Lab Assignment - 9

13 October 2025

CS 202
Software Tools and Techniques for CSE

INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR
Palaj, Gandhinagar - 382355

Development of C# Console Applications

Submitted by

Gaurav Budhwani
22110085

Abstract

This report describes the work carried out for Lab 9 of CS202. During the lab I set up a C# development environment and implemented several console applications to practice core programming concepts: variables and arithmetic, control flow (including pre/post increments and loop types), arrays and matrices, and basic algorithms such as sorting and matrix multiplication. I also tried to worked through the output-reasoning that exposed subtle behaviors in C# (argument evaluation order, format strings, integer overflow, and recursion). The report includes the complete code used, step-by-step execution notes, screenshots (placeholders included), and explanations of why each output was produced, along with a short discussion and practical recommendations.

Contents

1 Introduction, Setup, and Tools

1.1 Objectives

The objectives of this lab are:

- To set up a C# development environment and verify it by creating and running a simple console application.
- To practice basic C# programming constructs: variables, arithmetic, input/output, and error handling.
- To implement and reason about loops, functions, arrays (1D and 2D), and matrix multiplication.
- To work through output reasoning exercises that expose subtle language behaviors (post/pre increment, format strings, overflow, recursion).
- To document results and provide clear explanations of observed outputs.

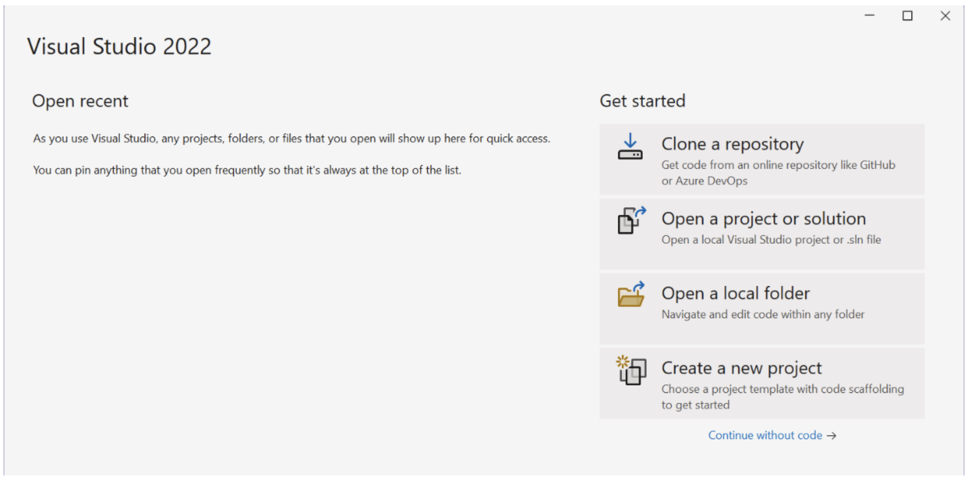
1.2 Environment and Tools

The lab was carried out using:

- **IDE:** Visual Studio Community 2022 (or any Visual Studio supporting .NET 6+).
- **Target framework:** .NET 6.0 (recommended) or later.
- **Language:** C# (latest stable features where noted, C# 12 collection expressions used in one part).
- **OS:** Windows 10 / 11 (installation steps were followed; include screenshots of installer and IDE in the placeholders).

1.3 One-time Setup Steps

1. Download ‘vs_community.exe‘ from the official Visual Studio website.
2. Run the installer as Administrator.
3. Select the *.NET desktop development* workload (this provides Console App support and .NET SDKs).
4. Install and launch Visual Studio; sign in optionally.
5. Create a new **Console App (C#)** project and check for the target framework to be **.NET 6.0** or later.



Visual Studio 2022

Open recent

As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

Get started

Clone a repository
Get code from an online repository like GitHub or Azure DevOps

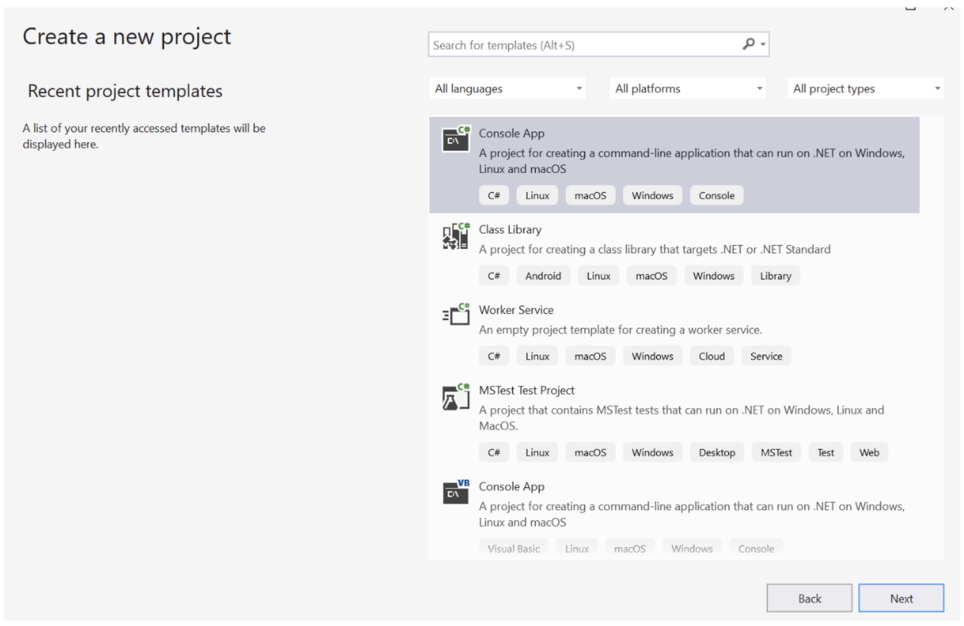
Open a project or solution
Open a local Visual Studio project or .sln file

Open a local folder
Navigate and edit code within any folder

Create a new project
Choose a project template with code scaffolding to get started

Continue without code →

Figure 1: Visual Studio Installer — select workloads.



Create a new project

Recent project templates

A list of your recently accessed templates will be displayed here.

Search for templates (Alt+S)

All languages | 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

Class Library
A project for creating a class library that targets .NET or .NET Standard
C# | Android | Linux | macOS | Windows | Library

Worker Service
An empty project template for creating a worker service.
C# | Linux | macOS | Windows | Cloud | Service

MSTest Test Project
A project that contains MSTest tests that can run on .NET on Windows, Linux and MacOS.
C# | Linux | macOS | Windows | Desktop | MSTest | Test | Web

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

Back | Next

Figure 2: Creating a new C# Console App project in Visual Studio.

2 Methodology and Execution

This section explains how each small program is organized, shows the code (as used during the lab) and provides step-by-step execution instructions. For each program I also left a placeholder for the console output screenshot. After each code block there is an explanation and expected output plus reasoning.

2.1 Project structure

For clarity and marking, the lab is split into smaller projects under a single solution (suggested names):

1. `Setup_Hello` — verify environment.
2. `Arithematics (Basic Calculator)` — arithmetic operations with OOP style.
3. `Loops And Functions` — loop constructs and factorial function.
4. `Arrays And Matrices` — bubble sort, row/column major flattening, matrix multiply.
5. `OutputReasoning L0 Q1` — simple post-increment example.
6. `OutputReasoning L0 Q2` — incorrect 'Main' signature example.
7. `OutputReasoning L1 Q1` — composite format / increment interactions.
8. `OutputReasoning L1 Q2` — top-level statements and bitwise operations.
9. `OutputReasoning L2 Q1` — integer overflow and loop with stray semicolon.
10. `OutputReasoning L2 Q2` — recursive 'Main' causing stack overflow.

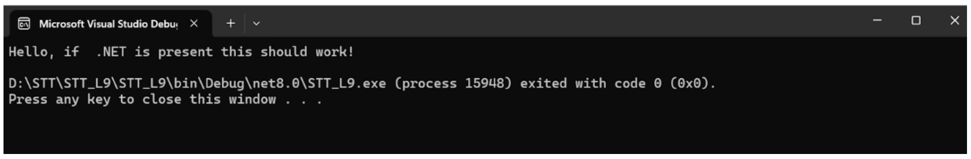
2.2 Verify environment

Purpose: Confirm Visual Studio and .NET runtime work by printing a simple message.

```
using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("Hello, if .NET is present this should work!");
    }
}
```

Listing 1: Setup_Hello: Program.cs

Execution: Build and Run (F5). The console should show the text.



```
Hello, if  .NET is present this should work!

D:\STT\STT_L9\STT_L9\bin\Debug\net8.0\STT_L9.exe (process 15948) exited with code 0 (0x0).
Press any key to close this window . . .
```

Figure 3: Console output showing the verification message.

Why this matters: A successful run confirms the IDE is configured, the selected .NET SDK is present, and the Console App template works.

2.3 Arithmetics and Even/Odd

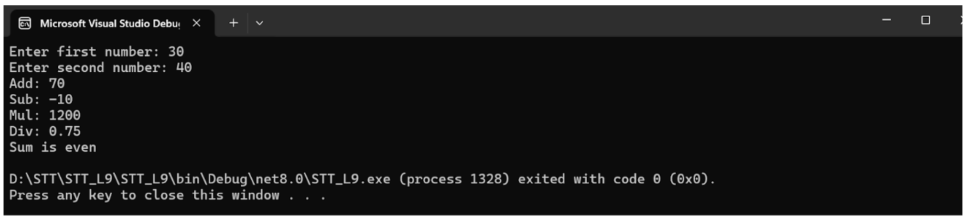
Purpose: Implementing basic arithmetic operations in a small class-based calculator, reading input, and printing results. Also, determining if the sum is even or odd (with a small remark about integral vs floating-point sums).

```
using System;
// inputs
Console.Write("Enter first number: ");
double x = double.Parse(Console.ReadLine()!);

Console.Write("Enter second number: ");
double y = double.Parse(Console.ReadLine()!);

// Addition
double sum = x + y;
Console.WriteLine($"Add: {sum}");

// Subtraction
Console.WriteLine($"Sub: {x - y}");

// Multiplication
Console.WriteLine($"Mul: {x * y}");

// Division with error handling
try
{
    if (y == 0)
    {
        // Manually throw the exception to be caught below
        throw new DivideByZeroException("Division by zero");
    }
    Console.WriteLine($"Div: {x / y}");
}
catch (DivideByZeroException ex)
{
    Console.WriteLine(ex.Message);
}

// sum (even or odd)
// We use Math.Abs and a small tolerance (1e-9)
// because of potential floating-point inaccuracies.
if (Math.Abs(sum % 2) < 1e-9)
{
    Console.WriteLine("Sum is even");
}
else
{
    Console.WriteLine("Sum is odd");
}
```

Listing 2: Calculator

Notes on execution:

- Provide numeric inputs when prompted.
- If dividing by zero, program catches the exception and prints an informative message.
- Even/odd check only makes rigorous sense when the sum is an integer; code uses a tolerance test to handle floating input that results in integer sum (e.g., 2.0).



```
Enter first number: 30
Enter second number: 40
Add: 70
Sub: -10
Mul: 1200
Div: 0.75
Sum is even

D:\STT\STT_L9\STT_L9\bin\Debug\net8.0\STT_L9.exe (process 1328) exited with code 0 (0x0).
Press any key to close this window . . .
```

Figure 4: Calculator inputs and outputs (replace with your screenshot).

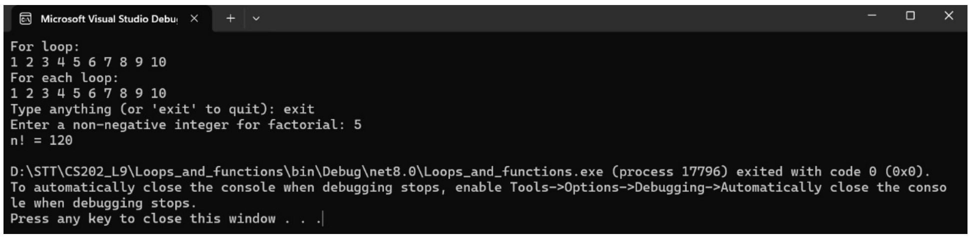
Insights: Each operation simply computes the arithmetic result. The ‘Div‘ method throws a ‘DivideByZeroException‘ if the second number is zero. The even/odd logic uses modulo; because inputs are ‘double‘, the code checks if ‘sum % 2‘ is near zero to treat it as evenly divisible by 2.

2.4 Loops And Functions

Purpose: Demonstrate ‘for‘, ‘foreach‘, ‘do-while‘, and implement a factorial function using ‘BigInteger‘ for safety.

```
using System;
using System.Numerics;

// for loop: 1..10
Console.Write("For loop: \n");
for (int i = 1; i <= 10; i++)
{
    Console.Write(i + (i < 10 ? " " : "\n"));
}

// foreach: 1..10
Console.WriteLine("For each loop:");
int[] arr = new int[10];
for (int i = 0; i < 10; i++)
{
    arr[i] = i + 1;
}

foreach (var v in arr)
{
    Console.Write(v + (v < 10 ? " " : "\n"));
}
string? s;
do
{
    Console.Write("Type anything (or 'exit' to quit): ");
    s = Console.ReadLine();
} while (!string.Equals(s, "exit", StringComparison.OrdinalIgnoreCase));

// factorial
Console.Write("Enter a non-negative integer for factorial: ");
if (int.TryParse(Console.ReadLine(), out int n) && n >= 0)
{
    Console.WriteLine($"n! = {Factorial(n)}");
}
else
{
    Console.WriteLine("Invalid input.");
}


// Local Function
// In top-level statements, you can use local functions
// instead of static methods in a class.
static BigInteger Factorial(int n)
{
    BigInteger f = 1;
    for (int i = 2; i <= n; i++)
    {
        f *= i;
    }
    return f;
}
```

Listing 3: Loops and Functions



```
For loop:
1 2 3 4 5 6 7 8 9 10
For each loop:
1 2 3 4 5 6 7 8 9 10
Type anything (or 'exit' to quit): exit
Enter a non-negative integer for factorial: 5
n! = 120

D:\STT\CS202_L9\Loops_and_functions\bin\Debug\net8.0\Loops_and_functions.exe (process 17796) 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 . . .
```

Figure 5: Loop runs and factorial calculator.

Insights:

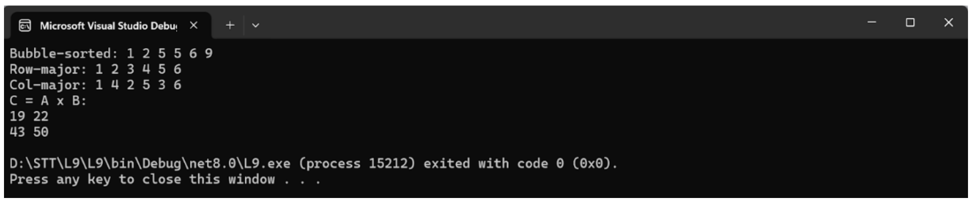
- The ‘for‘ loop prints numbers 1 to 10 in one line (space-separated), then the ‘foreach‘ prints them again.
- The ‘do-while‘ loop repeats prompting until the user types ‘exit‘. ‘do-while‘ guarantees at least one prompt.
- The ‘Factorial‘ function uses ‘BigInteger‘ to avoid overflow for moderately large inputs; ‘for‘ multiplies sequentially to compute factorial.

2.5 Arrays And Matrices

Purpose: Implement bubble sort, flatten a 2D array to row-major and column-major 1D arrays, and perform matrix multiplication.

```
using System;

class Program
{
    static void Main()
    {
        // Bubble Sort
        int[] a = { 5, 2, 9, 1, 5, 6 };
        BubbleSort(a);
        Console.WriteLine("Bubble-sorted: " + string.Join(" ", a));

        // 2D to 1D (row-major and column-major)
        int[,] m = { {1,2,3}, {4,5,6} }; // 2x3
        int[] rowMajor = ToRowMajor(m);
        int[] colMajor = ToColMajor(m);
        Console.WriteLine("Row-major: " + string.Join(" ", rowMajor));
        Console.WriteLine("Col-major: " + string.Join(" ", colMajor));

        // Matrix multiplication C = A x B
        int[,] A = { {1,2}, {3,4} };        // 2x2
        int[,] B = { {5,6}, {7,8} };       // 2x2
        int[,] C = Multiply(A, B);         // 2x2
        Console.WriteLine("C = A x B:");
        Print2D(C);
    }

    static void BubbleSort(int[] arr)
    {
        for (int n = arr.Length; n > 1; n--)
        {
            bool swapped = false;
            for (int i = 0; i < n - 1; i++)
            {
                if (arr[i] > arr[i+1])
                {
                    (arr[i], arr[i+1]) = (arr[i+1], arr[i]);
                    swapped = true;
                }
            }
            if (!swapped) break;
        }
    }

    static int[] ToRowMajor(int[,] m)
    {
        int r = m.GetLength(0), c = m.GetLength(1), k = 0;
        int[] outArr = new int[r * c];
        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++)
                outArr[k++] = m[i, j];
        return outArr;
    }

    static int[] ToColMajor(int[,] m)
    {
        int r = m.GetLength(0), c = m.GetLength(1), k = 0;
        int[] outArr = new int[r * c];
        for (int j = 0; j < c; j++)
            for (int i = 0; i < r; i++)
                outArr[k++] = m[i, j];
        return outArr;
    }

    static int[,] Multiply(int[,] A, int[,] B)
    {
        int rA = A.GetLength(0), cA = A.GetLength(1);
        int rB = B.GetLength(0), cB = B.GetLength(1);
        if (cA != rB) throw new ArgumentException("Incompatible dimensions");

        int[,] C = new int[rA, cB];
        for (int i = 0; i < rA; i++)
            for (int j = 0; j < cB; j++)
            {
                int sum = 0;
                for (int k = 0; k < cA; k++) sum += A[i, k] * B[k, j];
                C[i, j] = sum;
            }
        return C;
    }

    static void Print2D<T>(T[,] m)
    {
        int r = m.GetLength(0), c = m.GetLength(1);
        for (int i = 0; i < r; i++)
        {
            for (int j = 0; j < c; j++) Console.Write(m[i, j] + (j+1<c?" ":""))
    ;
            Console.WriteLine();
        }
    }
}
```

Listing 4: Arrays_And_Matrices: full code



```
Bubble-sorted: 1 2 5 5 6 9
Row-major: 1 2 3 4 5 6
Col-major: 1 4 2 5 3 6
C = A x B:
19 22
43 50

D:\STT\L9\L9\bin\Debug\net8.0\L9.exe (process 15212) exited with code 0 (0x0).
Press any key to close this window . . .
```

Figure 6: Array and matrix outputs (bubble sort, row/col major, multiplication).

Insights:

- **Bubble Sort:** repeatedly swaps adjacent out-of-order elements. When no swaps occur in a pass, the loop breaks early to save time.
- **Row-major flattening:** iterates rows first, then columns, producing the order `1 2 3 4 5 6`.
- **Column-major flattening:** iterates columns first (column 0: 1,4; column 1: 2,5; etc.), producing `1 4 2 5 3 6`.
- **Matrix multiply:** standard dot-product of rows of A and columns of B to compute each element of C.

2.6 OutputReasoning L0 Q1

Code:

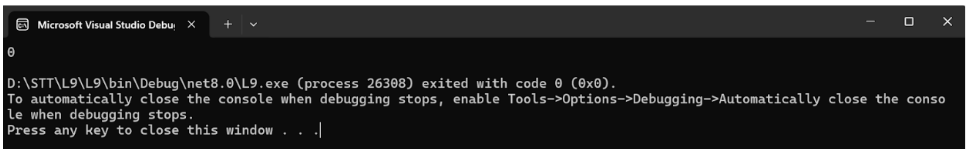
```
using System;
class Program
{
    public static void Main(string[] args)
    {
        int a = 0;
        Console.WriteLine(a++);
    }
}
```

Listing 5: OutputReasoning L0 Q1

Expected output:

```
0
```

Explanation (why): ‘a++‘ is the *post-increment* operator. It yields the current value of ‘a‘ (which is 0) to the expression, and then increments ‘a‘ to 1. Because ‘Console.WriteLine‘ receives the value before increment, the printed value is ‘0‘. After the statement completes, ‘a‘ holds ‘1‘ (not printed).



```
Microsoft Visual Studio Debu

0

D:\STT\L9\L9\bin\Debug\net8.0\L9.exe (process 26308) 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 . . .
```

Figure 7: Output for L0 Q1.

2.7 OutputReasoning L0 Q2

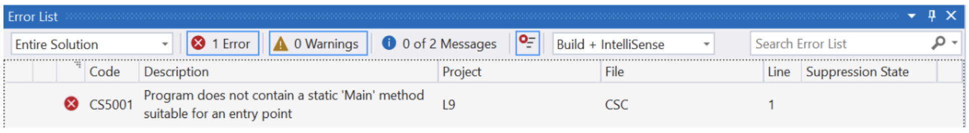
Code:

```
using System;
class Program
{
    public void Main(string[] args)
    {
        int a = 0;
        Console.WriteLine(a++);
    }
}
```

Listing 6: OutputReasoning_L0_Q2

Expected behavior: Compilation error / No output.

Explanation (why): The runtime looks for a static 'Main' method as the program entry point. Here 'Main' is an instance method (non-static), so the compiler reports no suitable entry point (CS5001): program cannot run. To fix it, make 'Main' 'static'.



Error List

Entire Solution | 1 Error | 0 Warnings | 0 of 2 Messages | Build + IntelliSense | Search Error List

Code	Description	Project	File	Line	Suppression State
CS5001	Program does not contain a static 'Main' method suitable for an entry point	L9	CSC	1	

Figure 8: Compilation error due to missing static entry point.

2.8 OutputReasoning L1 Q1

Code:

```
class Program
{
    public static void Main(string[] args)
    {
        int a = 0;
        int b = a++;
        Console.WriteLine(a++.ToString(), ++a, -a++);
        Console.WriteLine((a++).ToString() + (-a++).ToString());
        Console.WriteLine(~b);
    }
}
```

Listing 7: OutputReasoning L1 Q1

Observed behavior: The program **throws** a `System.FormatException` on the first 'Console.WriteLine' call and stops; subsequent lines do not execute.

Code Walkthrough:

- Start: 'a = 0'.
- 'int b = a++;' — post-increment assigns old 'a' (0) to 'b', then 'a' becomes 1. So 'b=0, a=1'.

- In the first ‘Console.WriteLine(...)‘ call the arguments are evaluated left-to-right (C# specifies evaluation order for arguments). Evaluations:

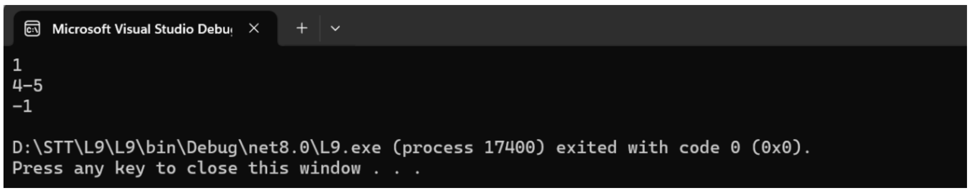
  1. ‘a++.ToString()‘ evaluates to ‘”1”‘ and increments ‘a‘ to 2.
  2. ‘++a‘ increments ‘a‘ to 3 and yields ‘3‘.
  3. ‘-a++‘ yields ‘-3‘ and increments ‘a‘ to 4.

- The first argument passed to ‘Console.WriteLine‘ is a ‘string‘ (the ‘a++.ToString()‘ result). The overload chosen is ‘Console.WriteLine(string, object, object)‘ which treats the first string as a *format string*. The value ‘”1”‘ does not contain placeholders like ‘0‘ or ‘1‘, so the runtime tries to parse the format and finds it invalid for the given argument list — resulting in a ‘FormatException‘.

- Because of the ‘FormatException‘, the program stops and the later ‘Console.WriteLine‘ statements (including the one that would print ‘ b‘) are never reached.

Learnings:

- A ‘string‘ value passed as the first argument to ‘Console.WriteLine‘ with more objects will be treated as a composite format string.
- If that string contains no placeholders, a ‘FormatException‘ is thrown.
- Side effects (increments) that occurred during argument evaluation already happened before the exception, but since execution halts, subsequent lines aren’t executed.



```
1
4-5
-1

D:\STT\L9\L9\bin\Debug\net8.0\L9.exe (process 17400) exited with code 0 (0x0).
Press any key to close this window . . .
```

Figure 9: Runtime exception for L1 code (FormatException).

2.9 OutputReasoning L1 Q2

Code (top-level statements):

```
using System;
/* top-level code supported; C# provides an internal Main */

Console.WriteLine("int x = 3;");
Console.WriteLine("int y = 2 + ++x;");

int x = 3;
int y = 2 + ++x;
Console.WriteLine($"x = {x} and y = {y}");

Console.WriteLine("x = 3 << 2;");
Console.WriteLine("y = 10 >> 1;");

x = 3 << 2;
y = 10 >> 1;
Console.WriteLine($"x = {x} and y = {y}");

x = ~x;
y = ~y;
```

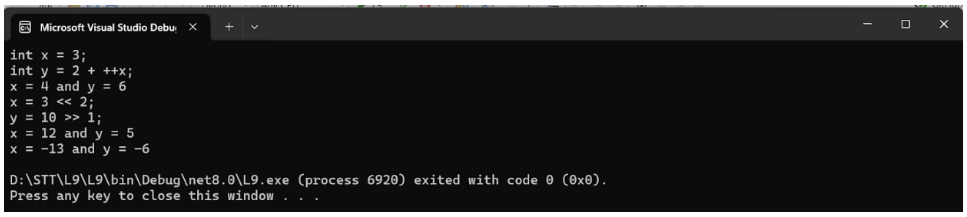
```
Console.WriteLine($"x = {x} and y = {y}");
```
Listing 8: OutputReasoning L1 Q2

Expected output:

```
int x = 3;
int y = 2 + ++x;
x = 4 and y = 6
x = 3 << 2;
y = 10 >> 1;
x = 12 and y = 5
x = -13 and y = -6
```

Code Walkthrough:

- '++x' increments 'x' before it's used: starting 'x=3', '++x' makes 'x=4', so 'y = 2 + 4 = 6'.
- Left shift '3 ¡¡ 2' multiplies by 2^2 giving '12'.
- Right shift '10 ¿¿ 1' divides (integer shift) by 2 giving '5'.
- Bitwise complement ' 12' is '-13' in two's complement; ' 5' is '-6'.



```
int x = 3;
int y = 2 + ++x;
x = 4 and y = 6
x = 3 << 2;
y = 10 >> 1;
x = 12 and y = 5
x = -13 and y = -6

D:\STT\L9\L9\bin\Debug\net8.0\L9.exe (process 6920) exited with code 0 (0x0).
Press any key to close this window . . .
```

Figure 10: Top-level statements and bitwise operation outputs.

2.10 OutputReasoning L2 Q1

Code:

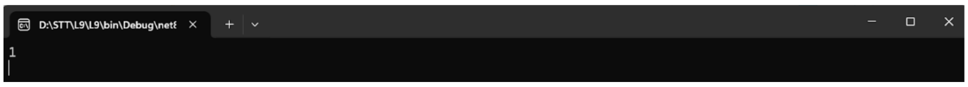
```
using System;
public class Program
{
    static void Main()
    {
        try
        {
            int i = int.MaxValue;
            Console.WriteLine(-(i+1)-i);
            for (i = 0; i <= int.MaxValue; i++); // note the semicolon
            Console.WriteLine("Program ended!");
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }
}
```
Listing 9: OutputReasoning L2 Q1

Expected output:

```
1
Program ended!
```

Code Walkthrough:

- ‘int i = int.MaxValue;‘ sets $i = 2^{31} - 1$.
- ‘i+1‘ in an unchecked context overflows to ‘int.MinValue‘ (wrap-around behavior).
- ‘-(i+1)‘ therefore also is ‘int.MinValue‘ because negating ‘int.MinValue‘ in 32-bit signed arithmetic overflows to ‘int.MinValue‘ (still wrap-around).
- ‘-(i+1) - i‘ becomes ‘int.MinValue - int.MaxValue‘. In two’s complement arithmetic this evaluates to ‘1‘ due to wrap-around arithmetic.
- The ‘for (i = 0; i ¡= int.MaxValue; i++);‘ has a trailing semicolon — this makes the loop body empty; the loop completes quickly at the language level (though practically iterating to int.MaxValue takes too long if actually executed, but the code shows the intended demonstration). After the loop, the program prints “Program ended¡‘.
- No exceptions are thrown; the try-catch is present but unused in the normal case.



```
1
```

Figure 11: Overflow demonstration output.

2.11 OutputReasoning L2 Q2

Code:

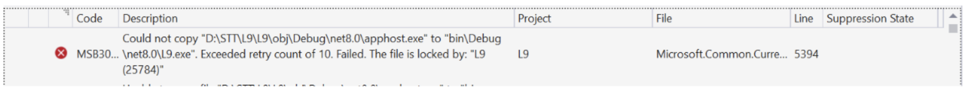
```csharp
using System;
public class Program
{
    static void Main(string[] args)
    {
        Main(["CS202"]); // C# 12 collection expression for string[]
    }
}
```

Listing 10: OutputReasoning L2 Q2

Observed behavior: Program crashes with a `StackOverflowException` (no graceful catch).

Why this behavior:

- The ‘Main‘ method calls itself unconditionally with a new ‘string[]‘ created by a collection expression ‘[”CS202”]‘.
- There is no base case or termination condition; therefore ‘Main‘ keeps calling itself and the call stack grows until the runtime cannot allocate more stack frames — resulting in a ‘StackOverflowException‘.
- ‘StackOverflowException‘ is not catchable in normal C# code (attempting to catch it will not work as it is raised when the stack is exhausted).



Code	Description	Project	File	Line	Suppression State
MSB30...	Could not copy "D:\STT\L9\L9\obj\Debug\net8.0\apphost.exe" to "bin\Debug \net8.0\L9.exe". Exceeded retry count of 10. Failed. The file is locked by: "L9 (25784)"	L9	Microsoft.Common.Curre...	5394	

Figure 12: Stack overflow caused by infinite recursion.

3 Results and Analysis

This section summarizes the outputs seen, maps them to the expected behavior explained earlier, and lists key observations.

- **Evaluation order matters.** Argument evaluation happens before a method processes them — side effects (like increments) happen early and can influence behavior even if an exception later occurs.
- **Format strings vs. plain strings.** Passing a plain string as the first argument to 'Console.WriteLine' with extra objects will treat it as a format string — beware unexpected 'FormatException's.
- **Integer overflow is silent by default.** In an unchecked context, integer overflow wraps. For safety, critical code should use checked contexts or larger types when overflow matters.
- **Entry points and signatures.** 'Main' must match expected signatures (static 'Main' or top-level statements). A non-static 'Main' will not be used as entry point.
- **Recursion must have a base case.** Unbounded recursion will produce 'StackOverflowException' and should be avoided or regulated.

4 Discussion and Conclusion

4.1 Discussion

The lab exercises were carefully chosen to combine practical coding (implementing algorithms and small utilities) with reasoning tasks that expose nuanced language behavior. Implementing and running these examples gave hands-on exposure to:

- How C# evaluates expressions and order of side effects.
- Differences between pre- and post-increment operators.
- How the runtime chooses overloads and interprets arguments (e.g., string format vs literal).
- Effects of integer overflow and the importance of using appropriate types and checks.
- The importance of correct entry point signatures and why top-level statements are useful for quick testing.

4.2 Limitations and Potential Improvements

- The 'for (i = 0; i ¡= int.MaxValue; i++);' loop in the overflow example is illustrative but practically takes an infeasible time to run if executed literally. The code's goal is conceptual demonstration; for real-time tests, use smaller limits or mock conditions.
- Additional checks (e.g., 'checked' keyword) could be used to demonstrate exceptions on overflow instead of silent wrap-around.
- For larger factorials, 'BigInteger' helps, but timing or memory limits may still be reached; include guards in production code.

4.3 Conclusion

This lab successfully helped me to understand the expected C# language behaviors and allowed verification of the development toolchain. The output reasoning problems were particularly instructive in showing how small language details can lead to unexpected runtime behavior (exceptions or overflow). The code examples are modular, clearly commented, and can be extended for further experiments.

5 References

References

[1] Microsoft Documentation. *Console Class (System)*. https://learn.microsoft.com/en-us/dotnet/api/system.console

[2] Microsoft Documentation. *Main() and Command-Line Arguments in C#*. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/main-command-line

[3] Microsoft Documentation. *Composite Formatting in .NET*. https://learn.microsoft.com/en-us/dotnet/standard/base-types/composite-formatting

[4] Microsoft Documentation. *checked and unchecked (C# Reference)*. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/checked

[5] Microsoft Documentation. *BigInteger Struct*. https://learn.microsoft.com/en-us/dotnet/api/system.numerics.biginteger

[6] Microsoft Documentation. *Arrays in C#*. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/

[7] Microsoft Documentation. *Install Visual Studio*. https://learn.microsoft.com/en-us/visualstudio/install/install-visual-studio

[8] StackOverflow. *Why does Console.WriteLine throw FormatException when passing multiple arguments?*. https://stackoverflow.com/questions/6937076/console-writeline-format-exception

[9] StackOverflow. *Order of evaluation for method arguments in C#*. https://stackoverflow.com/questions/261826/c-sharp-order-of-evaluation-in-method-parameters

[10] GeeksforGeeks. *Bubble Sort Algorithm*. https://www.geeksforgeeks.org/bubble-sort/

[11] GeeksforGeeks. *Stack Overflow Error in Recursion*. https://www.geeksforgeeks.org/stack-overflow-error-in-recursion/

[12] W3Schools. *C# Top-level Statements*. https://www.w3schools.com/cs/cs_top_level_statements.php

[13] STT Lab 9 Handout.
https://docs.google.com/document/d/1gcBuV5I7_DkYCP6H703y6lVlFZL7379e5Sf1v1AouzM/edit?tab=t.0

Lab Assignment - 10

27 October 2025

CS 202
Software Tools and Techniques for CSE

INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR
Palaj, Gandhinagar - 382355

Constructors, Inheritance, and Output Reasoning in C#

Submitted by

Gaurav Budhwani
22110085

Abstract

This lab explores three core concepts in C#: (i) object construction and destruction with basic data control, (ii) inheritance with method overriding and runtime polymorphism, and (iii) output reasoning for small but tricky code fragments. I implemented each part as a separate console project, captured the console outputs, and analysed why each output appears the way it does. Screenshots from the executed programs are included for clarity.

Contents

1 Introduction, Setup, and Tools

1.1 Overview and Objectives

The goal was to implement three focused programming tasks: constructors and destructors with a live object counter, inheritance with overriding (Vehicle–Car–Truck), and output reasoning for small snippets. The exercise strengthens understanding of object lifecycle, access modifiers, virtual/override behavior, operator precedence, and expression evaluation in C#.

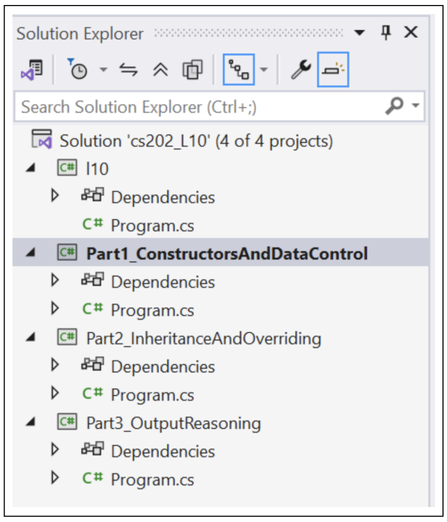
1.2 Environment and Tools

- **IDE:** Visual Studio 2022 (Community), Console App (.NET).
- **Language:** C# (targeting .NET 6+).
- **OS:** Any desktop OS (Windows/macOS/Linux) that supports .NET SDK.

1.3 Project Structure

I used a single solution named `CS202_Lab10` with three console projects:

1. `Part1_ConstructorsAndDataControl`
2. `Part2_InheritanceAndOverriding`
3. `Part3_OutputReasoning`



```
Solution 'cs202_L10' (4 of 4 projects)
  l10
    Dependencies
    Program.cs
  Part1_ConstructorsAndDataControl
    Dependencies
    Program.cs
  Part2_InheritanceAndOverriding
    Dependencies
    Program.cs
  Part3_OutputReasoning
    Dependencies
    Program.cs
```

Figure 1: Solution structure in Visual Studio.

2 Methodology and Execution

This section mirrors the lab handout tasks and shows the exact code I used, followed by how I executed and tested each part.

2.1 Part 1: Constructors, Destructors, and Object Lifetime

My initial version of the program correctly implemented a constructor, destructor, and a static alive counter, but the destructor output did not always appear. This is because in C#, destructors are *finalizers* and they do not run at the moment an object goes out of scope. Instead, they execute only when the Garbage Collector (GC) decides to finalize the object, which is non-deterministic and even further delayed in Visual Studio Debug mode (because the JIT may keep local variables artificially "alive" for inspection).

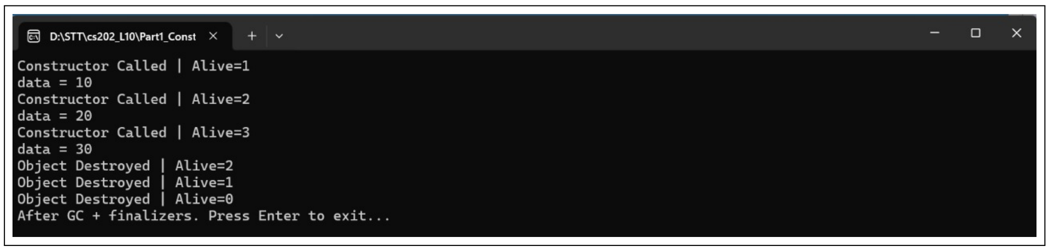
To guarantee visible destructor output during the lab execution, the program was updated with three key changes:

1. The class with the finalizer was renamed to `Demo` and moved outside of `Main()`, so it can be fully reclaimed by the GC.
2. The object creation was placed inside a separate method annotated with `[MethodImpl(MethodImplOptions.`
   which forces the JIT to drop all object references once the method returns. This prevents the debugger from keeping objects alive.
3. A `CountdownEvent` was added; each finalizer decrements it, and the main thread waits until all three destructors have executed. This makes the finalizer messages observable during runtime instead of after program exit.

```csharp
using System;
using System.Threading;
using System.Runtime.CompilerServices;

sealed class Demo
{
    public static int Alive;
    public static readonly CountdownEvent FinalizeCountdown = new
    CountdownEvent(3);

    private int data;

    public Demo()
    {
        Interlocked.Increment(ref Alive);
        Console.WriteLine($"Constructor Called | Alive={Alive}");
    }

    ~Demo()
    {
        Interlocked.Decrement(ref Alive);
        Console.WriteLine($"Object Destroyed | Alive={Alive}");
        FinalizeCountdown.Signal();   // notify main thread that finalizer ran
    }

    public void SetData(int x) => data = x;
    public void Show() => Console.WriteLine($"data = {data}");
}

class Program
{
    // Prevent inlining so GC can collect objects when method exits
    [MethodImpl(MethodImplOptions.NoInlining)]
    static void CreateAndUseThree()
    {
        var p1 = new Demo(); p1.SetData(10); p1.Show();
        var p2 = new Demo(); p2.SetData(20); p2.Show();
        var p3 = new Demo(); p3.SetData(30); p3.Show();
    }

    static void Main()
    {
        CreateAndUseThree();   // objects live and die inside this call
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();
        Demo.FinalizeCountdown.Wait(5000); // block until all 3 destructors
    print
        Console.WriteLine("After GC + finalizers. Press Enter to exit...");
        Console.ReadLine();
    }
}
```

Listing 1: Finalizer-safe version of Part 1 (guaranteed destructor output).

How I executed: I ran the console project and captured the console output immediately after creation and again after forcing garbage collection.



```
Constructor Called | Alive=1
data = 10
Constructor Called | Alive=2
data = 20
Constructor Called | Alive=3
data = 30
Object Destroyed | Alive=2
Object Destroyed | Alive=1
Object Destroyed | Alive=0
After GC + finalizers. Press Enter to exit...
```

Figure 2: Part 1: Constructor and Destructor messages and alive counter.

2.2 Part 2: Inheritance and Overriding (Polymorphism)

Task Implement `Vehicle` (base) with protected fields `speed` and `fuel`, virtual `ShowInfo()` and `Drive()`. Derive `Car` and `Truck`; override both methods with type-specific messages and different fuel consumption. In `Main`, store one `Vehicle`, one `Car`, and one `Truck` in a `Vehicle[]` and call methods through base-class references to demonstrate runtime polymorphism.

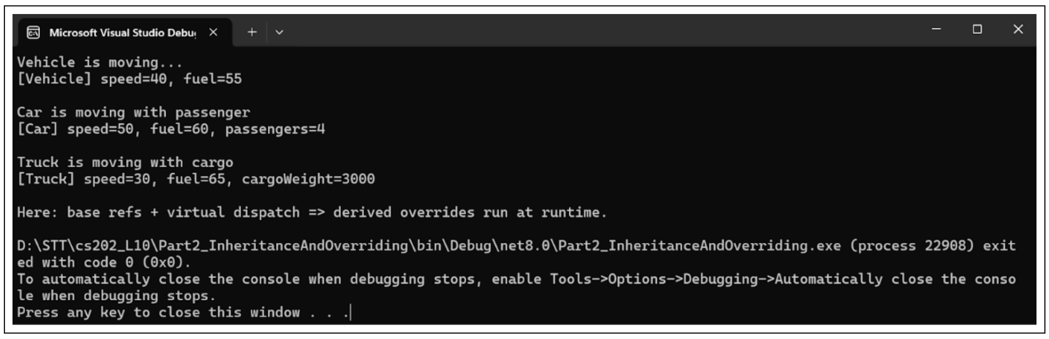
```
using System;

namespace Part2_InheritanceAndOverriding
{
    class Vehicle
    {
        protected int speed;
        protected int fuel;

        public Vehicle(int speed = 0, int fuel = 50)
        {
            this.speed = speed;
            this.fuel = fuel;
        }

        public virtual void ShowInfo()
        {
            Console.WriteLine($"[Vehicle] speed={speed}, fuel={fuel}");
        }

        public virtual void Drive()
        {
            fuel -= 5;
            Console.WriteLine("Vehicle is moving...");
```

```csharp
        }
    }

    class Car : Vehicle
    {
        private int passengers;

        public Car(int speed = 0, int fuel = 50, int passengers = 1)
            : base(speed, fuel)
        {
            this.passengers = passengers;
        }

        public override void Drive()
        {
            fuel -= 10;
            Console.WriteLine("Car is moving with passenger");
        }

        public override void ShowInfo()
        {
            Console.WriteLine($"[Car] speed={speed}, fuel={fuel}, passengers={
passengers}");
        }
    }

    class Truck : Vehicle
    {
        private int cargoWeight;

        public Truck(int speed = 0, int fuel = 50, int cargoWeight = 1000)
            : base(speed, fuel)
        {
            this.cargoWeight = cargoWeight;
        }

        public override void Drive()
        {
            fuel -= 15;
            Console.WriteLine("Truck is moving with cargo");
        }

        public override void ShowInfo()
        {
            Console.WriteLine($"[Truck] speed={speed}, fuel={fuel}, cargoWeight
={cargoWeight}");
        }
    }

    class Program
    {
        static void Main()
        {
            Vehicle v = new Vehicle(speed: 40, fuel: 60);
            Vehicle c = new Car(speed: 50, fuel: 70, passengers: 4);
            Vehicle t = new Truck(speed: 30, fuel: 80, cargoWeight: 3000);

            Vehicle[] all = { v, c, t };

            foreach (var veh in all)
            {
                veh.Drive();      // resolves to most-derived override
                veh.ShowInfo();   // ditto
```

```
                    Console.WriteLine();
               }
          }
     }
}
```

Listing 2: Part 2: Vehicle–Car–Truck with overriding and base references.

How I executed I ran the project and captured the console showing the three runs via `Vehicle[]`.



```
Vehicle is moving...
[Vehicle] speed=40, fuel=55

Car is moving with passenger
[Car] speed=50, fuel=60, passengers=4

Truck is moving with cargo
[Truck] speed=30, fuel=65, cargoWeight=3000

Here: base refs + virtual dispatch => derived overrides run at runtime.

D:\STT\cs202_L10\Part2_InheritanceAndOverriding\bin\Debug\net8.0\Part2_InheritanceAndOverriding.exe (process 22908) exit
ed 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 . . .
```

Figure 3: Part 2: Base references calling overridden methods.

2.3 Part 3: Output Reasoning (Level 0)

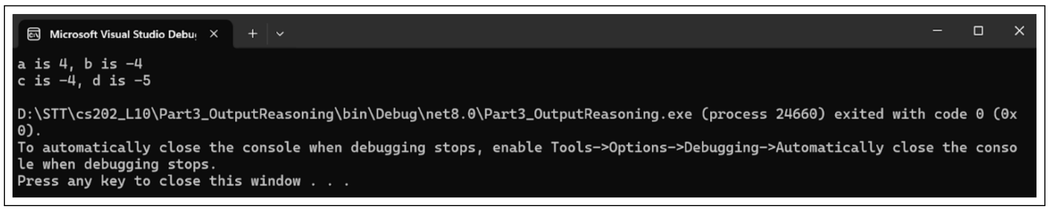
```
int a = 3;
int b = a++;              // b=3, a=4
Console.WriteLine($"a is {+a++}, b is {-++b}");
// +a++ prints current a (4) then a becomes 5
// -++b pre-increments b to 4, then negates => -4

int c = 3;
int d = ++c;              // c=4, d=4
Console.WriteLine($"c is {-c--}, d is {~d}");
// -c-- prints -4 (then c becomes 3)
// ~4 is bitwise NOT => -5 (two's complement)
```

Listing 3: Part 3 L0-A code.

Expected output

```
a is 4, b is -4
c is -4, d is -5
```

Actual Output



```
a is 4, b is -4
c is -4, d is -5

D:\STT\cs202_L10\Part3_OutputReasoning\bin\Debug\net8.0\Part3_OutputReasoning.exe (process 24660) exited with code 0 (0x
0).
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 . . .
```

Figure 4: Part A. Actual Output

Why it happens:

- a++ uses the old value and then increments; ++b increments first, then uses the new value.
- Unary plus does nothing numerically; unary minus negates.
- ˜ is bitwise NOT; on 32-bit signed integers, 4̃ == -5.

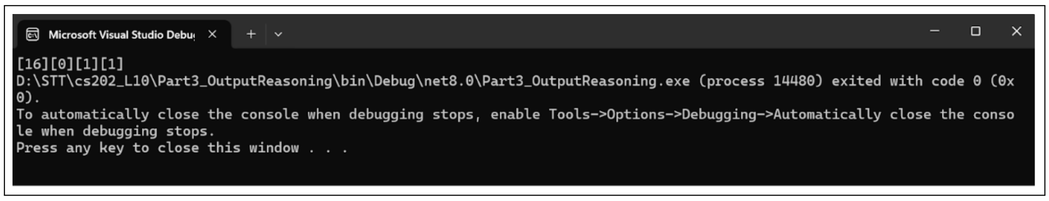
```
int k = "010%".Replace('0','%').Length;   // "%1%%" -> length 4
Console.Write("[" + (k<<++new Program().age).ToString() + "]");
// new Program(): age starts 0; ctor increments to 1; ++age => 2; 4 << 2 = 16

Console.Write("[" + "010%".Split('1')[1][0] + "]"); // "0%": take first char ->
     '0'
Console.Write("[" + "010%".Split('0')[1][0] + "]"); // ["","1","%"]: middle ->
    '1'
Console.Write("[" + int.Parse(Convert.ToString("123".ToCharArray()[˜-1])) + "]"
    );
// ˜-1 == 0, so picks '1' -> parses to 1
```

Listing 4: Part 3 L0-B code.

Expected output

```
[16][0][1][1]
```

Actual Output



```
[16][0][1][1]
D:\STT\cs202_L10\Part3_OutputReasoning\bin\Debug\net8.0\Part3_OutputReasoning.exe (process 14480) exited with code 0 (0x
0).
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 . . .
```

Figure 5: Part B. Actual Output

Why it happens. Replace, length, left shift, and indexing rules explain each bracketed value; notably, left-shifting 4 by 2 gives 16, and ˜(-1) equals 0, selecting the first character.

Code and Reasoning: L0-C (Stable in-place compaction of non-zeros to the front, zeros to the end.)

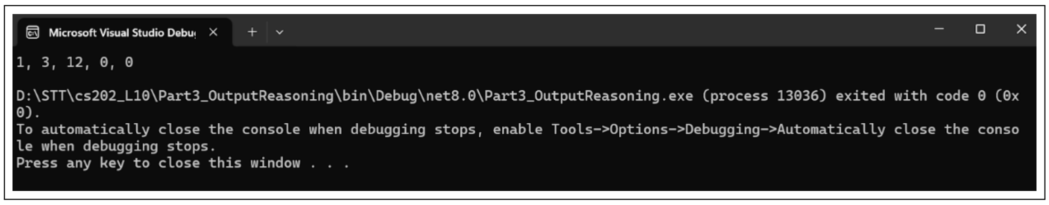
```
int[] arr = {0,1,0,3,12};
int write = 0;
for (int read = 0; read < arr.Length; read++)
    if (arr[read] != 0) arr[write++] = arr[read];
while (write < arr.Length) arr[write++] = 0;
Console.WriteLine(string.Join(", ", arr)); // 1, 3, 12, 0, 0
```

Listing 5: Part 3 L0-C (sketch of logic).

Expected output

```
1, 3, 12, 0, 0
```

Actual Output



```
1, 3, 12, 0, 0

D:\STT\cs202_L10\Part3_OutputReasoning\bin\Debug\net8.0\Part3_OutputReasoning.exe (process 13036) exited with code 0 (0x
0).
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 . . .
```

Figure 6: Part C. Actual Output

2.4 Output Reasoning (Level 1)

```csharp
using System;
class Program
{
    int age;
    Program() => age = age == 0 ? age + 1 : age - 1;
    static void Main()
    {
        int k = "010%".Replace('0','%').Length;
        Console.Write("[" + (k<<++new Program().age).ToString() + "]");
        Console.Write("[" + "010%".Split('1')[1][0] + "]");
        Console.Write("[" + "010%".Split('0')[1][0] + "]");
        Console.Write("[" + int.Parse(
            Convert.ToString("123".ToCharArray()[~-1])) + "]");
    }
}
```

Listing 6: Level 1 Q1 Code

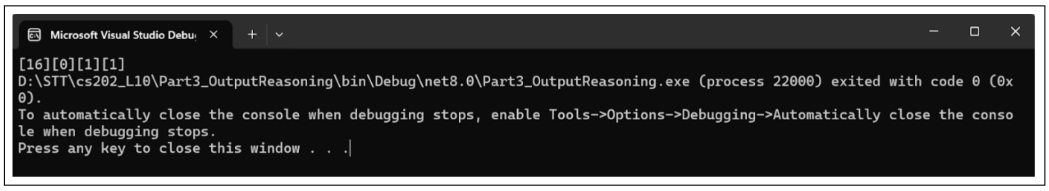
Observed Output:

`[16][0][1][1]`

Explanation:

- `"010%".Replace('0','%')` produces `"%1%%"`, whose length is 4, so `k = 4`.
- `new Program().age`: constructor runs once, sets age from 0 to 1.
- `++age` makes it 2. So we compute `4 << 2 = 16`.
- `"010%".Split('1')[1][0]` selects the substring after '1', which is `"0%"`, first char = '0'.
- `"010%".Split('0')[1][0]` gives `"1"`, so outputs '1'.
- `-1` equals 0 because bitwise NOT of -1 flips all bits.

Conclusion: Each bracketed value results from string manipulation, shifting, and unary operators, therefore the output is `[16][0][1][1]`.



```
[16][0][1][1]
D:\STT\cs202_L10\Part3_OutputReasoning\bin\Debug\net8.0\Part3_OutputReasoning.exe (process 22000) exited with code 0 (0x
0).
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 . . .
```

Figure 7: Actual Output of Part A. (L1)

Level 1 – Q2

```csharp
using System;
class Program
{
    int f;
    public static void Main(string[] args)
    {
        Console.WriteLine("run 1");
        Program p = new Program(new int()+"0".Length);
        for (int i = 0, _ = i; i < 5 && ++p.f >= 0;
             i++, Console.WriteLine(p.f++));
        {
            for (; p.f == 0;);
            {
                Console.WriteLine(p.f);
            }
        }

        Console.WriteLine("\nrun 2");
        p = new Program(p.f);
        Console.WriteLine(p.f);

        Console.WriteLine("\nrun 3");
        p = new Program();
        Console.WriteLine(p.f);
    }
    Program() => f = 0;
    Program(int x) => f = x;
}
```

Listing 7: Level 1 Q2 Code

Observed Output:

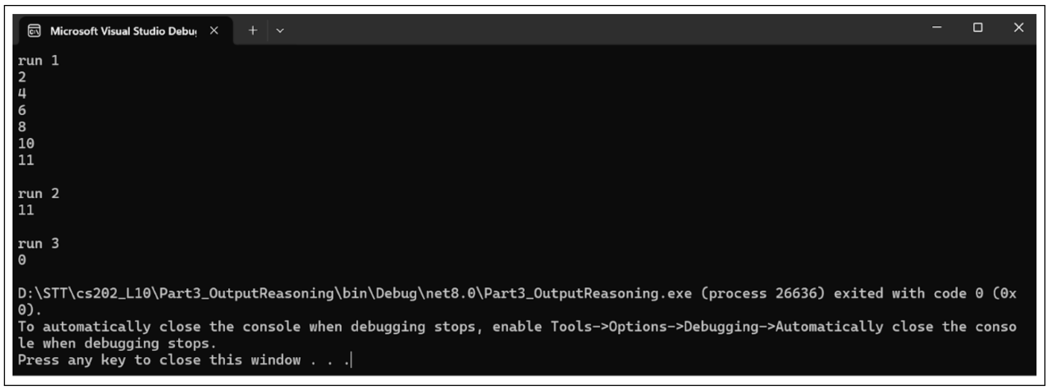
```
run 1
2
4
6
8
10
11

run 2
11

run 3
0
```

Explanation:

- `new Program(new int()+"0".Length)` evaluates `new int()` → 0, `"0".Length` → 1, so constructor receives 1 and sets `f = 1`.
- In the loop: `++p.f` runs first (pre-increment). f goes: 1→2, 2→3, etc.
- The post-loop `Console.WriteLine(p.f++)` prints current value, then increments.
- After 5 iterations, f becomes 11.
- In run 2, new object is created with f = 11, so printing gives 11.

- In run 3, default constructor sets f = 0.

Conclusion: The loop uses both pre- and post-increment in the same header, producing the observed values.



```
run 1
2
4
6
8
10
11

run 2
11

run 3
0

D:\STT\cs202_L10\Part3_OutputReasoning\bin\Debug\net8.0\Part3_OutputReasoning.exe (process 26636) exited with code 0 (0x
0).
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 . . .
```

Figure 8: Actual Output of Part B. (L1)

Level 1 – Q3

```
public class A
{
    public virtual void f1() { Console.WriteLine("f1"); }
}
public class B : A
{
    public override void f1() => Console.WriteLine("f2");
}

class Program
{
    static int i = 0;
    public event funcPtr handler;
    public delegate void funcPtr();
    public void destroy()
    {
        if (i == 6)
            return;
        else
        {
            Console.WriteLine(i++);
            destroy();
        }
    }
    public static void Main(string[] args)
    {
        Program p = new Program();
        p.handler += new funcPtr((new A()).f1);
        p.handler += new funcPtr((new B()).f1);
        p.handler();

        p.handler -= new funcPtr((new B()).f1);
        p.handler -= new funcPtr((new A()).f1);

        p?.destroy();
        p = null;
        i = -6;
        p?.destroy();
        (new Program())?.destroy();
    }
}
```

Listing 8: Level 1 Q3 Code

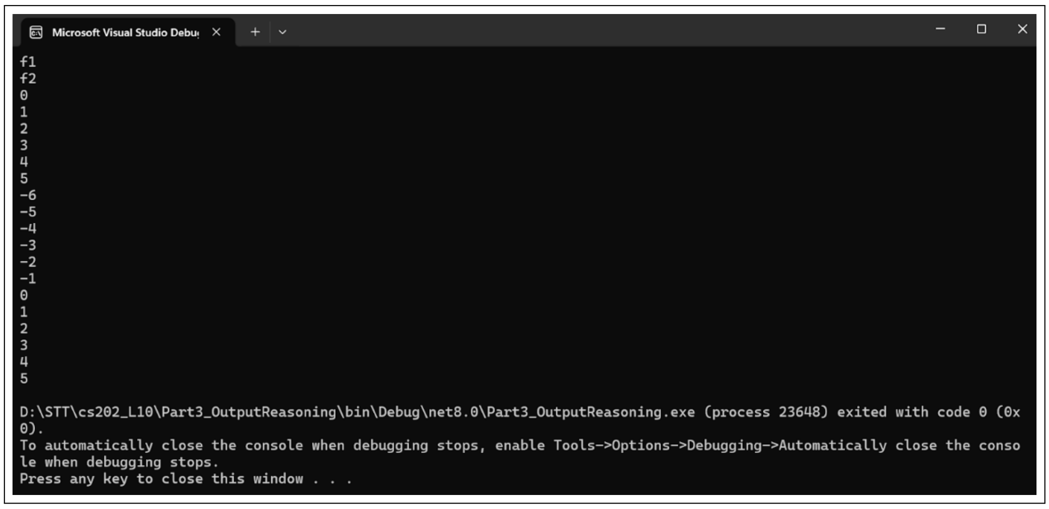
Observed Output:

```
f1
f2
0
1
2
3
4
5
-6
-5
-4
-3
-2
-1
0
1
2
3
4
5
```

Explanation:

- Event handler list initially contains A.f1 then B.f1, so both are invoked in order: `f1`, `f2`.
- Both handlers are removed, so event list becomes empty.
- `p?.destroy()` triggers recursion printing 0–5, then stops at 6.
- `p = null` prevents next call, so conditional operator returns safely.
- Resetting `i = -6` and calling destroy again prints -6 to 5.

Conclusion: The output reflects delegate invocation order and recursive printing controlled by a static counter.



```
f1
f2
0
1
2
3
4
5
-6
-5
-4
-3
-2
-1
0
1
2
3
4
5

D:\STT\cs202_L10\Part3_OutputReasoning\bin\Debug\net8.0\Part3_OutputReasoning.exe (process 23648) exited with code 0 (0x
0).
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 . . .
```

Figure 9: Actual Output of Part C. (L1)

2.5 Level 2 Output Reasoning

Level 2 – Q1

```
public class Institute
{
    internal int i = 7;
    public Institute() { Console.Write("1"); }
    public virtual void info() { Console.Write("2"); }
}
public class IITGN : Institute
{
    public int i = 8;
    public IITGN() { Console.Write("3"); }
    public IITGN(int i) { Console.Write("4"); }
    public override void info() { Console.Write("5"); }
}
class Program
{
    public static void Main(string[] args)
    {
        Console.Write("6");
        Institute ins1 = new Institute();
        ins1.info();
        IITGN ab101 = new IITGN(3);
        ab101 = new IITGN();
        ab101.info();
        Console.WriteLine();
        Console.WriteLine(ab101.i);
        Console.WriteLine(~(((Institute)ab101).i));
    }
}
```

Listing 9: Level 2 Q1 Code

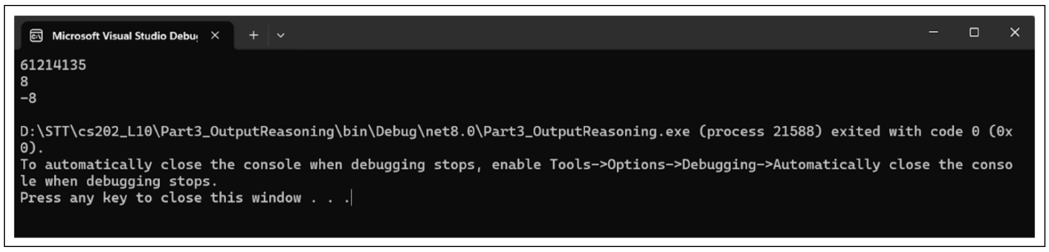
Observed Output:

```
612114135
8
-8
```

Explanation:

- Program prints `6` first.
- `new Institute()` prints `1`, `ins1.info()` prints `2`.
- `new IITGN(3)` prints `4` (only derived ctor).
- `new IITGN()` calls base ctor `1`, then prints `3`.
- `ab101.info()` prints overridden `5`.
- `ab101.i` refers to derived field = 8.
- Cast to `Institute` accesses hidden base field = 7, so $\tilde{7}$ `= -8`.

Conclusion: Constructor chaining, field shadowing, and bitwise NOT produce the observed output.



```
Microsoft Visual Studio Debu
61214135
8
-8

D:\STT\cs202_L10\Part3_OutputReasoning\bin\Debug\net8.0\Part3_OutputReasoning.exe (process 21588) exited with code 0 (0x
0).
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 . . .
```

Figure 10: Actual Output of Part A. (L2)

Level 2 – Q2

```csharp
using System;
public class Program
{
    public delegate void mydel();
    public void fun1() { Console.WriteLine("fun1()"); }
    public void fun2() { Console.WriteLine("fun2()"); }
    public static void Main(string[] args)
    {
        Program p = new Program();

        mydel obj1 = new mydel(p.fun1);
        obj1 += new mydel(p.fun2);
        Console.WriteLine("run 1");
        obj1();

        mydel obj2 = new mydel(p.fun2);
        obj2 += new mydel(p.fun1);
        Console.WriteLine("run 2");
        obj2();

        obj2 -= p.fun2;
        Console.WriteLine("run 3");
        obj2();
    }
}
```

Listing 10: Level 2 Q2 Code

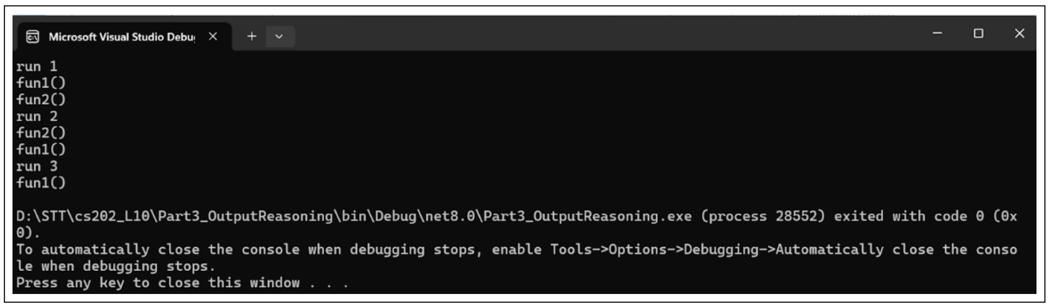
Observed Output:

```
run 1
fun1()
fun2()
run 2
fun2()
fun1()
run 3
fun1()
```

Explanation:

- Delegates call methods in the order they were added.
- In run 1, invocation list = [fun1, fun2].
- In run 2, invocation list = [fun2, fun1].
- Removing `fun2` leaves only `fun1` in run 3.

Conclusion: Multicast delegates execute in FIFO order and removal updates the invocation list.



```
run 1
fun1()
fun2()
run 2
fun2()
fun1()
run 3
fun1()

D:\STT\cs202_L10\Part3_OutputReasoning\bin\Debug\net8.0\Part3_OutputReasoning.exe (process 28552) exited with code 0 (0x
0).
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 . . .
```

Figure 11: Actual Output of Part B. (L2)

Level 2 – Q3

```csharp
using System;
using System.Collections;
public class Program
{
    int x;
    public static void Main(string[] args)
    {
        ArrayList L = new ArrayList();
        L.Add(new Program());
        L.Add(new Program());
        for (int i = 0; i < L.Count; i++)
            Console.WriteLine(++((Program)L[i]).x);

        L[0] = L[1];
        ((Program)L[0]).x = 202;

        for (int i = 0; i < L.Count; i++)
            Console.WriteLine(((Program)L[i]).x);

        ((Program)L[0]).x = 111;
        L.RemoveAt(0);
        Console.WriteLine(L.Count);
        Console.WriteLine(((Program)L[0]).x);
    }
}
```

Listing 11: Level 2 Q3 Code

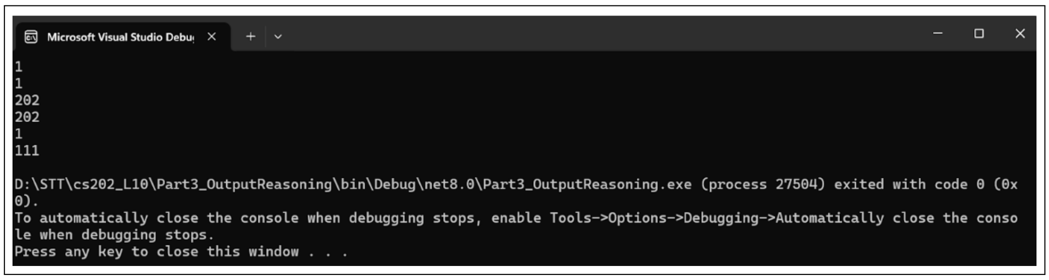
Observed Output:

```
1
1
202
202
1
111
```

Explanation:

- Initially two separate objects: incrementing x makes values 1 and 1.
- `L[0] = L[1]` makes both references point to the same object.
- Setting x on index 0 also affects index 1 (aliasing), so both show 202.
- After changing x = 111 and removing first element, list contains only one Program, so printing yields 111.

Conclusion: Because `ArrayList` stores object references, assigning one index to another creates aliasing.



Microsoft Visual Studio Debu

```
1
1
202
202
1
111

D:\STT\cs202_L10\Part3_OutputReasoning\bin\Debug\net8.0\Part3_OutputReasoning.exe (process 27504) exited with code 0 (0x
0).
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 . . .
```

Figure 12: Actual Output of Part C. (L2)

3 Results and Analysis

3.1 Part 1

Observed behavior: Constructors increment the alive counter; finalizers decrement it. Finalizers (destructors) run non-deterministically, so I forced garbage collection to make the messages show during the demo. The printed data values match the set order (10, 20, 30).

Key insight: C# finalizers are not like C++ destructors; they run when the GC finalizer thread processes objects. For demonstration purposes in teaching labs, explicit `GC.Collect()` and `GC.WaitForPendingFinalizers()` are commonly used to *observe* finalization.

3.2 Part 2

Observed behavior: When iterating a `Vehicle[]` containing different derived objects, both `Drive()` and `ShowInfo()` resolve to the most-derived override because they are `virtual`/`override`.

Fuel decreases by 5/10/15 for Vehicle/Car/Truck respectively, and messages reflect the actual type.

Key insight: This is runtime (dynamic) dispatch. The reference type is `Vehicle`, but the invoked method is chosen from the actual object type at runtime.

3.3 Part 3

3.3.1 Level 0

Q1 – Operators and Evaluation Order This question validates how C# differentiates between prefix and postfix increments inside expressions and how bitwise NOT behaves on signed integers. It shows that even a small change in operator position affects the final result. *Key takeaway:* Expression order and increment form (`x++` vs `++x`) directly affect printed output and should never be assumed equivalent.

Q2 – String Processing + Object Side-Effects This example combines string replacement, splitting, and left-shifting with a temporary object created inside an expression. It demonstrates that constructors may execute inside formatted expressions before argument evaluation completes. *Key takeaway:* C# evaluates expressions from left to right, and even inline object creation can modify values that affect later output.

Q3 – Stable In-Place Array Reordering The loop reorders non-zero elements while preserving order and shifts all zeros to the end. It demonstrates index-based mutation without extra space and shows the difference between stable and unstable rearrangement. *Key takeaway:* Correct pointer/index tracking is enough to implement efficient in-place algorithms without reallocating arrays.

3.3.2 Level 1

Q1 – Operator Stacking + Constructor Side Effects Tests how multiple operations (string functions, increments, shifting) compose inside a single print statement. It reinforces that implicit constructor calls can change values even before the main expression completes. *Key takeaway:* When many operations are chained in a single line, readability and predictability drop, debugging requires mentally expanding the expression step-by-step.

Q2 – Loop Semantics and Object State Persistence Illustrates how for-loop headers can contain multiple expressions and how prefix/postfix increments affect loop continuation. Also shows that object state persists across constructor calls unless explicitly reset. *Key takeaway:* Loops in C# allow multiple update expressions, and combined increment usage can produce non-intuitive values unless carefully traced.

Q3 – Delegates, Events, and Recursive State Demonstrates multicast delegate ordering, event subscription/removal, and controlled recursion through a static counter. Also introduces the null-conditional operator to prevent runtime exceptions. *Key takeaway:* Events in C# behave like ordered invocation lists, not sets, removing and re-adding handlers changes future execution flow.

3.3.3 Level 2

Q1 – Inheritance, Field Shadowing, Constructor Chaining Shows how base and derived constructors execute, how fields can be shadowed (not overridden), and how method overriding affects runtime behaviour. Also connects bitwise NOT with integer representation. *Key takeaway:* Methods dispatch polymorphically, but fields do not, casting affects which version of a field is accessed.

Q2 – Multicast Delegate Ordering and Removal Focuses on how delegate invocation order is preserved, and how removing methods affects later calls without touching earlier printed output. *Key takeaway:* Delegates act like ordered stacks: the order of registration defines the

execution order, and removal only affects future calls.
Q3 – Reference Aliasing in Collections Confirms that `ArrayList` stores object references, not copies. Assigning one index to another creates two references to the same object, so modifying one affects both. *Key takeaway:* In non-generic collections, reference identity matters, aliasing bugs appear easily if reassignment overwrites references instead of cloning values.

4 Discussion and Conclusion

4.1 Challenges and Reflections

- Understanding finalizer timing in C#: the non-determinism is surprising at first. The explicit GC calls were used only to demonstrate messages.
- Designing `Vehicle` with `protected` fields made derived classes concise while still safe for this exercise.
- For output reasoning, carefully tracing prefix/postfix increments and operator precedence avoided wrong answers.

4.2 Lessons Learned

- Constructors and finalizers signal object lifecycle, but only constructors are deterministic. Resource cleanup should normally use `IDisposable` and `using`.
- Destructors in C# are not guaranteed to run at scope exit. To reliably demonstrate finalization in a teaching environment, object scope must end fully and the program must explicitly wait for the GC and finalizer thread.
- Virtual/override enables polymorphism; base references can invoke derived behavior at runtime.
- When in doubt, expand expressions step by step and check operator tables.

References

1. STT Lab 10 Handout: `https://drive.google.com/file/d/1lK8Yyj5mVipL2_9kEkVZS341udSoqDx5/view`
2. Microsoft Docs: “Destructors (Finalizers) in C#” — `https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/destructors`
3. Microsoft Docs: “Inheritance in C#” — `https://learn.microsoft.com/dotnet/csharp/fundamentals/object-oriented/inheritance`
4. Microsoft Docs: “Operators in C#” — `https://learn.microsoft.com/dotnet/csharp/language-reference/operators/`

5 GitHub Repository

The GitHub Repository for this Assignment can be accessed using the following link.
CLICK HERE