Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГТУ»)**

**Институт компьютерных систем и информационной безопасности**

**Кафедра информационных систем и программирования**

**ЛАБОРАТОРНАЯ РАБОТА №2**

**Дисциплина: тестирование и отладка программного обеспечения**

Работу выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Я.С Давков

Направление подготовки: 09.03.04 Программная инженерия

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.Г. Волик

**1 Цель работы**

Изучить средства и возможности программной отладки и трассировки программ.

**2 Краткая теория**

Помимо встроенного в среду разработки отладчика, предоставляющие средства отладки в интерактивном режиме, существуют средства для организации программной отладки. К таким средствам относятся стандартные классы .NET, такие как Debug и Trace. Данный подход удобен в тех случаях, когда структура кода излишне сложна (например, имеется множество вложенных циклов со сложными условиями завершения) или возникают проблемы с локализацией ошибок (например, из-за разброса потенциальных проблемных точек по всему коду проекта). В таких случаях отладочная печать и программная остановка приложения позволяют упростить нахождение причины и место возникновения ошибки. Вы можете сами выбирать наборы данных, которые необходимы в данной точке программы, тогда как интерактивный режим отображает все заранее выбранные значения переменных и величин. При этом наличие в коде самих операторов управления отладкой позволяет сохранять все точки останова и прочую информацию между сеансами работы с проектом. Этот способ отладки исторически появился первым, до появления интегрированных сред разработки, однако по сей день пользуется заслуженной популярностью у программистов. При этом следует разделять два сходных понятия: трассировка и отладка. Трассировка кода представляет собой получение информационных сообщений о работе приложения во время выполнения. Отладка – это отслеживание и устранение ошибок программирования в приложении при разработке.

**3 Задание**

Задание 1. Написать программу для приближенного вычисления значения интеграла ∫ b a f (x)dx с заранее заданной точностью ε > 0 , т.е. вычисления продолжаются до тех пор, пока Δ > ε 2n , и прекращаются в тот момент, когда значение погрешности Δ2n из формулы Рунге становится меньше либо равно ε . В лабораторной работе читать ε = 0.0001 . Функцию выбрать из вариантов в разделе 5 в соответствии с номером в журнале. Для вычислений использовать формулу приближенного вычисления в соответствии с номером в журнале по модулю 3 (№%3): 0) Обобщенная формула прямоугольников. 1) Обобщенная формула Симпсона. 2) Обобщенная формула трапеций. Оценить погрешность вычислений с помощью правила Рунге. На каждой итерации производить трассировку всех значений изменяющихся переменных. Вовремя работы алгоритма проверять выход за границы интегрирования (интервал [a;b]) переменной x при помощи Assert. При вычислении значения погрешности и интеграла выводить информацию в лог трассировки на каждой итерации вычислений. При возникновении ошибок также выводить их в лог трассировки. Пусть FN – номер первой буквы имени, а LN – номер первой буквы фамилии студента. Найти значение интеграла на FN-ом шаге и вывести его в вывод отладчика. Найти значение интеграла на LN-ом шаге и вывести его в вывод трассировщика

Задание 2. Написать программу для вычисления суммы элементов последовательности. Последовательность выбрать из вариантов в разделе 5.2 в соответствии с номером в журнале. В программе проверить ситуацию арифметического переполнения при помощи Assert и при возникновении ошибки выводить в лог трассировки сообщение.

**4 Выполнение работы**

Выполним первое задание. Мой номер в журнале 4 - Обобщенная формула Симпсона.

Листинг программы 1:

using System;  
using System.Diagnostics;  
  
class Program  
{  
 const double *Epsilon* = 0.0001;  
 const double *A* = 0.0;  
 const double *B* = Math.*PI* / 2;  
 const int *FN* = 33; // Я  
 const int *LN* = 5; // Д  
  
 static double F(double x)  
 {  
 return Math.Exp(-Math.Tan(0.84 \* x)) \* (0.35 + Math.Cos(x));  
 }  
  
 static double Simpson(long n)  
 {  
 double h = (*B* - *A*) / n;  
 double sum = F(*A*) + F(*B*);  
  
 for (long i = 1; i < n; i++)  
 {  
 double x = *A* + i \* h;  
 Debug.Assert(x >= *A* && x <= *B*);  
 sum += (i % 2 == 1) ? 4 \* F(x) : 2 \* F(x);  
 }  
  
 return (h / 3) \* sum;  
 }  
  
 static void Main()  
 {  
 try  
 {  
 long n = 2;  
 double previousIntegral = Simpson(n);  
 double currentIntegral;  
 double rungeError;  
 int iteration = 1;  
 bool converged = false;  
  
 Trace.WriteLine($"Iteration {iteration}: n = {n}, Integral = {previousIntegral:F8}");  
  
 // идем до сходимости  
 do  
 {  
 n \*= 2;  
 currentIntegral = Simpson(n);  
 rungeError = Math.Abs(currentIntegral - previousIntegral) / 15.0;  
 iteration++;  
  
 Trace.WriteLine($"Iteration {iteration}: n = {n}, Integral = {currentIntegral:F8}, Error = {rungeError:F8}");  
  
 previousIntegral = currentIntegral;  
  
 if (rungeError <= *Epsilon* && !converged)  
 {  
 converged = true;  
 Console.WriteLine($"CONVERGED at iteration {iteration}");  
 Console.WriteLine($"Result: {currentIntegral:F8}");  
 Console.WriteLine($"Iterations: {iteration}");  
 break;  
 }  
  
 } while (iteration < 100);  
  
 // продолжаем до FN и LN шагов  
 int maxStep = Math.Max(*FN*, *LN*);  
 if (iteration < maxStep)  
 {  
 Console.WriteLine($"\nContinuing to reach FN={*FN*} and LN={*LN*} steps...");  
  
 while (iteration < maxStep)  
 {  
 n \*= 2;  
 currentIntegral = Simpson(n);  
 rungeError = Math.Abs(currentIntegral - previousIntegral) / 15.0;  
 iteration++;  
  
 Trace.WriteLine($"Iteration {iteration}: n = {n}, Integral = {currentIntegral:F8}, Error = {rungeError:F8}");  
  
 if (iteration == *LN*)  
 {  
 Trace.WriteLine($"Integral at step {*LN*}: {currentIntegral:F8}");  
 Console.WriteLine($"LN step ({*LN*}) reached: {currentIntegral:F8}");  
 }  
 if (iteration == *FN*)  
 {  
 Debug.WriteLine($"Integral at step {*FN*}: {currentIntegral:F8}");  
 Console.WriteLine($"FN step ({*FN*}) reached: {currentIntegral:F8}");  
 }  
  
 previousIntegral = currentIntegral;  
 }  
 }  
  
 Console.WriteLine($"Total iterations: {iteration}");  
 }  
 catch (Exception ex)  
 {  
 Trace.WriteLine($"ERROR: {ex.Message}");  
 Console.WriteLine($"ERROR: {ex.Message}");  
 }  
 }  
}

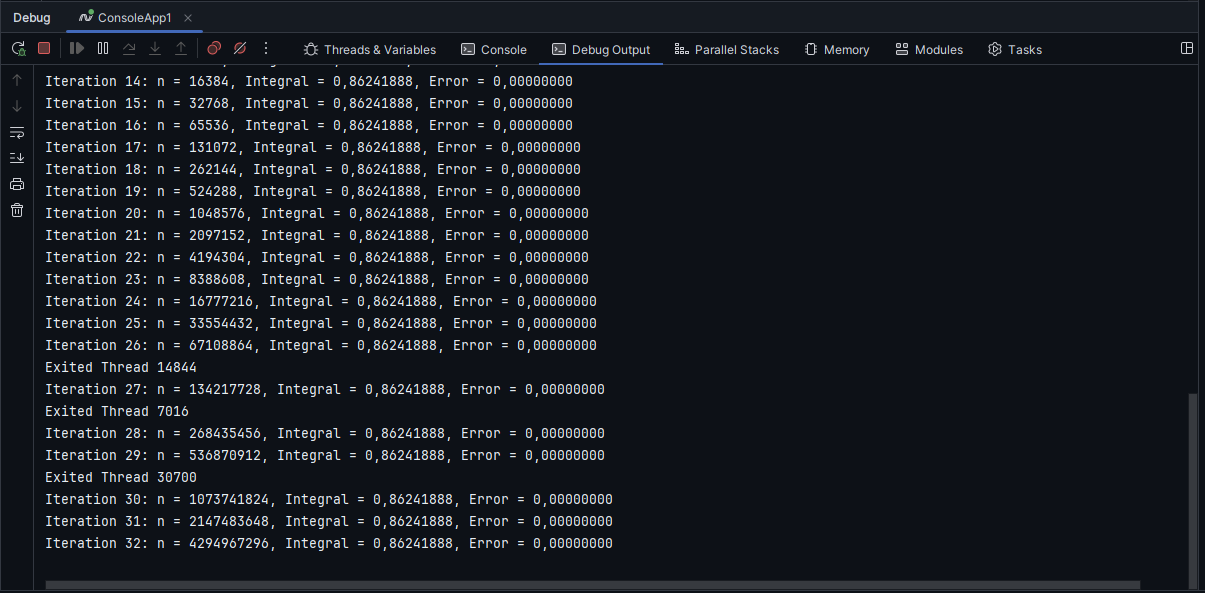


Рисунок 1 - Окно Debug Output

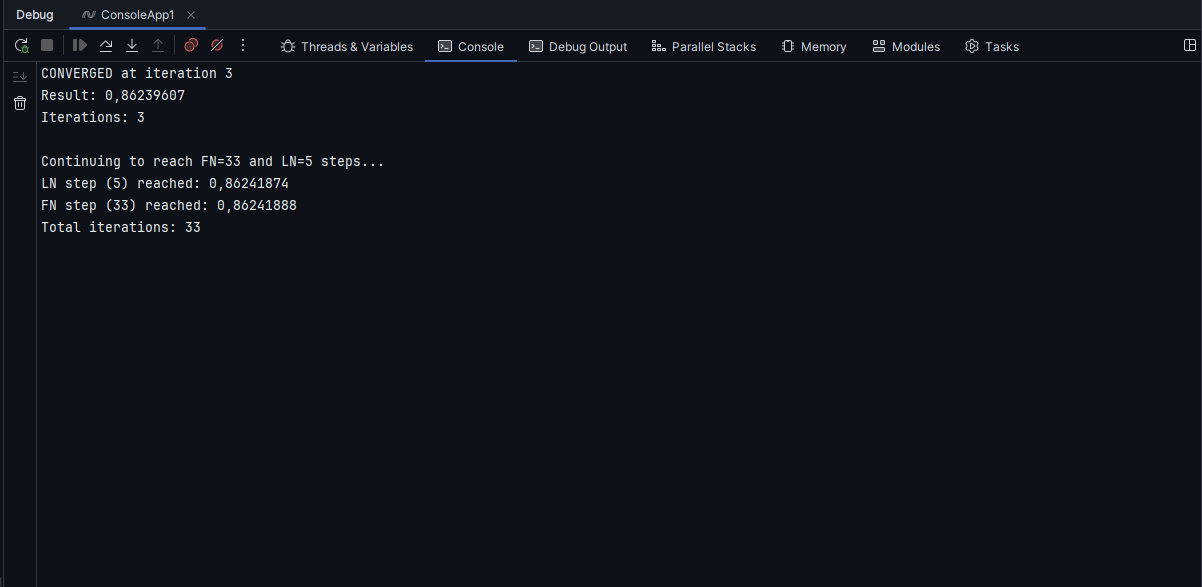


Рисунок 2 – Окно вывода

Выполним 2 задание. Мой номер в журнале 4 – геометрическая прогрессия.

Листинг программы 2:

using System;  
using System.Diagnostics;  
  
class Program  
{  
 static void Main()  
 {  
 try  
 {  
 Console.WriteLine("=== GEOMETRIC PROGRESSION SUM CALCULATOR ===");  
  
 Console.Write("Enter first element b1: ");  
 double b1 = Convert.ToDouble(Console.ReadLine());  
  
 Console.Write("Enter common ratio q: ");  
 double q = Convert.ToDouble(Console.ReadLine());  
  
 Console.Write("Enter number of elements n: ");  
 int n = Convert.ToInt32(Console.ReadLine());  
  
 Debug.Assert(n > 0, "Number of elements must be positive");  
 Trace.WriteLine($"INPUT: b1 = {b1}, q = {q}, n = {n}");  
  
 double sum = CalculateGeometricSum(b1, q, n);  
  
 Console.WriteLine($"\nSum of {n} elements of geometric progression: {sum}");  
 Trace.WriteLine($"RESULT: Sum = {sum}");  
 }  
 catch (Exception ex)  
 {  
 Console.WriteLine($"ERROR: {ex.Message}");  
 Trace.WriteLine($"ERROR: {ex.Message}");  
 }  
 }  
  
 static double CalculateGeometricSum(double b1, double q, int n)  
 {  
 try  
 {  
 double sum = 0;  
 double currentTerm = b1;  
  
 for (int i = 1; i <= n; i++)  
 {  
 if (double.IsInfinity(currentTerm) || double.IsNaN(currentTerm))  
 {  
 string errorMsg = $"Arithmetic overflow at element {i}: {currentTerm}";  
 Debug.Assert(false, errorMsg);  
 Trace.WriteLine($"OVERFLOW ERROR: {errorMsg}");  
 throw new OverflowException(errorMsg);  
 }  
  
 double newSum = sum + currentTerm;  
 if (double.IsInfinity(newSum) || double.IsNaN(newSum))  
 {  
 string errorMsg = $"Sum overflow at element {i}: sum = {sum}, currentTerm = {currentTerm}";  
 Debug.Assert(false, errorMsg);  
 Trace.WriteLine($"SUM OVERFLOW ERROR: {errorMsg}");  
 throw new OverflowException(errorMsg);  
 }  
  
 sum = newSum;  
 Trace.WriteLine($"Element {i}: b{i} = {currentTerm:F6}, Sum = {sum:F6}");  
  
 if (i < n)  
 {  
 double nextTerm = currentTerm \* q;  
  
 if (double.IsInfinity(nextTerm) || double.IsNaN(nextTerm))  
 {  
 string errorMsg = $"Multiplication overflow: {currentTerm} \* {q} = {nextTerm}";  
 Debug.Assert(false, errorMsg);  
 Trace.WriteLine($"MULTIPLICATION OVERFLOW: {errorMsg}");  
 throw new OverflowException(errorMsg);  
 }  
  
 currentTerm = nextTerm;  
 }  
 }  
  
 return sum;  
 }  
 catch (OverflowException)  
 {  
 throw;  
 }  
 catch (Exception ex)  
 {  
 Trace.WriteLine($"CALCULATION ERROR: {ex.Message}");  
 throw;  
 }  
 }  
}

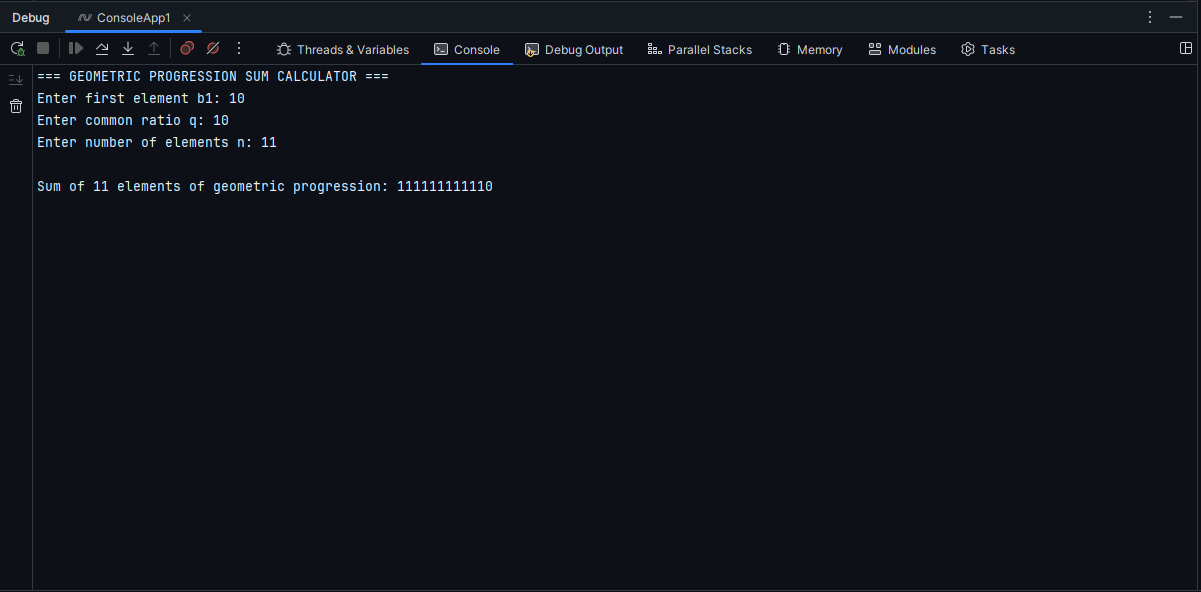


Рисунок 3 – окно вывода программы

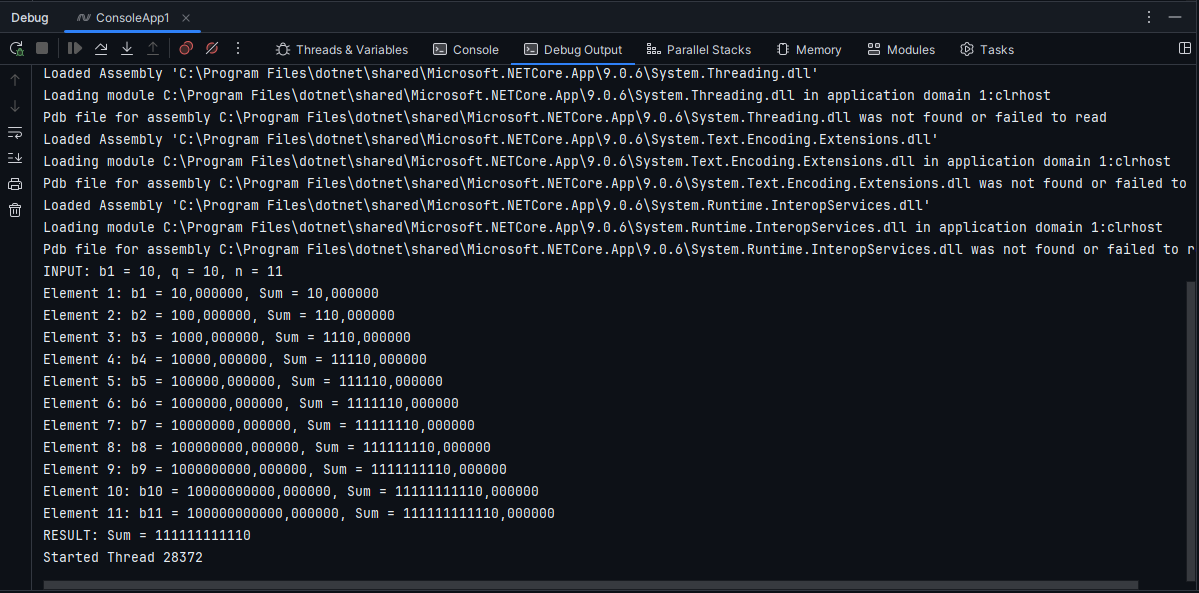


Рисунок 4 – окно Debug Output