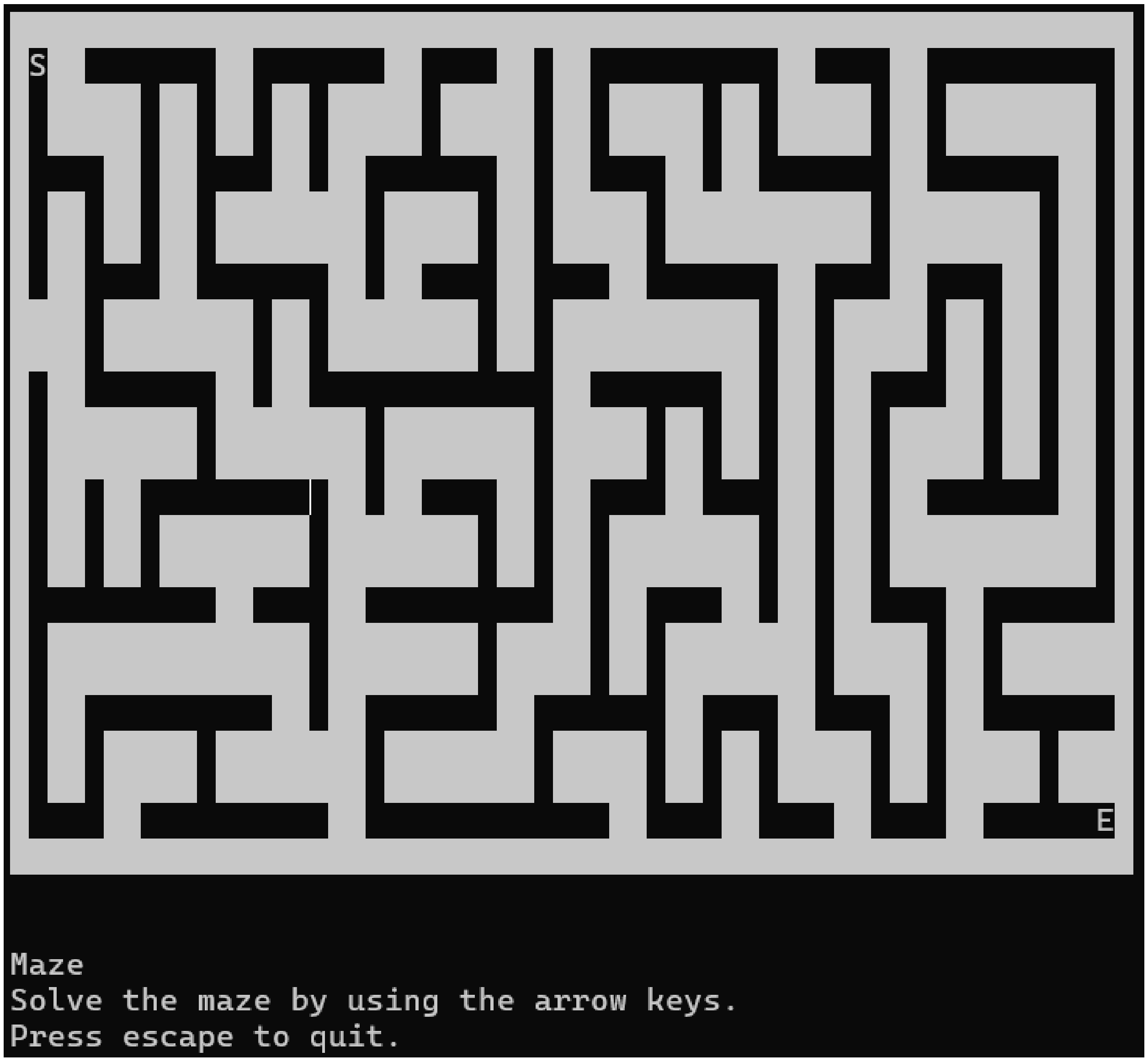
The `Console.ReadKey()` method by default **displays** the key that was pressed on the console. So, when a user pressed any key other than the arrow keys (which were handled), the key character still got printed on the screen — even though it wasn't being used for any logic.

The solution was to modify `Console.ReadKey()` to suppress the display of the key by passing the optional parameter `intercept: true`:



```
switch (Console.ReadKey(intercept: true).Key)
```

Now after the change, we will rebuild the project.

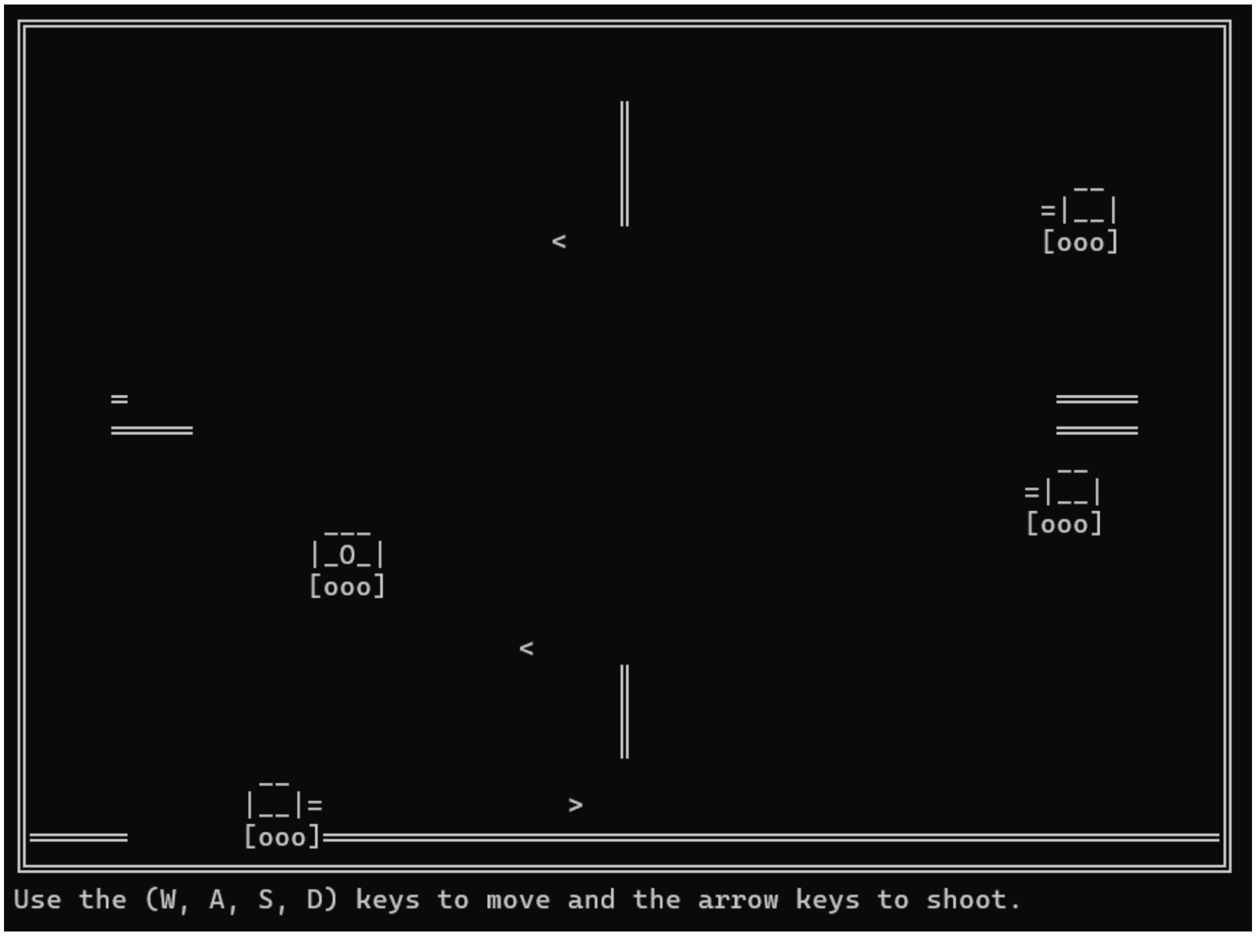


```
Maze
Solve the maze by using the arrow keys.
Press escape to quit.
```

Now, only the arrow keys (↑, ↓, ←, →) cause movement in the maze. All other key presses have no visual effect and do not clutter the console screen. This improves both the **usability** and the **visual clarity** of the game.

e) Bug-5

This bug was also found in the **Tanks** game. A visual bug was observed where certain walls would disappear when the player's tank or enemy tanks came into contact with them. This issue primarily affected the small vertical walls in the middle of the map and the thick horizontal wall at the bottom border. The root cause was traced back to incomplete collision detection in the **MoveCheck** method. The original implementation only checked for collisions with parts of the walls, meaning that when a tank (which is 5 characters wide and 3 lines tall) partially overlapped a wall, the game failed to register a collision, allowing the tank's rendering to overwrite the wall graphics. Additionally, the bottom border was not included in collision checks at all, so tanks could move right up against it, erasing parts of the wall.



```
Use the (W, A, S, D) keys to move and the arrow keys to shoot.
```

Clearly in the above image, we can see that the walls can be erased by the tanks simply by running through them.

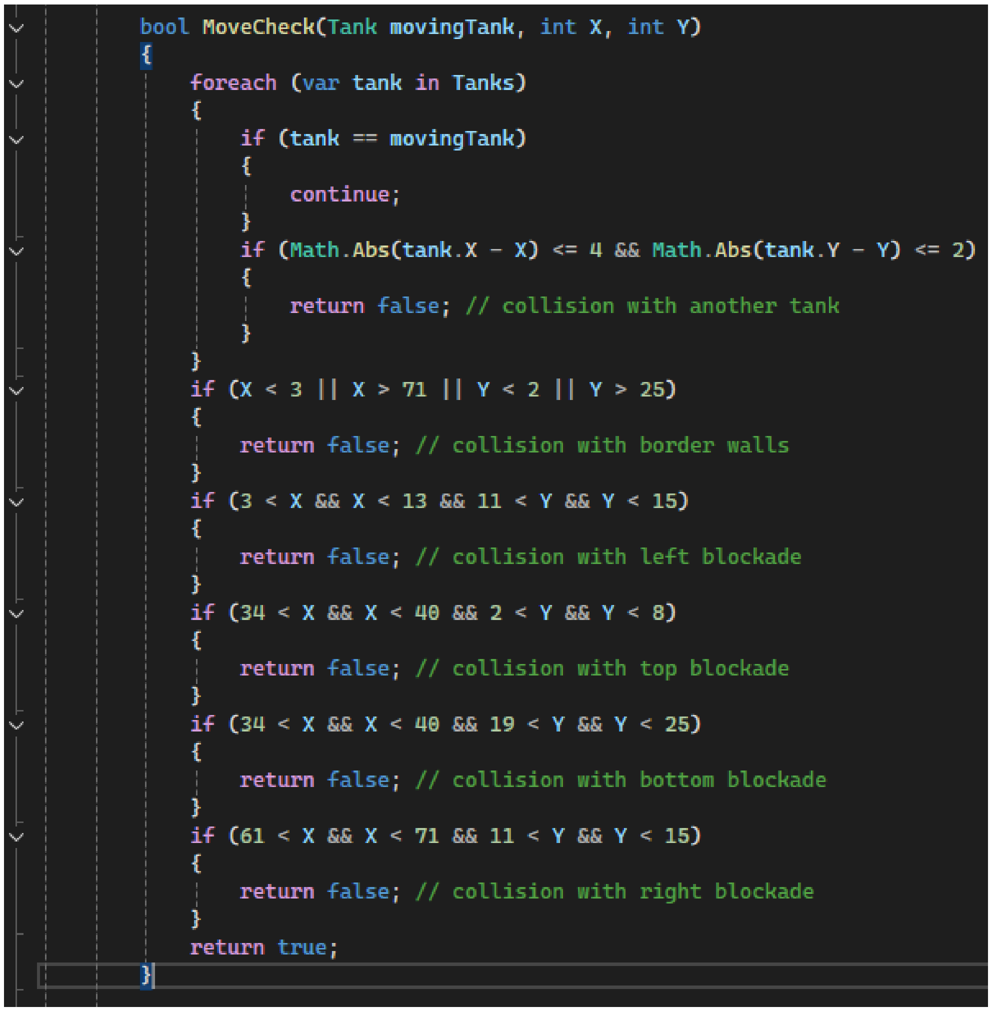
To fix the bug where walls disappear when tanks touch them, I strategically placed breakpoints in key functions to trace the issue. Here's how I approached it:

In the **MoveCheck()** Function, I added a breakpoint at the start of MoveCheck to log the tank's (X, Y) position whenever it moved. I noticed that when the tank approached the bottom border (Y == 25), the function wasn't returning false—meaning the game wasn't detecting the wall collision. I added an explicit check for Y == 25 to block tank movement at the border.

In the **TryMove()** Function. I set a breakpoint inside the Direction.Up case to see how the tank's position updated. The tank's old position was being cleared with spaces, but the map wasn't redrawn afterward, causing walls to "disappear." I ensured Render(Map) was called after movement to redraw walls properly.

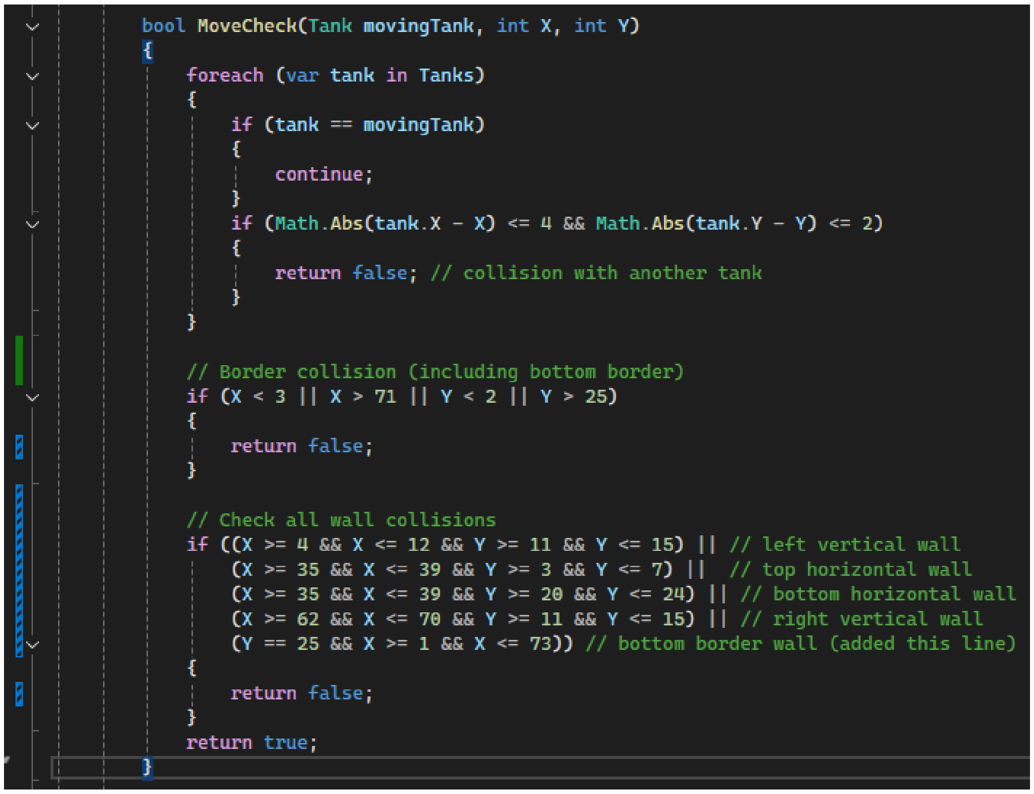
In the **Render()** Function, I added a breakpoint right before the tank's sprite was rendered (Console.SetCursorPosition(tank.X - 2, tank.Y - 1)). The tank was rendering over the walls instead of stopping at them. I confirmed MoveCheck() was blocking movement into walls, and the map was redrawn first.

The below is the original code which caused the issue:



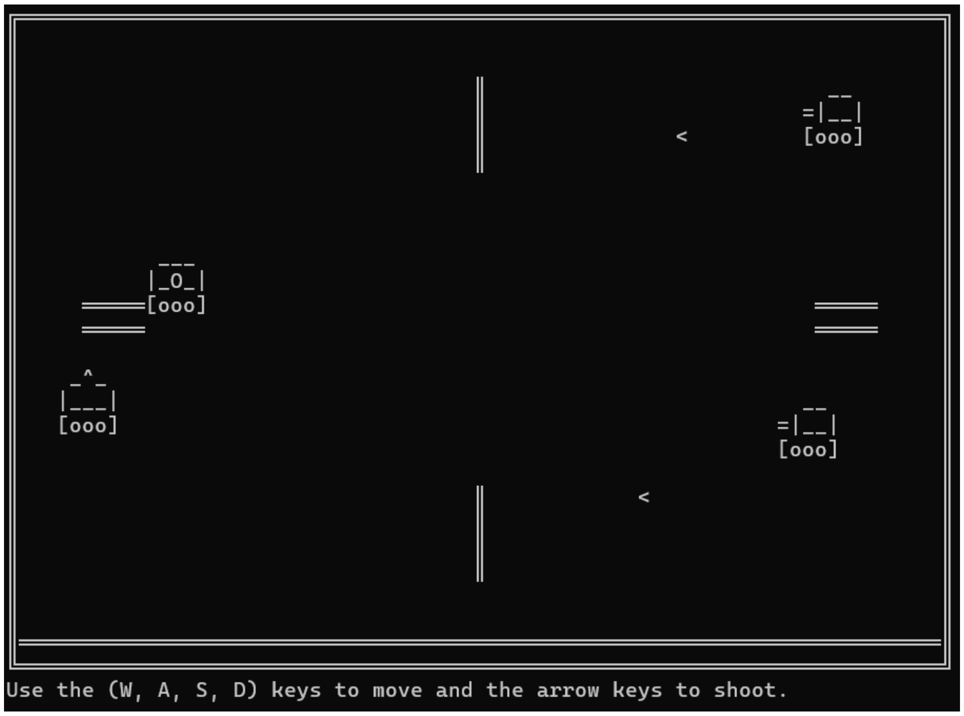
```
bool MoveCheck(Tank movingTank, int X, int Y)
{
    foreach (var tank in Tanks)
    {
        if (tank == movingTank)
        {
            continue;
        }
        if (Math.Abs(tank.X - X) <= 4 && Math.Abs(tank.Y - Y) <= 2)
        {
            return false; // collision with another tank
        }
    }
    if (X < 3 || X > 71 || Y < 2 || Y > 25)
    {
        return false; // collision with border walls
    }
    if (3 < X && X < 13 && 11 < Y && Y < 15)
    {
        return false; // collision with left blockade
    }
    if (34 < X && X < 40 && 2 < Y && Y < 8)
    {
        return false; // collision with top blockade
    }
    if (34 < X && X < 40 && 19 < Y && Y < 25)
    {
        return false; // collision with bottom blockade
    }
    if (61 < X && X < 71 && 11 < Y && Y < 15)
    {
        return false; // collision with right blockade
    }
    return true;
}
```

The below is the corrected code:



```
bool MoveCheck(Tank movingTank, int X, int Y)
{
    foreach (var tank in Tanks)
    {
        if (tank == movingTank)
        {
            continue;
        }
        if (Math.Abs(tank.X - X) <= 4 && Math.Abs(tank.Y - Y) <= 2)
        {
            return false; // collision with another tank
        }
    }

    // Border collision (including bottom border)
    if (X < 3 || X > 71 || Y < 2 || Y > 25)
    {
        return false;
    }

    // Check all wall collisions
    if ((X >= 4 && X <= 12 && Y >= 11 && Y <= 15) || // left vertical wall
        (X >= 35 && X <= 39 && Y >= 3 && Y <= 7) ||  // top horizontal wall
        (X >= 35 && X <= 39 && Y >= 20 && Y <= 24) || // bottom horizontal wall
        (X >= 62 && X <= 70 && Y >= 11 && Y <= 15) || // right vertical wall
        (Y == 25 && X >= 1 && X <= 73)) // bottom border wall (added this line)
    {
        return false;
    }
    return true;
}
```

The collision detection was expanded to cover the full height of vertical walls (from Y >= 11 to Y <= 15 instead of just Y == 13) and an explicit check was added for the bottom border (Y == 25).



```
Use the (W, A, S, D) keys to move and the arrow keys to shoot.
```

Now I rebuilt the game and played again to check if the issue was resolved. Clearly you can see in the above image that this time the walls are not disappearing upon contact with tanks and instead blocking the tanks properly.

The key takeaway from this issue is that collision detection must account for the full dimensions of moving objects—especially when they are larger than a single character.

Results and Analysis

To identify and resolve the bugs in the various games, proper debugging practices were employed using Visual Studio's integrated debugger. Breakpoints were strategically placed at suspected faulty lines, and the following debugging operations were used to trace the program's behavior:

Step In (F11): Used to dive into method calls and analyze logic line-by-line, especially helpful for tracing nested function behavior.

Step Over (F10): Used to execute a line of code without entering the internal method logic, useful for skipping library or non-critical methods.

Step Out (Shift + F11): Used to quickly return to the calling function after reviewing internal method behavior.

These techniques allowed us to pause the program's execution at critical points and observe the current state of variables, method calls, and control flow. This structured approach led to the successful identification and resolution of the following bugs:

Bug 1: Invalid Guess Range in Number Game

- **Game:** Guess a Number
- **Issue:** Inputs like `-1` and `0` were being accepted as valid guesses despite the game's number range starting from 1.
- **Fix:** Added a validation check to reject any guesses outside the acceptable range (1 to 100).

Bug 2: Bullet Spawns Through Wall in Tanks Game

- **Game:** Tanks
- **Issue:** When the tank was placed adjacent to a wall and fired a bullet, the bullet would appear on the other side of the wall. This happened because no collision check was performed at the bullet's spawn location.
- **Fix:** A check was added during bullet spawning to verify if the spawn location is a wall. If so, the bullet is not created.

Bug 3: Instant Snake Reversal Causes Self-Collision

- **Game:** Snake
- **Issue:** When the snake's direction was quickly reversed (e.g., from right to left), it immediately collided with itself, even when its length was only 2.
- **Fix:** Added logic to ignore direction changes that are directly opposite to the current movement.

Bug 4: Garbage Characters in Maze Game

- **Game:** Maze
- **Issue:** Non-arrow key inputs (like alphabets or special characters) were printed onto the console, cluttering the screen.
- **Fix:** Added a check to ignore all inputs except for arrow keys, preserving the visual integrity of the game.

Bug 5: Tank Overwrites Wall in Tanks Game

- **Game:** Tanks
- **Issue:** When the tank moved near certain walls, parts of the wall would disappear. This was caused by incomplete collision checks and the erasing of characters without checking if they were wall pieces.
- **Fix:** Enhanced the `MoveCheck` method to detect all tank-body collisions with walls, including borders.

Footer Questions and Answers

Question: If there is no Main() method in the program,where exactly is the entry point?
Answer: In modern versions of C#, starting from C# 9.0, it's possible to write programs without explicitly defining a `Main()` method. This is due to the introduction of **top-level statements**, a feature that simplifies the structure of small programs. In such cases, the compiler automatically treats the code written at the top level of a `.cs` file (that is, code not enclosed within any class or method) as the program's entry point. So, if there's no `Main()` method present, execution begins from the very first line of code written outside of any method or class. This approach helps reduce boilerplate code, especially in simple console applications—like many of those found in the `dotnet-console-games` repository.

Discussion and Conclusion

Challenges Faced

Many of the games lacked inline comments or external documentation, making it difficult to understand the original developer's intent. For instance, in the Tanks and Snake games, understanding the game loop and rendering logic required careful inspection.

Some bugs, like disappearing walls in the Tanks game or flickering text in the Maze game, were purely visual and not due to runtime errors. Debugging these required running the game repeatedly while observing the UI closely.

Some bugs were caused by incomplete or improperly implemented collision detection. Fixing these required not just code changes, but also understanding geometric relationships between game elements (e.g., tank size vs. wall size).

Lessons Learned

Effective Use of the Debugger Tools: Learning how to strategically use `Step In (F11)`, `Step Over (F10)`, and `Step Out (Shift+F11)` made the debugging process much smoother. These became essential in tracing control flow through large codebases.

Importance of Input Validation: The bug in the Guess a Number game highlighted how unchecked inputs—even seemingly harmless ones like 0 or -1—can impact game logic. Always validate user input before processing.

Attention to Detail in Visual Outputs: Visual bugs often don't throw exceptions, so noticing small glitches requires attention and patience. Fixing them deepened my understanding of how rendering and console output work in real time.

Summary

In this Lab Assignment, I worked on analyzing and debugging C# console games using Visual Studio. I explored a few games from the dotnet-console-games GitHub repo and used Visual Studio's debugging tools like **Step In (F11), Step Over (F10), and Step Out (Shift+F11)**, placing breakpoints at meaningful spots to trace the execution. I was able to find and fix five bugs across different games like Snake, Tanks, Maze, and Guess a Number. These bugs ranged from logical errors to visual glitches and improper input handling. Overall, this lab helped me get comfortable with using the Visual Studio Debugger and gave me a deeper understanding of how bugs occur and how to fix them effectively.

Note: All the code for the games analyzed and the fixes made can be found in the submitted GitHub repository link.

Resources

- Lecture 9 Slides
- Lecture 10 Slides
- https://github.com/dotnet/dotnet-console-games
- https://visualstudio.microsoft.com
- https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure
- https://learn.microsoft.com/en-us/visualstudio/debugger

CS202

Lab 12: Event-driven Programming for Windows Forms Apps in C#

Birudugadda Srivibhav 22110050

April 25, 2025

Introduction, Setup, and Tools

Overview

In this lab, I explored event-driven programming using C# Windows Forms Applications in Visual Studio. The main goal was to understand how the event-driven paradigm works and how the control flow of a program responds to various events, whether triggered by user interactions or specific states within the application.

Objectives

- Develop and analyze Windows Forms Applications using C# as the programming language.
- Understand the advantages of using the Visual Studio Toolbox for building interfaces.
- Grasp how, why, and when the control flow of an application changes based on user interactions or specific runtime conditions within the application.

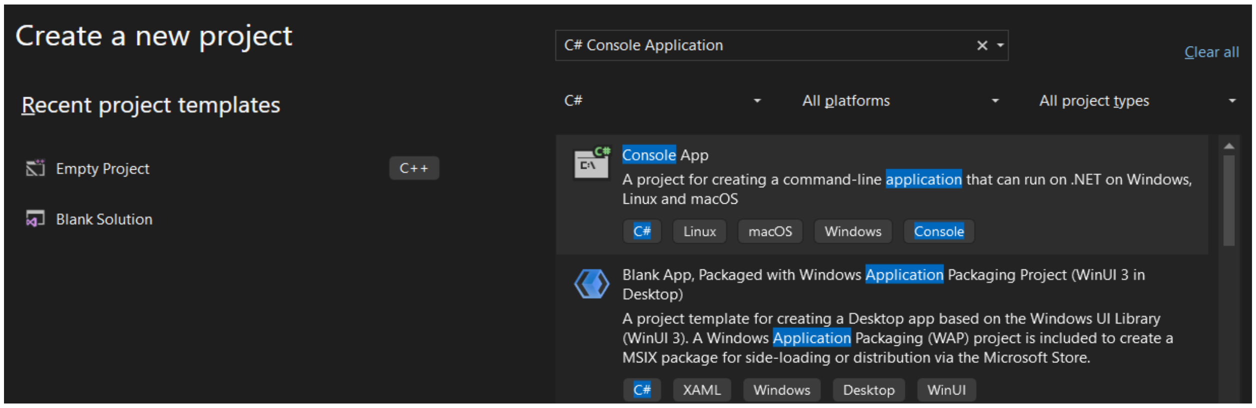
Environment Setup and Tools Used

- Operating System (Windows 11)
- Visual Studio 2022 (Community Edition), Visual Studio with .NET SDK
- C #

Methodology and Execution

a) Task 1

I have opened the Visual Studio and created a new C# Console application:


Create a new project

C# Console Application

Clear all

Recent project templates

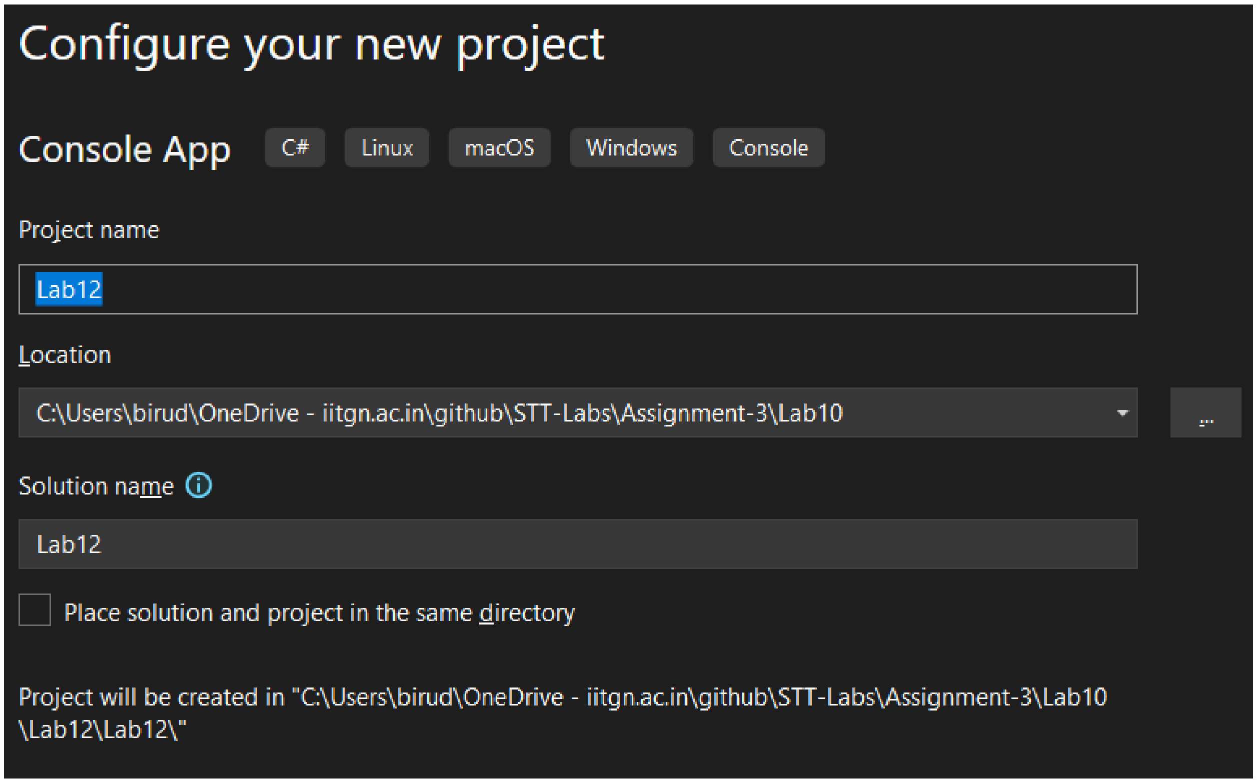
Empty Project C++

Blank Solution

C#　All platforms　All project types

Console App
A project for creating a command-line application that can run on .NET on Windows, Linux and macOS
C# Linux macOS Windows Console

Blank App, Packaged with Windows Application Packaging Project (WinUI 3 in Desktop)
A project template for creating a Desktop app based on the Windows UI Library (WinUI 3). A Windows Application Packaging (WAP) project is included to create a MSIX package for side-loading or distribution via the Microsoft Store.
C# XAML Windows Desktop WinUI



Configure your new project

Console App C# Linux macOS Windows Console

Project name

Lab12

Location

C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Labs\Assignment-3\Lab10

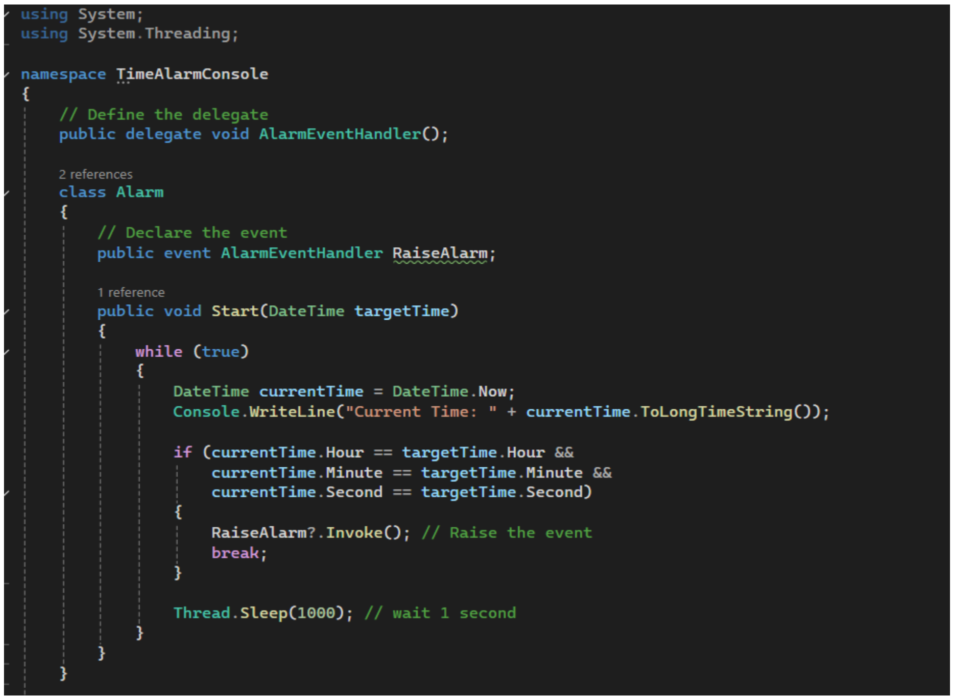
Solution name

Lab12

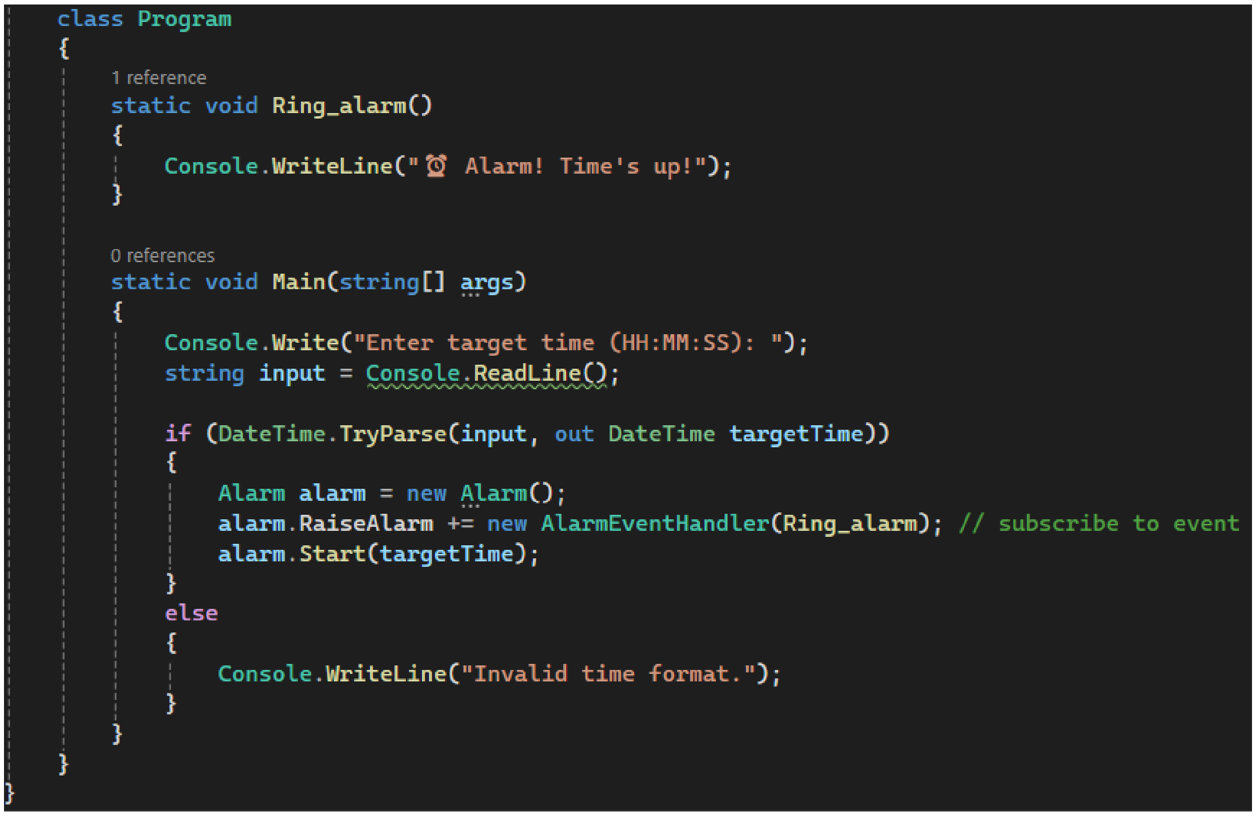
☐ Place solution and project in the same directory

Project will be created in "C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Labs\Assignment-3\Lab10\Lab12\Lab12\"

In the first part of the lab, I developed a console application in C#. The application prompts the user to input a target time in HH:MM:SS format. It then continuously checks the system time and compares it with the user-supplied time. When both times match, a custom event called RaiseAlarm is triggered, which in turn calls a subscriber method Ring_alarm() to print a message on the console.

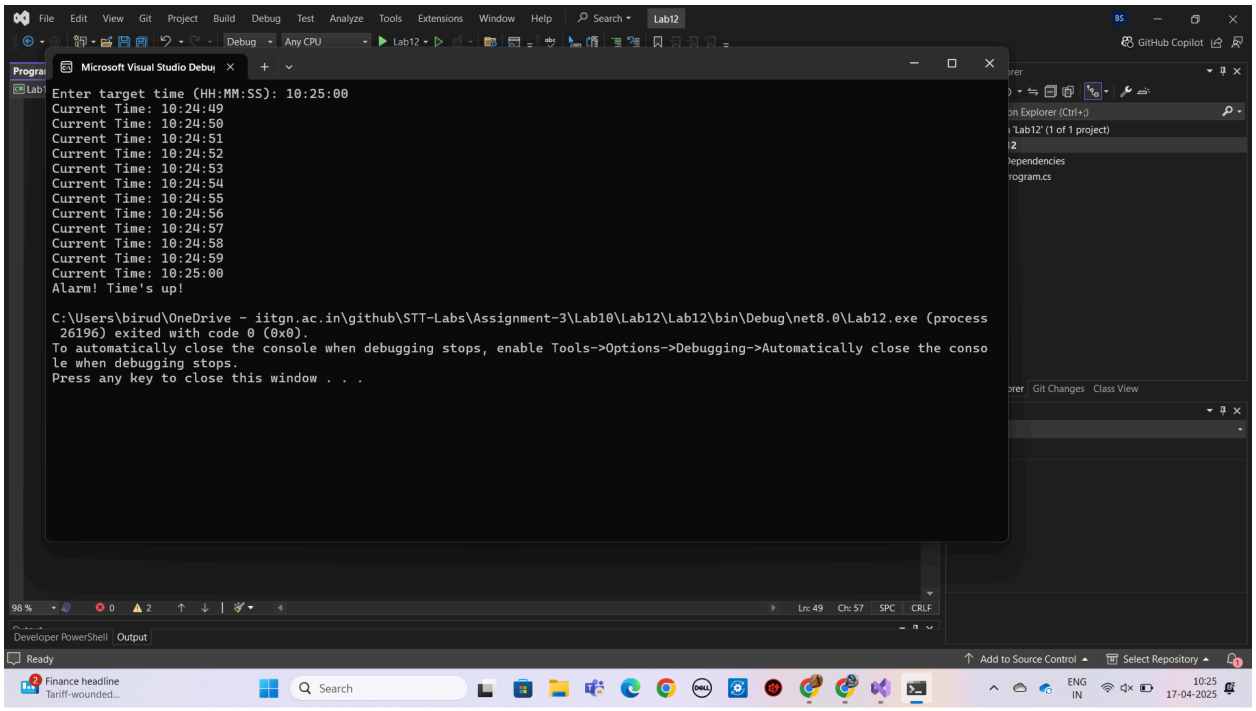


```
using System;
using System.Threading;

namespace TimeAlarmConsole
{
    // Define the delegate
    public delegate void AlarmEventHandler();

    2 references
    class Alarm
    {
        // Declare the event
        public event AlarmEventHandler RaiseAlarm;

        1 reference
        public void Start(DateTime targetTime)
        {
            while (true)
            {
                DateTime currentTime = DateTime.Now;
                Console.WriteLine("Current Time: " + currentTime.ToLongTimeString());

                if (currentTime.Hour == targetTime.Hour &&
                    currentTime.Minute == targetTime.Minute &&
                    currentTime.Second == targetTime.Second)
                {
                    RaiseAlarm?.Invoke(); // Raise the event
                    break;
                }

                Thread.Sleep(1000); // wait 1 second
            }
        }
    }
```



```
    class Program
    {
        1 reference
        static void Ring_alarm()
        {
            Console.WriteLine("⏰ Alarm! Time's up!");
        }

        0 references
        static void Main(string[] args)
        {
            Console.Write("Enter target time (HH:MM:SS): ");
            string input = Console.ReadLine();

            if (DateTime.TryParse(input, out DateTime targetTime))
            {
                Alarm alarm = new Alarm();
                alarm.RaiseAlarm += new AlarmEventHandler(Ring_alarm); // subscribe to event
                alarm.Start(targetTime);
            }
            else
            {
                Console.WriteLine("Invalid time format.");
            }
        }
    }
}
```

The below is the execution of the program:



```
Enter target time (HH:MM:SS): 10:25:00
Current Time: 10:24:49
Current Time: 10:24:50
Current Time: 10:24:51
Current Time: 10:24:52
Current Time: 10:24:53
Current Time: 10:24:54
Current Time: 10:24:55
Current Time: 10:24:56
Current Time: 10:24:57
Current Time: 10:24:58
Current Time: 10:24:59
Current Time: 10:25:00
Alarm! Time's up!

C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Labs\Assignment-3\Lab10\Lab12\Lab12\bin\Debug\net8.0\Lab12.exe (process
 26196) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Publisher/Subscriber Breakdown

- **Publisher**

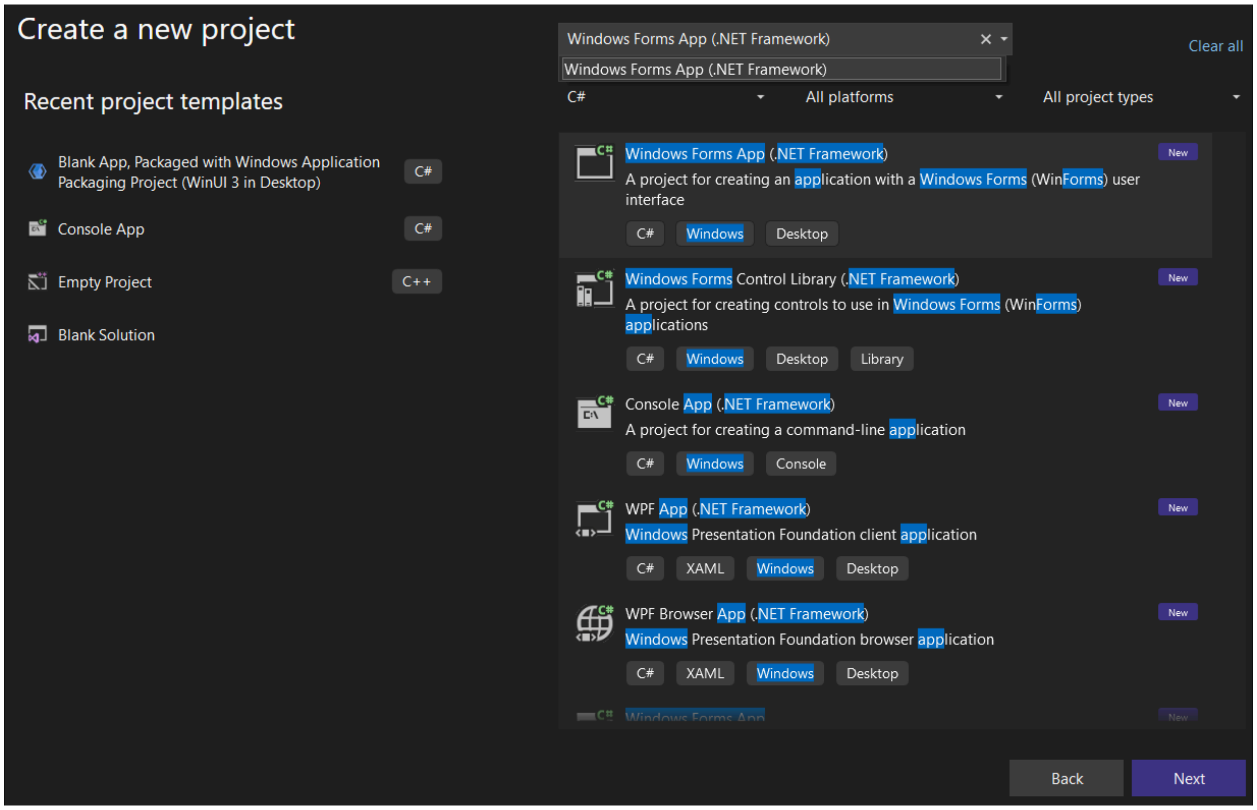
  - The part of my program that *detects when the time matches* is the **publisher**.
  - This section raises the event `RaiseAlarm` when the condition (system time == user time) is met.

- **Subscriber**

  - The function `Ring_alarm()` is the **subscriber**.
  - It's hooked to the `RaiseAlarm` event using the `+=` syntax, meaning it will be **called automatically when the event is triggered**.

b) Task 2

In the second part of the lab, I upgraded the console-based program into a full GUI application using Windows Forms.


Create a new project

Recent project templates

Blank App, Packaged with Windows Application Packaging Project (WinUI 3 in Desktop) C#
Console App C#
Empty Project C++
Blank Solution

Windows Forms App (.NET Framework)
Windows Forms App (.NET Framework)
C# | All platforms | All project types
Clear all

Windows Forms App (.NET Framework) New
A project for creating an application with a Windows Forms (WinForms) user interface
C# Windows Desktop

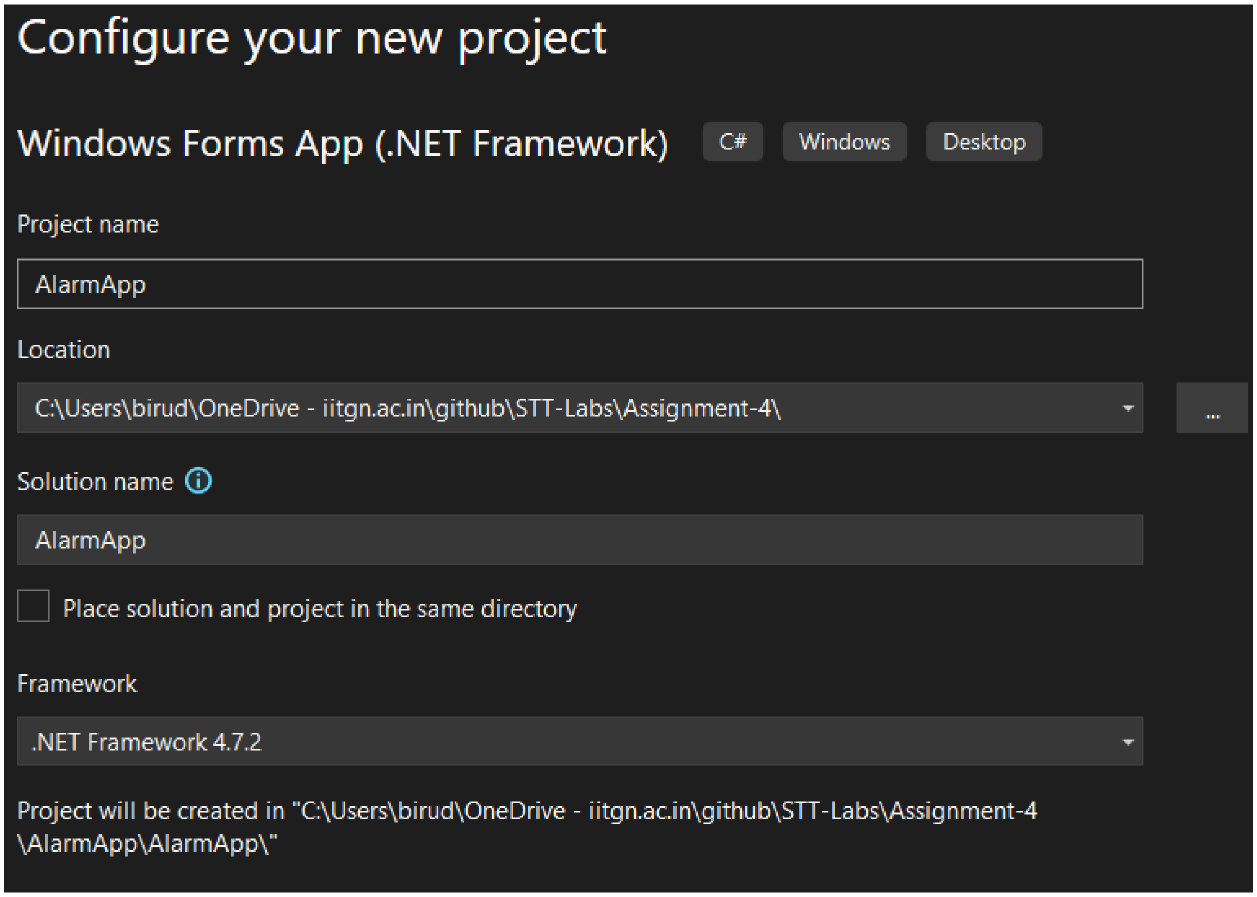
Windows Forms Control Library (.NET Framework) New
A project for creating controls to use in Windows Forms (WinForms) applications
C# Windows Desktop Library

Console App (.NET Framework) New
A project for creating a command-line application
C# Windows Console

WPF App (.NET Framework) New
Windows Presentation Foundation client application
C# XAML Windows Desktop

WPF Browser App (.NET Framework) New
Windows Presentation Foundation browser application
C# XAML Windows Desktop

Back Next


Configure your new project

Windows Forms App (.NET Framework) C# Windows Desktop

Project name

AlarmApp

Location

C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Labs\Assignment-4\

Solution name

AlarmApp

Place solution and project in the same directory

Framework

.NET Framework 4.7.2

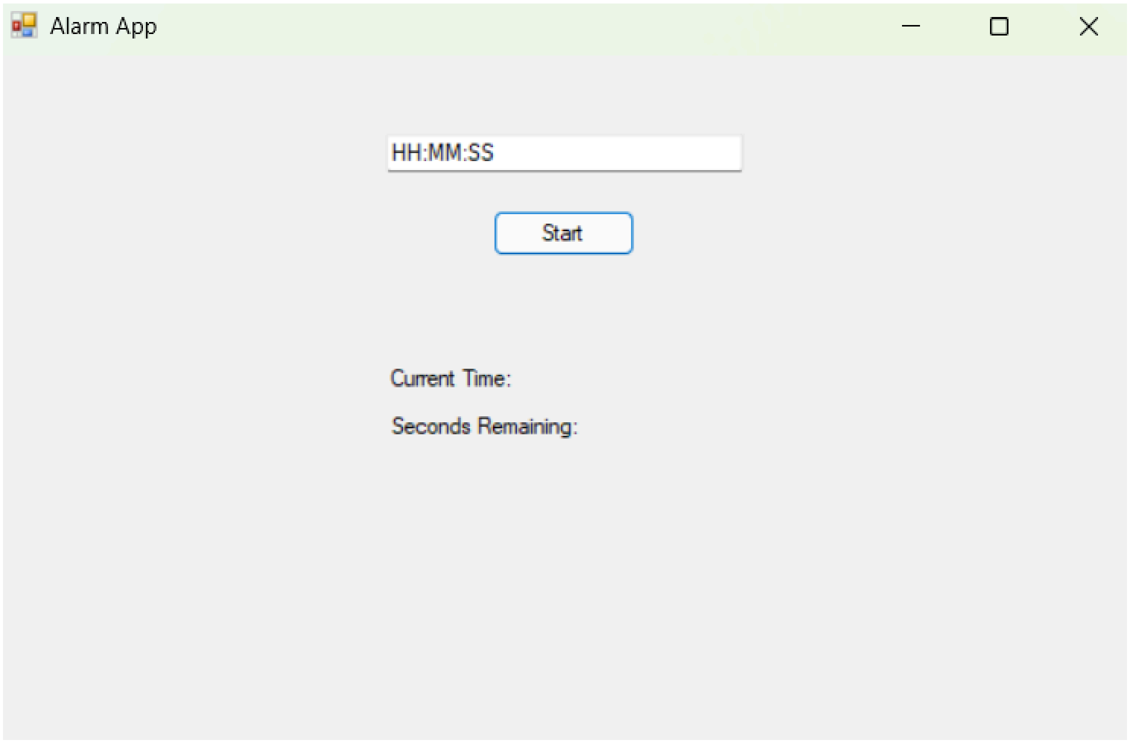
Project will be created in "C:\Users\birud\OneDrive - iitgn.ac.in\github\STT-Labs\Assignment-4
\AlarmApp\AlarmApp\"

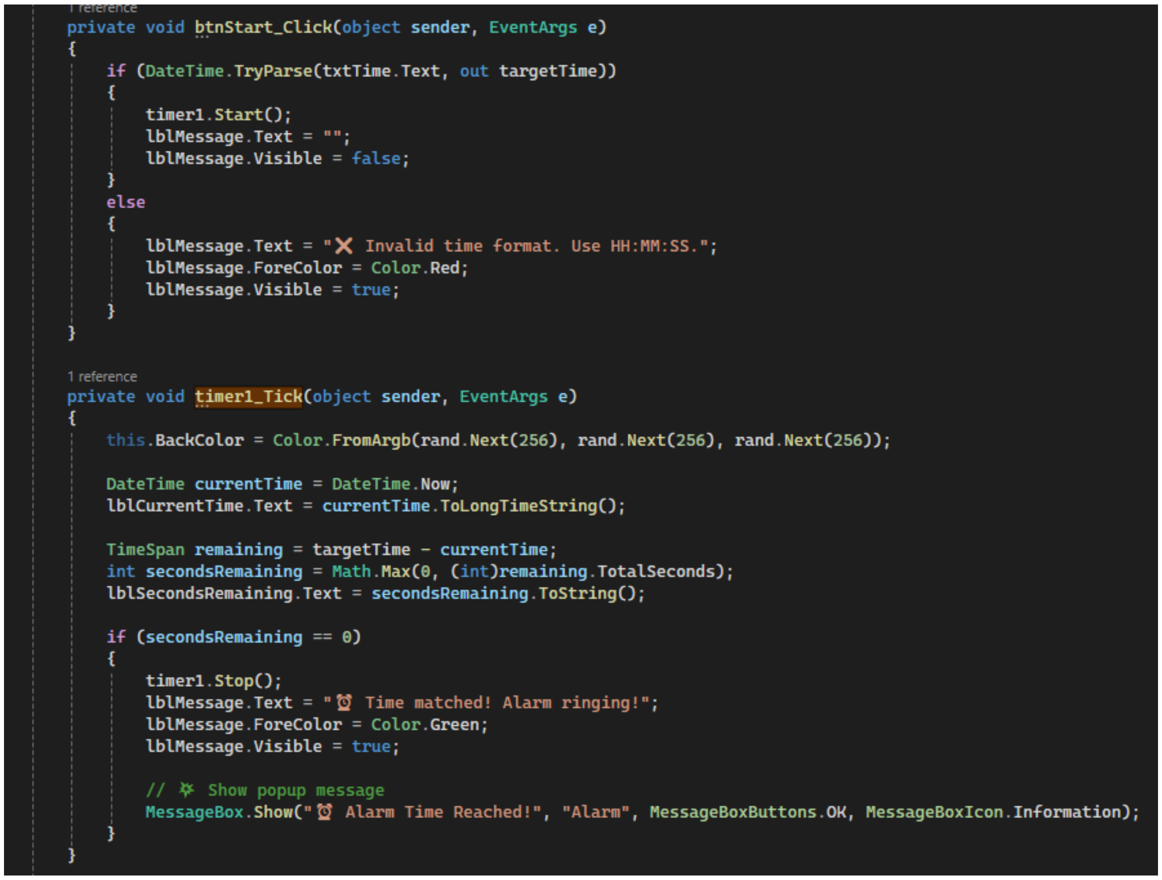
The main objective was to allow the user to enter a specific time in the format `HH:MM:SS`, and have the application notify the user when that time is reached. Initially, I set up the form with a textbox for input, a Start button, and a timer component. When the user inputs a time and clicks the Start button, the application parses the input, starts a countdown using the system clock, and continuously checks whether the current time matches the target time.

To improve the user experience, I included a **background color animation** that updates every second while the timer is active, giving a visual cue that the application is running. I also added two labels: one to display the **current time in real-time**, and another to show the **remaining time in seconds** until the alarm goes off. These additions make it easier for users to track when the alarm is expected.

Upon reaching the specified time, the application not only displays an on-screen message within the form but also triggers a **popup alert** to ensure the user is clearly notified. Additionally, input validation was handled to guide users if they enter the time in an incorrect format, and the interface updates accordingly with a red error message.

Below is the screenshot of the GUI:





```
1 reference
private void btnStart_Click(object sender, EventArgs e)
{
    if (DateTime.TryParse(txtTime.Text, out targetTime))
    {
        timer1.Start();
        lblMessage.Text = "";
        lblMessage.Visible = false;
    }
    else
    {
        lblMessage.Text = "❌ Invalid time format. Use HH:MM:SS.";
        lblMessage.ForeColor = Color.Red;
        lblMessage.Visible = true;
    }
}

1 reference
private void timer1_Tick(object sender, EventArgs e)
{
    this.BackColor = Color.FromArgb(rand.Next(256), rand.Next(256), rand.Next(256));

    DateTime currentTime = DateTime.Now;
    lblCurrentTime.Text = currentTime.ToLongTimeString();

    TimeSpan remaining = targetTime - currentTime;
    int secondsRemaining = Math.Max(0, (int)remaining.TotalSeconds);
    lblSecondsRemaining.Text = secondsRemaining.ToString();

    if (secondsRemaining == 0)
    {
        timer1.Stop();
        lblMessage.Text = "⏰ Time matched! Alarm ringing!";
        lblMessage.ForeColor = Color.Green;
        lblMessage.Visible = true;

        // 🔆 Show popup message
        MessageBox.Show("⏰ Alarm Time Reached!", "Alarm", MessageBoxButtons.OK, MessageBoxIcon.Information);
    }
}
```

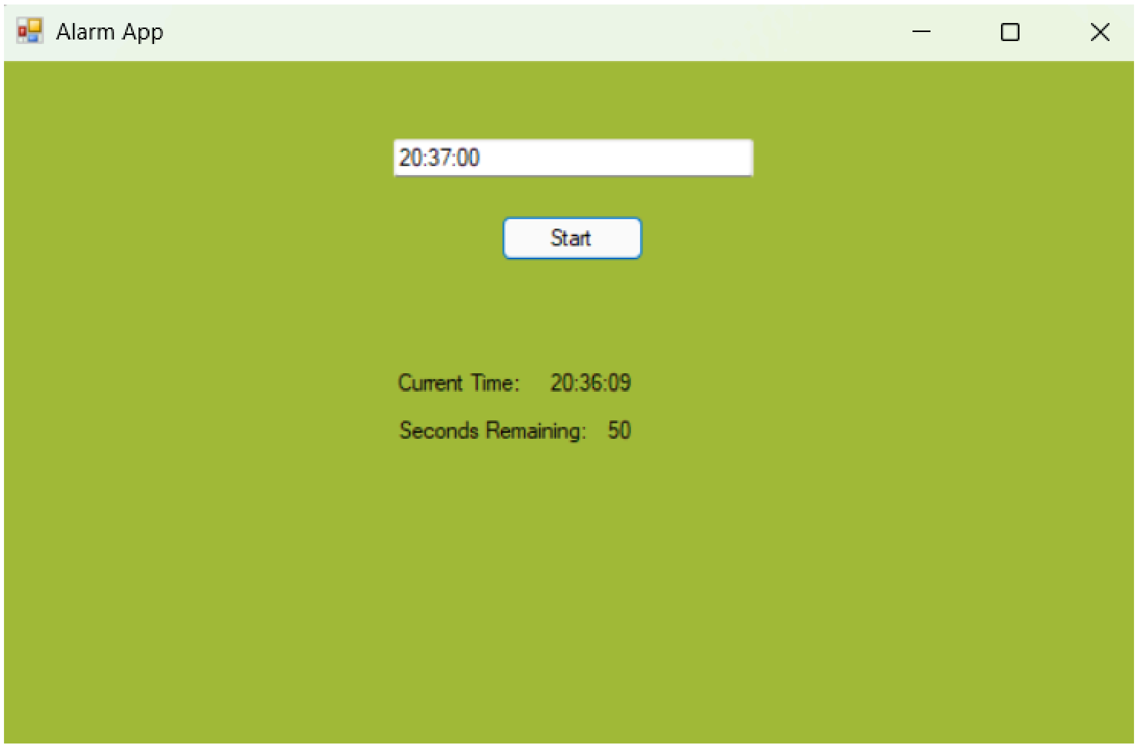
The first screenshot shows the initial interface of the application after launch. It includes:

- A text box (`txtTime`) where the user enters the target time in `HH:MM:SS` format.
- A `Start` button (`btnStart`) which begins the countdown process.
- Labels (`lblCurrentTime`, `lblSecondsRemaining`) that display real-time updates of the current time and time remaining.
- A message label (`lblMessage`) that shows alerts or errors based on the user's action.

The second screenshot captures the code from `Form1.cs` which binds user interactions to the appropriate event handlers:

- `btnStart_Click`: This event is triggered when the user clicks the Start button. It validates the time input and starts the timer.
- `timer1_Tick`: This event runs every second and updates the background color, displays the current system time, calculates the remaining time, and checks for alarm trigger conditions.
- `txtTime_Enter`: Clears placeholder text when the user clicks into the time input field.

When the timer starts, the background color of the Form changes every second:

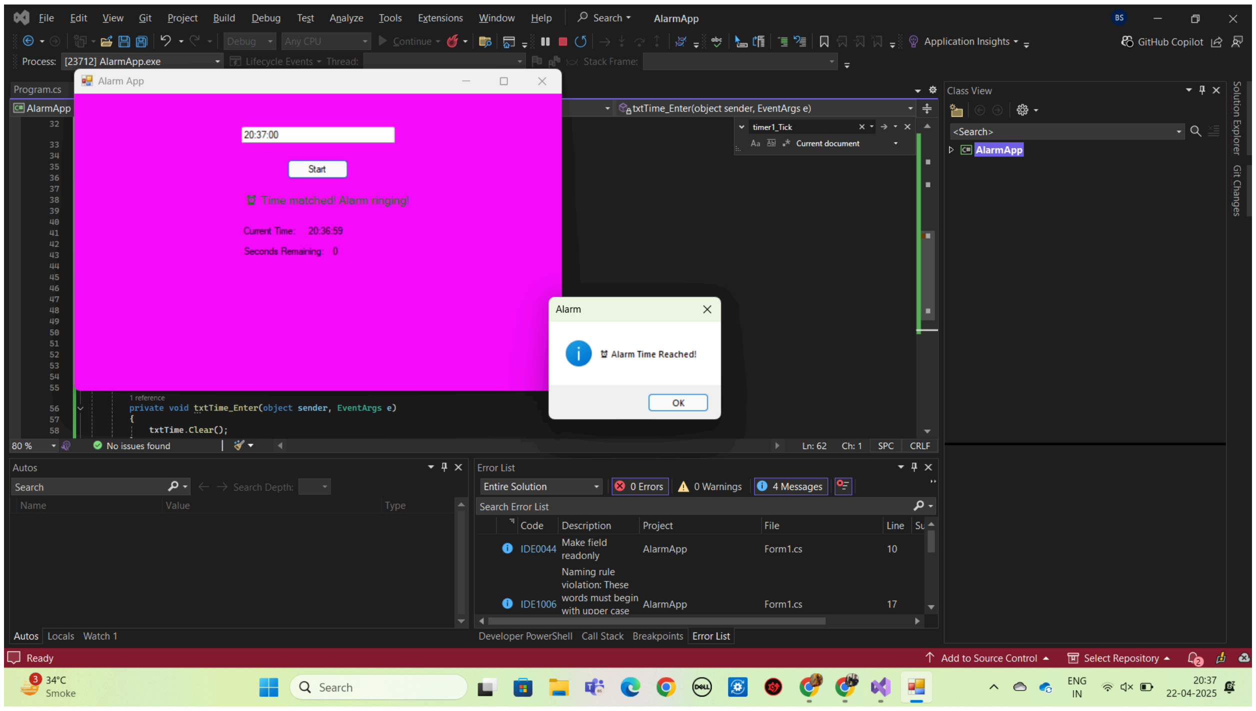


Current Time: 20:36:09

Seconds Remaining: 50

A screenshot mid-execution shows the background color changing dynamically and the labels updating with:

- The current system time.
- The countdown in seconds until the alarm triggers.


20:37:00

⏰ Time matched! Alarm ringing!

Current Time: 20:36:59

Seconds Remaining: 0

Alarm

⏰ Alarm Time Reached!

```
private void txtTime_Enter(object sender, EventArgs e)
{
    txtTime.Clear();
```

	Code	Description	Project	File	Line
	IDE0044	Make field readonly	AlarmApp	Form1.cs	10
	IDE1006	Naming rule violation: These words must begin with upper case	AlarmApp	Form1.cs	17

The above screenshot capture the moment the current time matches the target time:

- A success message (`"⏰ Time matched! Alarm ringing!"`) is shown in green on the GUI.
- A pop-up `MessageBox` alert appears simultaneously to notify the user that the alarm time has been reached.

The application follows an **event-driven programming model**, where:

- The `Start` button click acts as the main trigger event.
- The system clock updates via the `Timer` tick event.
- UI elements (textboxes, labels) respond to these events in real-time.
- Conditional logic checks time matches and provides user feedback.

By combining these elements, the application provides a smooth and interactive user experience, clearly demonstrating how user actions initiate and drive the application logic through events.

Discussions and Conclusion

Learning Outcomes

This lab offered practical exposure to **event-driven programming** in both console-based and GUI-based C# applications.

- **Event Handling Concepts**:
  In **Task 1**, I implemented a console-based alarm using the publisher-subscriber model, which helped me understand how to define, raise, and handle custom events.

- **Windows Forms and GUI Controls**:
  Task 2 deepened my understanding of the Visual Studio Toolbox and how GUI components interact through events. Using timers and message boxes helped create a responsive interface.

- **Control Flow via Events**:
  These tasks made it clear how application behavior is driven not by sequential code alone but by user actions or state changes at runtime.

- **Input Validation and Feedback**:
  Using `DateTime.TryParse()` and displaying appropriate messages taught me the importance of validating user input and handling errors gracefully in UI applications.

Challenges Faced

1. **Understanding Event Syntax (Task 1)**:
   Defining and subscribing to custom events was initially confusing, but became clearer later.

2. **Shifting to GUI Design (Task 2)**:
   Adapting logic from a console to a Windows Form required learning to use timers, form controls, and layout tools efficiently.

3. **Timer Accuracy and Control**:
   Managing the form's background color to change every second and then stop immediately at the target time needed careful control of `timer1.Stop()` to prevent unnecessary updates beyond the alarm trigger.

Conclusion

This lab clearly demonstrated how event-driven control flow works in both console and GUI-based C# applications. Task 1 helped me understand the basics of custom events and handlers, while Task 2 provided a visual, interactive experience using forms, timers, and message boxes. Together, these tasks built a strong foundation in responding to user actions and internal application states, an essential concept for modern software development.

Resources

- Lecture 11 Slides
- https://learn.microsoft.com/en-us/dotnet/csharp
- https://en.wikipedia.org/wiki/Event-driven_programming
- https://quix.io/blog/what-why-how-of-event-driven-programming