ENGG1410: Introductory Programming for Engineers

Lab 4: "Introduction to C Programming": Looping and Debugging a C Program

School of Engineering, University of Guelph
Fall 2024

Start Date: Week #5 2024
Duration: 1 Week

Objectives

The objective of this lab is to enhance your programming skills by writing C programs implementing looping for programs to compute series up to a specified iteration, Greatest Common Divosor between two numbers and displaying all the prime numbers in a given interval.

Instructions

Follow the instructions for each question. You are required to compile and test your programs. Submit your source code and screenshots of your program output via the learning management system. Once again, place your solution for each part in a separate directory, so that Visual Studio Code will not attempt to compile multiple files and link them together. Each of your directories should have one .c file corresponding to your solution for the individual part of the exercise. Please ensure you are selecting the folder with the .c you want to compile.

Part 1 Programming

1 Evaluating the value of π Using a Series

Problem Statement

Develop a program that prompts the user to enter the number of terms (iterations) to include in the series for estimating π. The series is defined as follows:

$$\pi \approx 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \dots\right)$$

Each additional term in the series improves the approximation by considering a subsequent fraction in the sequence. Your program will calculate the approximate value of π based on this series up to the specified number of terms.

Explanation of Series and Iterations

The series alternates between subtraction and addition of fractions, where the denominator is an odd number that increases by 2 with each term. More iterations of the series will yield a more accurate approximation of π. Each iteration accounts for one additional term in the series, enhancing the precision of the estimate.

Requirements

1. **Input Validation:** Ensure the user inputs a non-negative integer. If the input is invalid, prompt the user to enter a correct value.
2. **Calculation:** Implement the series calculation to estimate π. Loop through the series, adding and subtracting fractions according to the pattern shown, for the number of iterations specified by the user.
3. **Output:** Display the estimated value of π after computing the specified number of terms.

Sample Run

```
Enter the number of terms for \(\pi\) approximation: 1000
Estimated value of \(\pi\) after 1000 terms: 3.142
```

Hints

There are two ways you might implement the calculation of this series:

- Use a loop where you keep a running total. Use a variable to track whether to add or subtract the next fraction. This method is straightforward but can be lengthy in terms of code.
- A more concise method involves using a single loop with a **formula** to determine the sign and the next denominator in the series. The sign can be determined from the iteration number, `term = pow(-1, i)` where i is the iteration index starting from 0. Try to come up with such a formula to incorporates both the alternating sign and the incrementing denominator, reducing the need for additional variables or conditionals as would be needed in the first approach.

2 Computing the Greatest Common Divisor (GCD)

Problem Statement

Write a program that asks the user to input two non-negative integers and then outputs their GCD. The program must handle input validation and should provide clear and precise output.

Example

The GCD of 48 and 18 is 6, which can be determined using the Euclidean algorithm.

Requirements

1. **Input Validation:** Ensure that both user inputs are non-negative integers. If either input is invalid (e.g., negative, non-integer), the program should prompt the user to re-enter both numbers until valid inputs are provided.
2. **Processing:** Implement the Euclidean algorithm to compute the GCD. The algorithm involves repeatedly subtracting the smaller number from the larger one or, more efficiently, computing remainders using division until reaching a zero remainder.
3. **Output:** Display the GCD of the two numbers with an explanatory message.

Euclidean Algorithm Explanation

The Euclidean algorithm is based on the principle that the GCD of two numbers also divides their difference. To find the GCD of two numbers, a and b, where $a > b$, follow these steps:

1. Compute the remainder of a divided by b.
2. Replace a with b and b with the remainder from step 1.
3. Repeat the process until b equals 0. The non-zero remainder now will be the GCD of a and b.

Hints

You can implement this using a loop where you repeatedly update the values of a and b with b and $a\%b$ respectively, until b becomes zero. The final value of a at this point will be the GCD.

Sample Run

```
Enter two non-negative integers: 48 18
The GCD of 48 and 18 is 6.
```

3 Finding all Prime Numbers in an Interval

This part requires you to develop a C program that finds and displays all prime numbers within a specified interval. Understanding how to determine prime numbers efficiently is fundamental for solving problems in cryptography, number theory, and algorithms that require factorization or prime checking.

Problem Statement

Write a program that prompts the user to enter two integers representing the lower and upper bounds of an interval and then outputs all prime numbers within that interval, inclusive of the bounds.

Example

For the interval $[3, 10]$, the prime numbers are 3, 5, and 7.

Requirements

1. **Input Validation:** Ensure the user inputs two two non-negative integers where the first integer is less than or equal to the second. If the inputs are invalid or if the first integer is greater than the second, prompt the user to re-enter both numbers until valid inputs are provided.
2. **Processing:** Implement a method to check if a number is prime and use this method to find and display all primes within the given range.
3. **Output:** Display all prime numbers found in the specified interval separated by spaces or commas. If no prime numbers are found, output an appropriate message.

Method to Check if a Number is Prime

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. To determine if a number n is prime, you can follow these steps:

1. If n is less than 2, then n is not prime.
2. Check if n is divisible by any integer from 2 to the square root of n. If you find a divisor in this range, n is not prime.
3. If no divisors are found, then n is prime.

This method is efficient because it only tests divisibility up to the square root of n. Testing for divisors beyond the square root is unnecessary since if n had a factor larger than its square root, it would necessarily have a smaller co-factor that would have already been checked.

Hints

Use the trial division method described above for checking prime numbers. This method is effective for the ranges typically involved in programming assignments.

Sample Runs

```
Enter the lower and upper bounds of the interval: 10 3
Invalid Input.
Enter the lower and upper bounds of the interval:

Enter the lower and upper bounds of the interval: 3 10
Prime numbers in the interval [3, 10]: 3, 5, 7

Enter the lower and upper bounds of the interval: 14 19
Prime numbers in the interval [14, 19]: 17, 19
```

Part 2: Debugging

Problem Description

Ali and Catherine have developed a program that prompts the user to enter a sequence of numbers and arithmetic signs ('+' or '-'). The program should correctly calculate the cumulative sum of these numbers, considering the arithmetic signs. If the sum of the numbers falls below zero, the program should stop and output "Sum fell below zero."

Debugging Tasks

Unfortunately, the program is not functioning as expected. Your task is to identify and correct the errors in the program. You will be given a segment of the code that includes logical errors related to input validation and calculation.

Provided Code

```
#include <stdbool.h>
#include <stdio.h>

int main(void) {
    char userChar;
    int sum = 0, sign = 1;  // sign initialized to 1 for positive
    bool number = true;

    printf("Enter sequence of characters with numbers to add/subtract: ");
    scanf(" %c", &userChar);

    do {
        if (number) {
            while (userChar < '0' || userChar > '9') {
                printf("Invalid! Re-enter number.\n");
                scanf(" %c", &userChar);
            }
            sum += sign * (userChar - '0');
            number = false;
        } else {
            while (userChar != '+' && userChar != '-') {
                printf("Invalid! Re-enter sign.\n");
                scanf(" %c", &userChar);
            }
            sign = (userChar == '+') ? 1 : -1;
            number = true;
        }
        scanf(" %c", &userChar);
```

```
    } while (sum >= 0);

    printf("Sum fell below zero.\n");
    return 0;
}
```

Debugging Instructions

1. Verify that the loop conditions for checking numbers and signs are correctly implemented.
2. Ensure that the logic for updating the sum based on the sign is correctly handled.
3. Test the program with various inputs to confirm that it behaves as expected.

Demo and Report Submission

Demonstrate the correct operation of your programs to your TA and be readiy to answer the TA quesitons about your code. Submit a report of the lab in DropBox for evaluation. Include your `.c` files containing your source code for the diffrent programming parts. Your code should include appropriate comments explaining the implementation of the algorithms used. The file should have the group number, names of group members, date, and any additional notes at the top as comments.

General Demo and Report Evaluation Criteria

Herefollows a detailed criteria for assessing the students demonstrations, reports and projects. Each criterion focuses on a key aspect of software development, aiming to ensure that students not only complete the tasks but also adhere to best practices in coding and project documentation.
Note:*Depending on the lab specifics some of the below points might be excluded.*

- **Correctness of Algorithm Implementation:**
  - The program must correctly implement the algorithm as described in the project specifications.
  - All functionalities outlined must be fully operational without errors during the demo.
  - Special cases and edge conditions should be correctly handled by the algorithm.
- **Proper Handling of Input Validation:**
  - The program should include robust input validation to reject invalid inputs without crashing or producing incorrect results.
  - Error messages should be informative and guide the user towards providing correct input.
  - All inputs in the demo must be shown to be properly validated to prevent data type errors, out-of-bound errors, or any other type of input-related issues.
- **Clarity of the Output Provided to the User:**
  - Output must be clear, precise, and correctly formatted according to project specifications.

  - The program should provide outputs that are easy to understand and logically presented, with labels or explanations if necessary.
  - Any numerical results should be displayed with appropriate units and an adequate number of significant figures.

- **Code Readability and Documentation:**
  - The source code must be neatly formatted with consistent indentation and clear, logical organization.
  - Comments should be used to explain the purpose of functions, the logic behind critical sections of code, and any complex algorithms.
  - The documentation provided (either as comments in the code or in separate documentation files) should clearly explain the program's structure, provide a usage guide, and list all dependencies.