

Assignment Set 1

**Preambles with Logical Steps and Code for Problem Statement**

*B.Tech. 3rd Semester Assignments*

***1a. Take an extensive program written by others in C language. Perform the frequency analysis of characters, reserved words.***

Ans: Preamble for Analyzing Character and Reserved Word Frequencies in a C Program.

**Problem Statement:** The task at hand involves analysing an extensive C program written by others to perform frequency analysis on two key aspects: the characters used in the program and the occurrence of reserved words. This analysis will provide insights into the program's structure, coding style, and language usage.

**Preface:** To successfully perform this analysis, we need to specify the format and source of the C program, explain our approach to conducting the frequency analysis, and clarify the format of the output.

**Input Format:** The source of the C program will be provided as a text file or a collection of source code files. The program may consist of multiple files and contain code written in the C programming language.

**Approach to Perform Frequency Analysis:** Our approach to analysing the extensive C program includes the following steps: We read the source code, initialize data structures for character and reserved word frequency, analyse character and reserved word occurrences, and summarize the results for clear presentation.

**Format of the Output:** The output of the frequency analysis will be presented as a report or a text file. It will include two sections:

1. Character Frequency Analysis:

* A table or list showing the frequency of each character encountered in the code.
* The characters will be listed along with the number of times they appear.

1. Reserved Word Frequency Analysis:

* A list of reserved words from the C programming language along with the number of times each word appears in the code.

**Logical Steps:**

1. Start by reading the provided C program source code.
2. Initialize two dictionaries or lists for character frequency and reserved word frequency, setting all counters to zero.
3. For character frequency analysis:
   1. Iterate through the code character by character.
   2. For each character, increment the corresponding counter in the character frequency data structure.
4. For reserved word frequency analysis:
   1. Identify a list of reserved words used in C.
   2. Search for each reserved word in the code.
   3. When a reserved word is found, increment the corresponding counter in the reserved word frequency list.
5. After processing the entire code, create an output report.
6. In the report, present the character frequency analysis, followed by the reserved word frequency analysis.

**Program Code:**

#include <stdio.h>     // For Input/Output

#include <stdlib.h>    // For FileHandling

#include <string.h>    // For String

#include <ctype.h>     // For Characters

#include <sys/stat.h>  // For MKDIR

#include <sys/types.h> // For MKDIR

#define MAX\_WORD\_LEN 30

#define MAX\_FILENAME\_LENGTH 100

#define MAX\_OUTPUTFILENAME\_LENGTH 100

#define MAX\_FILEPATH\_LENGTH 200

// Structure to store information about reserved words

typedef struct

{

    char word[MAX\_WORD\_LEN];

    int frequency;

} ReservedWordInfo;

// Global declaration of keywords array

const char \*keywords[] = {

    "auto", "break", "case", "char", "const", "continue", "default", "do", "double", "else",

    "enum", "extern", "float", "for", "goto", "if", "int", "long", "register", "return", "short",

    "signed", "sizeof", "static", "struct", "switch", "typedef", "union", "unsigned", "void",

    "volatile", "while", "#define", "printf", "scanf", "#include", "bool"};

// Function to check if a character is a valid part of a word

int isValidChar(char ch)

{

    return isalnum(ch) || ch == '\_';

}

// Function to convert a string to lowercase

void toLowercase(char \*str)

{

    while (\*str)

    {

        \*str = tolower((unsigned char)\*str);

        str++;

    }

}

// Function to perform frequency analysis on characters

void analyzeCharacters(FILE \*file, int charFrequency[])

{

    int ch;

    while ((ch = fgetc(file)) != EOF)

    {

        // Increment the corresponding counter for all characters, including spaces

        charFrequency[ch]++;

    }

}

// Function to compare ReservedWordInfo structs for sorting

int compareReservedWordInfo(const void \*a, const void \*b)

{

    return ((ReservedWordInfo \*)b)->frequency - ((ReservedWordInfo \*)a)->frequency;

}

// Function to perform frequency analysis on reserved words

void analyzeReservedWords(FILE \*file, ReservedWordInfo reservedWords[], int \*totalReservedWords)

{

    char word[MAX\_WORD\_LEN];

    int ch;

    int index;

    while ((ch = fgetc(file)) != EOF)

    {

        if (isValidChar(ch) || ch == '#')

        {

            int i = 0;

            // Handle preprocessor directives

            if (ch == '#')

            {

                i = 0;

                word[i++] = ch;

                while ((ch = fgetc(file)) != EOF && isalnum(ch))

                {

                    word[i++] = ch;

                }

                word[i] = '\0';

                // Check if the word is #include or #define

                if (strcasecmp(word, "#include") == 0 || strcasecmp(word, "#define") == 0)

                {

                    // Update the frequency of the reserved word

                    int found = 0;

                    for (int j = 0; j < \*totalReservedWords; j++)

                    {

                        if (strcasecmp(word, reservedWords[j].word) == 0)

                        {

                            reservedWords[j].frequency++;

                            found = 1;

                            break;

                        }

                    }

                    // If the word is not in the array, add it

                    if (!found)

                    {

                        strcpy(reservedWords[\*totalReservedWords].word, word);

                        reservedWords[\*totalReservedWords].frequency = 1;

                        (\*totalReservedWords)++;

                    }

                }

                // Skip the rest of the line

                while (ch != '\n' && ch != EOF)

                {

                    ch = fgetc(file);

                }

                continue; // Move to the next iteration of the loop

            }

            // Process regular words

            while (isValidChar(ch) && i < MAX\_WORD\_LEN - 1)

            {

                word[i++] = ch;

                ch = fgetc(file);

            }

            word[i] = '\0';

            // Check if the word is a reserved word

            for (index = 0; index < sizeof(keywords) / sizeof(keywords[0]); index++)

            {

                if (strcasecmp(word, keywords[index]) == 0)

                {

                    // Update the frequency of the reserved word

                    int found = 0;

                    for (int j = 0; j < \*totalReservedWords; j++)

                    {

                        if (strcasecmp(word, reservedWords[j].word) == 0)

                        {

                            reservedWords[j].frequency++;

                            found = 1;

                            break;

                        }

                    }

                    // If the word is not in the array, add it

                    if (!found)

                    {

                        strcpy(reservedWords[\*totalReservedWords].word, word);

                        reservedWords[\*totalReservedWords].frequency = 1;

                        (\*totalReservedWords)++;

                    }

                    break;

                }

            }

        }

    }

}

// Function to analyze and count comments

void analyzeComments(FILE \*file, int \*singleLineComments, int \*multiLineComments)

{

    int ch;

    int insideComment = 0; // Flag to track whether currently inside a multi-line comment

    while ((ch = fgetc(file)) != EOF)

    {

        if (ch == '/')

        {

            ch = fgetc(file);

            if (ch == '/') // Single-line comment

            {

                (\*singleLineComments)++;

                while ((ch = fgetc(file)) != '\n' && ch != EOF)

                {

                    // Move to the end of the line

                }

            }

            else if (ch == '\*') // Multi-line comment

            {

                (\*multiLineComments)++;

                insideComment = 1;

                while (insideComment)

                {

                    while ((ch = fgetc(file)) != '\*')

                    {

                        if (ch == EOF)

                        {

                            insideComment = 0;

                            break;

                        }

                    }

                    ch = fgetc(file);

                    if (ch == '/')

                    {

                        insideComment = 0;

                    }

                }

            }

            else

            {

                // Not a comment, revert one character back

                ungetc(ch, file);

            }

        }

    }

}

// Function to save analysis results to a file

void saveToFile(char filename[], int charFrequency[], ReservedWordInfo reservedWords[], int totalReservedWords, int singleLineComments, int multiLineComments)

{

    const char \*folder\_name = "FrequenciesFiles";

    char outputFilename[MAX\_FILENAME\_LENGTH];

    // Create a new folder

    mkdir(folder\_name);

    snprintf(outputFilename, sizeof(outputFilename), "%s/%s\_freq.txt", folder\_name, filename);

    FILE \*outputFile = fopen(outputFilename, "w");

    if (!outputFile)

    {

        printf("Error opening output file!\n");

        return;

    }

    fprintf(outputFile, "Frequency Analysis of the %s program :\n\n", filename);

    // Save character frequency analysis to the file

    fprintf(outputFile, "Character Frequency Analysis:\n");

    for (int i = 32; i < 127; i++)

    {

        if (charFrequency[i] > 0)

        {

            fprintf(outputFile, "'%c': %d\n", i, charFrequency[i]);

        }

    }

    // Save reserved word frequency analysis to the file

    fprintf(outputFile, "\nReserved Word Frequency Analysis:\n");

    for (int i = 0; i < totalReservedWords; i++)

    {

        fprintf(outputFile, "%s: %d\n", reservedWords[i].word, reservedWords[i].frequency);

    }

    // Save comment analysis to the file

    fprintf(outputFile, "\nComment Analysis:\n");

    fprintf(outputFile, "Single-line comments: %d\n", singleLineComments);

    fprintf(outputFile, "Multi-line comments: %d\n", multiLineComments);

    fclose(outputFile);

}

int main()

{

    printf("\*\*\* Characters, Keywords, and Comments Frequency Analyzer \*\*\*\n\n");

    char filename[MAX\_FILENAME\_LENGTH];

    char filepath[MAX\_FILEPATH\_LENGTH];

    long unsigned int length;

    printf("Enter the Name of the File with its Extension = ");

    scanf("%s", filename);

    length = strlen(filename);

    if (((filename[length - 1]) != 'c') && ((filename[length - 1]) != 'C'))

    {

        printf("Only Files with Extension '.c' are Allowed!\n ");

        exit(1);

    }

    int choice;

    printf("Where does the File Exist ?\n");

    printf("1. In the same folder.\n");

    printf("2. I want to specify the filepath.\n");

    printf("Enter your choice = ");

    scanf("%d", &choice);

    FILE \*file;

    switch (choice)

    {

    case 1:

        // Open the C file for reading

        file = fopen(filename, "r");

        break;

    case 2:

        printf("Enter the Path of the File with its Name = ");

        scanf("%s", filepath);

        // Open the C file for reading

        file = fopen(filepath, "r");

        break;

    default:

        printf("Invalid Choice !!!\n");

        file = NULL;

        exit(1);

        break;

    }

    if (!file)

    {

        printf("Error Opening File!\n");

        printf("Or\n");

        printf("Can't Found the File %s .\n", filename);

        exit(1);

    }

    // Initialize character frequency array

    int charFrequency[256] = {0};

    // Analyze characters in the C program

    analyzeCharacters(file, charFrequency);

    // Close and reopen the file to reset the file pointer

    fclose(file);

    file = fopen(filename, "r");

    // Initialize reserved word frequency array

    ReservedWordInfo reservedWords[sizeof(keywords) / sizeof(keywords[0])] = {0};

    int totalReservedWords = 0;

    // Analyze reserved words in the C program

    analyzeReservedWords(file, reservedWords, &totalReservedWords);

    // Close and reopen the file to reset the file pointer

    fclose(file);

    file = fopen(filename, "r");

    int singleLineComments = 0;

    int multiLineComments = 0;

    // Analyze comments in the C program

    analyzeComments(file, &singleLineComments, &multiLineComments);

    fclose(file);

    // Sort reserved words by frequency in descending order

    qsort(reservedWords, sizeof(reservedWords) / sizeof(reservedWords[0]), sizeof(reservedWords[0]), compareReservedWordInfo);

// Display results for character frequency

    printf("Character Frequency Analysis:\n");

    int check = 0;

    for (int i = 32; i < 127; i++)

    {

        if (i <= 125)

        {

            if (charFrequency[i] > 0)

            {

                printf("'%c': %d  ,  ", i, charFrequency[i]);

                check++;

            }

            if (check > 8)

            {

                printf("\n");

                check = 0;

            }

        }

        else

        {

            printf("'%c': %d", i, charFrequency[i]);

        }

    }

    // Display results for reserved word frequency

    printf("\nReserved Word Frequency Analysis:\n");

    for (int i = 0; i < totalReservedWords; i++)

    {

        if (i <= totalReservedWords - 2)

        {

            printf("%s: %d   ,  ", reservedWords[i].word, reservedWords[i].frequency);

        }

        else

        {

            printf("%s: %d", reservedWords[i].word, reservedWords[i].frequency);

        }

    }

    // Display results for comments

    printf("\nComment Analysis:\n");

    printf("Single-line comments: %d\n", singleLineComments);

    printf("Multi-line comments: %d\n\n", multiLineComments);

    // Prompt the user to save the information to a file

    char saveToFileChoice[3];

    printf("\nDo you want to save the information to a file? (y/n) or (yes/no): ");

    scanf("%s", saveToFileChoice);

    // Convert the user's input to lowercase

    toLowercase(saveToFileChoice);

    if (strcmp(saveToFileChoice, "y") == 0 || strcmp(saveToFileChoice, "yes") == 0)

    {

        // Save the information to a file

        saveToFile(filename, charFrequency, reservedWords, totalReservedWords, singleLineComments, multiLineComments);

        printf("\nInformation saved to FrequenciesFiles/%s\_freq.txt\n\n", filename);

    }

    else if (strcmp(saveToFileChoice, "n") == 0 || strcmp(saveToFileChoice, "no") == 0)

    {

        printf("\nOk! Closing without saving in file.\n\n");

    }

    else

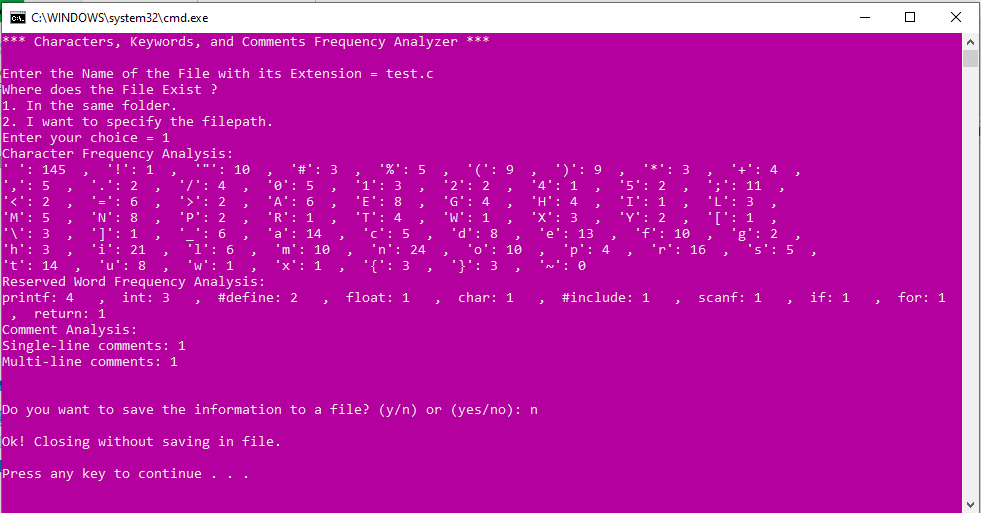
    {

        printf("\nInvalid Answer!!!\n\n");

    }

    return 0;

}

**Output Format: On Screen as well as Output Text File**

***1b. Take a series of ‘n’ numbers of characters and generate all permutations.***

Ans: Preamble for Generating All Permutations of a Series of Characters.

**Problem Statement:** The task at hand involves generating all possible permutations of a given series of 'n' characters. A permutation is an arrangement of elements in a specific order, and in this context, it means rearranging the characters in every possible way to create distinct combinations. For example, given the input 'abc', the permutations would be 'abc,' 'acb,' 'bac,' 'bca,' 'cab,' and 'cba.'

**Preface:** To approach this problem, we need to clarify certain aspects of our algorithm. We'll discuss the format of the input text file, our general approach to solving the problem, and the format of the output.

**Input Text File Format:** The input text file will contain a single line, representing the series of 'n' characters for which we want to generate permutations. The characters should be space-separated or provided in a comma-separated format.

**Approach to Perform Permutations:** Our approach to generating permutations will be based on a recursive algorithm. We will not delve into the specific implementation details or code in this document, as it is meant for the preamble and logical steps only. The algorithm will rely on recursion to systematically rearrange the characters, exploring all possible combinations.

**Format of the Output:** The output will be presented as a text file, containing each generated permutation on a separate line. The permutations will be written as strings with characters separated by spaces or commas, mirroring the format of the input text file.

**Logical Steps:**

1. Start by reading the input text file to obtain the series of 'n' characters.
2. Initialize a list to store the generated permutations.
3. Establish a recursive process that accepts the current characters, the characters not yet used, and the list of permutations as parameters.
4. In the recursive process:
   1. If there are no characters left to use, add the current permutation to the list.
   2. Otherwise, for each unused character, add it to the current permutation, call the process recursively with the updated parameters, and remove the added character to explore different combinations.
5. After the recursive calls, write the list of permutations to the output text file.

**Program Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_PERMUTATIONS 100000 // Maximum number of Permutations

#define MAX\_CHARACTERS 1000     // Maximum number of Characters

#define MAX\_HEADER\_LENGTH 1000  // Maximum length of Header

void generatePermutations(char \*current, char \*unused, int currentIndex, int n, char \*\*permutations, int \*count)

{

    if (currentIndex == n)

    {

        permutations[\*count] = strdup(current);

        (\*count)++;

        return;

    }

    for (int i = 0; i < strlen(unused); i++)

    {

        current[currentIndex] = unused[i];

        current[currentIndex + 1] = '\0'; // Null-terminate the current string

        char \*newUnused = (char \*)malloc((n - currentIndex) \* sizeof(char));

        // Copy the remaining unused characters to newUnused

        int k = 0;

        for (int j = 0; j < strlen(unused); j++)

        {

            if (j != i)

            {

                newUnused[k++] = unused[j];

            }

        }

        newUnused[k] = '\0'; // Null-terminate the newUnused string

        generatePermutations(current, newUnused, currentIndex + 1, n, permutations, count);

        free(newUnused);

    }

}

int main()

{

    char header[MAX\_HEADER\_LENGTH];

    char series[MAX\_CHARACTERS];

    const char \*inputFileName = "input.txt";

    const char \*outputFileName = "output.txt";

    // Open input file

    FILE \*inputFile = fopen(inputFileName, "r");

    if (inputFile == NULL)

    {

        fprintf(stderr, "Error opening input file\n");

        return 1;

    }

    fgets(header, sizeof(header), inputFile);

    // Read the series of 'n' characters

    fgets(series, sizeof(series), inputFile);

    if (series[0] == '\0' || series[0] == '\n')

    {

        printf("Error reading the series of characters.\n");

        fclose(inputFile);

        return 1;

    }

    printf("%s", header);     // Print the header for reference

    printf("'%s'\n", series); // Print the series for reference

    int n = strlen(series);

    if (n > 0 && series[n - 1] == '\n')

    {

        series[--n] = '\0'; // Remove the newline character if present

    }

    // Initialize variables

    char current[n + 1];

    char unused[n + 1];

    strcpy(unused, series);

    char \*permutations[MAX\_PERMUTATIONS];

    int count = 0;

    // Generate permutations

    generatePermutations(current, unused, 0, n, permutations, &count);

    // Open output file

    FILE \*outputFile = fopen(outputFileName, "w");

    if (outputFile == NULL)

    {

        fprintf(stderr, "Error opening output file\n");

        return 1;

    }

    fprintf(outputFile, "The Generated Permutations for the Characters %s are: \n\n", series);

    // Write permutations to the output file

    for (int i = 0; i < count; i++)

    {

        fprintf(outputFile, "%s\n", permutations[i]);

        free(permutations[i]);

    }

    fclose(outputFile);

    printf("Permutations generated for the characters %s and written to %s\n", series, outputFileName);

    return 0;

}

**Input Format: As Input Text File**

To Generate 5 Characters Permutations:

ABC

**Output Format: As Output Text File**

The Generated Permutations for the Characters ABC are:

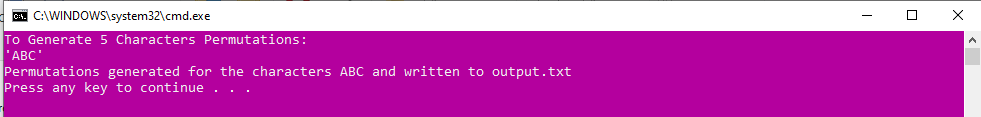
ABC

ACB

BAC

BCA

CAB

CBA

***1c. Take a ten-digit integer number and convert it to equivalent number representation in ROMAN and vice-versa.***

Ans: Preamble for Converting Integer Number to Roman Number and vice-versa.

**Problem Statement:** The task at hand involves converting a ten-digit integer number into its equivalent Roman numeral representation and vice versa. Roman numerals are a numeral system that originated in ancient Rome and were widely used in the Roman Empire.

**Preface:** To approach this problem, we need to create a program that takes a ten-digit integer as input and converts it into its Roman numeral representation. The input integer can range from 1,000,000,000 to 9,999,999,999. Additionally, the program should also be able to perform the reverse conversion, taking a Roman numeral as input and converting it back into its corresponding ten-digit integer.

**Input Text File Format:** The input is provided in a text file, with each line containing either a ten-digit integer or a Roman numeral for conversion. Use prefixes "INT:" for ten-digit integers and "ROMAN:" for Roman numerals. Each line represents a single conversion task.

**Approach to Convert the Numbers:** Our approach for this task involves mapping the Roman numerals (I, V, X, L, C, D, M) to their respective numeric values (1, 5, 10, 50, 100, 500, 1000). We will use this mapping to perform both conversions. The format of the input text file will consist of a single ten-digit integer per line for the integer-to-Roman conversion, and a Roman numeral per line for the Roman-to-integer conversion. The output will also be presented as a text file, with each line containing the input and its corresponding output.

**Format of the Output:** The output will be presented as a text file, containing one line for each conversion, with the format:

[Input] -> [Output]

For example, if the input is "1234567890," and it is converted to the Roman numeral "MCCXXXIV," the output in the text file will be:

1234567890 -> MCCXXXIV

**Logical Steps:**

1. Read the input text file line by line.
2. For each line, check if it represents a valid input:

* Use regular expressions or other appropriate methods to validate if the line is a valid ten-digit integer or a Roman numeral.
* If the line is a valid input, proceed to the conversion step; otherwise, skip it and display error message.

1. For each valid input, determine whether it's a ten-digit integer or a Roman numeral.
2. If the input is a ten-digit integer:
   1. Convert the ten-digit integer to its equivalent Roman numeral using the following steps:
      1. Break down the integer into its decimal places (billions, hundreds of millions, tens of millions, millions, etc.).
      2. Convert each decimal place to its Roman numeral equivalent independently.
      3. Combine the Roman numeral representations of all decimal places to form the complete Roman numeral.
   2. Write the input (ten-digit integer) and the corresponding Roman numeral to the output text file in the specified format: [Input] -> [Output]
3. If the input is a Roman numeral:
   1. Convert the Roman numeral to its corresponding ten-digit integer using the following steps:
      1. Initialize a variable to keep track of the total integer value.
      2. Iterate through the Roman numeral from left to right.
      3. Compare each Roman numeral character to the next character:

* If the value of the current character is less than the next character, subtract the current character's value from the total.
* If the value of the current character is greater than or equal to the next character, add the current character's value to the total.
  + 1. At the end of the iteration, the total will represent the equivalent ten-digit integer.
  1. Write the input (Roman numeral) and the corresponding ten-digit integer to the output text file in the specified format: [Input] -> [Output]

**Program Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

// Function to check if a given string is a valid ten-digit integer

int isValidInteger(const char \*line)

{

    int len = strlen(line);

    for (int i = 0; i < len; i++)

    {

        if (!isdigit(line[i]))

        {

            return 0;

        }

    }

    return 1;

}

// Function to check if a given string is a valid Roman numeral

int isValidRoman(const char \*line)

{

    const char \*validChars = "IVXLCDM̅";

    size\_t len = strlen(line);

    for (size\_t i = 0; i < len; i++)

    {

        if (strchr(validChars, line[i]) == NULL)

        {

            return 0;

        }

    }

    return 1;

}

// Function to check if a given string is a valid input (either integer or Roman numeral)

int isValidInput(const char \*line)

{

    return isValidInteger(line) || isValidRoman(line);

}

// Function to convert a ten-digit integer to its Roman numeral equivalent

void convertToRoman(long long number, char \*result)

{

    const char \*romanNumerals[] = {"\_M", "\_I\_M", "\_D", "\_I\_D", "\_C", "\_I\_C", "\_L", "\_I\_L", "\_X", "\_I\_X", "\_V", "\_I\_V", "M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};

    const long values[] = {1000000, 900000, 500000, 400000, 100000, 90000, 50000, 40000, 10000, 9000, 5000, 4000, 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};

    // Ensure the number is within the valid range

    if (number < 1 || number > 9999999) // Adjust the upper limit as needed

    {

        printf("Error: Input number is out of range\n");

        return;

    }

    int resIndex = 0; // Index for the result string

    for (int k = 0; k < 26; k++)

    {

        while (number >= values[k])

        {

            int arrIndex = k;

            // Append the Roman numeral to the result string

            const char \*currentSymbol = romanNumerals[arrIndex];

            size\_t len = strlen(currentSymbol);

            if (len == 1)

            {

                result[resIndex++] = currentSymbol[0];

            }

            else

            {

                for (size\_t i = 0; i < len; i++)

                {

                    result[resIndex++] = currentSymbol[i];

                }

            }

            number -= values[arrIndex];

            arrIndex++;

        }

    }

    // Null-terminate the result string

    result[resIndex] = '\0';

}

// Function to convert a Roman numeral to its corresponding ten-digit integer

long convertToInteger(const char \*roman)

{

    const char \*romanNumerals = "IVXLCDM";

    const int values[] = {1, 5, 10, 50, 100, 500, 1000};

    const char \*overline = "M\_";

    long total = 0;

    long prevValue = 0;

    for (int i = strlen(roman) - 1; i >= 0; i--)

    {

        char currentChar = roman[i];

        long value;

        if (i >= strlen(overline) && strncmp(&roman[i - strlen(overline) + 1], overline, strlen(overline)) == 0)

        {

            // If the current character is part of the overline sequence, skip it

            continue;

        }

        value = values[strchr(romanNumerals, currentChar) - romanNumerals];

        if (value < prevValue)

        {

            total -= value;

        }

        else

        {

            total += value;

        }

        prevValue = value;

    }

    return total;

}

// Function to process input file and write results to output file

int main()

{

    char line[20];

    char intResult[50];

    long number, integer;

    FILE \*inputFile = fopen("input.txt", "r");

    FILE \*outputFile = fopen("output.txt", "w");

    if (inputFile == NULL || outputFile == NULL)

    {

        perror("Error opening file");

        exit(EXIT\_FAILURE);

    }

    while (fgets(line, sizeof(line), inputFile) != NULL)

    {

        // Remove newline character if present

        size\_t len = strlen(line);

        if (len > 0 && line[len - 1] == '\n')

        {

            line[len - 1] = '\0';

        }

        if (isValidInput(line))

        {

            if (isdigit(line[0]))

            {

                number = atoll(line);

                convertToRoman(number, intResult);

                printf("%lld -> %s\n", number, intResult);

                fprintf(outputFile, "%lld -> %s\n", number, intResult);

            }

            else

            {

                integer = convertToInteger(line);

                printf("%s -> %lld\n", line, integer);

                fprintf(outputFile, "%s -> %lld\n", line, integer);

            }

        }

        else

        {

            printf("Invalid input: '%s'\n", line);

        }

        printf("\n");

    }

    fclose(inputFile);

    fclose(outputFile);

    return 0;

}

**Input Format: As Input Text File**

1234567

IX

27

635

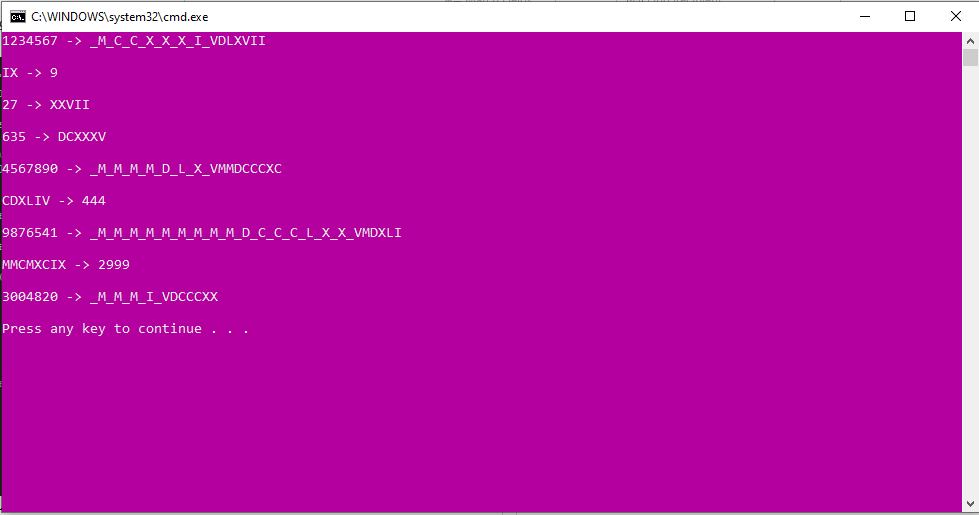
4567890

CDXLIV

9876541

MMCMXCIX

3004820

**Output Format: On Screen and as well as Output Text File**

***1d. Take a sufficiently long arithmetic expression with +,-,/,^ operators including parenthesis. Evaluate the expression.***

Ans: Preamble for Evaluating a Long Arithmetic Expression including parenthesis.

**Problem Statement:** The task at hand involves the evaluation of arithmetic expressions that can consist of various mathematical operations, including addition (+), subtraction (-), division (/), and exponentiation (^). These expressions may also include the use of parentheses to specify the order of operations. The ultimate goal is to determine the numerical result of the given expression.

**Preface:** To approach this problem, we are presented with the challenge of evaluating arithmetic expressions. These expressions are composed of common mathematical operators, such as addition, subtraction, division, and exponentiation including parenthesis. To ensure clarity and accuracy, it is essential to establish a structured approach to solving this problem.

**Input Text File Format:** The input is provided in a text file, it will contain one or more arithmetic expressions, each on a separate line. Each expression may consist of numbers, mathematical operators (+, -, /, ^), and parenthesis to indicate the order of operations.

**Approach to Evaluate Expression:** Our approach involves analysing arithmetic expressions retrieved from the input file, following standard order of operations, and carefully evaluating them step by step. We repeat this process for multiple expressions and document the results in the output file, all while avoiding programming-specific details.

**Format of the Output:** The output will be presented in a text file, with each line corresponding to the input expression and its corresponding result. The output file will accurately display the evaluated numerical values for each expression. For example, given the input expression "(3 + 5) \* 2," our approach evaluates it as "16" and records this result in the output file.

**Logical Steps:**

1. Retrieve the arithmetic expression from the input text file.
2. Analyse the expression to identify its key components, which include numbers, operators, and parentheses. This step breaks down the expression into manageable parts for accurate evaluation.
3. Follow the order of operations strictly. This means we prioritize operations within parentheses, ensuring they are calculated first, followed by exponentiation, then multiplication and division (from left to right), and finally, addition and subtraction (from left to right). Adhering to this order maintains the correct sequence of calculations.
4. Methodically evaluate the expression step by step, focusing on one operation at a time. By taking this meticulous approach, we ensure that each mathematical operation is executed accurately and in the proper sequence, leaving no room for computational errors.
5. Repeat the above steps for each expression in the input file, ensuring a consistent and precise evaluation process for all expressions.
6. Document both the input expression and its corresponding result in the output text file, preserving the original order, providing a clear and systematic record of the calculations performed.

**Program Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <ctype.h>

// Function to perform basic arithmetic operations

double operate(double operand1, char operator, double operand2)

{

    switch (operator)

    {

    case '+':

        return operand1 + operand2;

    case '-':

        return operand1 - operand2;

    case '\*':

        return operand1 \* operand2;

    case '/':

        if (operand2 != 0)

        {

            return operand1 / operand2;

        }

        else

        {

            printf("Error: Division by zero\n");

            exit(EXIT\_FAILURE);

        }

    case '^':

        return pow(operand1, operand2);

    default:

        printf("Error: Invalid operator\n");

        exit(EXIT\_FAILURE);

    }

}

// Function to check if a character is an operator

int isOperator(char ch)

{

    return ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '^';

}

double evaluateExpression(const char \*expression)

{

    int len = strlen(expression);

    double stackOperand[len];

    char stackOperator[len];

    int stackOperandIndex = -1;  // Stack index for operand initialization

    int stackOperatorIndex = -1; // Stack index for operator initialization

    int parenthesisCount = 0;

    for (int i = 0; i < len; ++i)

    {

        // If the current character is a digit or decimal point

        if (isdigit(expression[i]) || (expression[i] == '.' && isdigit(expression[i + 1])))

        {

            if (stackOperator[stackOperatorIndex] == '\*' || stackOperator[stackOperatorIndex] == '/' || stackOperator[stackOperatorIndex] == '^')

            {

                stackOperand[++stackOperandIndex] = strtod(expression + i, NULL);

                while (isdigit(expression[i]) || expression[i] == '.')

                {

                    ++i;

                }

                --i;

                // Perform the operation and push the result back to the stackOperand

                double operand2 = stackOperand[stackOperandIndex--];

                double operand1 = stackOperand[stackOperandIndex--];

                char operator= stackOperator[stackOperatorIndex--];

                stackOperand[++stackOperandIndex] = operate(operand1, operator, operand2);

            }

            else

            {

                stackOperand[++stackOperandIndex] = strtod(expression + i, NULL);

                while (isdigit(expression[i]) || expression[i] == '.')

                {

                    ++i;

                }

                --i;

            }

        }

        else if (isOperator(expression[i]))

        {

            // If the current character is an operator

            if (stackOperandIndex < 0)

            {

                printf("Error: Invalid expression - 1\n");

                exit(EXIT\_FAILURE);

            }

            else

            {

                stackOperator[++stackOperatorIndex] = expression[i];

            }

        }

        else if (expression[i] == '(')

        {

            stackOperator[++stackOperatorIndex] = expression[i];

            // If the current character is an opening parenthesis

            parenthesisCount++;

        }

        else if (expression[i] == ')')

        {

            // If the current character is a closing parenthesis

            if (parenthesisCount == 0)

            {

                printf("Error: Mismatched parenthesis\n");

                exit(EXIT\_FAILURE);

            }

            else if (stackOperator[stackOperatorIndex] == '(' && stackOperandIndex == 0 && isdigit(stackOperand[stackOperandIndex]))

            {

                stackOperatorIndex--;

                parenthesisCount--;

            }

            else

            {

            repeat:

                if (stackOperator[stackOperatorIndex] == '(')

                {

                    // Discard the opening parenthesis from the stackOperator

                    stackOperatorIndex--;

                    parenthesisCount--;

                    if ((stackOperator[stackOperatorIndex] == '\*' || stackOperator[stackOperatorIndex] == '/' || stackOperator[stackOperatorIndex] == '^') && expression[0] != '(')

                    {

                        // Perform the operation and push the result back to the stackOperand

                        double operand2 = stackOperand[stackOperandIndex--];

                        double operand1 = stackOperand[stackOperandIndex--];

                        char operator= stackOperator[stackOperatorIndex--];

                        stackOperand[++stackOperandIndex] = operate(operand1, operator, operand2);

                    }

                }

                else

                {

                    // Perform the operation inside the parenthesis

                    double operand2 = stackOperand[stackOperandIndex--];

                    char operator= stackOperator[stackOperatorIndex--];

                    double operand1 = stackOperand[stackOperandIndex--];

                    stackOperand[++stackOperandIndex] = operate(operand1, operator, operand2);

                    goto repeat;

                }

            }

        }

    }

    // Perform any remaining operations

    while (stackOperatorIndex != -1 && stackOperandIndex >= 1 && parenthesisCount == 0)

    {

        double operand2 = stackOperand[stackOperandIndex--];

        char operator=(char) stackOperator[stackOperatorIndex--];

        double operand1 = stackOperand[stackOperandIndex--];

        stackOperand[++stackOperandIndex] = operate(operand1, operator, operand2);

    }

    if (stackOperandIndex != 0)

    {

        printf("Error: Invalid expression - 2\n");

        exit(EXIT\_FAILURE);

    }

    return stackOperand[0];

}

int main()

{

    FILE \*inputFile, \*outputFile;

    char expression[100];

    // Open input and output files

    inputFile = fopen("input.txt", "r");

    if (inputFile == NULL)

    {

        perror("Error opening input file");

        exit(EXIT\_FAILURE);

    }

    outputFile = fopen("output.txt", "w");

    if (outputFile == NULL)

    {

        perror("Error opening output file");

        exit(EXIT\_FAILURE);

    }

    printf("\*\*\* Simplifier \*\*\*\n\n");

    fprintf(outputFile, "Simplified Expression by Simplifier: \n\n");

    // Process each expression in the input file

    while (fscanf(inputFile, "%99[^\n]\n", expression) == 1)

    {

        // Evaluate the expression

        double result = evaluateExpression(expression);

        // Display the expression and result and also print it to the output file

        printf("%s = %.2f\n", expression, result);

        fprintf(outputFile, "%s = %.2f\n", expression, result);

    }

    printf("\nExpressions has been saved to output file...\n\n");

    // Close files

    fclose(inputFile);

    fclose(outputFile);

    return 0;

}

**Input Format:**

2+5\*4-1

(25+3)\*(4-1)^2/2

(1.32+2^3)/(5-2\*2)

10\*(2+3)/(4-1)+6^2

(8/2)+(4\*3)-(7^2/(2+1))

80+((((10+(9+3)/2)\*3-1)+21\*2)/4)+20

**Output Format: On Screen as well as on Output Text File**

Simplified Expression by Simplifier:

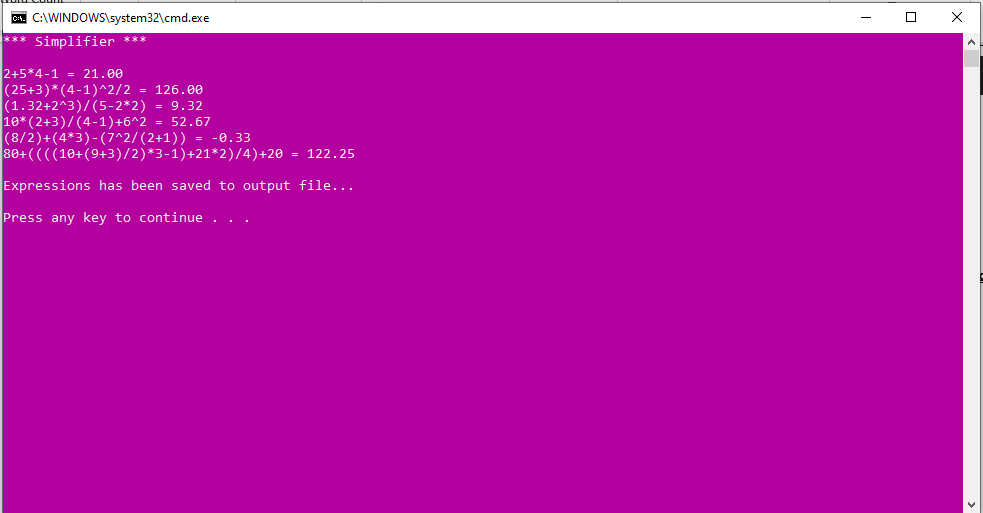
2+5\*4-1 = 21.00

(25+3)\*(4-1)^2/2 = 126.00

(1.32+2^3)/(5-2\*2) = 9.32

10\*(2+3)/(4-1)+6^2 = 52.67

(8/2)+(4\*3)-(7^2/(2+1)) = -0.33

****80+((((10+(9+3)/2)\*3-1)+21\*2)/4)+20 = 122.25

***1e. Find the exact execution time of a program but not by timestamp only.***

Ans: Preamble for Measuring the Exact Execution Time of a Program.

**Problem Statement:** The task at hand involves the measuring of the execution time of a program, it is crucial for optimizing software performance and ensuring efficient resource utilization. However, relying solely on timestamps may not provide the exact execution time due to system fluctuations and variations. This problem statement addresses the need to find the exact execution time of a program by employing a method that goes beyond timestamp recording.

**Preface:** To approach this problem and to accurately determine the execution time of a program, we will not solely rely on timestamp-based measurements. Instead, we will employ a method that includes additional techniques to measure execution time more precisely. We will assume that the program in question takes input from a text file and produces output in another text file. Our approach will involve careful analysis of the program's execution to capture the start and end times, as well as any other relevant data.

**Input Text File Format:** The input is provided in a text file; The input text file should contain the necessary data that the program requires for execution. The specific format will depend on the program under examination.

**Approach to Measure Execution Time:** Our approach to finding the exact execution time of a given program entails going beyond timestamp-based measurements. We prioritize the collection of relevant data during program execution to ensure precise and accurate measurement. By avoiding sole reliance on timestamps and incorporating performance metrics and repeated runs, we aim to obtain a more comprehensive and dependable assessment of the program's execution time, allowing for improved software optimization and resource management.

**Format of the Output:** The output will be presented in a text file that should contain the captured execution time along with any other relevant information. The exact format of this file will be determined during the implementation of the solution.

**Logical Steps:**

1. Read the input text file and load the necessary data into memory for program execution.
2. Record the start time before the program execution begins. This can be achieved using a timestamp-based method.
3. Execute the program, allowing it to perform its tasks as it normally would.
4. Record the end time after the program execution completes. Again, use a timestamp-based method for this purpose.
5. Calculate the time elapsed between the start time and end time to obtain an initial estimate of the execution time. This estimate may still be subject to inaccuracies due to system fluctuations.
6. Employ an iterative or repetition mechanism to execute the program multiple times, preferably under varying system conditions. This will aid in the recognition and alleviation of disparities in execution time.
7. Gather additional data during each program run, such as CPU usage, memory consumption, and any other relevant performance metrics. These metrics will contribute to a more accurate measurement of execution time.
8. Analyse the collected data to identify and mitigate any outliers or anomalies in the execution time measurements. Statistical techniques may be employed for this purpose.
9. Calculate the final execution time by taking into account the average or median of the recorded execution times, factoring in any relevant performance metrics, and adjusting for potential variations.
10. Write the final execution time and relevant performance metrics to the output text file for further analysis or reporting.
11. The resulting execution time measurement, combined with additional data, will provide a more accurate representation of the program's performance than timestamp-based measurements alone.

**Program Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <Windows.h>

#include <psapi.h>

#define MAX\_PATH\_LENGTH 100

#define NUM\_RUNS 10

void measureExecutionTime(const char \*programPath)

{

    double totalExecutionTime = 0;

    double totalCpuTime = 0;

    double totalMemoryUsage = 0;

    char compileCommand[MAX\_PATH\_LENGTH + 20]; // +20 for additional options like -o, -std, etc.

    sprintf(compileCommand, "gcc -std=gnu99 -o temp %s", programPath);

    system(compileCommand);

    for (int i = 0; i < NUM\_RUNS; ++i)

    {

        // Record the starting time

        clock\_t start\_time = clock();

        // Run the compiled program

        int result = system("temp.exe");

        // Record the ending time

        clock\_t end\_time = clock();

        // Check if the specified program ran successfully

        if (result == 0)

        {

            // Calculate the execution time in seconds

            double execution\_time = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

            totalExecutionTime += execution\_time;

            // Measure CPU time

            FILETIME createTime, exitTime, kernelTime, userTime;

            GetProcessTimes(GetCurrentProcess(), &createTime, &exitTime, &kernelTime, &userTime);

            double cpu\_time = (kernelTime.dwHighDateTime + userTime.dwHighDateTime) \* 1e-7 +

                              (kernelTime.dwLowDateTime + userTime.dwLowDateTime) \* 1e-7;

            totalCpuTime += cpu\_time;

            // Measure memory usage

            PROCESS\_MEMORY\_COUNTERS\_EX pmc;

            if (GetProcessMemoryInfo(GetCurrentProcess(), (PROCESS\_MEMORY\_COUNTERS \*)&pmc, sizeof(pmc)))

            {

                totalMemoryUsage += (double)pmc.PrivateUsage / (1024.0 \* 1024.0); // Convert to MB

            }

            // Print the measurements for each run

            printf("Run %d - Execution Time: %f seconds, CPU Time: %f seconds, Memory Usage: %f MB\n",

                   i + 1, execution\_time, cpu\_time, totalMemoryUsage);

        }

        else

        {

            printf("Error running temp.exe\n");

            return; // Exit if there is an error in any run

        }

    }

    // Calculate and print the average measurements

    double averageExecutionTime = totalExecutionTime / NUM\_RUNS;

    double averageCpuTime = totalCpuTime / NUM\_RUNS;

    double averageMemoryUsage = totalMemoryUsage / NUM\_RUNS;

    printf("For File-%s, Average - Execution Time: %f seconds, CPU Time: %f seconds, Memory Usage: %f MB\n", programPath, averageExecutionTime, averageCpuTime, averageMemoryUsage);

}

int main()

{

    // Get the program path from the user

    char programPath[MAX\_PATH\_LENGTH];

    printf("Enter the program path to measure execution time: ");

    scanf("%s", programPath);

    // Measure the execution time, CPU time, and memory usage of the specified program

    measureExecutionTime(programPath);

    return 0;

}

**Input Format: As C Program File to find execution time**

**Sample Program C File as Input**

// SampleProgram.c

#include <stdio.h>

int main()

{

    printf("Hello, this is a sample program!");

    long long x = 0;

    // Some time-consuming task

    for (int i = 0; i < 10; ++i)

    {

        x++;

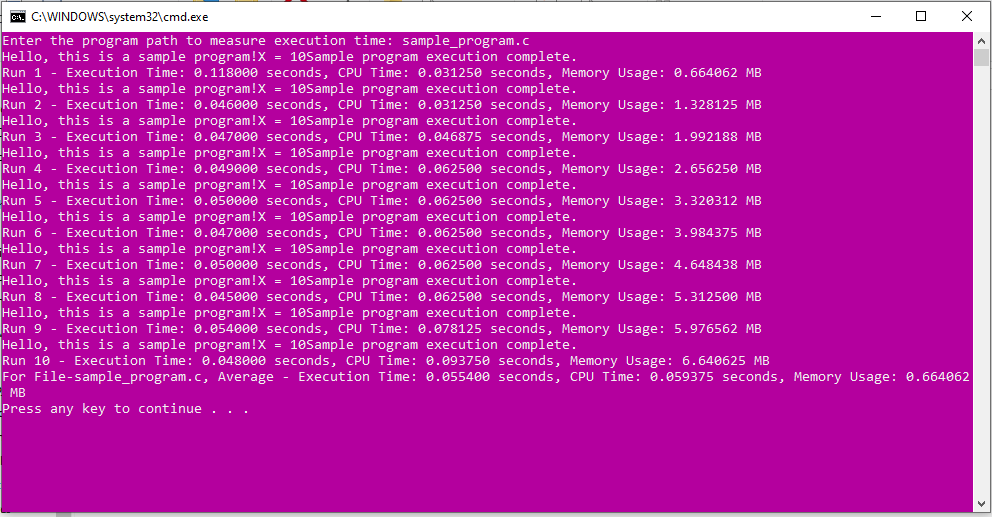
    }

    printf("X = %lld", x);

    printf("Sample program execution complete.\n");

    return 0;

}

**Output Format:**

***1f. For two large square matrix (not to fit individually in primary memory), compute the resultant multiplication matrix.***

Ans: Preamble for Multiplying Large Square Matrices without using computer resources.

**Problem Statement:** The task at hand involves multiplying large square matrices, which is essential but computationally intensive. These matrices find applications in various fields, but their size often exceeds primary memory capacity. We aim to develop an efficient algorithm for matrix multiplication without relying on specialized hardware or distributed computing, increasing its accessibility across applications.

**Preface:** To tackle this problem, we will consider a general approach that involves reading the matrices from input text files, processing them, and generating the resulting multiplication matrix as the output. It is essential to understand the format of the input text files, the fundamental steps of the algorithm, and the structure of the output.

**Input Text File Format:** The input text files will contain the two large square matrices. Each matrix will be stored as rows of space-separated numerical values. The dimensions of the matrices (e.g., N x N) will be specified within the files.

**Approach to Multiply 2 Large Matrices without using resources:** Our approach will be that we will employ a matrix multiplication algorithm that does not require loading the entire matrices into memory simultaneously. This algorithm is based on the divide-and-conquer approach and will operate on smaller submatrices to calculate the final result.

**Format of the Output:** The output will be presented in a text file containing the resultant multiplication matrix. It will follow the same format as the input matrices, with rows of space-separated numerical values. Additionally, the dimensions of the resultant matrix will be specified in the output file.

**Logical Steps:**

1. Read the input text files containing the two large square matrices, M1 and M2, and extract their dimensions (N x N).
2. Initialize an empty matrix R to store the resultant multiplication matrix. Set its dimensions to N x N.
3. Divide the matrices M1 and M2 into smaller submatrices (A, B, C, and D) to create a recursive framework for matrix multiplication.
4. For each submatrix A, B, C, and D:
5. If the submatrices are of a small enough size to fit in memory, calculate their multiplication using the traditional matrix multiplication algorithm.
6. If the submatrices are still large, recursively divide them into smaller submatrices and perform matrix multiplications until they reach a size that can be handled by traditional multiplication.
7. Combine the results from step 4 to populate the resultant matrix R, following the divide-and-conquer strategy.
8. Write the elements of matrix R into an output text file in the specified format, along with the dimensions of the resultant matrix.
9. The algorithm terminates, and you obtain the multiplication of the two large square matrices stored in the output file.

**Program Code:**

#include <stdio.h>

#include <stdlib.h>

// Function to read a matrix from a file

void readMatrixFromFile(const char \*filename, int \*\*matrix, int N)

{

    FILE \*file = fopen(filename, "r");

    if (file == NULL)

    {

        printf("Error opening file %s.\n", filename);

        exit(EXIT\_FAILURE);

    }

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            fscanf(file, "%d", &matrix[i][j]);

        }

    }

    fclose(file);

}

// Function to write a matrix to a file

void writeMatrixToFile(const char \*filename, int \*\*matrix, int N)

{

    FILE \*file = fopen(filename, "w");

    if (file == NULL)

    {

        printf("Error opening file %s for writing.\n", filename);

        exit(EXIT\_FAILURE);

    }

    fprintf(file, "%d\n", N);

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            fprintf(file, "%d ", matrix[i][j]);

        }

        fprintf(file, "\n");

    }

    fclose(file);

}

// Function to allocate memory for a matrix

int \*\*allocateMatrix(int N)

{

    int \*\*matrix = (int \*\*)malloc(N \* sizeof(int \*));

    for (int i = 0; i < N; i++)

    {

        matrix[i] = (int \*)malloc(N \* sizeof(int));

    }

    return matrix;

}

// Function to deallocate memory for a matrix

void deallocateMatrix(int \*\*matrix, int N)

{

    for (int i = 0; i < N; i++)

    {

        free(matrix[i]);

    }

    free(matrix);

}

// Function to perform matrix multiplication using a simpler approach

void multiplyMatrix(int \*\*M1, int \*\*M2, int \*\*R, int N)

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            for (int k = 0; k < N; k++)

            {

                R[i][j] += M1[i][k] \* M2[k][j];

            }

        }

    }

}

int main()

{

    printf("\*\*\* Large Matrix Multiplication \*\*\*\n\n");

    // Read input matrices from files

    const char \*filename1 = "matrix1.txt";

    const char \*filename2 = "matrix2.txt";

    int N; // Matrix dimension

    FILE \*file = fopen(filename1, "r");

    if (file == NULL)

    {

        printf("Error opening file %s.\n", filename1);

        exit(EXIT\_FAILURE);

    }

    fscanf(file, "%d", &N);

    fclose(file);

    // Allocate memory for matrices

    int \*\*M1 = allocateMatrix(N);

    int \*\*M2 = allocateMatrix(N);

    int \*\*R = allocateMatrix(N);

    // Read matrices from files

    readMatrixFromFile(filename1, M1, N);

    printf("Matrix 1.txt was read.\n");

    readMatrixFromFile(filename2, M2, N);

    printf("Matrix 2.txt was read.\n");

    // Initialize result matrix R to zeros

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            R[i][j] = 0;

        }

    }

    printf("Multiplying Matrix 1 and Matrix 2 and Calculating Resultant Matrix : \n");

    // Perform matrix multiplication

    multiplyMatrix(M1, M2, R, N);

    // Write the result matrix to a file

    const char \*resultFilename = "result\_matrix.txt";

    writeMatrixToFile(resultFilename, R, N);

    // Deallocate memory

    deallocateMatrix(M1, N);

    deallocateMatrix(M2, N);

    deallocateMatrix(R, N);

    printf("Resultant Matrix has been saved to result\_matrix.txt\n\n");

    return 0;

}

**Input Format: Both Matrices are Given as Input in Separate Text File.**

**Matrix 1: Matrix 2:**

4 4

1 2 3 4 17 18 19 20

5 6 7 8 21 22 23 24

9 10 11 12 25 26 27 28

13 14 15 16 29 30 31 32

**Output Format: The Output was saved in an Output as Result Matrix Text File.**

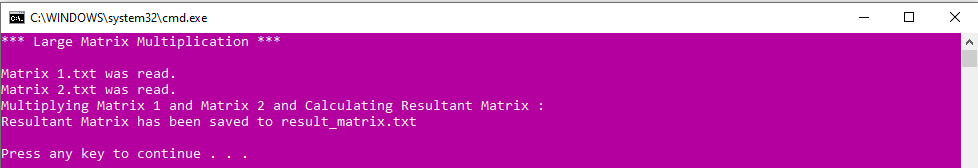
**Result Matrix:**

4

168 226 236 246

456 526 548 570

760 894 932 970

1064 1262 1316 1370

***1*g*. From a given set of points with (x,y) coordinates in either clockwise or anti-clockwise order. Find the maximum polygon (convex polygon)/ Intersection/ Union/ Inside or Outside?***

Ans: Preamble for analysing the relationships between the given polygons.

**Problem Statement:** The task at hand involves to determine the relationships between two sets of points represented as convex or non-convex polygons. Given a set of points with (x, y) coordinates, arranged either in a clockwise or anti-clockwise order, this problem seeks to find the maximum polygon, which could be a convex polygon, the intersection of two polygons, the union of two polygons, or whether one polygon is entirely inside or outside the other.

**Preface:** To tackle this problem, we need to establish a systematic approach for analysing the relationships between the given polygons. The input is expected to be provided in the form of text files, with each line representing a point and the order (clockwise or anti-clockwise) of these points determining the shape of the polygon. The output format will depend on the specific task being performed: it might involve providing coordinates of the resulting polygon or indicating whether one polygon is inside or outside the other.

**Input Text File Format:** The input is provided in a text file, The input text file should consist of a single character representing the type of operation: "M" for Maximum Polygon, "I" for Intersection, "U" for Union, or "IO" for Inside/Outside Analysis, followed by a series of lines, each containing (x, y) coordinates of points.

**Approach to Efficient Analysis of Polygons:** Our approach is to create an algorithm for efficient polygon analysis, including finding maximum polygons, intersections, unions, and inside/outside comparisons. This has broad applications in geographic information systems, image processing, and computational geometry. The approach includes input handling, polygon representation, specific algorithmic tasks, and output generation. The algorithm's optimized implementation supports practical deployment.

**Format of the Output:** The output will be presented in a text file containing a concise representation of the results, including the (x, y) coordinates of points forming the maximum convex polygon, intersection, or union polygon in order, or a single character "I" for Inside or "O" for Outside in the case of Inside/Outside Analysis.

**Logical Steps:**

1. Read the coordinates of points from input files and determine the order of points.
2. Create polygons based on the order of points.
3. Analyse the points to find the convex hull of each polygon, identifying the maximum polygon.
4. Compare points of two polygons to find intersection points and construct a new polygon.
5. Identify common boundary points between two polygons to create a new polygon that combines both.
6. Determine whether one polygon is fully contained within the other or if any points lie outside.
7. Generate output specific to the task (maximum polygon, intersection, union, or inside/outside analysis).
8. Save the results in an output text file with the specified format.
9. Terminate the program.

**Program Code:**

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>  // For MKDIR

#include <sys/types.h> // For MKDIR

#define MAX\_FILENAME\_LENGTH 100

struct Point

{

    int x, y;

};

struct Polygon

{

    struct Point \*points;

    int numPoints;

};

struct Polygon globalConvexHull; // Global variable with a different name

int orientation(struct Point p, struct Point q, struct Point r)

{

    int val = (q.y - p.y) \* (r.x - q.x) - (q.x - p.x) \* (r.y - q.y);

    if (val == 0)

        return 0;

    return (val > 0) ? 1 : 2;

}

int comparePoints(const void \*a, const void \*b)

{

    struct Point p1 = \*(struct Point \*)a;

    struct Point p2 = \*(struct Point \*)b;

    int o = orientation(p1, globalConvexHull.points[0], p2);

    if (o == 0)

        return ((p2.x - globalConvexHull.points[0].x) \* (p2.x - globalConvexHull.points[0].x) + (p2.y - globalConvexHull.points[0].y) \* (p2.y - globalConvexHull.points[0].y) >= (p1.x - globalConvexHull.points[0].x) \* (p1.x - globalConvexHull.points[0].x) + (p1.y - globalConvexHull.points[0].y) \* (p1.y - globalConvexHull.points[0].y)) ? -1 : 1;

    return (o == 2) ? -1 : 1;

}

void convexHull(struct Point points[], int n, struct Polygon \*resultConvexHull)

{

    resultConvexHull->points = (struct Point \*)malloc(n \* sizeof(struct Point));

    resultConvexHull->numPoints = 0;

    resultConvexHull->points[resultConvexHull->numPoints++] = points[0];

    resultConvexHull->points[resultConvexHull->numPoints++] = points[1];

    for (int i = 2; i < n; i++)

    {

        while (resultConvexHull->numPoints > 1 && orientation(resultConvexHull->points[resultConvexHull->numPoints - 2],

                                                              resultConvexHull->points[resultConvexHull->numPoints - 1], points[i]) != 2)

        {

            resultConvexHull->numPoints--;

        }

        resultConvexHull->points[resultConvexHull->numPoints++] = points[i];

    }

}

// Rest of the code remains unchanged

void findIntersection(struct Polygon \*poly1, struct Polygon \*poly2, struct Polygon \*intersection)

{

    // Assuming simple intersection by finding common points

    int i, j;

    intersection->points = (struct Point \*)malloc(poly1->numPoints \* sizeof(struct Point));

    intersection->numPoints = 0;

    for (i = 0; i < poly1->numPoints; i++)

    {

        for (j = 0; j < poly2->numPoints; j++)

        {

            if (poly1->points[i].x == poly2->points[j].x && poly1->points[i].y == poly2->points[j].y)

            {

                intersection->points[intersection->numPoints++] = poly1->points[i];

                break; // Break inner loop to avoid duplicate points in intersection

            }

        }

    }

}

void findUnion(struct Polygon \*poly1, struct Polygon \*poly2, struct Polygon \*unionPoly)

{

    // Simple union by combining points from both polygons

    unionPoly->points = (struct Point \*)malloc((poly1->numPoints + poly2->numPoints) \* sizeof(struct Point));

    unionPoly->numPoints = 0;

    for (int i = 0; i < poly1->numPoints; i++)

    {

        unionPoly->points[unionPoly->numPoints++] = poly1->points[i];

    }

    for (int i = 0; i < poly2->numPoints; i++)

    {

        unionPoly->points[unionPoly->numPoints++] = poly2->points[i];

    }

}

int isPolygonInside(struct Polygon \*innerPoly, struct Polygon \*outerPoly)

{

    // Assuming simple check by comparing points

    for (int i = 0; i < innerPoly->numPoints; i++)

    {

        int isInside = 0;

        for (int j = 0; j < outerPoly->numPoints; j++)

        {

            if (innerPoly->points[i].x == outerPoly->points[j].x && innerPoly->points[i].y == outerPoly->points[j].y)

            {

                isInside = 1;

                break;

            }

        }

        if (!isInside)

        {

            // If any point of innerPoly is not found in outerPoly, then it's not fully inside

            return 0;

        }

    }

    // All points of innerPoly are found in outerPoly, so it's fully inside

    return 1;

}

void readPointsFromFile(char \*filename, struct Point points[], int \*n)

{

    FILE \*file = fopen(filename, "r");

    if (file == NULL)

    {

        perror("Error opening file");

        exit(EXIT\_FAILURE);

    }

    fscanf(file, "%d", n);

    for (int i = 0; i < \*n; i++)

    {

        fscanf(file, "%d %d", &points[i].x, &points[i].y);

    }

    fclose(file);

}

void writePointsToFile(char \*filename, struct Point points[], int n, char \*word)

{

    FILE \*file = fopen(filename, "w");

    if (file == NULL)

    {

        perror("Error opening file");

        exit(EXIT\_FAILURE);

    }

    if (n > 0)

    {

        for (int i = 0; i < n; i++)

        {

            fprintf(file, "%d %d\n", points[i].x, points[i].y);

        }

    }

    else

    {

        fprintf(file, "No %s points found.", word);

    }

    fclose(file);

}

int main()

{

    printf("\*\*\* Polygon Analyzer \*\*\*\n\n\n");

    struct Point points1[100], points2[100];

    int n1, n2;

    const char \*folder\_name = "PolygonsDetails";

    char outputFilename[MAX\_FILENAME\_LENGTH];

    const char \*operations[] = {"union", "intersection", "output1", "output2"};

    mkdir(folder\_name);

    // Step 1: Read coordinates of points from input files

    printf("Read Points of Polygon1\n");

    readPointsFromFile("input1.txt", points1, &n1);

    printf("Read Points of Polygon2\n");

    readPointsFromFile("input2.txt", points2, &n2);

    // Step 2: Create polygons based on the order of points

    struct Polygon polygon1, polygon2;

    convexHull(points1, n1, &polygon1);

    convexHull(points2, n2, &polygon2);

    // Step 3: Analyze points to find the convex hull of each polygon, identifying the maximum polygon

    // (Already done in the convexHull function)

    // Step 4: Compare points of two polygons to find intersection points and construct a new polygon

    struct Polygon intersectionPoly;

    findIntersection(&polygon1, &polygon2, &intersectionPoly);

    // Step 5: Identify common boundary points between two polygons to create a new polygon that combines both

    struct Polygon unionPoly;

    findUnion(&polygon1, &polygon2, &unionPoly);

    // Step 6: Determine whether one polygon is fully contained within the other or if any points lie outside

    int isPoly1Inside2 = isPolygonInside(&polygon1, &polygon2);

    int isPoly2Inside1 = isPolygonInside(&polygon2, &polygon1);

    // Step 7: Generate output specific to the task (maximum polygon, intersection, union, or inside/outside analysis)

    // Writing the results to output files

    printf("Intersecting Points between Polygon 2 and Polygon 1 are saved in intersection.txt\n");

    snprintf(outputFilename, sizeof(outputFilename), "%s/%s.txt", folder\_name, operations[1]);

    writePointsToFile(outputFilename, intersectionPoly.points, intersectionPoly.numPoints, "intersection");

    printf("Union Points between Polygon 2 and Polygon 1 are saved in union.txt\n");

    snprintf(outputFilename, sizeof(outputFilename), "%s/%s.txt", folder\_name, operations[0]);

    writePointsToFile(outputFilename, unionPoly.points, unionPoly.numPoints, "union");

    // Additional analysis outputs based on the inside/outside checks

    if (isPoly1Inside2)

    {

        printf("Polygon 1 is fully inside Polygon 2.\n");

    }

    else

    {

        printf("Polygon 1 is not fully inside Polygon 2.\n");

    }

    if (isPoly2Inside1)

    {

        printf("Polygon 2 is fully inside Polygon 1.\n");

    }

    else

    {

        printf("Polygon 2 is not fully inside Polygon 1.\n");

    }

    // Step 8: Save the results in an output text file with the specified format

    printf("Convex Hull Points of Polygon 1 is saved in output1.txt\n");

    snprintf(outputFilename, sizeof(outputFilename), "%s/%s.txt", folder\_name, operations[2]);

    writePointsToFile(outputFilename, polygon1.points, polygon1.numPoints, "convex hull");

    printf("Convex Hull Points of Polygon 2 is saved in output2.txt\n");

    snprintf(outputFilename, sizeof(outputFilename), "%s/%s.txt", folder\_name, operations[3]);

    writePointsToFile(outputFilename, polygon2.points, polygon2.numPoints, "convex hull");

    printf("Details saved in %s folder.\n\n", outputFilename);

    // Step 9: Terminate the program

    return 0;

}

**Input Format: Set of co-ordinates of Both Polygons are given in Separate Input Text File.**

**Input1: Polygon1 Input2: Polygon2**

6 5

1 1 2 2

2 5 4 4

3 1 6 6

4 9 8 8

5 2 10 10

6 7

**Output Format: Output will be saved in the Output Text File with their respective names in Polygon Details Folder.**

**Intersection.txt**

No intersection points found.

**Union.txt**

1 1

3 1

5 2

6 7

2 2

10 10

**Output1.txt**

1 1

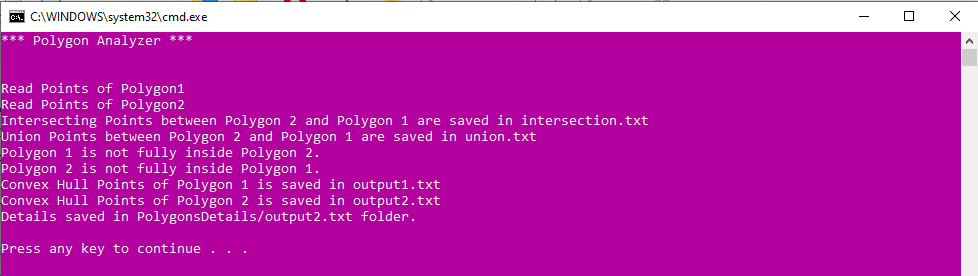
3 1

5 2

6 7

**Output2.txt**

2 2

****10 10

***1h. Find the factorial of any integer number. The numbers could be at most ten-digit long.***

Ans: Preamble for calculating the factorial of any integer number.

**Problem Statement:** The task at hand involves computing the factorial of any given integer number, with the constraint that the numbers provided will be at most ten digits long. Factorial, denoted by the symbol '!', is the product of all positive integers from 1 to the given integer. The importance of this problem lies in its relevance in various mathematical and computational domains, including combinatorics, probability, and algebra.

**Preface:** To tackle this problem, we compute factorials for integers up to ten digits long. Factorials are fundamental in mathematics and science, making this versatile algorithm valuable in diverse applications. Its simplicity and universal nature enable easy integration into different environments, with user-friendly input and output formats.

**Input Text File Format:** The input is provided in a text file, The input text file should contain a single line containing a positive integer, representing the number for which we need to compute the factorial. The integer will be at most ten digits long.

**Approach to Calculate the Factorial of an Integer:** Our approach is to calculate the factorial of an integer; we will use a straightforward iterative algorithm that multiplies each integer from 1 to the given number. We will start with an accumulator initialized to 1 and iterate from 1 to the input number, multiplying the current value of the accumulator by the current iteration variable. The result in the accumulator will be the final factorial of the input number.

**Format of the Output:** The output will be presented in a text file containing a single line containing the computed factorial, which is also a positive integer.

**Logical Steps:**

1. Read the input from the text file, which contains a single integer, and store it in a variable (let's call it 'n').
2. Initialize an accumulator variable (let's call it 'factorial') to 1. This variable will store the final factorial value.
3. Iterate from 1 to 'n', performing the following operation in each iteration:

* Multiply 'factorial' by the current value of the iteration variable.

1. After the iteration completes, 'factorial' will contain the factorial of 'n'.
2. Write the value of 'factorial' to the output text file.
3. Close the input and output files, completing the factorial calculation process.

**Program Code:**

#include <stdio.h>

#include <gmp.h>

int main()

{

    // Step 1: Read input from the text file

    FILE \*inputFile = fopen("input.txt", "r");

    if (inputFile == NULL)

    {

        perror("Error opening input file");

        return 1;

    }

    mpz\_t n;

    mpz\_init(n);

    fscanf(inputFile, "%Zd", n);

    // Step 2: Initialize accumulator variable

    mpz\_t factorial;

    mpz\_init\_set\_ui(factorial, 1);

    // Step 3: Calculate factorial

    for (mpz\_t i; mpz\_cmp\_ui(i, 1) <= 0; mpz\_sub\_ui(i, i, 1))

    {

        mpz\_mul(factorial, factorial, i);

    }

    // Step 4: Write the result to the output text file

    FILE \*outputFile = fopen("output.txt", "w");

    if (outputFile == NULL)

    {

        perror("Error opening output file");

        fclose(inputFile);

        return 1;

    }

    gmp\_fprintf(outputFile, "%Zd", factorial);

    // Step 5: Close files

    fclose(inputFile);

    fclose(outputFile);

    // Free allocated memory

    mpz\_clear(n);

    mpz\_clear(factorial);

    return 0;

}

**Input Format: As Input Text File**

**Output Fomat: As Output Text File**

***1i. Indentation of a C- program file.***

Ans: Preamble for Automating the Indentation of the Given C File.

**Problem Statement:** The task at hand is to develop a program that automates the process of indenting a C-program file. Proper code indentation is crucial for code readability and maintainability. This program will take an unindented C-program file as input and produce an indented version of the same code as output.

**Preface:** To tackle and achieve this task, we will assume that the input file contains valid C code and follow the commonly accepted conventions for C code indentation. The input text file will be assumed to contain unindented C code, and the program will not perform any syntax checking but focus solely on the indentation. The program will use if-else statements to analyse the code structure and add appropriate indentation for blocks of code.

**Input Text File Format:**

* The input will be a text file containing unindented C code.
* The file will have a '.c' extension.
* The code within the file must be well-formed C code.

**Approach to Automate the Indentation of a C File:**

Our approach involves systematically analysing unindented C code line by line, employing if-else statements to determine proper indentation. We adjust the indentation level based on opening and closing curly braces while maintaining existing indentation for lines starting with '#'. The result is neatly indented code that adheres to standard C formatting practices, enhancing code readability and maintainability.

**Format of the Output:**

* The program will create an output text file with the same '.c' extension.
* The output file will contain the C code with proper indentation.

**Logical Steps:**

1. Read the input file name as provided by the user.
2. Check if the file exists. If not, display an error message and terminate the program.
3. Attempt to open and read the input file. If unsuccessful, display an error message and terminate the program.
4. Create an output file with the same name as the input file, but with '\_indented' appended to it.
5. Initialize an indentation level variable to keep track of the current level of indentation (e.g., start with 0).
6. Initialize a Boolean variable to keep track of whether the current line is within a block (e.g., set it to false initially).
7. Read the input file line by line.
8. For each line, analyse its content using if-else statements to determine the appropriate indentation level.
9. If the line contains an opening curly brace '{', increase the indentation level.
10. If the line contains a closing curly brace '}', decrease the indentation level.
11. If the line is not a comment and does not contain '{' or '}', add the current level of indentation before the line.
12. If a line starts with '#', it may indicate a preprocessor directive; keep the indentation level the same.
13. Write the indented line to the output file.
14. Repeat steps 7-9 for all lines in the input file.
15. Close both the input and output files.
16. Display a success message to the user indicating that the indentation process is complete, and the indented code has been saved in the output file.
17. Terminate the program.

**Problem Statement:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#define MAX\_FILENAME\_LENGTH 100

int main()

{

    printf("\*\*\* Prettsy - A C File Indenter \*\*\* \n\n\n");

    int increaseIntent = 0;

    // 1. Read the input file name as provided by the user.

    char inputFileName[MAX\_FILENAME\_LENGTH];

    printf("Enter the File Name to prettify it = ");

    scanf("%s", inputFileName);

    size\_t length = strlen(inputFileName);

    if (((inputFileName[length - 1]) != 'c') && ((inputFileName[length - 1]) != 'C'))

    {

        printf("Only Files with Extension '.c' are Allowed!\n ");

        exit(1);

    }

    // Open the C file for reading

    FILE \*inputFile = fopen(inputFileName, "r");

    if (inputFile == NULL)

    {

        printf("Error: File '%s' not found.\n", inputFileName);

        return 1;

    }

    // Create the output file

    char outputFilename[MAX\_FILENAME\_LENGTH];

    snprintf(outputFilename, sizeof(outputFilename), "Indented\_%s", inputFileName);

    FILE \*outputFile = fopen(outputFilename, "w");

    if (outputFile == NULL)

    {

        printf("Error: Unable to create the output file.\n");

        fclose(inputFile);

        return 1;

    }

    // 4. Initialize an indentation level variable to keep track of the current level of indentation (e.g., start with 0).

    int indentationLevel = 0;

    // 6. Read the input file line by line.

    char line[1000];

    while (fgets(line, sizeof(line), inputFile) != NULL)

    {

        // Trim leading and trailing whitespace

        char \*trimmedLine = line;

        while (\*trimmedLine == ' ' || \*trimmedLine == '\t')

        {

            trimmedLine++;

        }

        size\_t len = strlen(trimmedLine);

        while (len > 0 && (trimmedLine[len - 1] == ' ' || trimmedLine[len - 1] == '\t' || trimmedLine[len - 1] == '\n'))

        {

            trimmedLine[--len] = '\0';

        }

        // 8. For each line, analyze its content using if-else statements to determine the appropriate indentation level.

        // 9. If the line contains an opening curly brace '{', increase the indentation level.

        if ((strstr(trimmedLine, "{") != NULL) && (strstr(trimmedLine, "(") != NULL) && (strstr(trimmedLine, ")") != NULL))

        {

            increaseIntent = 2;

        }

        else if (strstr(trimmedLine, "{") != NULL)

        {

            increaseIntent = 1;

        }

        // 10. If the line contains a closing curly brace '}', decrease the indentation level.

        else if (strstr(trimmedLine, "}") != NULL)

        {

            indentationLevel--;

        }

        // 11. Add the current level of indentation before the line.

        for (int i = 0; i < indentationLevel; i++)

        {

            fprintf(outputFile, "\t");

        }

        // 13. Write the indented line to the output file.

        if (increaseIntent == 2)

        {

            char \*newTrimmedLine = strtok(trimmedLine, "{");

            fprintf(outputFile, "%s\n", newTrimmedLine);

            for (int i = 0; i < indentationLevel; i++)

            {

                fprintf(outputFile, "\t");

            }

            fprintf(outputFile, "{\n");

        }

        else

        {

            fprintf(outputFile, "%s\n", trimmedLine);

        }

        if (increaseIntent == 1 || increaseIntent == 2)

        {

            indentationLevel++;

            increaseIntent = 0;

        }

    }

    // 15. Close both the input and output files.

    fclose(inputFile);

    fclose(outputFile);

    // 16. Display a success message to the user indicating that the indentation process is complete.

    printf("Code indentation is complete. Check %s for the indented code.\n", outputFilename);

    // 17. Terminate the program.

    return 0;

}

**Input Format: A C File which is without any indentation as Input File.**

**Sample Program:**

// SampleProgram.c

#include <stdio.h>

int main()

{

printf("Hello, this is a sample program!\n");

long long x = 0;

// Some time-consuming task

for (int i = 0; i < 10; ++i){

// Increasing x

x++;

int y=10;

while(y!=0){

printf("Make In India!!!");

y--;

}

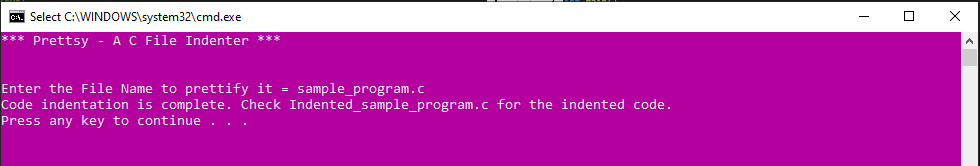
}

printf("X = %lld\n", x);

printf("Sample program execution complete.\n\n");

return 0;

}

**Output Format: Indented C file as Output file.**

**Indented Sample C File:**

// SampleProgram.c

#include <stdio.h>

int main()

{

    printf("Hello, this is a sample program!\n");

    long long x = 0;

    // Some time-consuming task

    for (int i = 0; i < 10; ++i)

    {

        // Increasing x

        x++;

        int y=10;

        while(y!=0)

        {

            printf("Make In India!!!");

            y--;

        }

    }

    printf("X = %lld\n", x);

    printf("Sample program execution complete.\n\n");

    return 0;

}

***1*j*. Integer Number to Word Conversion.***

Ans: Preamble for Conversion of an Integer Number to Word.

**Problem Statement:** The task at hand is to create a program that can convert an integer number into its corresponding word representation. This conversion from numeric form to words is essential for various applications such as generating written checks, invoices, or any scenario where the presentation of a numerical value in words is more reader-friendly. For instance, converting the number 12345 to "twelve thousand three hundred forty-five.

**Preface:** To tackle, achieve and accomplish this task, we will develop an algorithm that takes an integer as input and returns its word representation as output. The input will be provided through a text file, where each line contains a single integer value. The output will also be stored in a text file with each line corresponding to the word representation of the input number. Our approach will involve a series of If-Else statements to cover a wide range of integer values. The goal is to ensure that any positive integer within the range of the data type used for representing numbers in the program can be converted into words.

**Input Text File Format:** The input is provided through a text file, where each line contains a single integer value for word conversion.

**Approach to Convert Integer Number into Word:** Our approach involves reading integers from an input text file, process them, and convert each number to its word representation. By breaking numbers into groups of three digits and applying If-Else statements, we create a comprehensive word representation. The results are saved in an output text file, enhancing readability for applications like check writing and financial documentation.

**Format of the Output:** The output is stored in a text file, with each line corresponding to the word representation of the input number.

**Logical Steps:**

1. Read the input text file containing integer numbers.
2. For each line in the input file, extract the integer value as the current number to be converted.
3. Check if the current number is zero. If it is, add "Zero" to the output text file and proceed to the next number.
4. Divide the current number into groups of three digits from right to left. For example, if the number is 1234567, it will be divided into three groups: 1,234,567.
5. For each group of three digits, convert it into words by examining the hundreds, tens, and units’ positions. Use If-Else statements to handle different cases. For example, "123" becomes "One Hundred Twenty-Three."
6. Combine the word representations of groups to form the complete word representation of the current number. Ensure the appropriate use of words like "Thousand," "Million," etc., to indicate the scale of the number.
7. Write the word representation of the current number to the output text file.
8. Repeat steps 3 to 7 for all the numbers in the input text file.
9. Close both the input and output files.
10. The program has successfully converted integer numbers into their word representations, and the output text file now contains the desired results.

**Program Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

char \*units[] = {"", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine"};

char \*teens[] = {"", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"};

char \*tens[] = {"", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};

char \*scale[] = {"", "Thousand", "Million", "Billion", "Trillion"};

void convertNumberToWords(long long number, char \*result)

{

    // Initialize result with an empty string

    result[0] = '\0';

    // Handle the special case when the input number is 0

    if (number == 0)

    {

        sprintf(result, "Zero");

        return;

    }

    int scaleIndex = 0;

    int flag = 0;

    while (number > 0)

    {

        int temp = number % 1000;

        int temp2 = temp % 100;

        int temp3 = (temp - temp2) / 100;

        int temp4 = temp2 % 10;

        int temp5 = (temp2 - temp4) / 10;

        char tempResult[500] = ""; // Use a temporary buffer

        if (temp3 > 0)

        {

            strcat(tempResult, units[temp3]);

            strcat(tempResult, " Hundred");

            if (temp2 > 0)

            {

                strcat(tempResult, " and ");

            }

        }

        if (temp2 >= 11 && temp2 <= 19)

        {

            if (temp5 > 0 && result[0] == '\0')

            {

                strcat(tempResult, " and ");

            }

            strcat(tempResult, teens[temp2 - 10]);

        }

        else

        {

            if (temp5 > 0)

            {

                strcat(tempResult, tens[temp5]);

                if (temp4 > 0)

                {

                    strcat(tempResult, "-");

                }

            }

            if (temp4 > 0)

            {

                strcat(tempResult, units[temp4]);

            }

        }

        if (flag > 0)

        {

            strcat(tempResult, " ");

            strcat(tempResult, scale[flag]);

        }

        strcat(tempResult, " ");

        strcat(tempResult, result); // Append to the existing result

        strcpy(result, tempResult); // Copy the temporary buffer back to the result

        flag++;

        number = number / 1000;

    }

}

int main()

{

    printf("\*\*\* Integer to Word Conversion (Upto 10 Digits) \*\*\*\n\n");

    FILE \*inputFile = fopen("input.txt", "r");

    FILE \*outputFile = fopen("output.txt", "w");

    printf("Reading Integers from input.txt\n\n");

    if (inputFile == NULL || outputFile == NULL)

    {

        printf("Error opening files.\n");

        return 1;

    }

    char line[20];

    while (fgets(line, sizeof(line), inputFile) != NULL)

    {

        long long number = atoi(line);

        char \*result = (char \*)malloc(500 \* sizeof(char));

        if (result == NULL)

        {

            printf("Memory allocation error.\n");

            fclose(inputFile);

            fclose(outputFile);

            return 1;

        }

        convertNumberToWords(number, result);

        // Check if the result is empty

        if (result[0] == '\0')

        {

            printf("Conversion error for input: %s\n", line);

            free(result);

            continue;

        }

        fprintf(outputFile, "%s\n", result);

        free(result);

    }

    printf("Integers have been converted and saved to output.txt\n\n");

    fclose(inputFile);

    fclose(outputFile);

    return 0;

}

**Input Format: As Input Text File**

1239784

4567

89012

3456789

1325684736

12324454

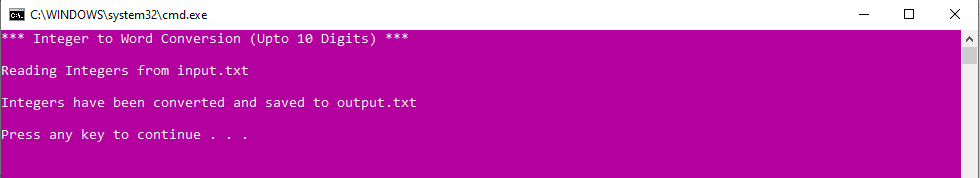
999

456

1234

2134786898

0

**Output Format: As Output Text File**

One Million Two Hundred and Thirty-Nine Thousand Seven Hundred and Eighty-Four

Four Thousand Five Hundred and Sixty-Seven

Eighty-Nine Thousand and Twelve

Three Million Four Hundred and Fifty-Six Thousand Seven Hundred and Eighty-Nine

One Billion Three Hundred and Twenty-Five Million Six Hundred and Eighty-Four Thousand Seven Hundred and Thirty-Six

Twelve Million Three Hundred and Twenty-Four Thousand Four Hundred and Fifty-Four

Nine Hundred and Ninety-Nine

Four Hundred and Fifty-Six

One Thousand Two Hundred and Thirty-Four

Two Billion One Hundred and Thirty-Four Million Seven Hundred and Eighty-Six Thousand Eight Hundred and Ninety-Eight

Zero

***1k. Write a TSR program to add a series of integer numbers.***

Ans: Preamble for TSR program to add a series of integer numbers.

**Problem Statement:** The task at hand is to write a TSR (Terminate and Stay Resident) program that can add a series of 'n' integer numbers. A TSR program is a type of software that remains active and resident in memory even after the main program has completed execution. In this case, the program should be able to accept a series of 'n' integer numbers, calculate their sum, and remain active for further use.

**Preface:** To tackle, achieve and accomplish this task, we will create a TSR program that can be initiated within a specific environment, such as a DOS shell. The program will be designed to accept a sequence of 'n' integer numbers from the user and continuously update the sum as new numbers are provided. The result can be displayed on demand, and the program will remain active in the memory.

**Input Text File Format:** The input to the program will be provided by the user will be prompted to enter a series of integer numbers, one at a time. The program will continue to accept input until a specific command is given to calculate and display the sum.

**Approach to Convert Integer Number into Word:** Our approach involves creating a TSR program that remains active in memory, continuously accepting user input for integer numbers, and displaying the sum on demand. The program efficiently manages user interaction, making it a versatile and always-ready tool for performing continuous additions in various contexts. There is no traditional file-based input or output in this context.

**Format of the Output:** The program will display the sum of the entered numbers on request. The result on the screen within the environment where the program is running.

**Logical Steps:**

1. Initialize the TSR program within the desired environment.
2. Create a variable 'sum' and set it initially to 0. This variable will be used to store the sum of the entered numbers.
3. Continuously monitor user input.
   1. If user inputs an integer, add that integer to the 'sum' and continue waiting for the next input.
   2. If user inputs a specific command (e.g., 'calculate' / 'show') proceed to step 4.
4. Upon receiving the command to calculate and display the sum:
   1. Calculate the sum of all the previously entered integers stored in the 'sum' variable.
   2. Display the calculated sum on the screen.
5. The program will remain active and resident in memory, allowing the user to input more integers and calculate the sum again by repeating steps 3 and 4.
6. Optionally, provide an exit command to terminate the TSR program and release memory resources when the user is done.

**Problem Statement:**

#include <dos.h>

#include <stdio.h>

#include <conio.h>

#define CTRL\_C 0x2E03 // Ctrl+C key combination

void interrupt (\*oldInt9)(); // Pointer to the original interrupt handler

int sum = 0;                 // Variable to store the sum

void interrupt newInt9()

{

    int key = \_bios\_keybrd(\_NKEYBRD\_READY);

    if (key != 0)

    {

        int ascii = \_bios\_keybrd(\_NKEYBRD\_READ);

        if (ascii >= '0' && ascii <= '9')

        {

            int number = ascii - '0';

            sum += number;

            printf("%d ", number);

        }

        else if (ascii == CTRL\_C)

        {

            printf("\nSum: %d\n", sum);

            printf("Press any key to continue or Ctrl+C to exit...\n");

            sum = 0;

        }

    }

    (\*oldInt9)();

}

void main()

{

    clrscr();

    printf("TSR Program: Enter integers, use Ctrl+C to calculate and display sum.\n");

    oldInt9 = getvect(0x09); // Get the current interrupt handler

    setvect(0x09, newInt9);  // Set the new interrupt handler

    keep(0, \_SS); // Stay resident in memory

    // The program will continue to run in the background, and the user can input integers.

    // Pressing Ctrl+C will display the sum.

    // To exit the program, the user can use an external utility or a specific exit command.

    // Note: This code is specific to DOS and may not work on modern systems.

}