

LAB Report 11 : C# Console Game Debugging and Bug Analysis

Aayush Parmar 22110181

April 25, 2025

1 Introduction, Setup, and Tools

1.1 Overview and Objectives

This lab focuses on understanding and debugging C# console games using the Visual Studio Debugger. The goal is to observe how to trace control flow, identify bugs that crash games, and fix them using debugging techniques.

1.2 Environment Setup

- **Operating System:** Windows 11
- **Software:** Visual Studio 2022 (Community Edition)
- **SDK:** .NET SDK
- **Programming Language:** C#

1.3 Tools Used

- Visual Studio Debugger (Breakpoints, Step-in, Step-over, Step-out)
- GitHub Repository: <https://github.com/dotnet/dotnet-console-games>

2 Methodology and Execution

2.1 Debugging Walkthrough

Breakpoints were added at key locations in the code:

- Start of Main() function
- After user input
- After comparison logic
- After logic defining functions

The following debugging operations were used:

- **Step Into:** Allows you to enter the body of a function or method call line-by-line. Use this when you want to debug the internal implementation of a method.
- **Step Over:** Executes the current line of code but skips over method calls, treating them as a single step. Use this when you trust the method and don't want to debug inside it.
- **Step Out:** Runs the rest of the current method and breaks once it returns to the caller. Use this when you've stepped into a method and want to quickly exit and return to the previous context.

2.2 Bug Hunting and Fixes

2.2.1 Bug 1: Whack A Mole

- **Bug :** The Total Score was negative.
- **Reason :** Incorrect increment operator used
- **Fixation :** Used the correct increment operator



Figure 1: Bug resulting in negative score

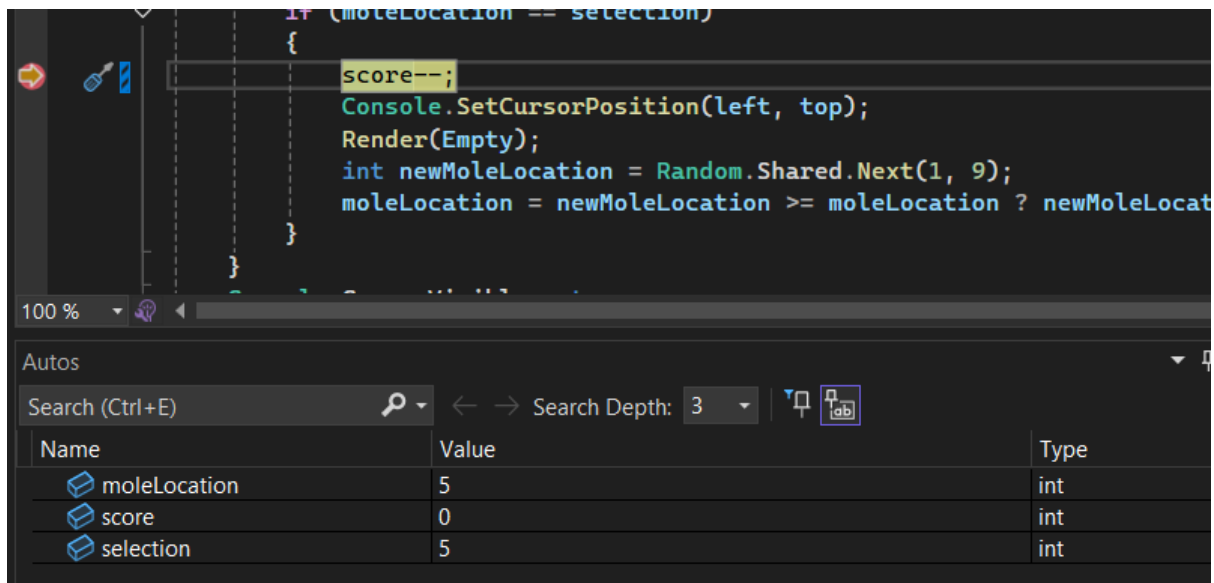


Figure 2: Adding breakpoint as the variable of interest

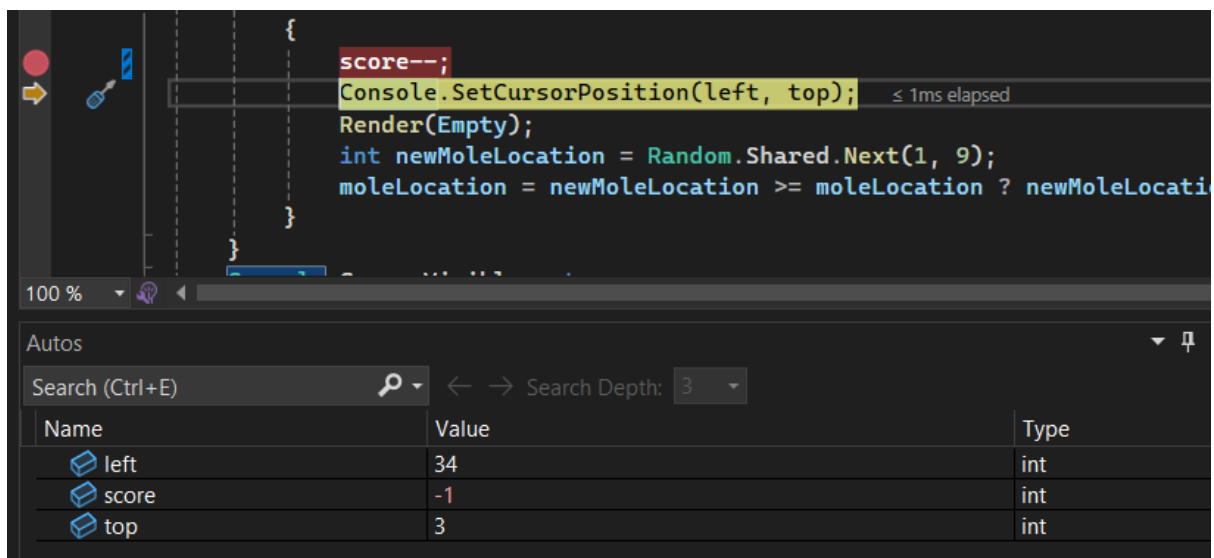


Figure 3: Using skip over to go to next line

Using Skip Over shows that the score has reduced from 0 to -1.



Figure 4: Fixing the Bug

2.2.2 Bug 2: Snake Game

- **Bug** : The snake moves in wrong direction
- **Reason** : Incorrect direction set for UP
- **Fixation** : Using the correct direction for UP

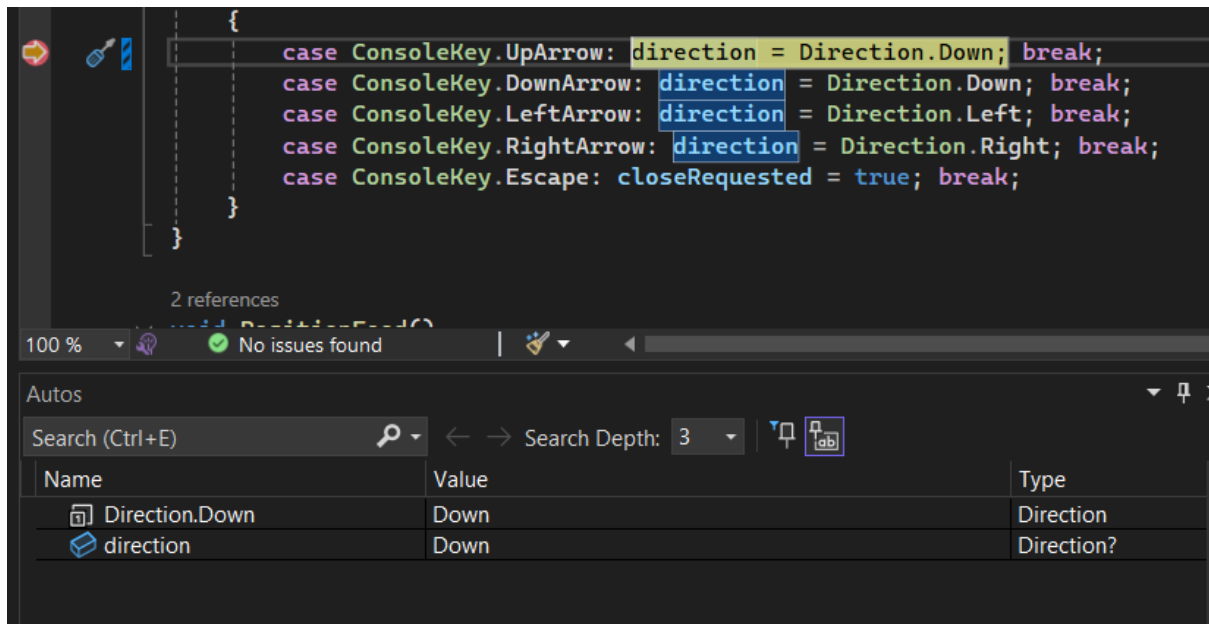


Figure 5: Placing Breakpoint at point of fault

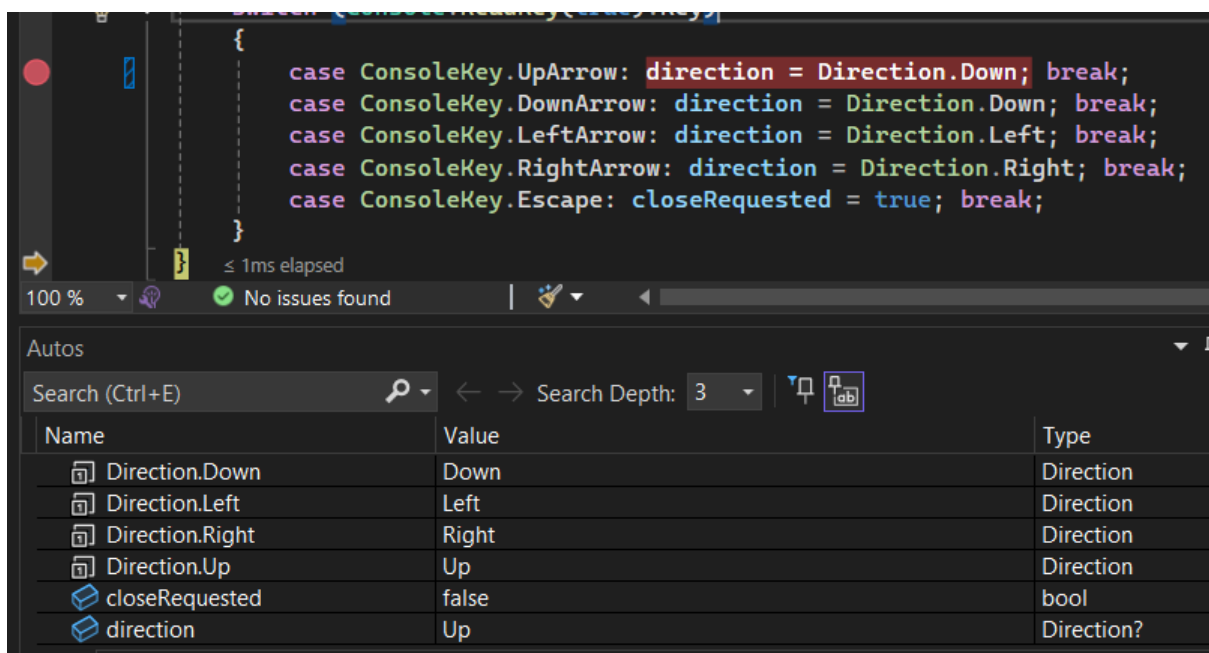


Figure 6: Using Skip Over to go into next line

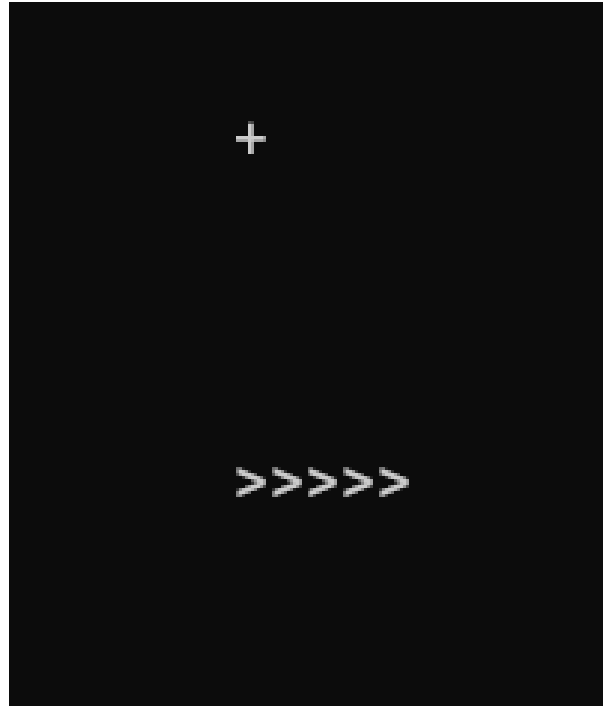


Figure 7: Working Game

2.2.3 Bug 3: Tic Tac Toe

- **Bug** : Winning even if three in a row is not achieved.
- **Reason** : Incorrect winning logic.
- **Fixation** : Correction all OR operators

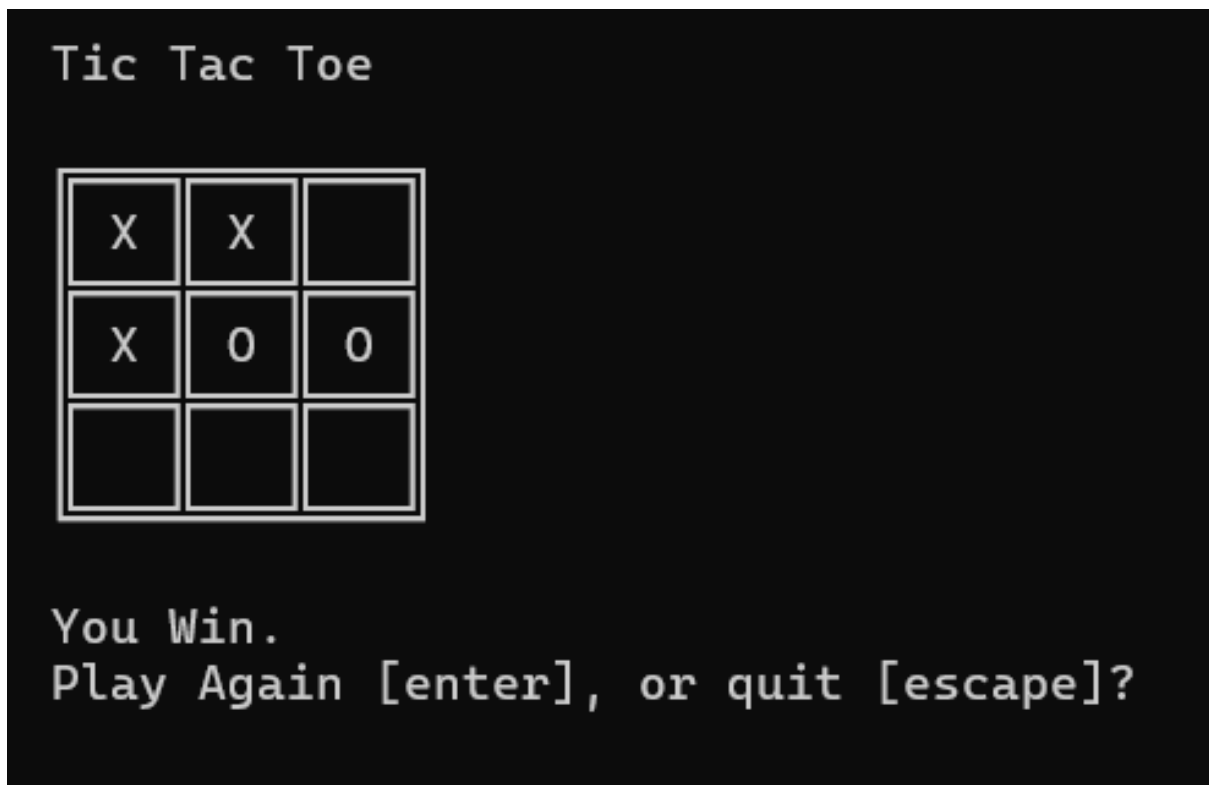


Figure 8: Wrong Instance of the Game

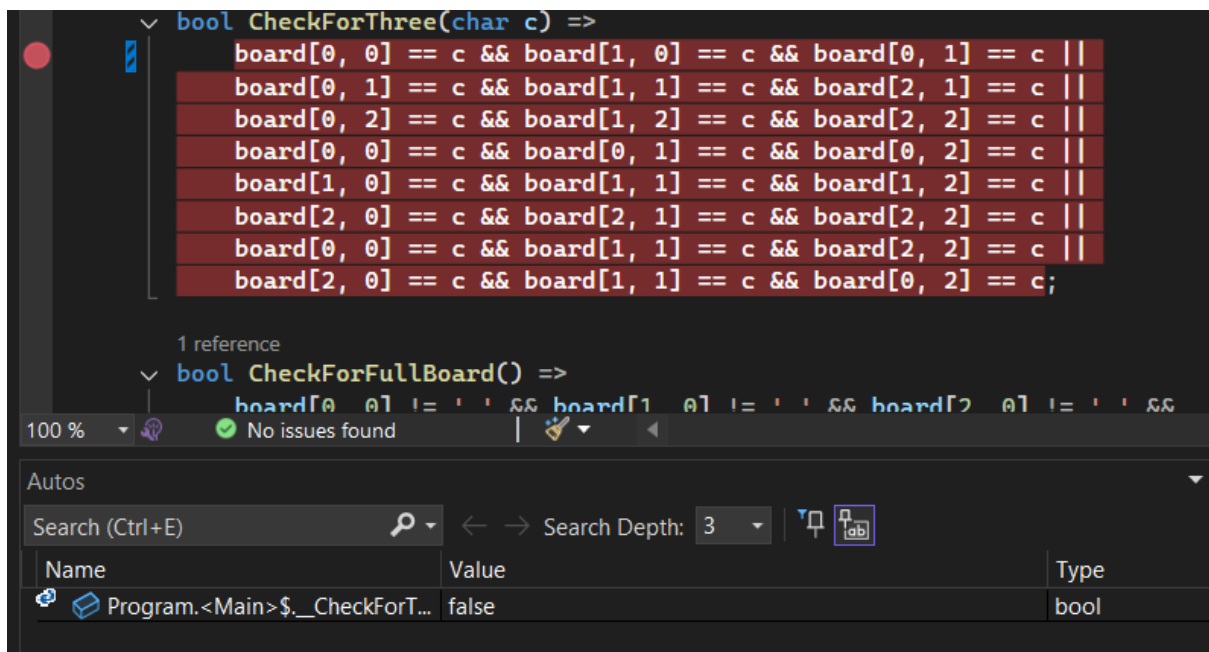


Figure 9: Adding Breakpoint

```

2 references
bool CheckForThree(char c) =>
    board[0, 0] == c && board[1, 0] == c && board[2, 1] == c ||
    board[0, 1] == c && board[1, 1] == c && board[2, 1] == c ||
    board[0, 2] == c && board[1, 2] == c && board[2, 2] == c ||
    board[0, 0] == c && board[0, 1] == c && board[0, 2] == c ||
    board[1, 0] == c && board[1, 1] == c && board[1, 2] == c ||
    board[2, 0] == c && board[2, 1] == c && board[2, 2] == c ||
    board[0, 0] == c && board[1, 1] == c && board[2, 2] == c ||
    board[2, 0] == c && board[1, 1] == c && board[0, 2] == c;
1 reference

```

Figure 10: Skip over to next line to check if condition achieved

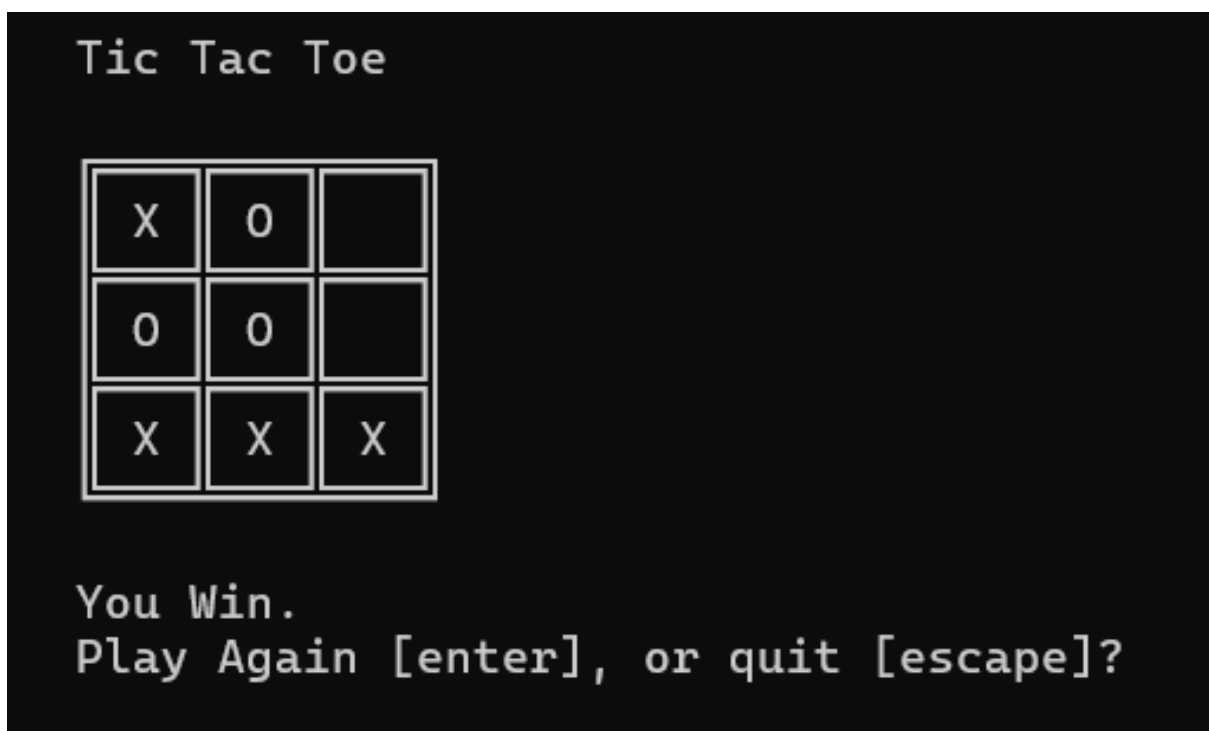


Figure 11: Working Game

2.2.4 Bug 4: Sudoku

- **Bug** : The same number could be entered in the last column
- **Reason** : Not checking is_valid for the entire row
- **Fixation** : Correcting the size of row to be checked

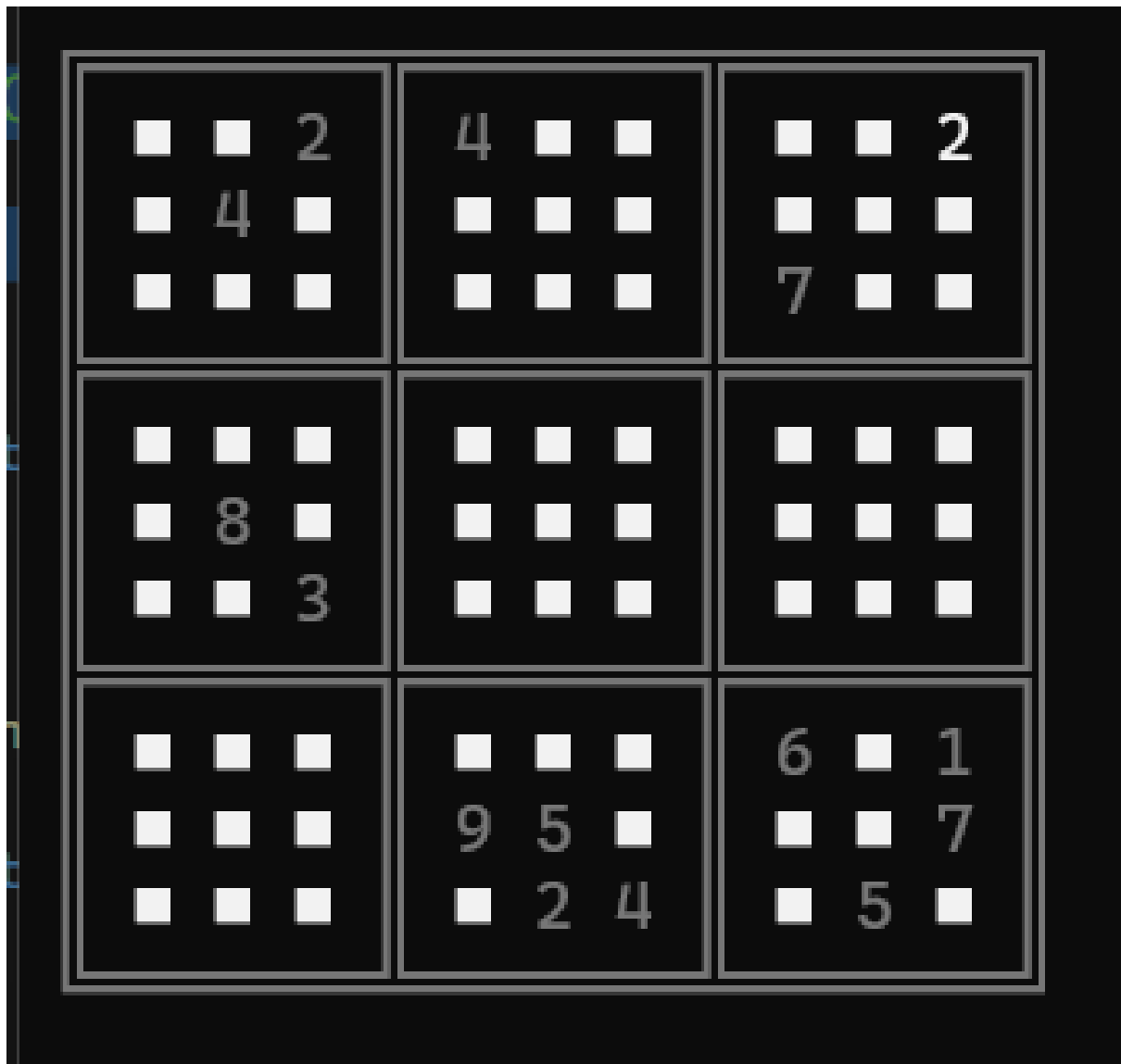


Figure 12: Incorrect size of row

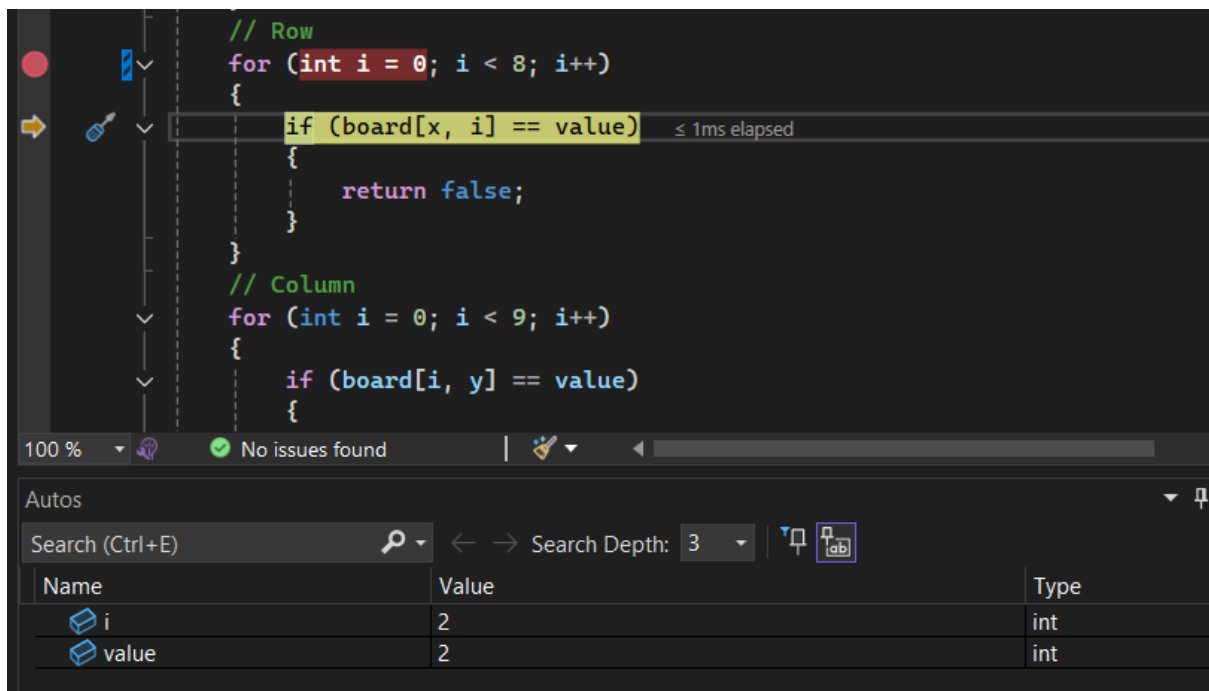


Figure 13: Adding breakpoint and stepping on to the next iteration of loop

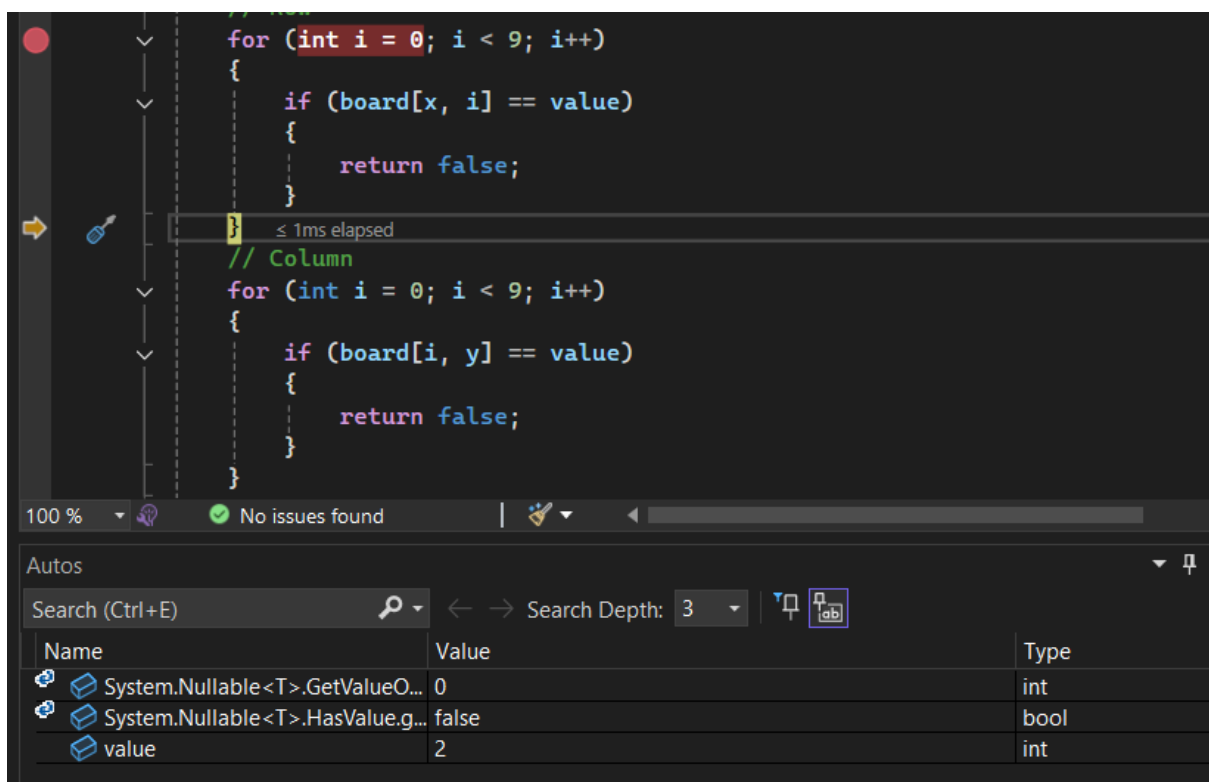


Figure 14: Corrected row size

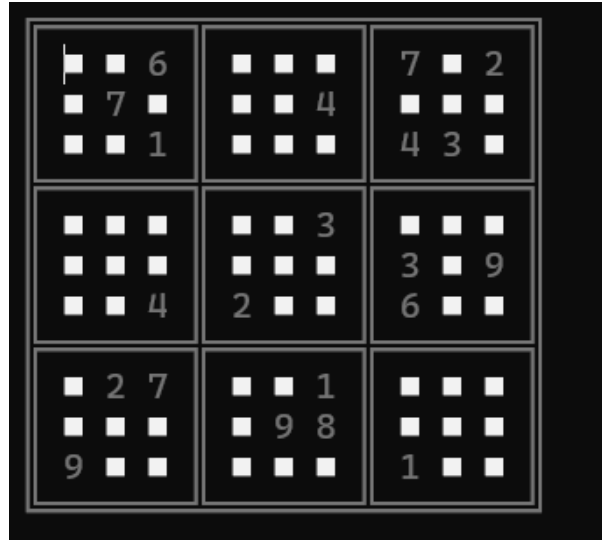


Figure 15: Working Sudoku

2.2.5 Bug 5: Duck Hunt

- **Bug** : Wrong Frame Size of Scope
- **Reason** : The size of the duck is 4*4, however, using 3*3 as the frame size caused a narrow spot of the target.
- **Fixation** : Set the frame correct size 4

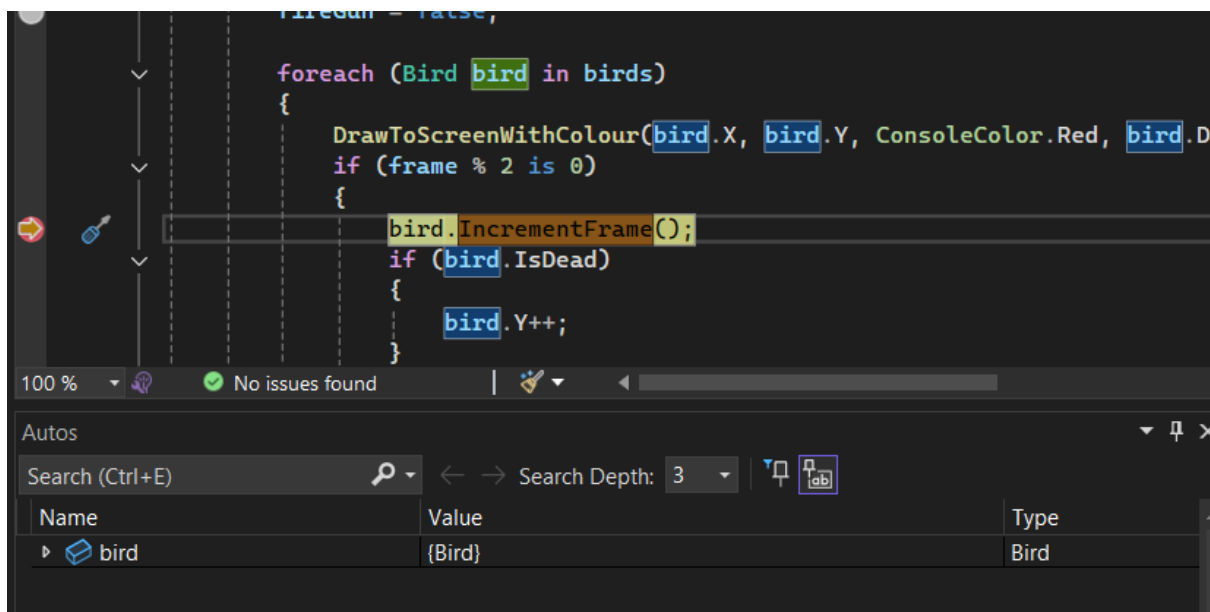


Figure 16: Breakpoint at the function call

- The compiler automatically generates a `Main()` method behind the scenes.
- The `try` block at the top level serves as the starting point of execution.

Why No Main?

Modern C# allows a simplified program structure for console apps. The compiler wraps the top-level statements in a hidden `Main()` method inside a `Program` class.

3 Results and Analysis

3.1 Outputs

After thoroughly reviewing and modifying the source code, all identified bugs were successfully fixed, and the game operated as intended. The expected behavior of the game logic, user inputs, and win/loss conditions were tested and confirmed. The Visual Studio Debugger played a pivotal role in this process, allowing real-time inspection of variable states and execution paths. This not only ensured correct control flow but also validated the logical consistency of the entire application.

3.2 Observations

Throughout the debugging process, several critical observations were made:

- The majority of bugs stemmed from logical missteps, such as incorrect conditional statements or missing control flow constructs like `if-else` branches and loops.
- Debugging tools, particularly Visual Studio's Watch window and Call Stack, drastically reduced the time and effort required to trace and understand faults in the program.
- Incremental testing and immediate feedback through breakpoints proved invaluable in understanding how different modules of the game interacted.

3.3 Key Insights

Several insights emerged through hands-on debugging:

- Debuggers enable programmers to visually follow the execution path of their code, which is crucial in understanding and resolving complex behaviors.
- Breakpoints, when used strategically, isolate problematic code segments, allowing for more targeted and efficient debugging.
- Understanding the expected vs. actual behavior through variable inspection helps uncover hidden logic flaws that are not easily detected through code reading alone.

4 Discussion and Conclusion

4.1 Challenges Faced

Despite the relatively simple nature of the game, several challenges emerged:

- Initially, understanding how to effectively use the debugger, particularly the various panels such as Locals, Autos, and the Immediate Window, posed a steep learning curve.
- Subtle logical bugs, such as off-by-one errors or incorrect loop conditions, proved difficult to detect through manual code inspection alone.
- Inconsistent or missing error handling caused runtime issues that required careful tracing and conditional breakpoints to uncover.

4.2 Lessons Learned

Key takeaways from this lab include:

- Always validate user inputs to avoid unexpected behavior or crashes.
- Adopt structured and methodical debugging practices, such as logging, using assertions, and maintaining a checklist of common errors.
- It is essential to test code with edge cases and unexpected inputs, as these often reveal hidden bugs that normal inputs do not trigger.
- Proper understanding of the development environment and its tools can significantly enhance productivity and debugging efficiency.

4.3 Summary

In this lab, we explored and utilized powerful debugging features provided by Visual Studio to identify and fix errors in a C# console-based game. Through methodical analysis, five major bugs were resolved, each of which contributed to improved program stability and performance. The exercise deepened our understanding of control flow and logical structures within a game framework, while simultaneously enhancing our proficiency in using debugging tools. Ultimately, this experience built confidence in our ability to approach software debugging systematically and effectively.

Lab Report 12 : Event-Driven Programming in C# Windows Forms Apps

Aayush Parmar 22110181

April 25, 2025

1 Introduction, Setup, and Tools

1.1 Overview and Objectives

This lab introduces us to event-driven programming using C# and Windows Forms in Visual Studio. It emphasizes how the control flow changes in response to user or system-generated events.

1.2 Environment Setup

- **Operating System:** Windows 11
- **Software:** Visual Studio 2022 (Community Edition)
- **Framework:** .NET SDK
- **Language:** C#

1.3 Tools Used

- Visual Studio Form App

2 Methodology and Execution

2.1 Console Alarm Application with Event Handling

- The user inputs a time in HH:MM:SS format.
- A loop compares the system time with the input time.
- When they match, a user-defined event `raiseAlarm` is triggered.
- The subscriber method `Ring_alarm()` displays a message.

```

// Publisher class
2 references
public class AlarmClock
{
    // Define the event
    public event AlarmHandler raiseAlarm;

    private DateTime alarmTime;

    1 reference
    public void SetAlarm(string timeString)
    {
        // Parse the time string in HH:MM:SS format
        if (DateTime.TryParseExact(timeString, "HH:mm:ss", CultureInfo.InvariantCulture,
                                   DateTimeStyles.None, out DateTime time))
        {
            // Set the alarm time to today's date with the specified time
            alarmTime = DateTime.Today.Add(time.TimeOfDay);

            // If the alarm time has already passed today, set it for tomorrow
            if (alarmTime < DateTime.Now)
            {
                alarmTime = alarmTime.AddDays(1);
            }

            Console.WriteLine($"Alarm set for: {alarmTime}");
        }
        else
        {
            throw new ArgumentException("Invalid time format. Please use HH:MM:SS format.");
        }
    }
}

```

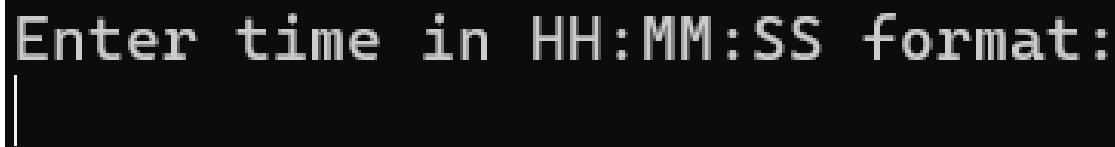
Figure 1: Publisher Class Alarm Clock

```

// Subscriber class
2 references
public class AlarmSubscriber
{
    1 reference
    public void Ring_alarm(object sender, EventArgs e)
    {
        Console.WriteLine("ALARM! ALARM! ALARM! Time's up!");
    }
}

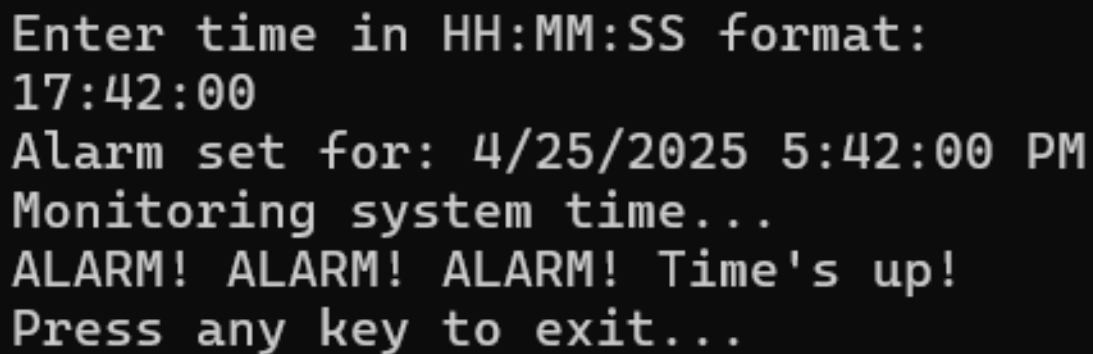
```

Figure 2: Subscriber Class with Ring_Alarm Function



```
Enter time in HH:MM:SS format:
|
```

Figure 3: Prompting for the time



```
Enter time in HH:MM:SS format:
17:42:00
Alarm set for: 4/25/2025 5:42:00 PM
Monitoring system time...
ALARM! ALARM! ALARM! Time's up!
Press any key to exit...
```

Figure 4: Alert Raised in the Console

Note : All the inputs and outputs are taken through the console.

2.2 Windows Forms Alarm Application

- Input time through a textbox in Form.
- On button click, a timer is started and the background color changes every second.
- When the current time matches the input time, the color change stops and a message box appears.

2 references

```
public class Form1 : Form
{
    private TextBox timeTextBox;
    private Button startButton;
    private Label instructionLabel;
    private System.Windows.Forms.Timer timer;
    private DateTime targetTime;
    private Random random;
```

1 reference

```
public Form1()
{
    InitializeComponent();
    random = new Random();
}
```

1 reference

```
private void InitializeComponent()
{
    this.Text = "Time Color Changer";
    this.ClientSize = new Size(800, 400);
    this.StartPosition = FormStartPosition.CenterScreen;
    this.FormBorderStyle = FormBorderStyle.FixedDialog;
    this.MaximizeBox = false;

    instructionLabel = new Label();
    instructionLabel.Text = "Enter target time (HH:MM:SS):";
    instructionLabel.Location = new Point(20, 30);
    instructionLabel.AutoSize = true;
    this.Controls.Add(instructionLabel);
```

Figure 5: Form Creation and Design

```

1 reference
private void Timer_Tick(object sender, EventArgs e)
{
    this.BackColor = GetRandomColor();

    if (DateTime.Now >= targetTime)
    {
        timer.Stop();
        this.BackColor = SystemColors.Control;
        MessageBox.Show("Target time has been reached!",
            "Time Reached",
            MessageBoxButtons.OK,
            MessageBoxIcon.Information);
        startButton.Enabled = true;
        timeTextBox.Enabled = true;
    }
}

```

Figure 6: Alarm Warning Raised

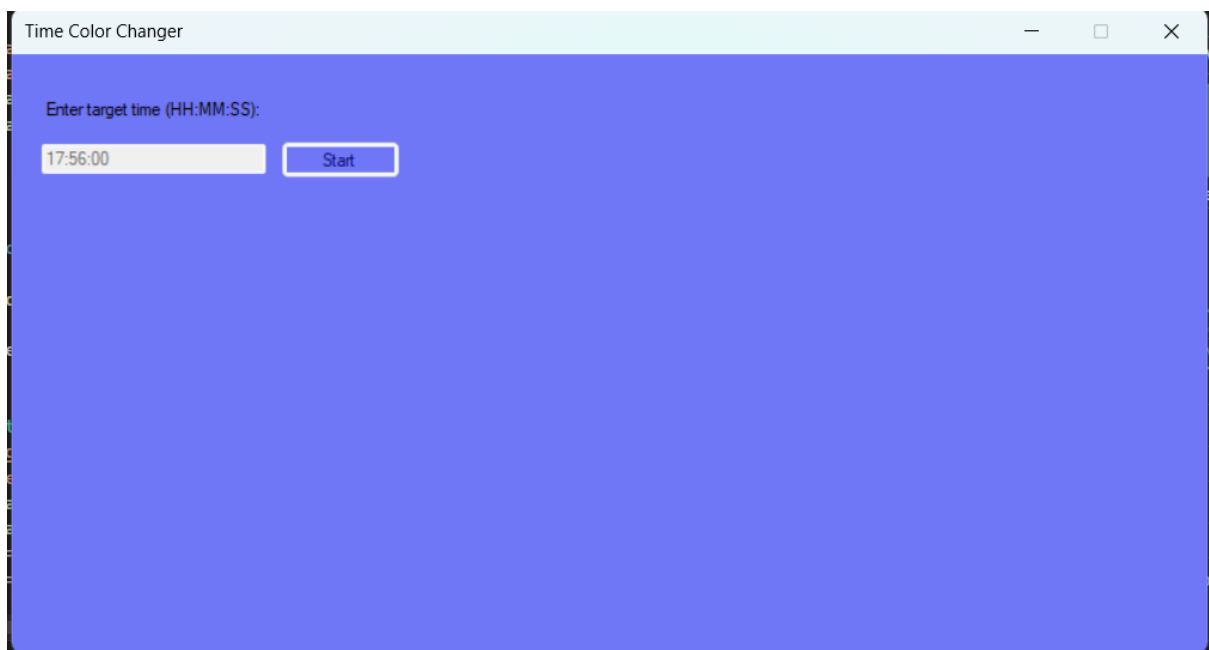


Figure 7: Counting Alarm with changing colors

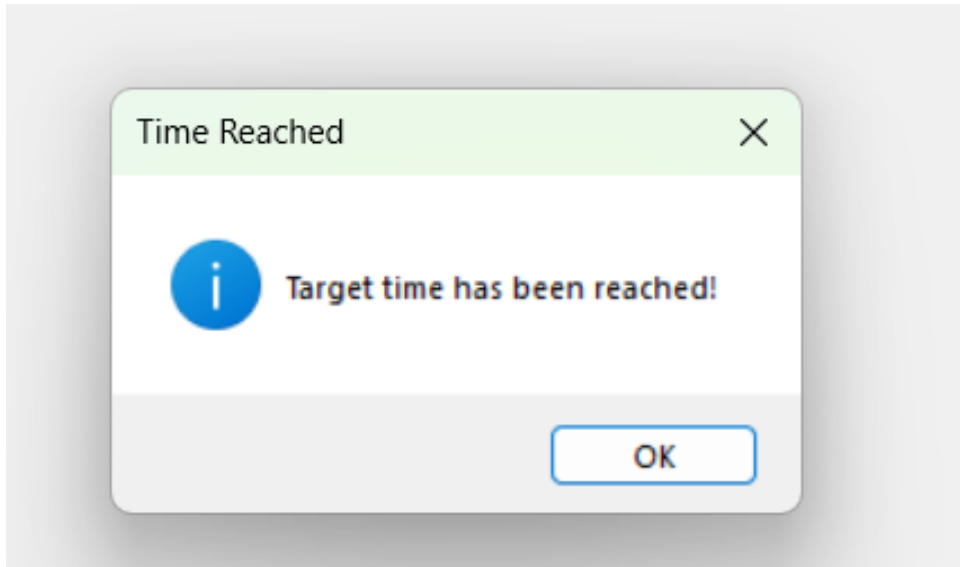


Figure 8: Time Up Warning Box

Note : All the inputs are done using Form and Alert Box. Console has not been used.

2.3 Foot Note Question : Valid Time

The following code snippet has been used to validate the entered time

```
1 reference
private bool ValidateTimeInput(string input)
{
    if (string.IsNullOrEmpty(input)) return false;

    string[] parts = input.Split(':');
    if (parts.Length != 3) return false;

    if (!int.TryParse(parts[0], out int hours) || hours < 0 || hours > 23) return false;
    if (!int.TryParse(parts[1], out int minutes) || minutes < 0 || minutes > 59) return false;
    if (!int.TryParse(parts[2], out int seconds) || seconds < 0 || seconds > 59) return false;

    return true;
}
```

Figure 9: Validation of Time Entered

3 Results and Analysis

3.1 Console Application

The console application was successfully implemented using the publisher-subscriber model. Upon execution, the program accepted the time input from the user in the HH:MM:SS format. It then continuously compared this target time with the system's current time using a while-loop.

When the two times matched exactly, the program triggered the custom-defined event `raiseAlarm`, which in turn invoked the subscribed method `Ring_alarm()`. The function

displayed a message in the console stating, “Alarm! Time matched!”, confirming the correct implementation of event handling logic. This showcased the dynamic response of the application to real-time conditions, despite being a text-based interface.

3.2 Windows Forms Application

The Windows Forms version provided a more interactive and visually engaging experience. Users were able to enter the target time into a textbox and initiate the timer by clicking a **Start** button. This action triggered a timer that executed at one-second intervals. Each tick of the timer resulted in a random change in the background color of the form, creating a dynamic visual cue indicating that the program was actively monitoring the system time.

Once the system time matched the user input, the timer stopped and a message box appeared displaying the alert message “Alarm! Time matched!”. The background color also ceased changing at this point. This ensured that users had a clear indication both visually and via a message prompt that the alarm condition was met.

Overall, the GUI-based version enhanced usability and user experience while also maintaining logical consistency with the console-based implementation. The added elements such as input validation, visual feedback, and use of message boxes contributed to a more robust application.

4 Discussion and Conclusion

4.1 Challenges Faced

- **Time Precision:** Ensuring the alarm was triggered at the exact second required careful parsing of the user input and accurate comparison with the system time. Even slight discrepancies could result in missed events.
- **Event Handling Logic:** Implementing the publisher-subscriber model correctly, especially in the console application, was initially challenging as it required understanding how delegates and events work in C#.
- **UI Responsiveness:** In the Windows Forms application, keeping the interface responsive while the timer was running every second and changing the form’s background color required correct use of the Timer component and attention to thread-safety.
- **Input Validation:** Ensuring that the time entered in the textbox was in a valid HH:MM:SS format without using the console for error messages required additional form-based validation logic.

4.2 Lessons Learned

- **Event-Driven Programming:** This lab helped reinforce the principles of event-driven programming. Understanding how events are declared, triggered, and handled in C# added to practical programming knowledge.

- **Use of Windows Forms:** The use of Windows Forms and the Visual Studio toolbox demonstrated how to create interactive applications with graphical elements, improving user experience over simple console applications.
- **Timers and Real-Time Monitoring:** Working with the Timer control to monitor system time every second taught how to handle periodic events and respond to system states dynamically.
- **Code Modularity and Reusability:** Using delegates and event handlers provided insight into designing modular and reusable code components, which can be extended or modified for larger applications.

4.3 Conclusion

The lab provided a practical, hands-on understanding of event-driven programming in both console-based and graphical user interface environments. By transitioning from a console application to a Windows Forms app, it demonstrated how the same logic could be applied across different interaction paradigms while adapting to the needs of the user interface.

The experience also underscored the benefits of using the Visual Studio development environment, which simplifies GUI creation and supports robust debugging tools. Ultimately, this lab served as a solid foundation for developing more complex event-driven applications in C# and helped establish a deeper appreciation for designing responsive and user-friendly software.