CS202 - Lab Assignment 11

- Hitesh Kumar
- 22110098
- https://github.com/Hit2737/CS202_A3

Introduction, Setup, and Tools

Overview

This lab introduces students to debugging C# console-based games using Visual Studio. The primary goal is to explore the control flow of game programs, identify bugs (both existing and injected), and fix them using the built-in Visual Studio Debugger tools. This lab emphasizes practical debugging skills and provides a deeper understanding of how code behaviour can lead to game crashes.

Repository: *dotnet-console-games*

System Requirements

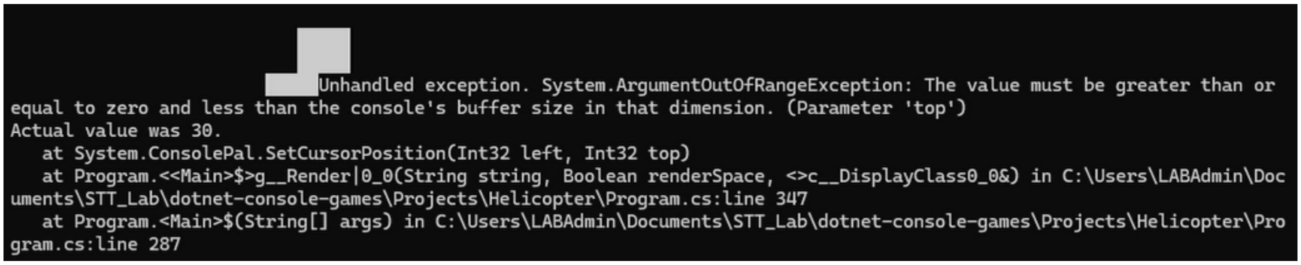
- **Operating System:** Windows
- **Software:** Visual Studio 2022 Community Edition, .NET SDK
- **Programming Language:** C# (latest stable version)

Methodology and Execution

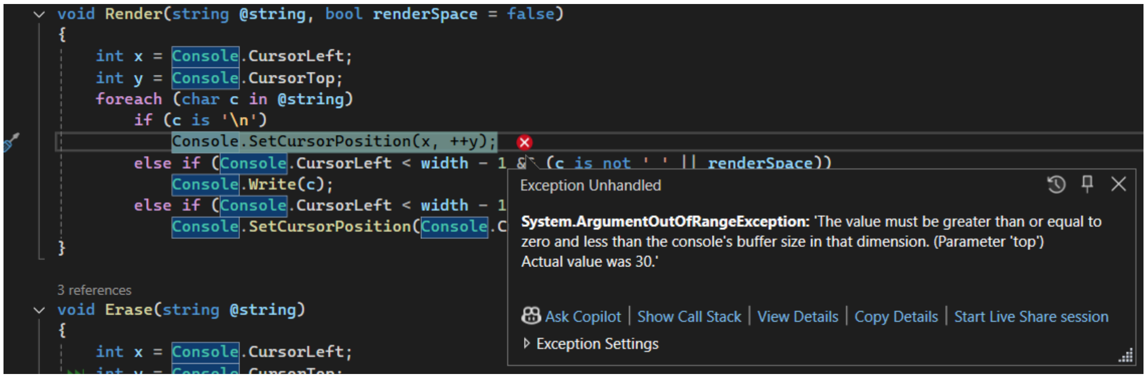
Bug Hunting and Fixing

Bug 1:

- **Game:** Helicopter
- **Bug:** Unhandled Exception (Inherently present in the code)



```
Unhandled exception. System.ArgumentOutOfRangeException: The value must be greater than or
equal to zero and less than the console's buffer size in that dimension. (Parameter 'top')
Actual value was 30.
   at System.ConsolePal.SetCursorPosition(Int32 left, Int32 top)
   at Program.<<Main>$>g__Render|0_0(String string, Boolean renderSpace, <>c__DisplayClass0_0&) in C:\Users\LABAdmin\Doc
uments\STT_Lab\dotnet-console-games\Projects\Helicopter\Program.cs:line 347
   at Program.<Main>$(String[] args) in C:\Users\LABAdmin\Documents\STT_Lab\dotnet-console-games\Projects\Helicopter\Pro
gram.cs:line 287
```



```
void Render(string @string, bool renderSpace = false)
{
    int x = Console.CursorLeft;
    int y = Console.CursorTop;
    foreach (char c in @string)
        if (c is '\n')
            Console.SetCursorPosition(x, ++y);
        else if (Console.CursorLeft < width - 1 && (c is not ' ' || renderSpace))
            Console.Write(c);
        else if (Console.CursorLeft < width - 1
            Console.SetCursorPosition(Console.C
}

3 references
void Erase(string @string)
{
    int x = Console.CursorLeft;
    int y = Console.CursorTop;
```

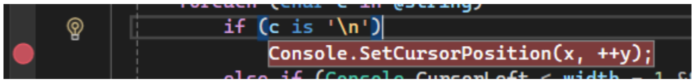
Exception Unhandled

System.ArgumentOutOfRangeException: 'The value must be greater than or equal to zero and less than the console's buffer size in that dimension. (Parameter 'top') Actual value was 30.'

Ask Copilot | Show Call Stack | View Details | Copy Details | Start Live Share session
Exception Settings

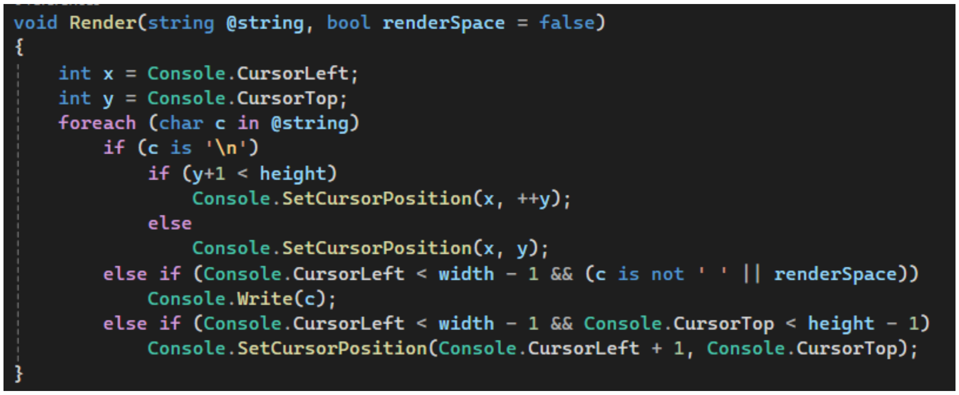
- **Analysis and Fix:**

  Identified the cause of the issue using breakpoints by understanding the code flow.



```
        if (c is '\n')
            Console.SetCursorPosition(x, ++y);
```

  Added a condition to prevent overflow from occurring on passing a greater value of y than expected. This fixes the error, and the game works correctly after that.

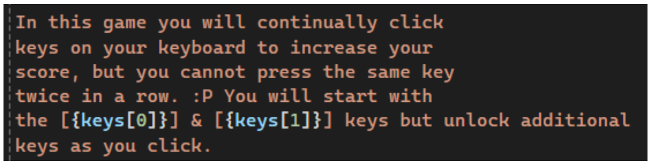


```
void Render(string @string, bool renderSpace = false)
{
    int x = Console.CursorLeft;
    int y = Console.CursorTop;
    foreach (char c in @string)
        if (c is '\n')
            if (y+1 < height)
                Console.SetCursorPosition(x, ++y);
            else
                Console.SetCursorPosition(x, y);
        else if (Console.CursorLeft < width - 1 && (c is not ' ' || renderSpace))
            Console.Write(c);
        else if (Console.CursorLeft < width - 1 && Console.CursorTop < height - 1)
            Console.SetCursorPosition(Console.CursorLeft + 1, Console.CursorTop);
}
```

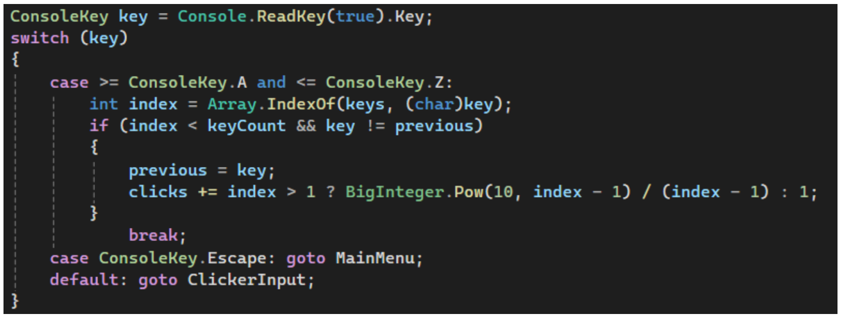
Bug 2:

- **Game:** Clicker
- **Bug:** Missing Logic (Inherently present in the code)

  As per the instructions, the game should end when the same key is pressed twice in a row. But the game still continues.



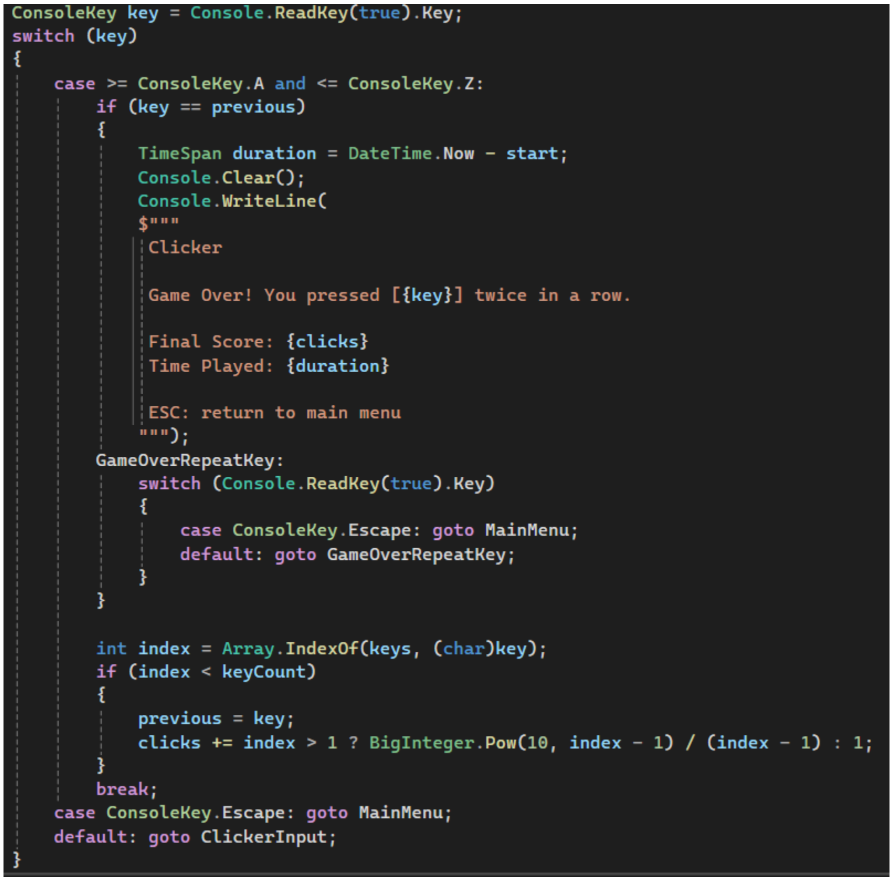
```
In this game you will continually click
keys on your keyboard to increase your
score, but you cannot press the same key
twice in a row. :P You will start with
the [{keys[0]}] & [{keys[1]}] keys but unlock additional
keys as you click.
```



```
ConsoleKey key = Console.ReadKey(true).Key;
switch (key)
{
    case >= ConsoleKey.A and <= ConsoleKey.Z:
        int index = Array.IndexOf(keys, (char)key);
        if (index < keyCount && key != previous)
        {
            previous = key;
            clicks += index > 1 ? BigInteger.Pow(10, index - 1) / (index - 1) : 1;
        }
            break;
    case ConsoleKey.Escape: goto MainMenu;
    default: goto ClickerInput;
}
```

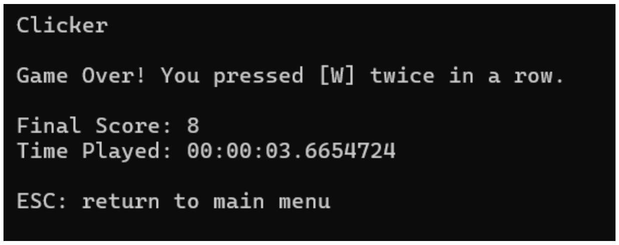
- **Analysis and Fix:**

  In the current logic, nothing happens if the same key is pressed twice because of the missing check for (key == previous):



```
ConsoleKey key = Console.ReadKey(true).Key;
switch (key)
{
    case >= ConsoleKey.A and <= ConsoleKey.Z:
        if (key == previous)
        {
            TimeSpan duration = DateTime.Now - start;
            Console.Clear();
            Console.WriteLine(
            $"""
             Clicker

             Game Over! You pressed [{key}] twice in a row.

             Final Score: {clicks}
             Time Played: {duration}

             ESC: return to main menu
            """);
        GameOverRepeatKey:
            switch (Console.ReadKey(true).Key)
            {
                case ConsoleKey.Escape: goto MainMenu;
                default: goto GameOverRepeatKey;
            }
        }

        int index = Array.IndexOf(keys, (char)key);
        if (index < keyCount)
        {
            previous = key;
            clicks += index > 1 ? BigInteger.Pow(10, index - 1) / (index - 1) : 1;
        }
        break;
    case ConsoleKey.Escape: goto MainMenu;
    default: goto ClickerInput;
}
```

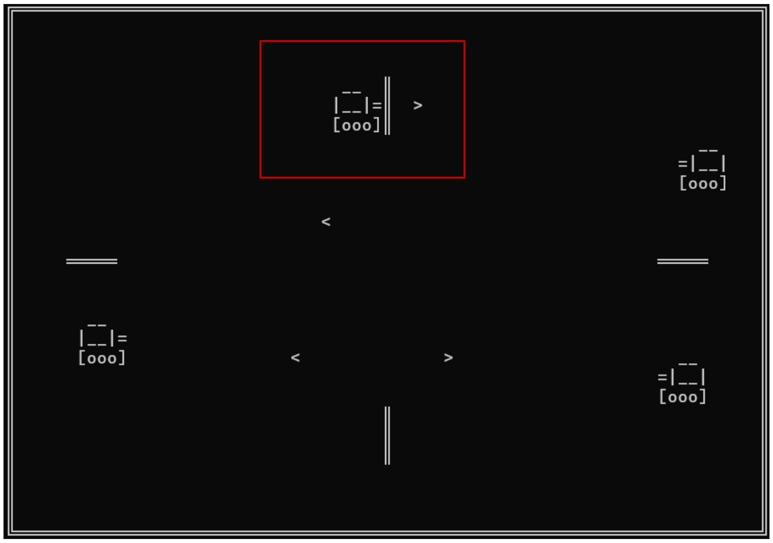
  The game behaves as expected after adding the required logic for the condition (key == previous).



```
Clicker

Game Over! You pressed [W] twice in a row.

Final Score: 8
Time Played: 00:00:03.6654724

ESC: return to main menu
```

Bug 3:

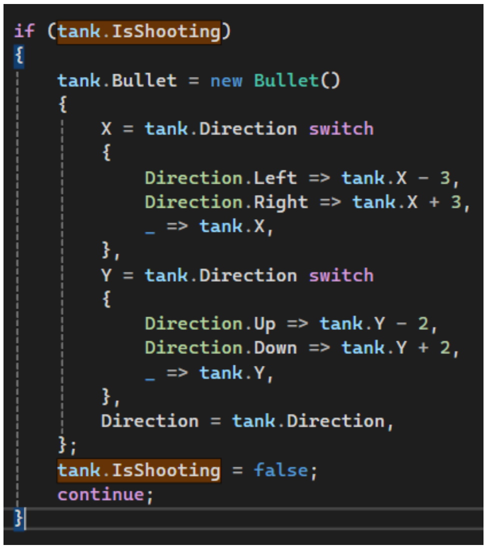
- **Game:** Tanks
- **Bug:** Bullets passing through the wall (Inherently present in the code)



The code logic is incorrect for bullets hitting the wall when the tank stands very next to the wall.

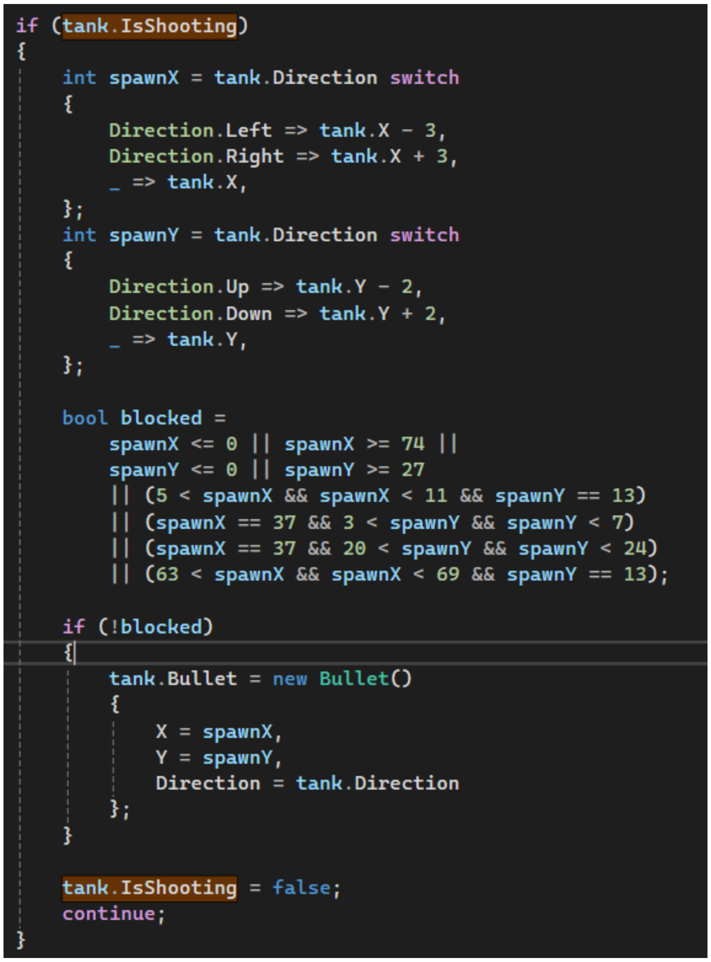
- **Analysis and Fix:**

No check for the wall is found while shooting. Therefore, the bullet can spawn on the other side of the wall, ignoring the presence of the wall.



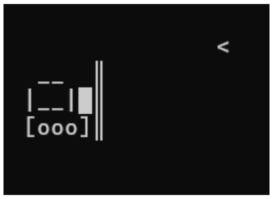
```
if (tank.IsShooting)
{
    tank.Bullet = new Bullet()
    {
        X = tank.Direction switch
        {
            Direction.Left => tank.X - 3,
            Direction.Right => tank.X + 3,
            _ => tank.X,
        },
        Y = tank.Direction switch
        {
            Direction.Up =>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

There needs to be a specific check for the blockage by the wall before spawning the bullet.



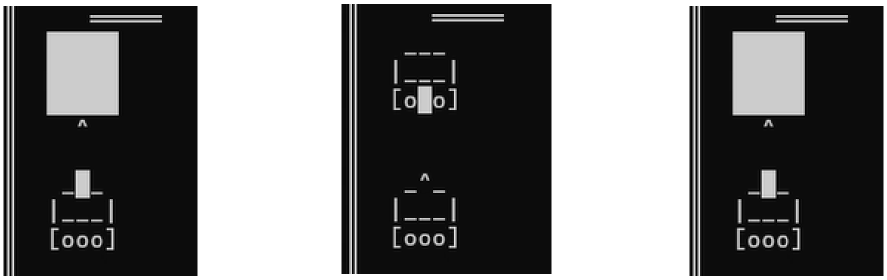
```
if (tank.IsShooting)
{
    int spawnX = tank.Direction switch
    {
        Direction.Left => tank.X - 3,
        Direction.Right => tank.X + 3,
        _ => tank.X,
    };
    int spawnY = tank.Direction switch
    {
        Direction.Up => tank.Y - 2,
        Direction.Down => tank.Y + 2,
        _ => tank.Y,
    };

    bool blocked =
        spawnX <= 0 || spawnX >= 74 ||
        spawnY <= 0 || spawnY >= 27
        || (5 < spawnX && spawnX < 11 && spawnY == 13)
        || (spawnX == 37 && 3 < spawnY && spawnY < 7)
        || (spawnX == 37 && 20 < spawnY && spawnY < 24)
        || (63 < spawnX && spawnX < 69 && spawnY == 13);

    if (!blocked)
    {
        tank.Bullet = new Bullet()
        {
            X = spawnX,
            Y = spawnY,
            Direction = tank.Direction
        };
    }

    tank.IsShooting = false;
    continue;
}
```

If the wall does not block the bullet, then it will spawn otherwise, the bullet won’t spawn.



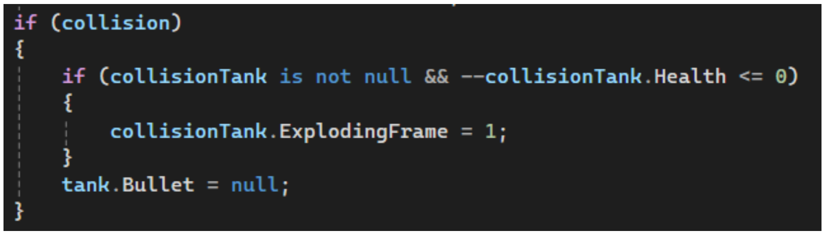
Bug 4:

- **Game:** Tanks
- **Bug:** Dead Tank Remains Visible After Health Depletion (Inherently present in the code)



The shooting goes on infinitely, and the targeted tank doesn't vanish.

This is because of the insufficient condition check before setting the collisionTank.ExplodingFrame, even if the ExplodingFrame is 1, it will

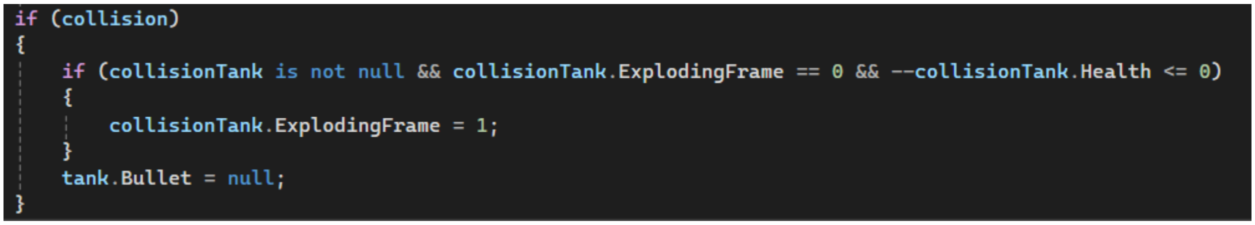


```
if (collision)
{
    if (collisionTank is not null && --collisionTank.Health <= 0)
    {
        collisionTank.ExplodingFrame = 1;
    }
    tank.Bullet = null;
}
```

keep setting the ExplodingFrame to 1, leading to repetitive starting of the ExplodingFrame.

- **Analysis and Fix:**

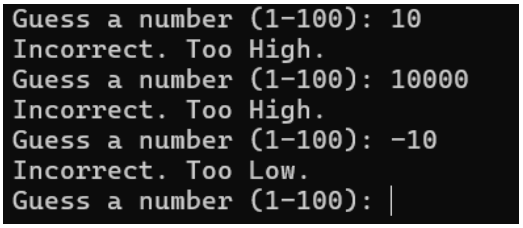
  This can simply be fixed by adding a condition before setting the ExplodingFrame, just check if it is 0, as if it is already set, then there is no point in again setting it to 1, as it will only reset the ExplodingFrame, keeping the process never ending.



```
if (collision)
{
    if (collisionTank is not null && collisionTank.ExplodingFrame == 0 && --collisionTank.Health <= 0)
    {
        collisionTank.ExplodingFrame = 1;
    }
    tank.Bullet = null;
}
```

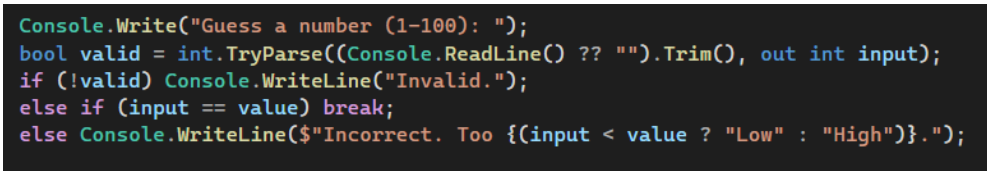
Bug 5:

- **Game:** Guess the Number
- **Bug:** Valid range is (1-100), still other numbers (>100) & (<1) are treated as valid inputs.



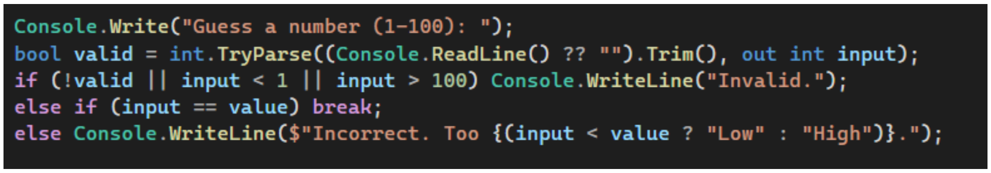
```
Guess a number (1-100): 10
Incorrect. Too High.
Guess a number (1-100): 10000
Incorrect. Too High.
Guess a number (1-100): -10
Incorrect. Too Low.
Guess a number (1-100):
```

The reason is straightforward: there is no check for valid input.



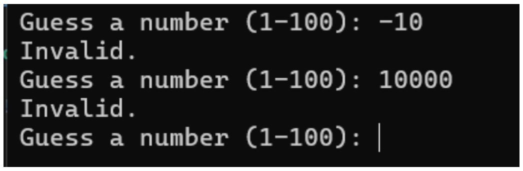
```
Console.Write("Guess a number (1-100): ");
bool valid = int.TryParse((Console.ReadLine() ?? "").Trim(), out int input);
if (!valid) Console.WriteLine("Invalid.");
else if (input == value) break;
else Console.WriteLine($"Incorrect. Too {(input < value ? "Low" : "High")}.");
```

- **Analysis and Fix:**



```
Console.Write("Guess a number (1-100): ");
bool valid = int.TryParse((Console.ReadLine() ?? "").Trim(), out int input);
if (!valid || input < 1 || input > 100) Console.WriteLine("Invalid.");
else if (input == value) break;
else Console.WriteLine($"Incorrect. Too {(input < value ? "Low" : "High")}.");
```

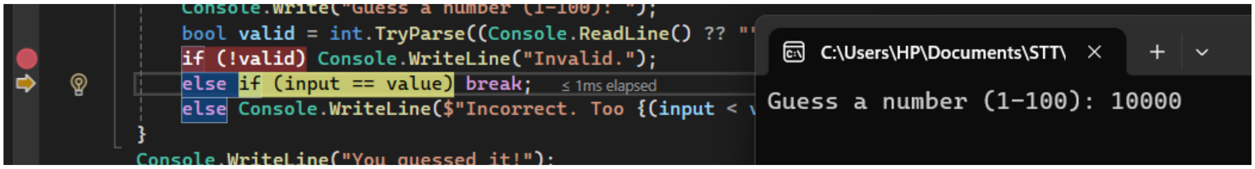
After adding simple validating condition before comparing the input the bug can be fixed.



```
Guess a number (1-100): -10
Invalid.
Guess a number (1-100): 10000
Invalid.
Guess a number (1-100):
```

Illustration of Debug Mode:

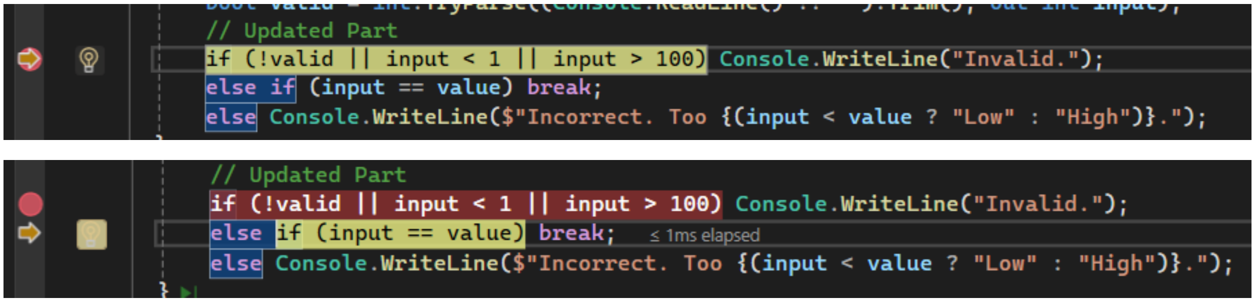
Used the Visual Studio Debugger to step through the program's execution to track down where the input validation was missing. By setting breakpoints and using the step-into/step-over/step-out operations, I could see that the input was being accepted without proper validation and added the necessary condition to fix the issue.



```
    Console.Write("Guess a number (1-100): ");
    bool valid = int.TryParse((Console.ReadLine() ?? "
    if (!valid) Console.WriteLine("Invalid.");
    else if (input == value) break;   ≤ 1ms elapsed
    else Console.WriteLine($"Incorrect. Too {(input < v
}
Console.WriteLine("You guessed it!");
```

```
Guess a number (1-100): 10000
```

During no validation, the invalid input is still going into the next check, which can be checked using step-over or step-into, as both will jump to the next ins. which is shown in the image



```
bool valid = int.TryParse((Console.ReadLine() ?? "").Trim(), out int input);
// Updated Part
if (!valid || input < 1 || input > 100) Console.WriteLine("Invalid.");
else if (input == value) break;
else Console.WriteLine($"Incorrect. Too {(input < value ? "Low" : "High")}.");
```

```
// Updated Part
if (!valid || input < 1 || input > 100) Console.WriteLine("Invalid.");
else if (input == value) break;  ≤ 1ms elapsed
else Console.WriteLine($"Incorrect. Too {(input < value ? "Low" : "High")}.");
}
```

The validation can be done for the input range on changing the if condition, and only the valid numbers will be passed on to the next check.

Results and Analysis

All five bugs were successfully identified and fixed. This included both pre-existing logical bugs and intentionally injected mutations. Using breakpoints and runtime inspection significantly helped trace and correct these issues.

Each bug was documented with screenshots and analyzed regarding its origin and resolution process.

Discussion and Conclusion

Challenges and Reflections

Understanding unfamiliar game code structures required careful analysis of the flow of control. Injecting bugs required creativity to maintain the realism of the crash scenarios. Using the Visual Studio Debugger became easier with repeated practice, especially combining step-into/over/out.

Ques: If there is no Main() method in the program, where exactly is the entry point?

Ans: If a C# console application does not contain a visible Main() method, it’s likely using top-level statements, a feature introduced in C# 9 (available with .NET 5 and later). This feature lets us write code directly in a .cs file without explicitly declaring a Main() method—the compiler automatically generates one behind the scenes.

Learnings

- Debugging tools are crucial for locating the origin of unexpected behaviour in console games.
- Breakpoints and step-based operations simplify tracing program flow.
- Identifying and fixing bugs improves overall code comprehension.

Summary

This lab reinforced debugging as a critical skill in software development. Through hands-on exploration of console games, students gained experience in navigating control flow and resolving bugs using Visual Studio. The exercise demonstrated the power of debugging tools and highlighted practical problem-solving strategies in C# development.

NOTE: The illustration of Step-Into, Step-Out, and Step-Over was documented only for Bug-5, as it was also done in the previous assignment, and again, doing it for all the Bugs is repetitive.

CS202 - Lab Assignment 12

- **Hitesh Kumar**
- **22110098**
- https://github.com/Hit2737/CS202_A3

Introduction, Setup, and Tools

Overview

This lab explores event-driven programming in C# through Windows Forms applications. Its primary goals are to grasp the event-driven model, build a time-triggered alarm system, and show how user actions and changes in the application's state influence the program's behavior.

System Requirements

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

Methodology and Execution

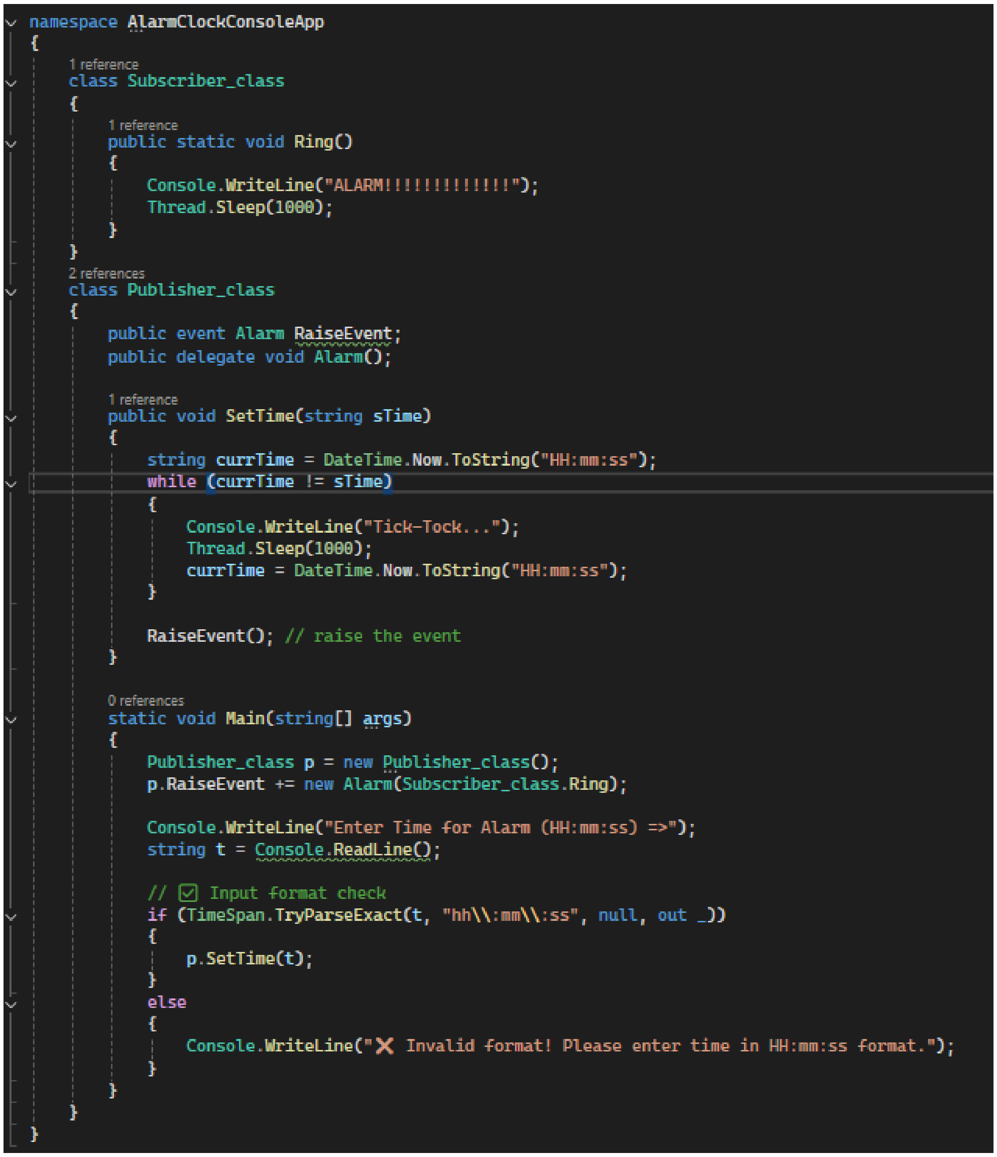
Part 1: Console Application for Alarm

Design

Create a console application that:

- Asks the user to enter a target time in the format HH:MM:SS (24 hour format).
- Raises Error on wrong input format.
- Continuously monitors the current system time and prints `Tick-Tock` until target time is reached
- Prints `ALARM!!!!!!!!!` once target time is reached.

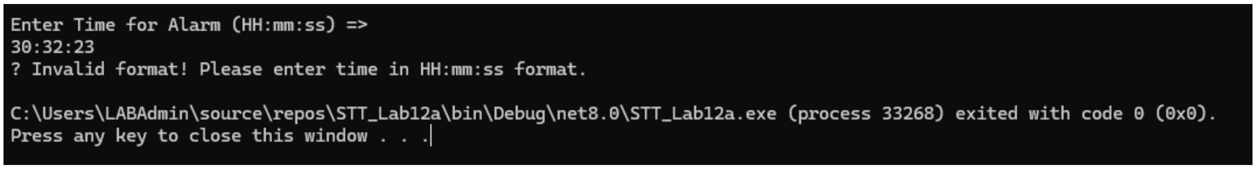
Implementation



```
namespace AlarmClockConsoleApp
{
    1 reference
    class Subscriber_class
    {
        1 reference
        public static void Ring()
        {
            Console.WriteLine("ALARM!!!!!!!!!!!!!");
            Thread.Sleep(1000);
        }
    }
    2 references
    class Publisher_class
    {
        public event Alarm RaiseEvent;
        public delegate void Alarm();

        1 reference
        public void SetTime(string sTime)
        {
            string currTime = DateTime.Now.ToString("HH:mm:ss");
            while (currTime != sTime)
            {
                Console.WriteLine("Tick-Tock...");
                Thread.Sleep(1000);
                currTime = DateTime.Now.ToString("HH:mm:ss");
            }

            RaiseEvent(); // raise the event
        }

        0 references
        static void Main(string[] args)
        {
            Publisher_class p = new Publisher_class();
            p.RaiseEvent += new Alarm(Subscriber_class.Ring);

            Console.WriteLine("Enter Time for Alarm (HH:mm:ss) =>");
            string t = Console.ReadLine();

            // ✅ Input format check
            if (TimeSpan.TryParseExact(t, "hh\\:mm\\:ss", null, out _))
            {
                p.SetTime(t);
            }
            else
            {
                Console.WriteLine("❌ Invalid format! Please enter time in HH:mm:ss format.");
            }
        }
    }
}
```

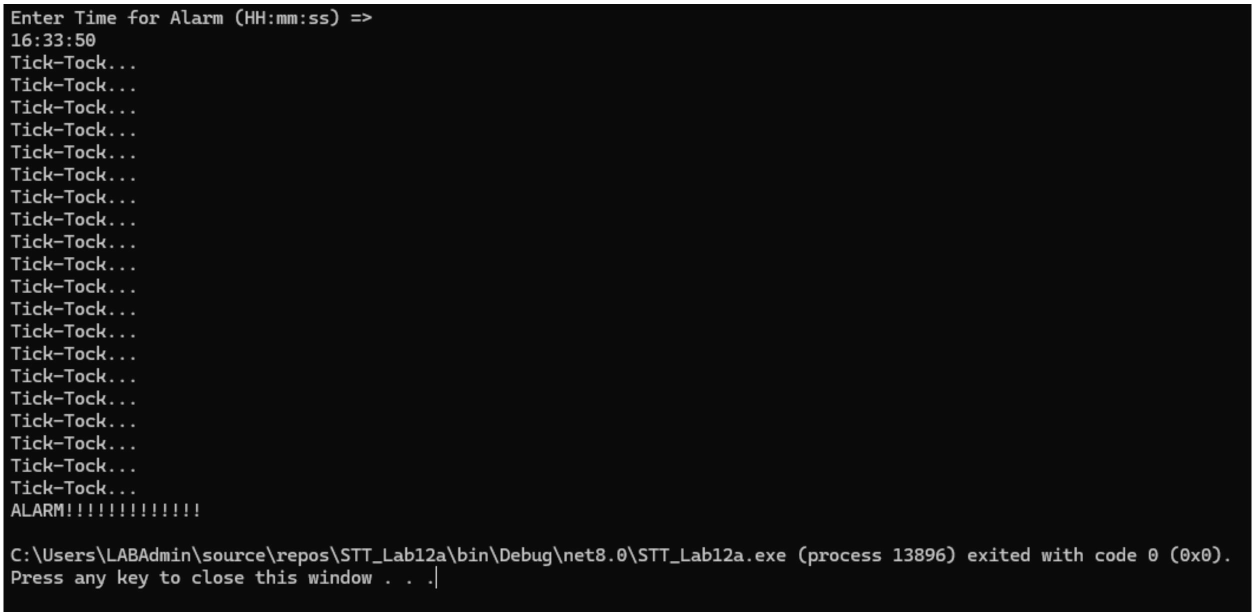
Fig. Code Implementation for the Console App

Output



```
Enter Time for Alarm (HH:mm:ss) =>
30:32:23
? Invalid format! Please enter time in HH:mm:ss format.

C:\Users\LABAdmin\source\repos\STT_Lab12a\bin\Debug\net8.0\STT_Lab12a.exe (process 33268) exited with code 0 (0x0).
Press any key to close this window . . .
```

Fig. Output on Invalid Input



```
Enter Time for Alarm (HH:mm:ss) =>
16:33:50
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
Tick-Tock...
ALARM!!!!!!!!!!!!!

C:\Users\LABAdmin\source\repos\STT_Lab12a\bin\Debug\net8.0\STT_Lab12a.exe (process 13896) exited with code 0 (0x0).
Press any key to close this window . . .
```

Fig. Output on valid Input and Alarm.

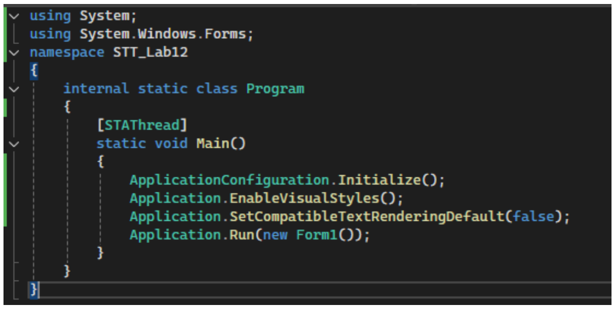
Part 2: Windows Forms App for Event-Driven Alarm

Design

Update the console application to use a Windows Form for all user interaction:

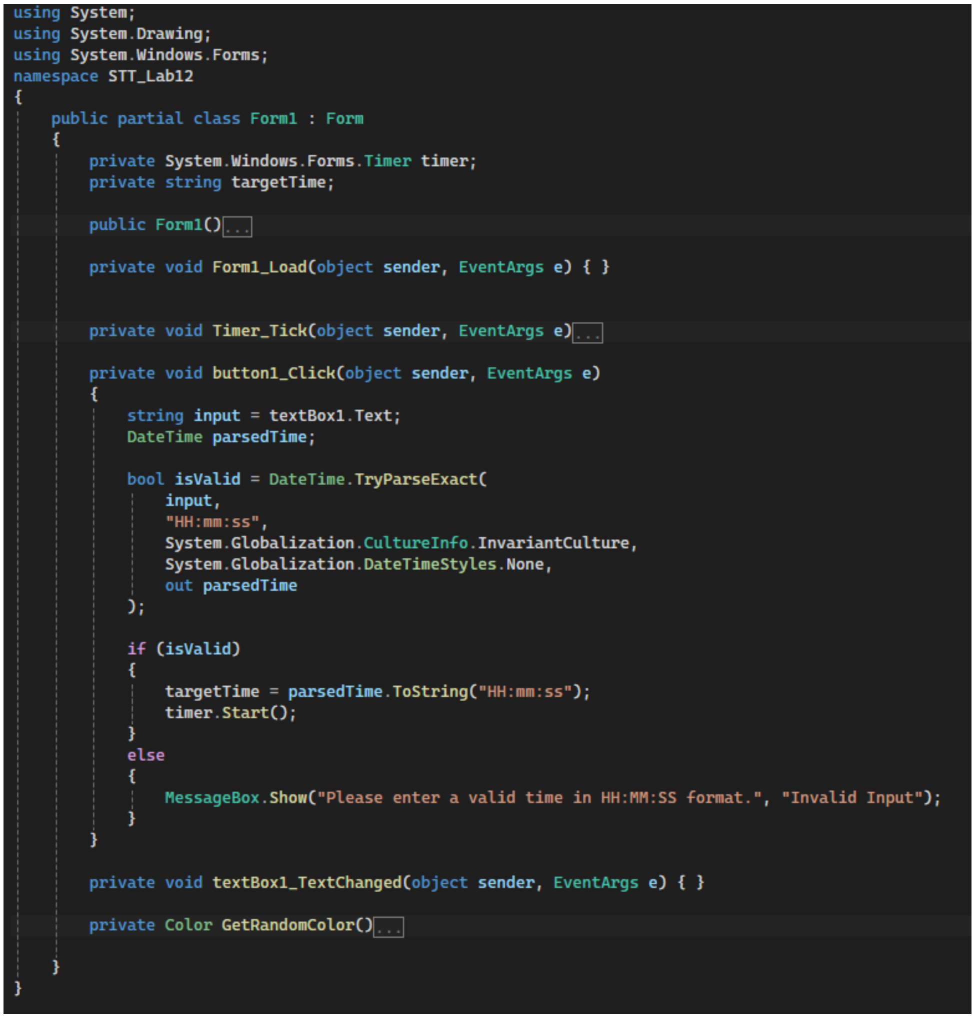
- Include a textbox for entering the target time (validated in HH:MM:SS format) and a start button.
- When the start button is clicked, the form's background colour should change every second.
- Once the current system time reaches the specified target time, the colour change stops and a message box appears.

Implementation



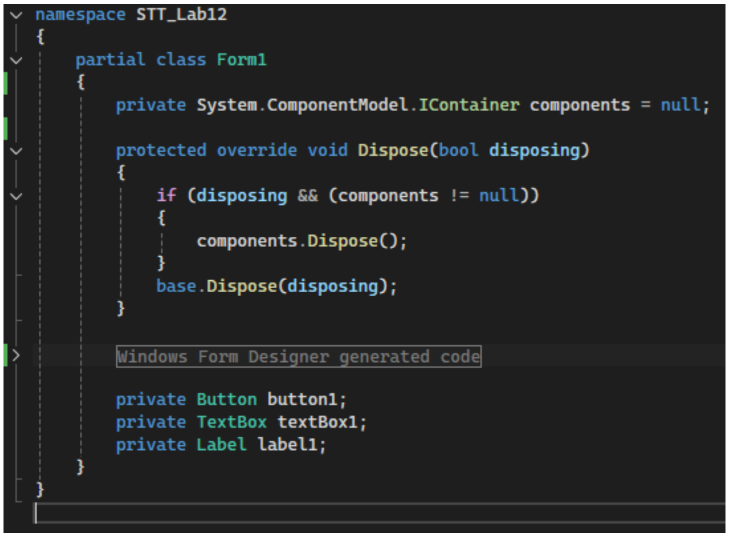
```csharp
using System;
using System.Windows.Forms;
namespace STT_Lab12
{
    internal static class Program
    {
        [STAThread]
        static void Main()
        {
            ApplicationConfiguration.Initialize();
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Fig. Program.cs file (main entry point of the program)



```
using System;
using System.Drawing;
using System.Windows.Forms;
namespace STT_Lab12
{
    public partial class Form1 : Form
    {
        private System.Windows.Forms.Timer timer;
        private string targetTime;

        public Form1()...

        private void Form1_Load(object sender, EventArgs e) { }


        private void Timer_Tick(object sender, EventArgs e)...

        private void button1_Click(object sender, EventArgs e)
        {
            string input = textBox1.Text;
            DateTime parsedTime;

            bool isValid = DateTime.TryParseExact(
                input,
                "HH:mm:ss",
                System.Globalization.CultureInfo.InvariantCulture,
                System.Globalization.DateTimeStyles.None,
                out parsedTime
            );

            if (isValid)
            {
                targetTime = parsedTime.ToString("HH:mm:ss");
                timer.Start();
            }
            else
            {
                MessageBox.Show("Please enter a valid time in HH:MM:SS format.", "Invalid Input");
            }
        }

        private void textBox1_TextChanged(object sender, EventArgs e) { }

        private Color GetRandomColor()...

    }
}
```

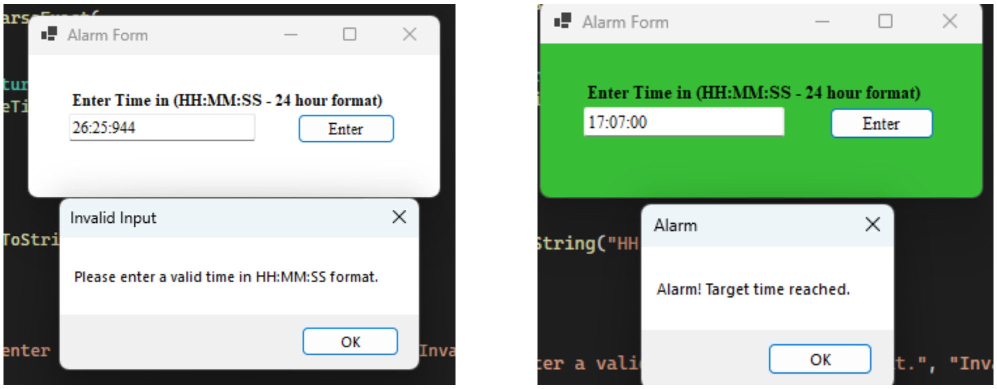
Fig. Form1.cs (C# Code for the functionality of the App)



```
namespace STT_Lab12
{
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;

        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        Windows Form Designer generated code

        private Button button1;
        private TextBox textBox1;
        private Label label1;
    }
}
```

Fig. Form1.Designer.cs (The Design of the Form)

Output



Enter Time in (HH:MM:SS - 24 hour format)

26:25:944

Invalid Input

Please enter a valid time in HH:MM:SS format.

Enter Time in (HH:MM:SS - 24 hour format)

17:07:00

Alarm

Alarm! Target time reached.

Fig. Output Message on Invalid Input Format and on reaching target time

---

Results and Analysis

Part 1: Console Application Results

- The console-based alarm program accurately captures the user-defined target time and continuously monitors the system clock.
- Once the system time matches the input, the code notifies the user by printing `ALARM!!!!!!!` on the console.

Part 2: Windows Forms Application Results

- The Windows Forms version correctly reads and validates the time input from the user interface.
- After initiating the timer, the form background changes color every second.
- When the system time reaches the specified target, the timer halts, and a message box appears to alert the user.

Discussion and Conclusion

Challenges and Reflections

- Gaining familiarity with the event-driven paradigm in Windows Forms involved understanding how events invoke corresponding methods.
- Implementing time-based functionality and managing input validation through a graphical interface presented notable challenges.

Learnings

- Event-driven programming enables dynamic, responsive user interfaces and streamlines application control flow.
- Windows Forms offers a comprehensive toolkit for building interactive desktop applications efficiently.
- Mastering the publisher/subscriber event model is essential for developing event-driven features in C#.

Summary

This lab provided practical experience with event-driven development in C# using both console and GUI applications. By building a functional alarm system in both formats, the exercise highlighted the advantages of event-based design and demonstrated effective debugging and implementation techniques within Visual Studio.