ENGG1410: Introductory Programming for Engineers
Lab 2: "Introduction to C Programming": Complex Calculations and Decision Making, Debugging a C Program

School of Engineering, University of Guelph
Fall 2024

Start Date: Week #3 2024
Duration: 1 Week

1 Objectives:

The primary objective of this laboratory exercise is to enable students to put operators and math library into use and incorporate simple decision making and logic statements to perform required tasks.

- understanding the numbering systems: Base 10 **Decimal** and Base two **Binary** and converting numbers from one base to the other.
- Going through the steps of implementing a simple algorithm using C with more spohisticated operations and using functions from `<math.h>`
- Simple decison making within a program using `if` condition
- Diagnosing and rectifying both compile-time and run-time errors when they arise, enhancing the problem-solving and debugging skills.

2 Introduction

In lab 2 you will be writing four C prgrams and debugging a fifth program. The lab aims to deepen your understanding of basic C programming concepts while applying mathematical reasoning and decision-making skills to real-world problems. In this lab, you will work through several tasks that require the use of number systems, trigonometric calculations, time conversion, and debugging practices. By solving these problems, you will practice writing and debugging C programs, handling user input, performing calculations, and managing program flow with conditional statements. Each task builds upon fundamental programming concepts and reinforces your skills in translating problem-solving steps into efficient C code.

Throughout the lab, you will also develop an understanding of error diagnostics and resolution in C, learning to recognize and fix both logical and runtime errors. This hands-on experience will be invaluable as you progress to more complex programming challenges.

2.1 Lab sepcific objective

1. Understand and apply number systems: Convert decimal numbers to binary and familiarize yourself with the Base 2 system.
2. Perform geometric calculations: Use the cosine rule to compute angles between two vectors on a 2D plane.
3. Convert and manipulate time: Convert total minutes into hours and round to the nearest quarter hour, utilizing division, modulus operations, and conditional logic.
4. Implement basic arithmetic algorithms: Write a program to calculate the sum of digits of a given integer, applying basic math and control structures in C.
5. Develop debugging skills: Fix a buggy C program by diagnosing errors, applying logic to correct the program's flow, and using debugging tools to ensure correct functionality.

These exercises will prepare you to handle more complex programming problems by reinforcing core C programming skills such as input/output, mathematical functions, conditionals, and debugging.

3 Part 1 – Conversion to Base 2

As you begin your journey into the world of computer programming, it's important to understand how computers represent numbers using "bits" (short for binary digits). From now on, you will frequently encounter binary numbers, also known as Base 2 representation. Your task is to write a program that converts a decimal whole number i.e. Base 10 integer, no greater than 15, into its Base 2 representation

Write a C program that asks the user to enter an integer number no greater than 15 , and outputs on separate lines, the four digits of the base 2 representation of that entered number. Below is a sample input and output sequence for the program.

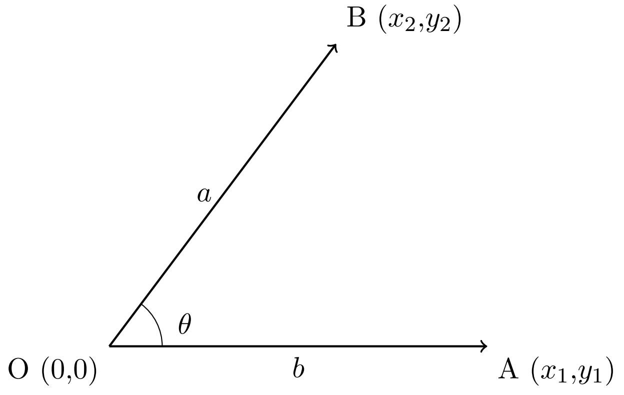
```
   Enter number to convert to base 2: 13<enter>
The four digits of that number are as follows:
Most significant digit: 1
Next digit: 1
Next digit: 0
Least significant digit: 1
```

For the example above, the program convert 13 in base 10 to 1101 in base 2. So $(13)_{10} \rightarrow (1101)_2$ The left-most digit of any number is referred to as the "most significant digit", and the right-most digit is referred to as the "least significant digit".

Note: if the user enters a number bigger than 16, output a message indicating you only handle numbers less than 16 and exit the program. With numbers less than 16 you should always output four bits, regardless of the size of the input number. Once we are introduced in lectures to more programming methods and techniques *looping and more decision making capapbilities* we will see how we can modify the programs to handle any number. For now we are restricting the input to be inthe range [0-15]. For example, if the input number is '3', then the output four digits will be 0011 . You will find it helpful to make use of the modulo operator (%) which outputs the remainder after a division.
Your C program should be saved in a file named `Lab2Part1Binary.c`.

4 Part 2 – Trigonometry - using the cosine rule

Given two points in a 2D plane, $A(x_1, y_1)$ and $B(x_2, y_2)$, Write a program to find the angle θ between the vectors from the origin $O(0, 0)$ to these two points using the cosine rule.



$c^2 = a^2 + b^2 - 2ab\cos(\theta)$

The formula to find the angle θ is derived using the dot product and magnitudes of the vectors. The angle is given by:

$$\cos(\theta) = \frac{x_1 \cdot x_2 + y_1 \cdot y_2}{\sqrt{x_1^2 + y_1^2} \cdot \sqrt{x_2^2 + y_2^2}} \tag{1}$$

Write a program to output the angle θ for any tow user input points.

You should ask the user to enter the coordinates of the two points and you will print as output the angle between them and the origin, using the **cosine rule** where point A and B are (x_1, y_1) and (x_2, y_2) respectively. **Hint** scanf the coordinate of each point using one scanf so for each point. So for point A for eample you will use `scanf("%lf,%lf", &x1, &y1)`
Note: look closely at how the %lf format spexcifiers are written and the **"coma"** between them

Sample Output:

```
Enter point A as (x1,y1): 3,4<enter>
Enter point B as (x2,y2): 4,3<enter>
The anlge theta between the two points is: 0.238 radian or 16.26 degree.
```

Notes

- Use the `math.h` library to calculate the necessary square root and for the value of π use (`M_PI`).
- Ensure that you handle floating-point input correctly.
- The values should be displayed in the required format.

Your C program should be saved in a file named `Lab2Part2CosineRule.c`.

5 Part 3 – Conversion of Minutes to Hours and Nearest Quarter Hour

It is important to understand how various mathematical concepts can be applied in practical programs. In this task, you will write a program that converts a given total number of minutes into hours and minutes, and then approximates this time to the nearest quarter hour. This will help solidify your understanding of division, modulus operations, and simple conditional statements.

Write a C program that asks the user to input a total number of minutes as an integer. The program should perform the following tasks:

- Convert the total number of minutes into hours and remaining minutes.
- Round the remaining minutes to the nearest quarter of an hour (0, 15, 30, or 45 minutes).

Below is a sample input and output sequence for the program:

```
    Enter total minutes: 138<enter>
That is equivalent to:
2 hours and 15 minutes (rounded to nearest quarter hour)
```

For the example above, the program converts 138 minutes to 2 hours and 18 minutes, then rounds this to the nearest quarter hour, giving 2 hours and 15 minutes. The rounding follows these rules:

- If the remaining minutes are between 0 and 7, round to 0 minutes.
- If the remaining minutes are between 8 and 22, round to 15 minutes.
- If the remaining minutes are between 23 and 37, round to 30 minutes.
- If the remaining minutes are between 38 and 52, round to 45 minutes.
- If the remaining minutes are between 53 and 59, round up to the next hour.

Note: You should handle large values for the total number of minutes. If the total number of minutes results in an approximation that rounds up to the next hour, adjust the hour count accordingly.

Once we cover more advanced programming concepts like loops and additional decision-making techniques in later lectures, we will revisit this program to handle more complex requirements. For now, this problem will focus on converting and rounding time using basic operations and conditions.

Your C program should be saved in a file named `Lab2Part3MinToHrs.c`.

6 Part 4 — Summing the Digits of an Integer (with Input Limits)

In this task, you will write a program that computes the sum of the digits of a given integer number. To simplify the problem, the input will be restricted to numbers between 0 and 99999. This task will help you practice using basic arithmetic operations, which we have already covered in lectures.

Write a C program that asks the user to input an integer number within the range 0 to 99999, and then outputs the sum of its digits, ignoring the sign of the number. The program should perform the following steps:

- Accept an integer number between 0 and 99999 as input from the user.
- Extract each digit of the number and compute the sum of all digits, ignoring the sign of the number.

Below is a sample input and output sequence for the program:

```
Enter an integer number between -99999 and 99999: 246<enter>
The sum of the digits is: 2 + 4 + 6 = 12
```

For the example above, the program reads the input 246, extracts the digits 2, 4, and 6, and computes their sum, which is 12.

Note: You do not need to handle numbers outside of the range $[0, 99999]$. If the user enters a number outside of this range, simply display the message

```
Out of Range
```

and exit the program without processing it. Once we cover more programming concepts such as conditions and loops, we will revisit this problem and extend the range of inputs we can handle.

For now, ensure your program works for all valid numbers in the specified range.

Your C program should be saved in a file named `Lab2Part4SumDigits.c`.

6.1 Optional

Extend your program to handle negative as well as positive numbers of 5 digits? (i.e. integer number between -99999 and 99999.

7 Part 5 Debugging

Patrick wants to use a 4-digit combination lock on his school locker, where he stores his personal items during the day. As a security measure, Chris changes the combination every week. To keep track of the combinations without risking someone discovering them, he writes them down using a coding scheme. If anyone finds his note, they still won't be able to figure out the real combination.

The coding scheme Chris uses involves swapping the 1st and 4th digits of the real combination, and replacing the 2nd and 3rd digits with their 9's complement. In other words, a combination of the form **AbcD** is encoded as $D(9-b)(9-c)A$.

For example, if the actual combination is 0428, the encrypted combination will be 8570 (note that 0 and 8 are swapped, 4 is replaced by 9-4, and 2 is replaced by 922. *A key feature of this coding scheme is that applying the same process again to the encrypted combination will return the original combination.*

Patrick is still learing to program using C so he wrote the following C program to implement his secret coding scheme and asked his lab partner Maria to test his code program. She entered 8021 But the program did not output the correct result of 1978!

```c
#include <stdio.h>
int main(void) {
  int encComb;
  printf("Enter an encrypted 4-digit combination: ");
  scanf("%d", &encComb);
  // Determine the 4 digits of the encrypted combinaiton.
  int d4, d3, d2, d1;
  encComb = encComb / 1000;
  d4 = encComb % 1000;
  encComb = encComb / 100;
  d3 = encComb % 100;
  encComb = encComb / 10;
  d2 = encComb % 10;
  d1 = encComb;
  // printing the decryped combination: d4 and d1 are swaped.
  // d3 and d2 are 9s complemented
  printf("\nThe real combination is: %d%d%d%d\n", d1, 9 - d3, 9 - d2, d4);
return 0;
}
```

You were asked to come to the rescue and fix the errors in the program.
Use debugger by adding watchpoints, examining the program variables progressing through the execution of the program step by step. Note: You may assume that the entered combination is a valid 4-digit integer number. No negative numbers will be entered by the user. This means the numbers will be between 0000 and 9999. When reading the 4-digit combination from user input, you are not allowed to scan in individual characters; instead, you should scan in a single integer using `scanf`.

8 Requirements

1. Write, compile and **execute** the program in Parts 1, 2, 3 and 4 and Demo them to your TA.
2. Fix the program in part 5 to work correctly. (consider different test cases).

9 Deliverables

9.1 DEMO:

- You will need to demonstrate to the Teaching Assistant that your programs fulfill the requirements and are correct. TAs might run test cases to check correctness.
- The DEMO will take place during the LAB hours and not outside these hours.
- For full demo marks at least 2 parts need to be correct.

9.2 REPORT

- You will provide a report with your solutions to the different parts of the lab and your source code and executables. All these should be uploaded to the coure DropBox in due date to get your full grade and avoid penalties.

9.2.1 Lab Report File:

Below is the general format of a final report (when required).
Note: Not all sections mentioned below are needed, they are given as a guidance.

1. Title Page: *Required*
   - Course #, and Date
   - Lab # and name of experiment
   - Your Group #, and Names (if lab is carried as a group)

2. **Start a new page** and explain how you implemented your code by providing the following:
    (a) **Problem Statement**,
        i. Briefly describe the problem(s) solved in the lab.
    (b) **Assumptions and Constraints** - if there were any.
        i. Assumptions could be for example specific number used e.g. taxes 7%.
        ii. Constraints could be for example no optimization with compilation.
    (c) **How you solved the Problem**,
        i. Pseudo Code.
        ii. Flowchart.
        iii. Block Diagram.
        iv. Another method ...
    (d) **System Overview & Justification of Design** *optional*
        i. Give an overview of the system to be designed.
        ii. Briefly explain how the system works and reasons behind the design.
3. **Error Analysis** *optional*
    (a) Confirm no syntax errors are present.
    (b) Confirm no semantic and logical errors are present.
    (c) Describe any problems with your program.
    (d) If no problems in the final system, describe problems/errors encountered during the development and how they were resolved.
4. **Conclusion and self assessment** *Required*
    (a) What you learned in the lab.
    (b) What level of understanding or experteese you think you attained.
    (c) Any suggestions *optional*

9.2.2 Source Files and Executables:

Inlcude your source code to the different lab parts.

The Report as well as your sourcecode files and executables should be uploaded to the course Dropbox in due date to get your grade and avoid penalties. You are responsible to provide correct files.