NAME: IFRA HABIB
REG NO: SP22-BCS-032

LAB MID

QUESTION 1:

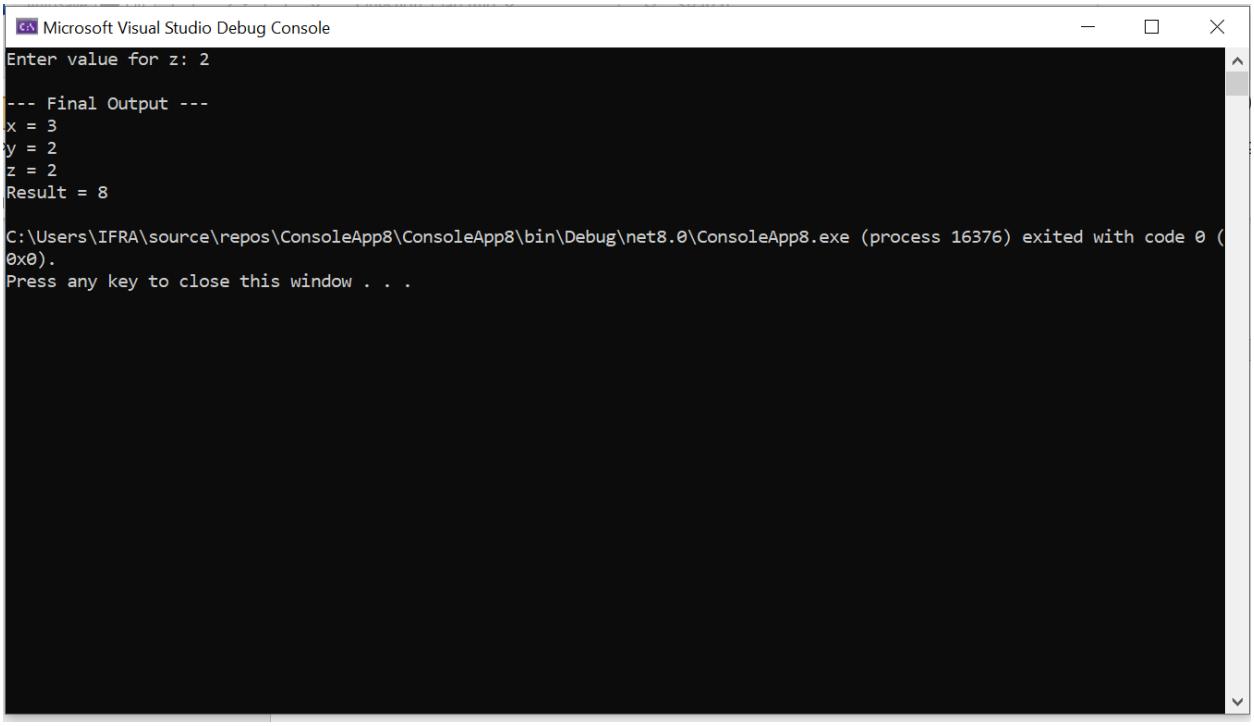
CODE:

```
using System;
using System.Collections.Generic;


class Program
{
   static void Main()
   {
      // The original string format (modified with fixed values)
      string inputTemplate = "x:3; y:2; z:userinput; result: x * y + z;";

      // Dictionary to store parsed values
      Dictionary<string, int> variables = new Dictionary<string, int>();

      // Split the string on semicolons
      string[] parts = inputTemplate.Split(';');

      string expression = "";

      foreach (string part in parts)
      {
```

```
    if (string.IsNullOrWhiteSpace(part)) continue;

    string[] keyValue = part.Split(':');
    string key = keyValue[0].Trim();
    string value = keyValue[1].Trim();

    if (value.ToLower() == "userinput")
    {
        // Only ask for z value
        Console.Write($"Enter value for {key}: ");
        int userVal = int.Parse(Console.ReadLine());
        variables[key] = userVal;
    }
    else if (key.ToLower() == "result")
    {
        expression = value; // Store expression for later
    }
    else
    {
        // Fixed numbers (x=3, y=2)
        variables[key] = int.Parse(value);
    }
}

// Get values from dictionary
int x = variables["x"];  // Will be 3
```

```
        int y = variables["y"];  // Will be 2
        int z = variables["z"];  // User input

        // Perform the calculation (x * y + z)
        int result = x * y + z;

        // Final output showing all values
        Console.WriteLine("\n--- Final Output ---");
        Console.WriteLine($"x = {x}");
        Console.WriteLine($"y = {y}");
        Console.WriteLine($"z = {z}");
        Console.WriteLine($"Result = {result}");
    }
}
```

OUTPUT



QUESTION 2:

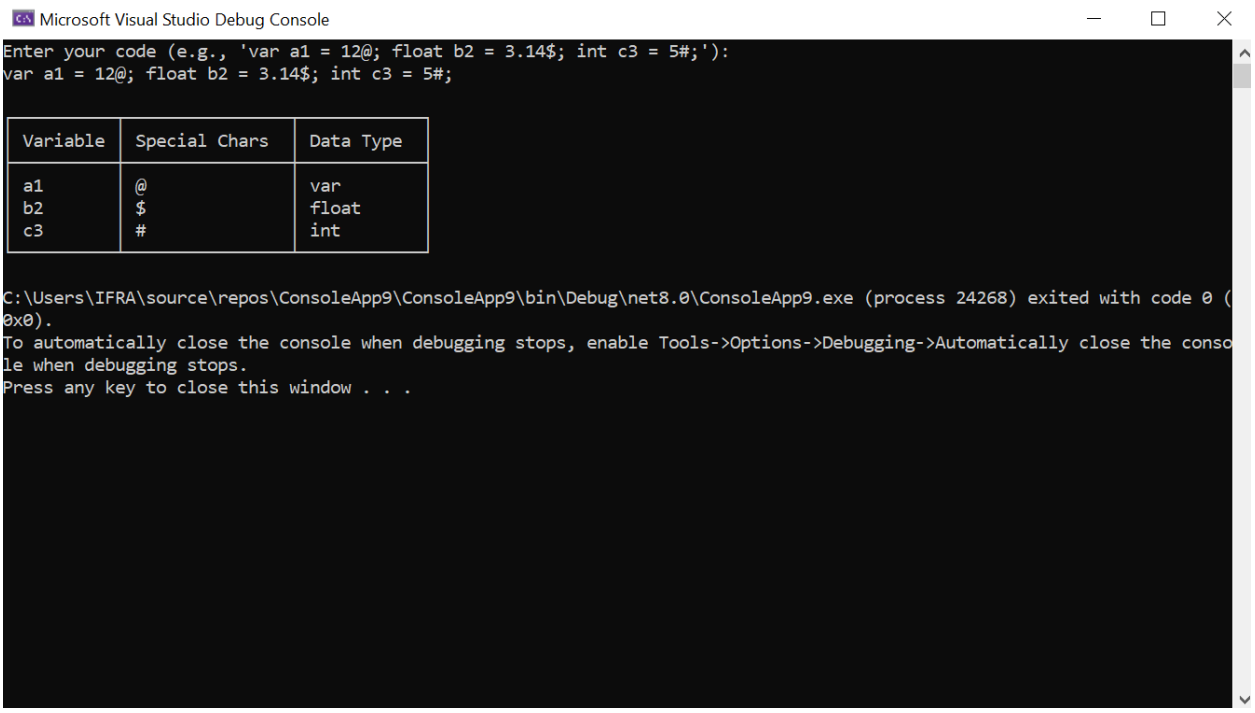
CODE:

```
using System;

using System.Text.RegularExpressions;

using System.Collections.Generic;


class Program

{

   static void Main()

   {

      Console.WriteLine("Enter your code (e.g., 'var a1 = 12@; float b2 = 3.14$; int c3 =
5#;'):");

      string inputCode = Console.ReadLine();
```

```
    // Regex to match variables starting with a/b/c, ending with digit, with special chars
    string pattern = @"(var|int|float|double)\s+([abc][a-zA-Z]*\d+)\s*=\s*([^;]+);";

    MatchCollection matches = Regex.Matches(inputCode, pattern);
    List<(string VarName, string SpecialSymbol, string TokenType)> extracted = new
List<(string, string, string)>();

    foreach (Match match in matches)
    {
      string tokenType = match.Groups[1].Value;
      string varName = match.Groups[2].Value;
      string value = match.Groups[3].Value;

      // Find all special characters
      MatchCollection specialChars = Regex.Matches(value, @"[^\w\s.]");
      if (specialChars.Count > 0)
      {
        extracted.Add((varName, string.Join("", specialChars), tokenType));
      }
    }

    // Display results in a box-style table
    if (extracted.Count > 0)
    {
      Console.WriteLine("\n┌──────────┬──────────────┬──────────┐");
```

```
        Console.WriteLine("│ {0,-8} │ {1,-14} │ {2,-10} │", "Variable", "Special Chars", "Data
Type");
        Console.WriteLine(" ├──────────┼────────────────┼────────────┤ ");


        foreach (var item in extracted)
        {
           Console.WriteLine("│ {0,-8} │ {1,-14} │ {2,-10} │",
                    item.VarName,
                    item.SpecialSymbol,
                    item.TokenType);
        }

        Console.WriteLine("└──────────┴────────────────┴────────────┘");
      }
      else
      {
        Console.WriteLine("\nNo valid variables found with special characters.");
        Console.WriteLine("Example input: var a1 = 12@; float b2 = 3.14$; int c3 = 5#;");
      }
    }
}
```

OUTPUT



QUESTION 3:

CODE :

```
using System;

using System.Collections.Generic;

using System.Text.RegularExpressions;


namespace SymbolTableApp

{

  // Represents a single symbol table entry

  class SymbolTableEntry

  {

    public string Type { get; }

    public string Name { get; }

    public string Value { get; }

    public int LineNumber { get; }
```

```
    public SymbolTableEntry(string type, string name, string value, int lineNumber)
    {
        Type = type;
        Name = name;
        Value = value;
        LineNumber = lineNumber;
    }
}

// Handles validation logic like syntax and palindrome checking
class Validator
{
    private static readonly Regex DeclarationPattern =
        new Regex(@"^(int|float|char|string)\s+([a-zA-Z_]\w*)\s*=\s*(.+);$");

    public static bool IsValidDeclaration(string input, out string type, out string name, out string value)
    {
        var match = DeclarationPattern.Match(input);
        if (match.Success)
        {
            type = match.Groups[1].Value;
            name = match.Groups[2].Value;
            value = match.Groups[3].Value.Trim();
            return true;
```

```
    }

    type = name = value = null;
    return false;
}

public static bool ContainsPalindromeSubstring(string text)
{
    for (int i = 0; i < text.Length; i++)
    {
        for (int j = i + 2; j < text.Length; j++)
        {
            var substr = text.Substring(i, j - i + 1);
            if (IsPalindrome(substr))
                return true;
        }
    }
    return false;
}

private static bool IsPalindrome(string input)
{
    int left = 0, right = input.Length - 1;
    while (left < right)
    {
        if (input[left++] != input[right--])
```

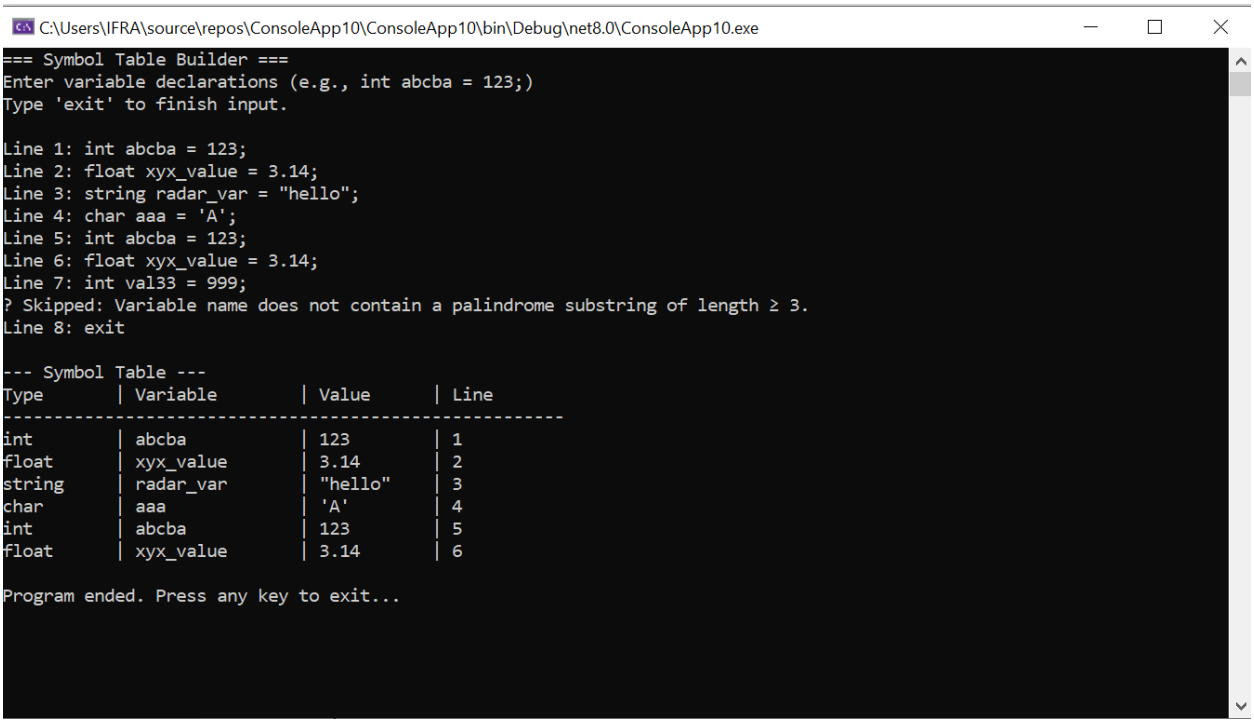
```
            return false;
        }
        return true;
    }
}

// Manages symbol table operations
class SymbolTableManager
{
    private readonly List<SymbolTableEntry> entries = new();
    private int lineCounter = 1;

    public void Start()
    {
        Console.WriteLine("=== Symbol Table Builder ===");
        Console.WriteLine("Enter variable declarations (e.g., int abcba = 123;)");
        Console.WriteLine("Type 'exit' to finish input.\n");

        while (true)
        {
            Console.Write($"Line {lineCounter}: ");
            var input = Console.ReadLine()?.Trim();

            if (string.Equals(input, "exit", StringComparison.OrdinalIgnoreCase))
                break;
```

```
            ProcessLine(input);
            lineCounter++;
        }

        DisplaySymbolTable();
    }

    private void ProcessLine(string input)
    {
        if (Validator.IsValidDeclaration(input, out string type, out string name, out string value))
        {
            if (Validator.ContainsPalindromeSubstring(name))
            {
                entries.Add(new SymbolTableEntry(type, name, value, lineCounter));
            }
            else
            {
                Console.WriteLine("✗ Skipped: Variable name does not contain a palindrome substring of length ≥ 3.");
            }
        }
        else
        {
            Console.WriteLine("✗ Invalid syntax. Format must be: <type> <varName> = <value>;");
        }
```

```
        }

        private void DisplaySymbolTable()
        {
            Console.WriteLine("\n--- Symbol Table ---");
            Console.WriteLine("{0,-10} | {1,-15} | {2,-10} | {3}", "Type", "Variable", "Value", "Line");
            Console.WriteLine(new string('-', 55));

            if (entries.Count == 0)
            {
                Console.WriteLine("No valid entries found.");
                return;
            }

            foreach (var entry in entries)
            {
                Console.WriteLine($"{entry.Type,-10} | {entry.Name,-15} | {entry.Value,-10} |
{entry.LineNumber}");
            }
        }
    }

    class Program
    {
        static void Main()
        {
```

```
        var manager = new SymbolTableManager();
        manager.Start();

        Console.WriteLine("\nProgram ended. Press any key to exit...");
        Console.ReadKey();
    }
  }
}
```

OUTPUT



QUESTION 4:

CODE :

```
using System;

using System.Collections.Generic;

using System.Linq;


class CFGProcessor

{

   static Dictionary<string, List<string>> productions = new();

   static Dictionary<string, HashSet<string>> firstSet = new();

   static Dictionary<string, HashSet<string>> followSet = new();

   static HashSet<string> nonTerminals = new();

   static HashSet<string> terminals = new();

   static string startingSymbol = "";
```

```
static void Main()
{
    Console.WriteLine("Enter grammar rules (e.g., E -> T X | e). Type 'exit' to finish.\n");

    // Accept grammar rules
    while (true)
    {
        Console.Write("Rule: ");
        string input = Console.ReadLine()?.Trim();
        if (input?.ToLower() == "exit") break;

        if (!input.Contains("->"))
        {
            Console.WriteLine("Invalid format. Use 'A -> alpha | beta'");
            continue;
        }

        string[] parts = input.Split("->", 2);
        string lhs = parts[0].Trim();
        string[] rhs = parts[1].Split('|').Select(p => p.Trim()).ToArray();

        if (productions.ContainsKey(lhs))
            productions[lhs].AddRange(rhs);
        else
            productions[lhs] = new List<string>(rhs);
```

```
        if (startingSymbol == "")
            startingSymbol = lhs;

        nonTerminals.Add(lhs);
    }

    IdentifyTerminals();

    if (CheckLeftRecursion() || CheckAmbiguity())
    {
        Console.WriteLine("\nGrammar invalid for top-down parsing.");
        return;
    }

    GenerateFirstSets();
    GenerateFollowSets();

    Console.WriteLine("\n=== FIRST SETS ===");
    PrintTable(firstSet);

    Console.WriteLine("\n=== FOLLOW SETS ===");
    PrintTable(followSet);
}

static void IdentifyTerminals()
{
```

```
    foreach (var (lhs, rules) in productions)
    {
        foreach (var rule in rules)
        {
            var symbols = rule.Split(' ', StringSplitOptions.RemoveEmptyEntries);
            foreach (var sym in symbols)
            {
                if (!productions.ContainsKey(sym) && sym != "e")
                    terminals.Add(sym);
            }
        }
    }

    static bool CheckLeftRecursion()
    {
        foreach (var (lhs, rules) in productions)
        {
            foreach (var rule in rules)
            {
                var firstSymbol = rule.Split(' ',
StringSplitOptions.RemoveEmptyEntries).FirstOrDefault();
                if (firstSymbol == lhs)
                    return true;
            }
        }
```

```
    return false;
}

static bool CheckAmbiguity()
{
    foreach (var (lhs, rules) in productions)
    {
        var seen = new HashSet<string>();
        foreach (var rule in rules)
        {
            if (!seen.Add(rule)) return true;
        }
    }
    return false;
}

static void GenerateFirstSets()
{
    foreach (var nt in nonTerminals)
        firstSet[nt] = new HashSet<string>();

    bool changed;
    do
    {
        changed = false;
        foreach (var (nt, rules) in productions)
```

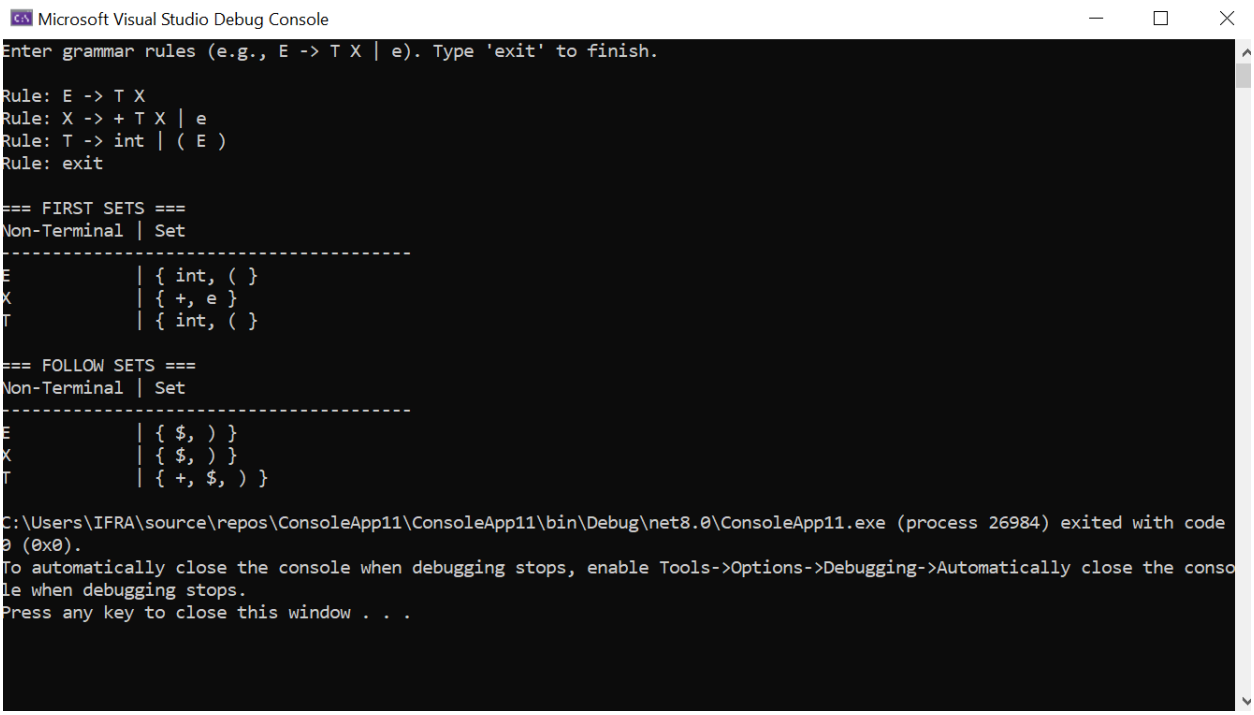
```
{
    foreach (var rule in rules)
    {
        var symbols = rule.Split(' ', StringSplitOptions.RemoveEmptyEntries);
        if (symbols.Length == 0) continue;

        bool canHaveEpsilon = true;

        for (int i = 0; i < symbols.Length && canHaveEpsilon; i++)
        {
            string symbol = symbols[i];

            if (symbol == "e")
            {
                changed |= firstSet[nt].Add("e");
                canHaveEpsilon = false;
            }
            else if (terminals.Contains(symbol))
            {
                changed |= firstSet[nt].Add(symbol);
                canHaveEpsilon = false;
            }
            else if (nonTerminals.Contains(symbol))
            {
                foreach (var item in firstSet[symbol])
                {
```

```
                    if (item != "e")
                        changed |= firstSet[nt].Add(item);
                }
                canHaveEpsilon = firstSet[symbol].Contains("e");
            }
        }

        if (canHaveEpsilon)
            changed |= firstSet[nt].Add("e");
    }
  } while (changed);
}

static void GenerateFollowSets()
{
  foreach (var nt in nonTerminals)
    followSet[nt] = new HashSet<string>();

  followSet[startingSymbol].Add("$");

  bool changed;
  do
  {
    changed = false;
    foreach (var (lhs, rules) in productions)
```

```
{
    foreach (var rule in rules)
    {
        var symbols = rule.Split(' ', StringSplitOptions.RemoveEmptyEntries);
        for (int i = 0; i < symbols.Length; i++)
        {
            string current = symbols[i];
            if (!nonTerminals.Contains(current)) continue;

            bool epsilonPossible = true;
            for (int j = i + 1; j < symbols.Length && epsilonPossible; j++)
            {
                string next = symbols[j];
                epsilonPossible = false;

                if (terminals.Contains(next))
                {
                    changed |= followSet[current].Add(next);
                    break;
                }

                foreach (var item in firstSet[next])
                {
                    if (item == "e") epsilonPossible = true;
                    else changed |= followSet[current].Add(item);
                }
```

```
                }

                if (epsilonPossible || i == symbols.Length - 1)
                {
                    foreach (var f in followSet[lhs])
                        changed |= followSet[current].Add(f);
                }
            }
        }
    } while (changed);
  }

  static void PrintTable(Dictionary<string, HashSet<string>> setDict)
  {
    Console.WriteLine("{0,-12} | {1}", "Non-Terminal", "Set");
    Console.WriteLine(new string('-', 40));
    foreach (var nt in nonTerminals)
    {
      string items = string.Join(", ", setDict[nt]);
      Console.WriteLine($"{nt,-12} | {{ {items} }}");
    }
  }
}
```

OUTPUT:


Microsoft Visual Studio Debug Console
Enter grammar rules (e.g., E -> T X | e). Type 'exit' to finish.
Rule: E -> T X
Rule: X -> + T X | e
Rule: T -> int | (E)
Rule: exit
=== FIRST SETS ===
Non-Terminal | Set
E | { int, (}
X | { +, e }
T | { int, (}
=== FOLLOW SETS ===
Non-Terminal | Set
E | { $,) }
X | { $,) }
T | { +, $,) }
C:\Users\IFRA\source\repos\ConsoleApp11\ConsoleApp11\bin\Debug\net8.0\ConsoleApp11.exe (process 26984) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .