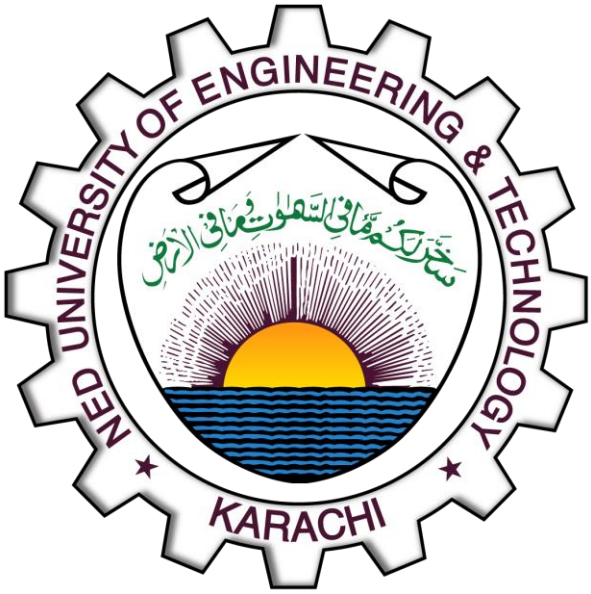
Currency Converter

CCP Report

PF (CT-175)



Group Name:

ZenCoders

Submitted By:

- Muhammad Tayyab Khan (CT-25088)
- Muhammad Ayaan (CT-24288)
- Vishal Kumar (CT-25081)

Teachers:

Sir Abdullah

Index

Table of Contents

1. Abstract

This report presents a currency converter application developed in C language as part of the CCP course (PF CT-175). The goal was to create a scalable, efficient, and user-friendly tool that allows users to convert between over 150 currencies using real-world exchange rates.

The program uses core C concepts such as structures (struct), file handling (fopen, fscanf, fclose), and input validation to build a robust console-based application. Exchange rates are stored in an external file (rates.txt), allowing updates without modifying the code. This separation of logic and data reflects good software design and makes the system highly maintainable.

To simplify conversions, the program uses a Base-USD model. Instead of calculating every possible currency pair, it converts all currencies to USD first, then to the target currency. This reduces complexity and improves performance.

The application includes a menu-driven interface with options to convert currencies, list available codes, and exit. It also handles invalid inputs gracefully, ensuring a smooth user experience.

Overall, this project provided hands-on experience with practical programming techniques and demonstrated how foundational C skills can be applied to solve real-world problems.

2. Introduction

This report provides a comprehensive overview of the "Currency Converter" application, a project for the Programming Fundamentals (PF CT-175) course. This document outlines the project's objectives, the technical design, and the logical processes that power the application.

2.1. Project Objective

The primary objective of this project was to design and build a fully functional console application in the C language that can convert monetary values between a wide range of global currencies. The project was required to be **robust** and **scalable.**

2.2. Problem Statement

Many simple conversion programs are "hard-coded," meaning the currency rates are written directly into the source code. This is highly inefficient, as any change in rates requires the programmer to edit the .c file and recompile the entire application.Our solution was to design a program that separates the **logic** (the C code) from the **data** (the currency rates). This was achieved by storing all conversion rates in an external rates.txt file, which the C program reads upon startup.

2.3. Scope & Limitations

The program operates as a menu-driven console application. Its scope includes:

- Loading over 180 currency rates from rates.txt.
- A menu to (1) Convert, (2) List all currencies, or (3) Exit.
- Handling invalid user input (e.g., text instead of numbers).
- The primary limitation is that the rates.txt file is static and must be updated manually. The program does not connect to the internet to fetch live rates.

3. Code Analysis & Usage

3.1. Code Understanding

The core of this program is its scalability, which is achieved through two key C concepts: structs and FILE handling.

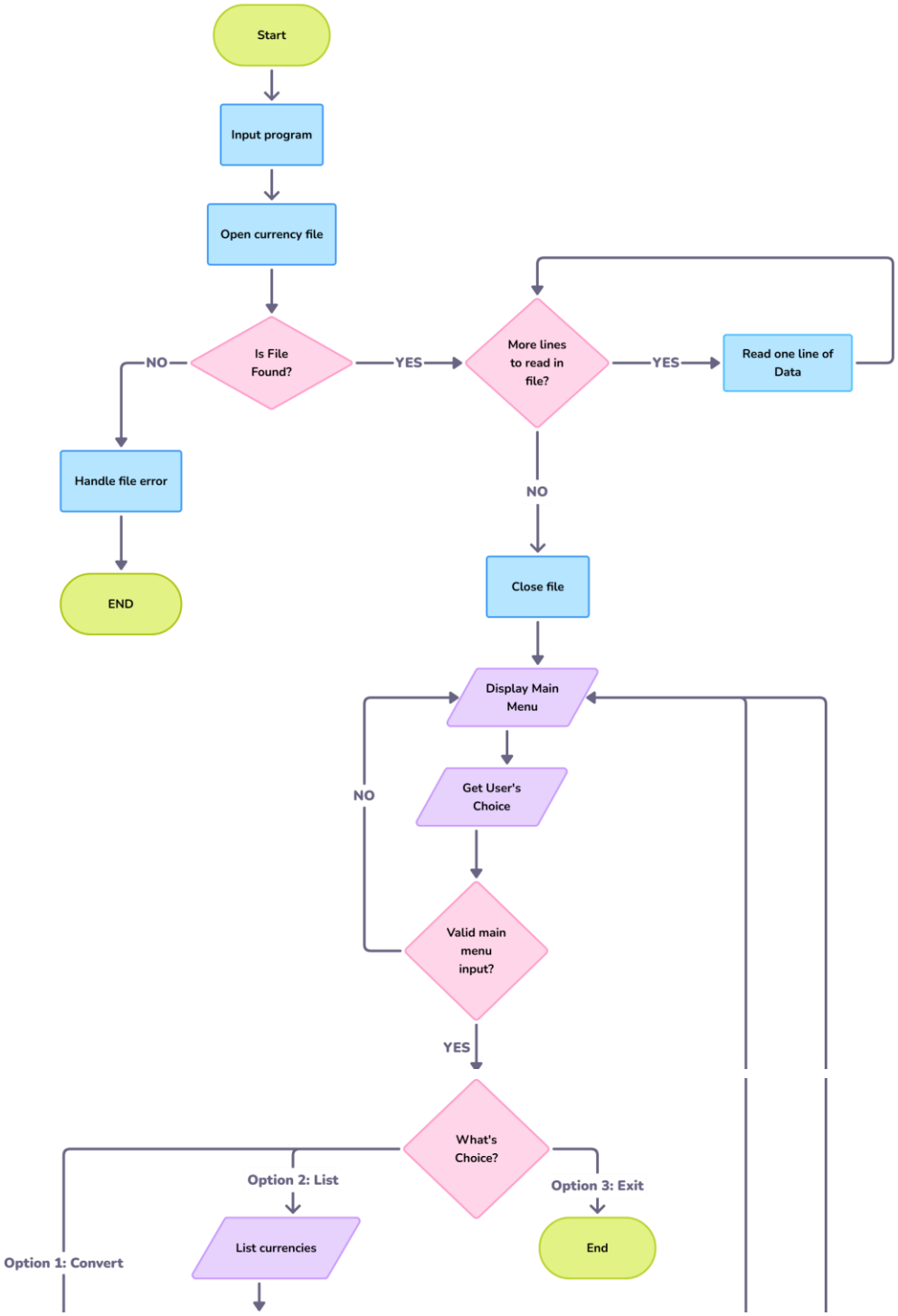
- **struct Currency:** A struct is used as a "blueprint" to hold all the data for a single currency (code, name, and rate). This is far more organized than using separate arrays.
- **FILE* and fopen():** On startup, the program opens rates.txt. This external file holds all 180+ currency rates. This means we can add, remove, or update currency rates without ever recompiling the C code.
- **fscanf() Loop:** The program reads the rates.txt file line-by-line using fscanf inside a while loop. This loop automatically loads all currencies into an array of structs (struct Currency currencies[200]).
- **Base-USD Logic:** The program does not need 22,000+ conversion pairs. It uses a single base currency (USD). To convert PKR to EUR, it first converts (PKR -> USD), and then (USD -> EUR). This two-step formula works for all currencies.

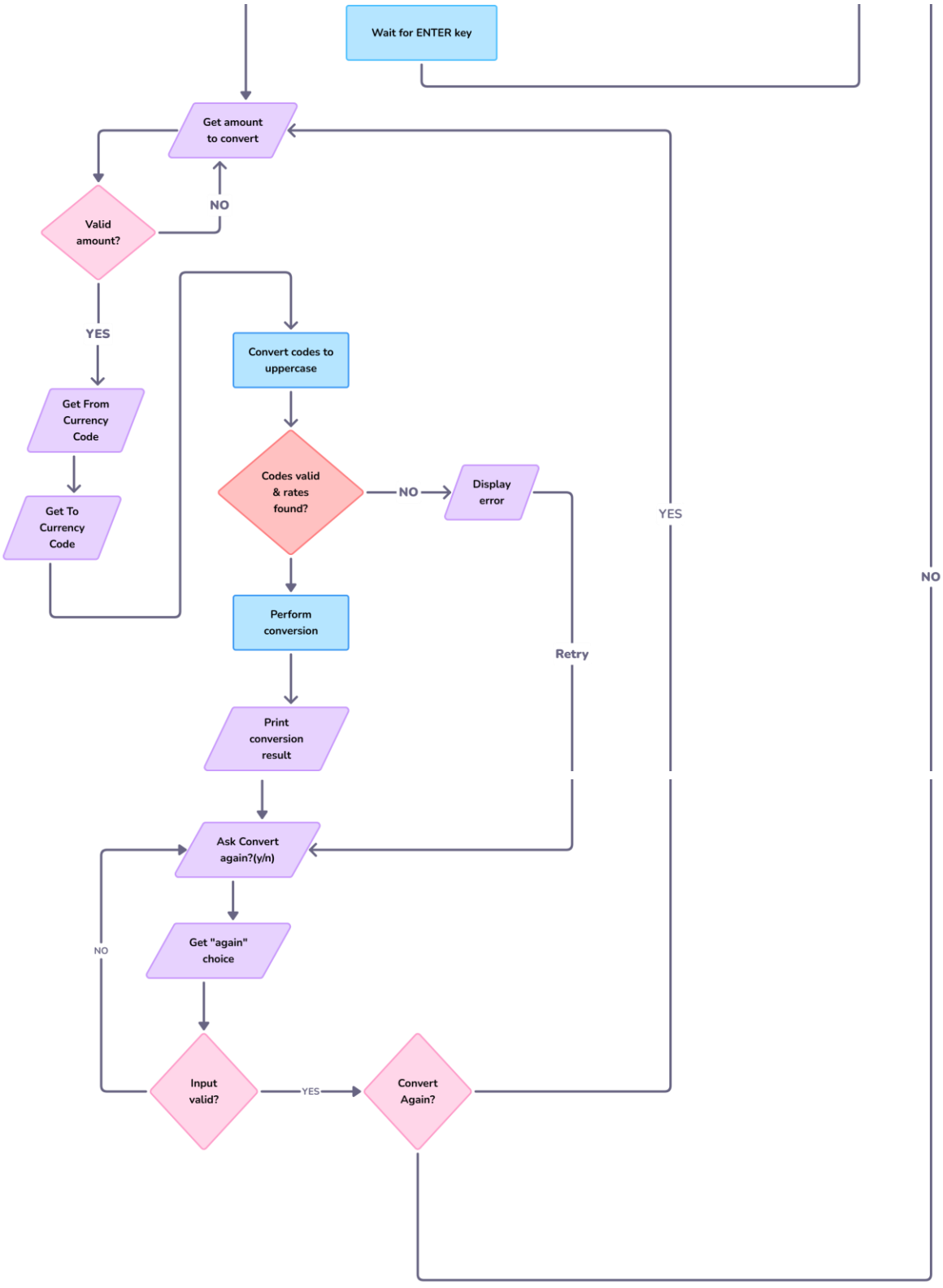
3.2. Program Usage

The program is a simple, menu-driven console application:

- **Menu 1 (Convert):** The user provides an amount, a "from" code (e.g., PKR), and a "to" code (e.g., USD). The program finds the rates and prints the result. It also handles user errors (like "pkr") by converting all input to uppercase.
- **Menu 2 (List Currencies):** This prints all 150+ currencies loaded from the rates.txt file, so the user knows which codes are available to use.
- **Menu 3 (Exit):** This safely closes the program.

4. Program Flowchart




Wait for ENTER key
Get amount to convert
Valid amount?
NO
YES
Get From Currency Code
Get To Currency Code
Convert codes to uppercase
Codes valid & rates found?
NO
Display error
Retry
YES
NO
Perform conversion
Print conversion result
Ask Convert again?(y/n)
Get "again" choice
NO
Input valid?
YES
Convert Again?

5. Key Code Snippets

5.1. Snippet 1: The Data Structure (struct)

Code:

```
struct Currency {
    char code[10];
    char name[50];
    double rate_to_usd;
};

struct Currency currencies[200];
int currency_count = 0;
```

Working: This struct is the blueprint for our entire program. Instead of using separate arrays (which is messy), we group all the data for a *single* currency into one organized variable. code[10] stores the 3-letter code like "PKR", name[50] stores "Pakistani Rupee", and rate_to_usd stores the rate.

We then create a single array called currencies that can hold 200 of these structs. This organized array is what we use to store all the data we read from the file.

5.2. Snippet 2: File Opening & Error Handling

Code:

```
FILE *file;
file = fopen("rates.txt", "r");

if (file == NULL) {
    printf("Error: Could not open rates.txt file.\n");
    printf("Please make sure the file is in the same folder as the program.\n");
    printf("Press Enter to exit.\n");
    getchar();
    return 1;
```

```
}
```

Working: This code is responsible for opening the data file. It uses a FILE pointer and the fopen function to open rates.txt in "read" mode ("r").

The if (file == NULL) block is a critical safety check. If the file is missing or cannot be opened, fopen returns NULL. This code catches that error, prints a user-friendly message, and exits the program safely. This prevents a "Segmentation Fault" crash.

5.3. Snippet 3: The fscanf Data Parsing Loop

Code:

```
while(fscanf(file, "%[^,],%[^,],%lf\n",
      currencies[currency_count].code,
      currencies[currency_count].name,
      &currencies[currency_count].rate_to_usd) == 3)
{
   currency_count++;
   if (currency_count >= 200) {
      printf("Warning: Hit currency limit. Loaded 200 currencies.\n");
      break;
   }
}
fclose(file);
```

Working: This while loop is the engine that reads the data. fscanf returns the number of items it successfully read. We expect 3 items per line.

The format string "%[^,],%[^,],%lf\n" is the most important part. [^,] tells fscanf to "read all characters *until* you hit a comma." This allows us to read the code, then the name (including spaces), and then the double (rate). We store this data directly into the currencies array at the current currency_count index, and then add 1 to the count. Finally, we fclose(file).

5.4. Snippet 4: The scanf Input Buffer Bug Fix

Code:

```
if (scanf("%d", &choice) != 1) {
    while(getchar() != '\n');
    printf("\nInvalid input. Please enter a number.\n\n");
    continue;
}
```

Working: This code block makes our program robust. A famous C bug is that if the user types letters (like 'abc') when scanf("%d", ...) expects a number, the program will break or go into an infinite loop.

Our code fixes this. scanf returns the number of items it read (we expect 1). If it returns something else (like 0), we know the input was bad. The line while(getchar() != '\n'); runs, which "flushes" all the bad 'abc' text out of the input buffer. We then print an error and continue to restart the main menu loop.

5.5. Snippet 5: Search & Conversion Logic

Code:

```
double from_rate = 0.0, to_rate = 0.0;
int from_found = 0, to_found = 0;

for(int i = 0; i < currency_count; i++) {
    if (strcmp(from_upper, currencies[i].code) == 0) {
        from_rate = currencies[i].rate_to_usd;
        from_found = 1;
    }
    if (strcmp(to_upper, currencies[i].code) == 0) {
        to_rate = currencies[i].rate_to_usd;
        to_found = 1;
    }
}
if (from_found == 0 || to_found == 0) {
    printf("Error: Invalid currency code entered.\n");
} else {
    double base_amount = amount / from_rate;
    double final_amount = base_amount * to_rate;
```

```
    printf("%.2f %s = %.2f %s\n", amount, from_upper, final_amount, to_upper);
}
```

Working: This is the "Base-USD" algorithm. First, we loop through our currencies array from 0 to currency_count. We use strcmp (string compare) to find a match for the "from" and "to" codes (which were converted to uppercase earlier).

After the loop, we check our from_found and to_found flags. If either is 0 (false), we print an error.

If both are found, we apply the two-step formula:

1. base_amount = amount / from_rate (This converts the source amount to USD).
2. final_amount = base_amount * to_rate (This converts the USD amount to the target currency).

Finally, we print the result formatted to two decimal places.

6. Program Output (Screenshots)

Figure6.1: Main Menu

This screenshot shows the main menu when the program first starts. It correctly identifies that **152** currencies were loaded from the rates.txt file.

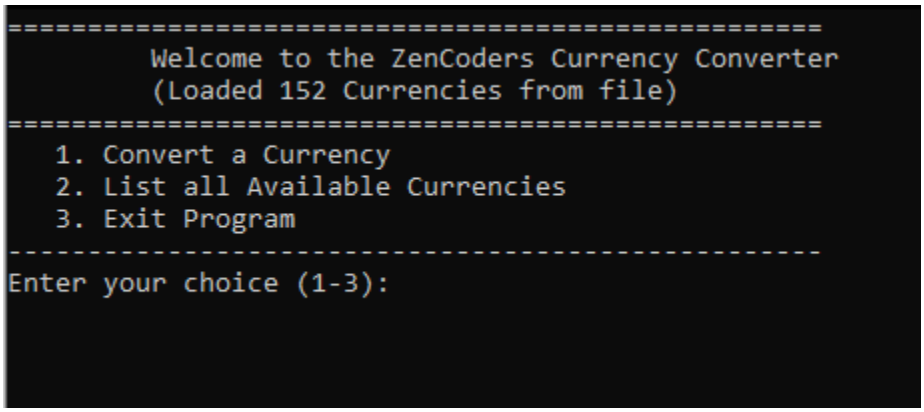


Figure 6.2: Successful Conversion

This screenshot shows the process for case 1. The user converts 5000 PKR to EUR. The program correctly finds the rates and prints the converted amount.

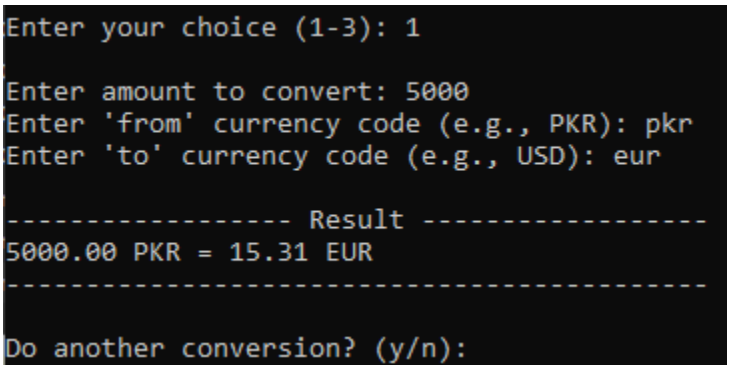
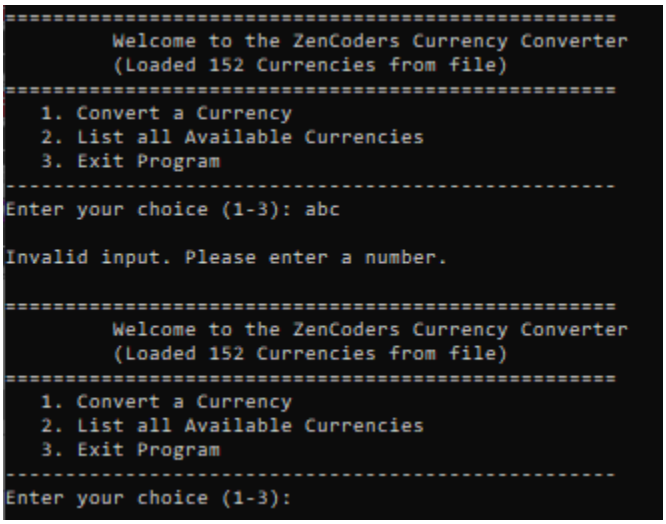


Figure 6.3: List All Currencies

This screenshot shows the output for case 2. The program prints a neatly formatted list of all **152** available currency codes and their names, so the user knows what codes to use.

Figure 6.4: Error Handling

This screenshot demonstrates the robust input handling. The user entered "abc" at the main menu, and the program correctly identified it as invalid input, printed an error, and re-loaded the menu instead of crashing.



7. Code Improvement

This C project is a strong foundation that successfully meets all core requirements. However, there are several ways this program could be improved and expanded in the future.

7.1. Graphical User Interface (GUI)

The current program runs in a text-based console. The most significant improvement would be to build a GUI.

- **Method:** This could be done in C using a library like GTK+, or by porting the logic to C++ (Qt) or Python (Tkinter).
- **Benefit:** A GUI would be far more user-friendly, replacing text-based code entry (e.g., "PKR") with drop-down menus, which would eliminate user error.

7.2. Live Exchange Rates via API

The current rates.txt file is static and must be updated manually. A more advanced version would connect to the internet to get live data.

- **Method:** This would require using a library like libcurl in C to make an HTTP request to a free currency API (like ExchangeRate-API). The program would parse the (JSON) response to get the latest rates every time it starts.
- **Benefit:** This would make the converter accurate to the minute and would eliminate the need for any manual file management.

7.3. More Robust Error Handling

Further enhancements could include adding checks for negative numbers and implementing a search function for the "List Currencies" menu. A search would improve usability by making it easier to find a code in the **152**-entry list.

8. Conclusion

This project was a comprehensive and practical exercise in C programming that went far beyond basic syntax. As **ZenCoders**, our group successfully designed and built a console application that is not just functional, but also robust, scalable, and efficient—principles that are critical in professional software development.

The most important takeaway was learning to **separate logic from data**. By using FILE handling to read rates.txt, we created a program that can be updated (with new currencies or new rates) without ever touching or recompiling the C code. This is a powerful, real-world concept.

We gained direct experience with key C topics:

- **Data Structuring:** Using structs to organize data was a major lesson in efficiency, making the code far cleaner than managing parallel arrays.

- **File I/O:** We mastered fopen, fscanf, and fclose, including the critical step of checking for NULL to prevent crashes if the file is missing.
- **Robust Input:** We learned the hard way *why* scanf is tricky. Implementing the if (scanf(...) != 1) and while(getchar() != '\n') loops taught us the importance of handling bad user input and managing the input buffer to prevent bugs.
- **Algorithmic Logic:** Designing the two-step "Base-USD" conversion formula was a key insight. It simplified a problem that seemed to require 22,000+ calculations down to one single, reusable formula.

Ultimately, this project was a success. We achieved all our objectives and produced a final program that is complete, user-friendly, and demonstrates a strong, practical understanding of core C programming.