Lab 2: Scientific Computations and Matrix Multiplication

Name: Syed Muhammad Ghufran Alvi ID: sa08235 Section: L1

Objective

To explore scientific computations and to get familiarity with using loops to solve for matrix operations in Keil simulation environment.

In-Lab

Task 1: Use of arithmetic operator

Task 2: Use of logical operator

Task 3: 3x3 matrix multiplication

Task 4: Armstrong number

Debugging in Keil Environment

1. In Keil uVision, go to your pack installer and follow the steps shown in Fig. 1.
2. Install the mentioned STM32 package in pack-installer as highlighted in the Fig. 2.

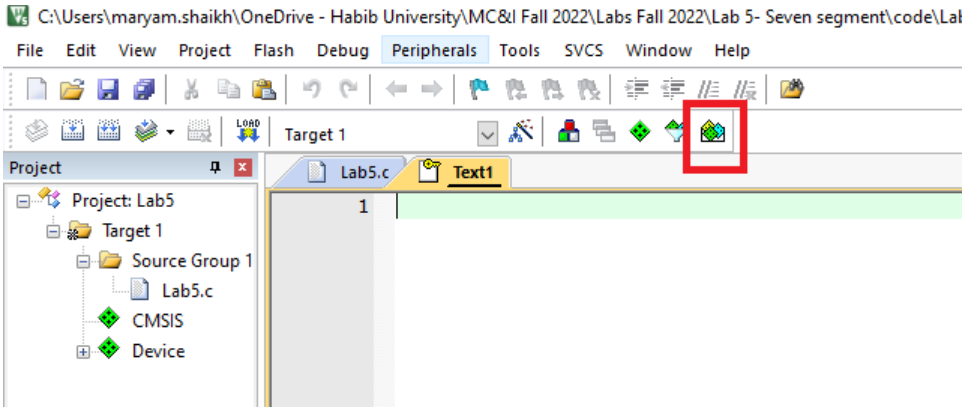


Figure 1

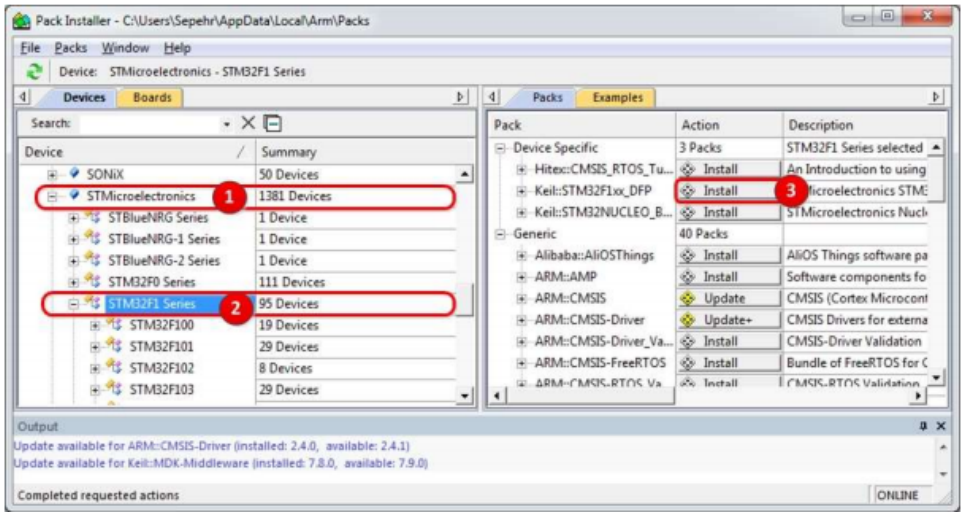


Figure 2

Simulation Mode in Keil

1. Create new project in Kiel using steps from previous Lab 01 and select ”STM32F103C8” as shown in Fig. 3 and check the items given in Fig. 4.
2. Make a new project file by adding new item to ”Source Group 1” with (.c) file as per lab01 steps.
3. Set the ”Options for Target” as shown in Fig. 5.

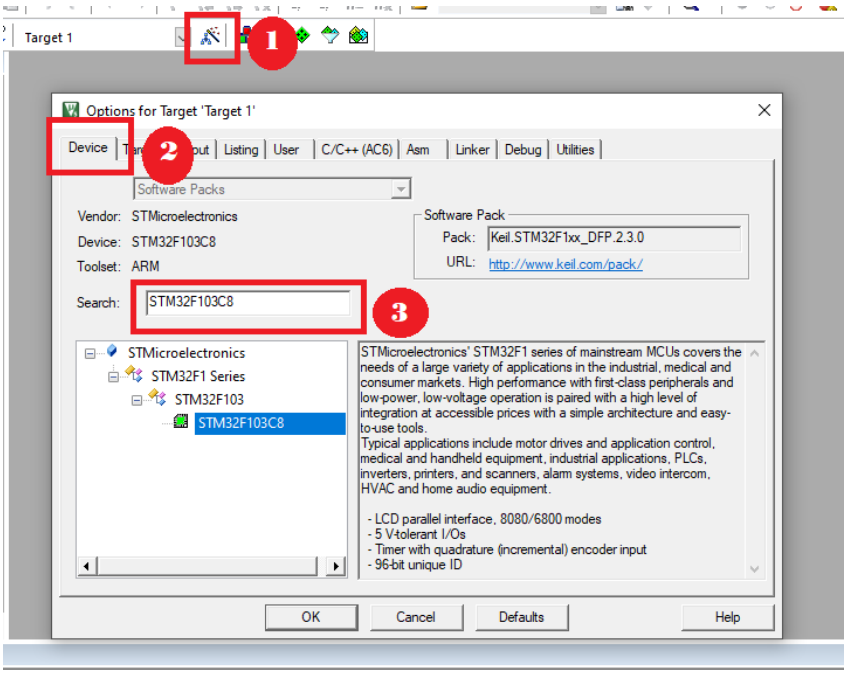


Figure 3

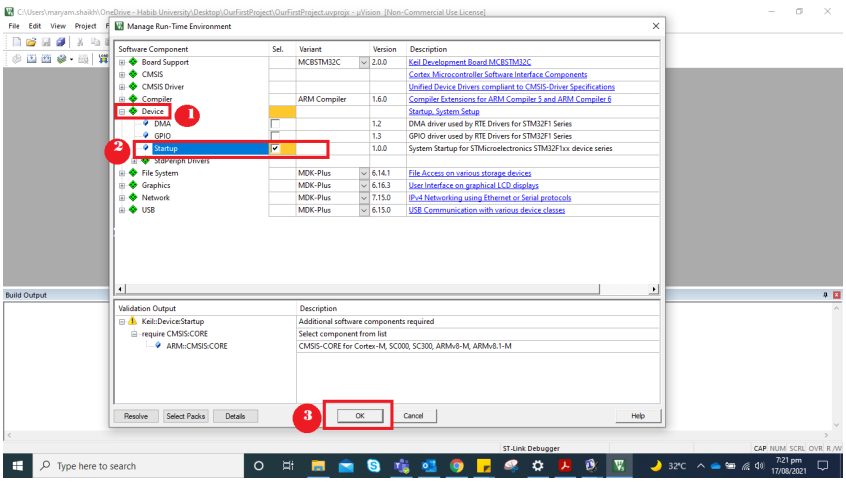


Figure 4

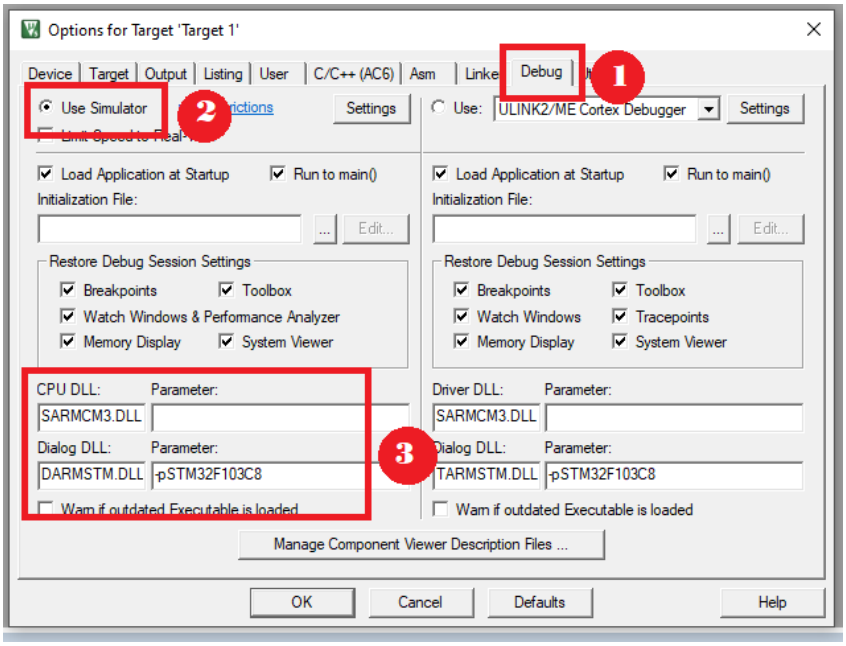


Figure 5

Introduction to Data Types

Computer store data as a sequence of binary bits. A bit is the smallest quantity of information. Each bit has a value of either one or zero, and thus it is called a binary bit. However, it is more efficient for computers to load, process, store, and transmit a group of bits simultaneously. Therefore, bits are divided into a sequence of fixed-length logic units. A byte is a group of 8 bits, a halfword consist of 16 bits (or 2 bytes), a word has 32 bits (or 4 bytes), and a double-word contains 64 bits (or 8 bytes).

Specific Data types

The C standard specifies the minimum size of basic data types. Table 1 and 2 gives the typical size of some data types and integer data type respectively.

Data type	Size
char	1 byte
unsigned char	1 byte
short int	2 bytes
unsigned short int	2 bytes
long	4 bytes
unsigned long	4 bytes
long long	8 bytes
unsigned long long	8 bytes

Table 1: Data type

Data type	Size
int8_t	1 byte
uint8_t	1 byte
int16_t	2 bytes
uint16_t	2 bytes
int32_t	4 bytes
uint32_t	4 bytes
int64_t	8 bytes
uint64_t	8 bytes

Table 2: Integer data type

Arithmetic and Logical operators

Arithmetic and logical operators are two essential operators of data processing. Logical operators produce n-bit results of the corresponding logical operation. Figure 6 and 7 shows standard arithmetic and logical operators with C examples respectively.

```
int i, j, k;            // 32-bit signed integers
uint16_t m, n, p;       // 8-bit unsigned numbers

i = j + k;              // add 32-bit integers
m = n - 5;              // subtract 8-bit numbers
j = i * k;              // multiply 32-bit integers
m = n / p;              // quotient of 8-bit divide
m = n % p;              // remainder of 8-bit divide
i = (j + k) * (i - 2); // arithmetic expression
```

Figure 6: Examples of Arithmetic operators

```
C = A & B;      A   0 1 1 0 0 1 1 0
(AND)           B   1 0 1 1 0 0 1 1
                C   0 0 1 0 0 0 1 0

C = A | B;      A   0 1 1 0 0 1 0 0
(OR)            B   0 0 0 1 0 0 0 0
                C   0 1 1 1 0 1 0 0

C = A ^ B;      A   0 1 1 0 0 1 0 0
(XOR)           B   1 0 1 1 0 0 1 1
                C   1 1 0 1 0 1 1 1

B = ~A;         A   0 1 1 0 0 1 0 0
(COMPLEMENT)    B   1 0 0 1 1 0 1 1
```

Figure 7: Examples of Logical operators

Before moving further, let's see program organization in C-language.

Figure 8 shows a basic C program structure. It can be divided into various blocks as shown with rectangular boxes.

1. The first red rectangular box shows inclusion of header files. Keil MDK-ARM provides a derivative-specific "header file" for each microcontroller, which defines memory addresses and symbolic labels for CPU and peripheral function register addresses. This eliminates the need of specifying address locations.
2. Next, we initialize global variables. These variables are accessible by all functions in a program.
3. Function prototype are used to tell the compiler about the number of arguments and about the required datatypes of a function parameter, it also tells about the return type of the function. By this information, the compiler cross-checks the function signatures before calling it.
4. Function definition provides actual body of the function.
5. Every C program has a primary (main) function that must be named main. The main function serves as the starting point for program execution. It usually controls program execution by directing the calls to other functions in the program.
6. Main function is organized as follows: variables declaration and initialization, second initialization of I/O ports and function registers. Lastly, in an endless while loop the actual program is added. The function can be called in this while loop or as required.

```
1  #include <stm32f10x.h>
2  /*Global variables - accessible by all functions */
   int count, bob;
5
3  /*function prototype*/
   int function1(char z);
8
9  /*Function definitions*/
4  int function1(char x){
     int i,j;
12   //--instructions to implement the function
13 }
14
5  /*Main program*/
   int main(void){
17   unsigned char swi;
6    int k;
19   /*Initialization section*/
20 7 //--instructions to initialize variables, I/O ports, devies, function registers
21   /*Endless loop*/
22   while(1){
23 8 //--instructions to be repeated
24   }
25 }
```

Figure 8: C-language program structure

Debug (printf) Viewer window

The Debug (printf) Viewer window displays data streams that are transmitted sequentially via ITM Stimulus Port 0. It works in similar manner to printing strings on output terminal using the "print" command for various programming languages. However for Keil, to use the viewer for printf debugging, follow these steps:

1. Open the Manage RTE dialog (the symbol towards the right of "Options for Target") and expand the Compiler – I/O folder.
2. Check the STDERR, STDIN and STDOUT components and set the Variant to ITM for each component as shown in Fig. 9 and press OK.
3. You will have to add the following command in your code:
```
#include <stdio.h>
#include <stm32f10x.h>
```
4. Whatever result or string you want to print, add a debugging trace messages using printf, for example: `printf("Sum = 0x%04X\r\n", sum);`
5. Once you have compiled/build your code, you will have to enter into the "debugging mode" by pressing "Ctr+F5" or clicking "Start/stop debug session" button. Once in debug mode, open the window from the menu View - Serial Windows - Debug (printf) Viewer.

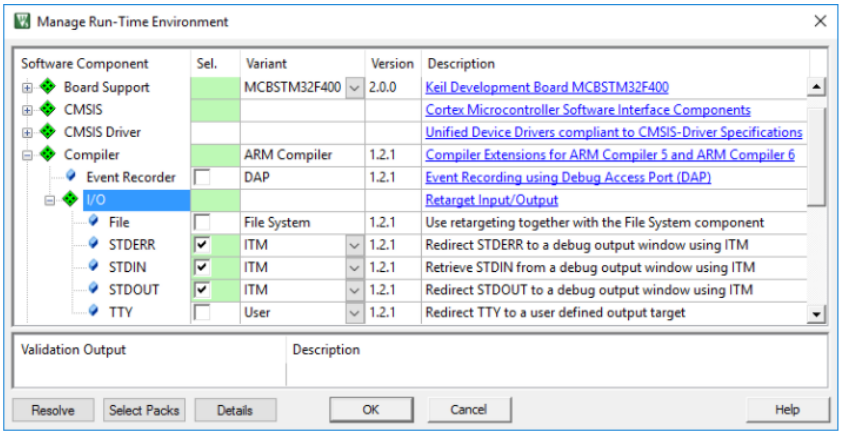


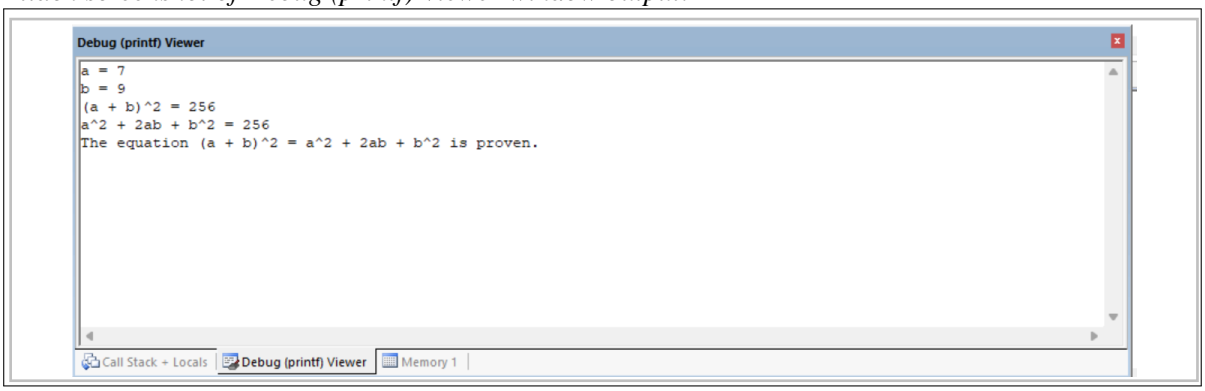
Figure 9

Task 1

Write a c-language program to prove the following equation: $(a+b)^2 = a^2 + 2ab + b^2$. Use values of your choice for variables a and b. Read further at link "**How to use printf in C**" to use "printf" command in C-language. Provide your code below.

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

int main() {
  int a = 7;
  int b = 9;
    // Calculate (a + b)^2
    int left_side = (a + b) * (a + b);

    // Calculate a^2 + 2ab + b^2
    int right_side = (a * a) + (2 * a * b) + (b * b);

    // Print the values and the results
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("(a + b)^2 = %d\n", left_side);
    printf("a^2 + 2ab + b^2 = %d\n", right_side);

    // Check if both sides are equal
    if (left_side == right_side) {
        printf("The equation (a + b)^2 = a^2 + 2ab + b^2 is proven.\n");
    } else {
        printf("There is an error in the calculation.\n");
    }

    return 0;
}
```

Attach screenshot of Debug (printf) Viewer window output:



Task 2

It is often required to make your data secure. For this, you have to encrypt the 'user-data' using a 'Key'. The XOR gate is helpful in this. You can easily encrypt and decrypt the user-data using following equations:

$$\begin{aligned} encrypted\_data &= user\_data \oplus Key \\ decrypted\_data &= encrypted\_data \oplus Key \end{aligned} \tag{1}$$

Let's suppose that we have user_data = "Habib University" and we enrypt it using the key=XY which are the last 2-digits of your HUID = $(abcXY)_{16}$. Each character of the user_data is 8-bits ASCII code. Each character has to be xor-ed with the $Key = (XY)_{16}$. For example, if your HUID is 01234, then the $Key = (34)_{16} = (00110100)_2$. The ASCII code of character 'H' is $(A8)_{16} = (10101000)_2$. When you do encryption and decryption given in the above equations, you will get:

$$\begin{aligned} encrypted\_data &= user\_data \oplus Key \\ &= (10101000)_2 \oplus (00110100)_2 \\ &= (10011100)_2 \end{aligned}$$

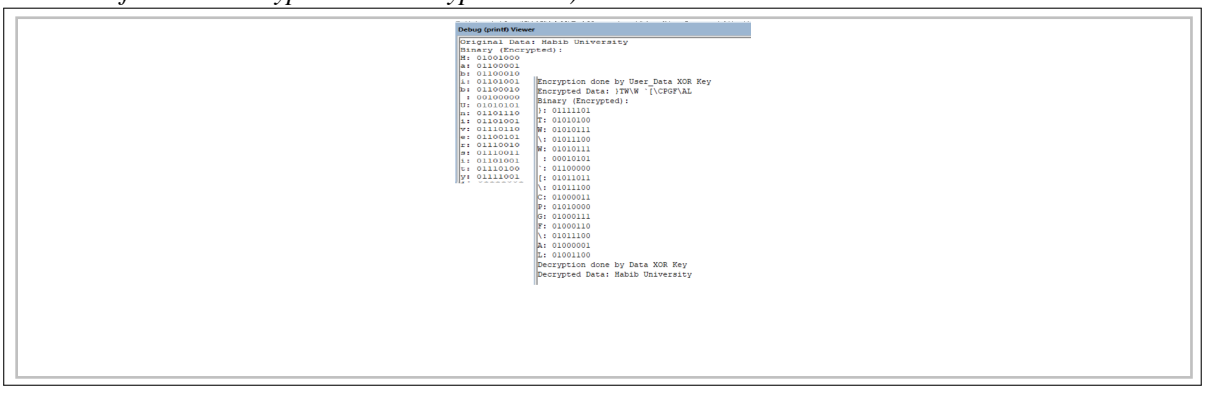
$$\begin{aligned} decrypted\_data &= encrypted\_data \oplus Key \\ &= (10011100)_2 \oplus (00110100)_2 \\ &= (10101000)_2 \\ &= user\_data \end{aligned}$$

which is correct ASCII for 'H'. Verify that you get the same text after this process. Please note that above notation is just for purpose of explanation and not the correction programming syntax.
Write a C-language program for this encryption and decryption application and prove Equation set 1 with appropriate comments.

Hint: Use array structure and XOR operator of C-language.

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

// Function to XOR each character with the key
void xorEncryptDecrypt(char *data, char key) {
  // size_t is used as the type of counting variable i so that we can compare unsigned integer with unsigned integer
    for (size_t i = 0; i < strlen(data); i++) {
        data[i] ^= key; // XORing each bit
    }
}
// Function to repersent each character with their respective binary number
void printBinary(char c) {
    for (int i = 7; i >= 0; i--) {
        printf("%d", (c >> i) & 1);
    }
}

int main() {
    char data[] = "Habib University"; // Data
    // Key derived from HUID (08235) -> last two digits 35 in hexadecimal
    char key = 0x35; // 0x35 in hex is 00110101 in binary
    printf("Original Data: %s\n", data);
  printf("Binary (Encrypted):\n");
  //Binary repersentation
    for (size_t i = 0; i < strlen(data); i++) {
        printf("%c: ", data[i]);
        printBinary(data[i]);
        printf("\n");
    }
    // Encrypting data
    xorEncryptDecrypt(data, key);
  printf("Encryption done by User_Data XOR Key\n");
  printf("Encrypted Data: %s\n" , data);
  printf("Binary (Encrypted):\n");
  //Binary repersentation
    for (size_t i = 0; i < strlen(data); i++) {
        printf("%c: ", data[i]);
        printBinary(data[i]);
        printf("\n");
    }
    // Decrypting data
    xorEncryptDecrypt(data, key);
  printf("Decryption done by Data XOR Key\n");
    printf("Decrypted Data: %s\n", data);
    return 0;
}
```

Attach screenshot of Debug (printf) Viewer window (Should display both the binary and its equivalent character for both encrypted and decrypted data):



Introduction to Loop and Conditional Statements

A program is usually not limited to a linear sequence of instructions since during its process it may bifurcate, repeat code or bypass sections. Control Structures are the blocks that analyze variables and choose directions in which to go based on given parameters.
The basic Control Structures in programming languages are:

Conditionals (or Selection): which are used to execute one or more statements if a condition is met or select one set of statements to be executed from several options, depending on one or more conditions.
Loops (or Iteration): which purpose is to repeat a statement a certain number of times or while a condition is fulfilled or until some condition is met.

Conditional

IF-THEN structure: It execute a set of statements if and only if some condition is met (refer Figure below).
IF-THEN-ELSE structure: It execute one set of statements if a condition is met and an alternate set if the condition is not met (refer Figure below).
Switch statement: It is used for multiway decision that tests one variable or expression for a number of constant values.

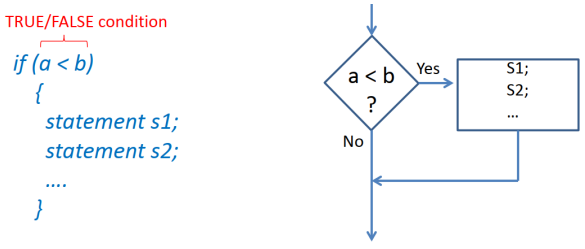


Figure 10: if-then structure

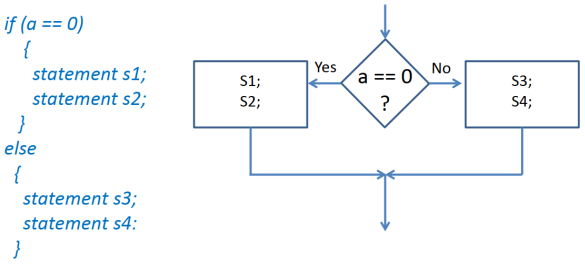


Figure 11: if-then-else structure

```
switch ( n) {       //n is the variable to be tested
   case 0:  statement1;  //do if n == 0
   case 1:  statement2;  // do if n == 1
   case 2:  statement3;  // do if n == 2
   default: statement4;  //if for any other n value
}
```

Figure 12: Switch statement

Loops

Loops repeat a sequence of steps as often as necessary, it gives computers much of their power.
While loop: It repeats a set of statements as long as some condition is met.
Do-while loop: It repeats a set of statements (one "loop") until some condition is met.
For loop: It repeats a set of statements (one "loop") while some condition is met.

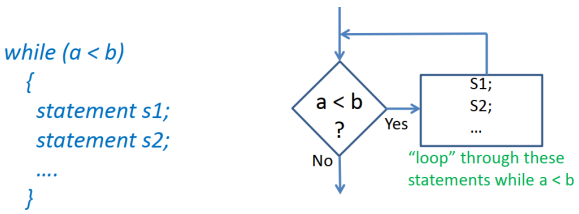


Figure 13: while loop

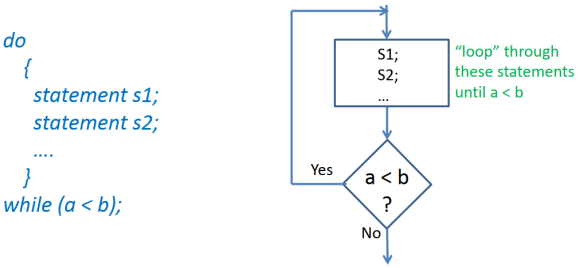


Figure 14: do-while loop

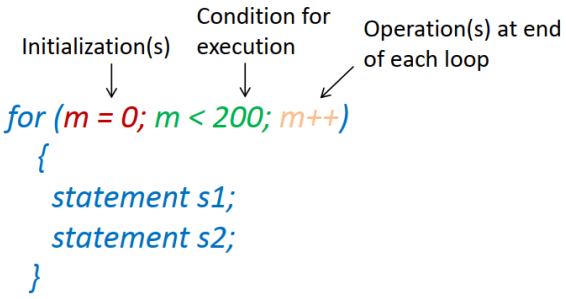


Figure 15: For loop

Task 3

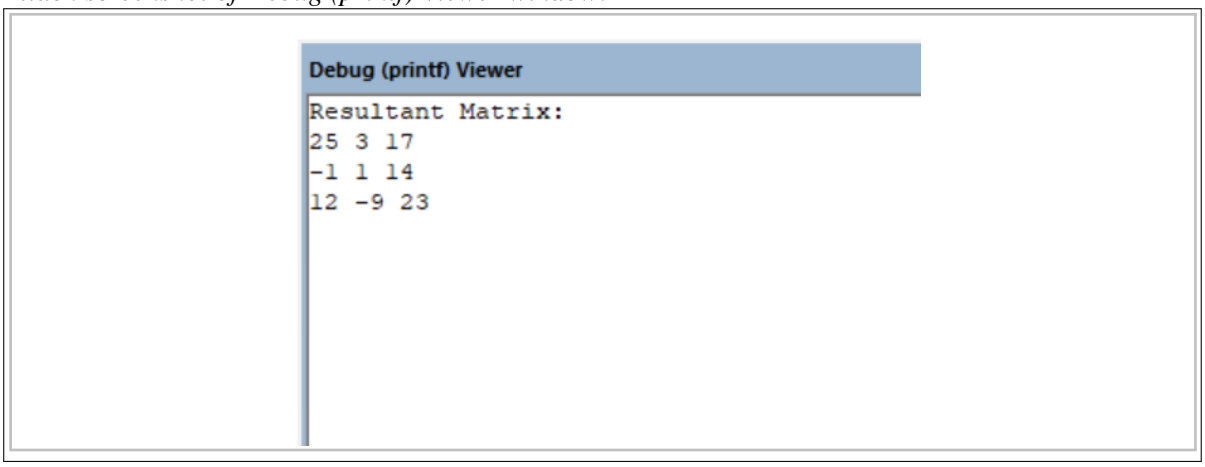
Write a c-language program to find multiplication product of two matrices given below:

$$A = \begin{bmatrix} 2 & 1 & 4 \\ 4 & 0 & -1 \\ 5 & -1 & 2 \end{bmatrix}, B = \begin{bmatrix} 1 & 0 & 4 \\ 3 & 7 & 1 \\ 5 & -1 & 2 \end{bmatrix}$$

Provide the C-program/code for your implementation below:

```c
#include <stdio.h>
//symbolic repersentation is used instead of int type variable for clarity of the code
#define ROWS_A 3
#define COLS_A 3
#define ROWS_B 3
#define COLS_B 3
//this function first creats a null matrix, then fills it with values
void multiplyMatrices(int firstMatrix[ROWS_A][COLS_A], int secondMatrix[ROWS_B][COLS_B], int resultMatrix[ROWS_A][COLS_B]) {
    for (int i = 0; i < ROWS_A; i++) {
        for (int j = 0; j < COLS_B; j++) {
            resultMatrix[i][j] = 0; //creating null matrix
            for (int k = 0; k < COLS_A; k++) {
                resultMatrix[i][j] += firstMatrix[i][k] * secondMatrix[k][j]; //filling the blanks spaces, replacing zeros
            }
        }
    }
}

void printMatrix(int matrix[ROWS_A][COLS_B]) {
    for (int i = 0; i < ROWS_A; i++) {
        for (int j = 0; j < COLS_B; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int A[ROWS_A][COLS_A] = {
        {2, 1, 4},
        {4, 0, -1},
        {5, -1, 2}
    };

    int B[ROWS_B][COLS_B] = {
        {1, 0, 4},
        {3, 7, 1},
        {5, -1, 2}
    };

    int result[ROWS_A][COLS_B];

    multiplyMatrices(A, B, result);

    printf("Resultant Matrix:\n");
    printMatrix(result);

    return 0;
}
```

Attach screenshot of Debug (printf) Viewer window:



Task 4

An n-digit number is Armstrong if the sum of the nth powers of its digits equals the number it self. For example, the following are the Armstrong numbers

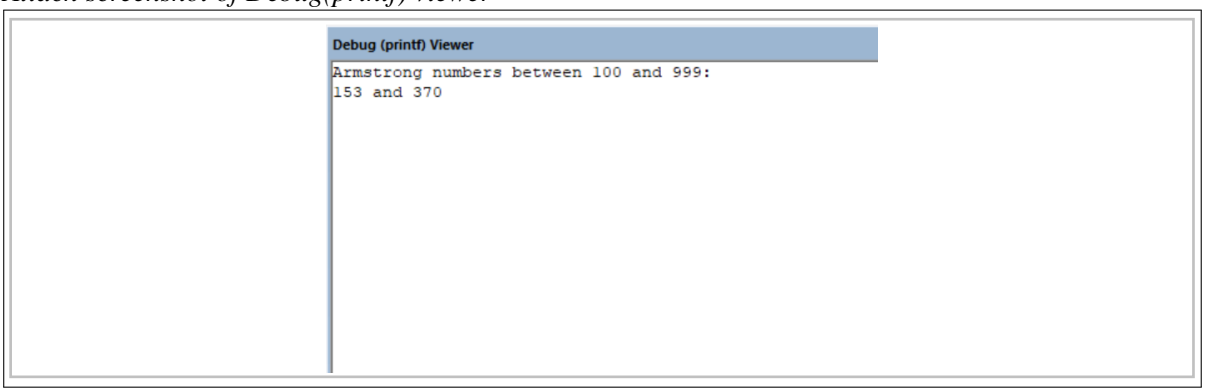
$$\begin{aligned} 371 &= 3^3 + 7^3 + 1^3 \\ 1634 &= 1^4 + 6^4 + 3^4 + 4^4 \end{aligned} \tag{2}$$

Write a C-Language program to find atleast 2 armstrong number for the range of 3-digit numbers (100-999).

Provide your code here

```
#include <stdio.h>

// function to check if a number is an Armstrong Number
int isArmstrong(int num) {
    int originalNum = num;
    int sum = 0;
    int digit;
  //every time we check the next number we take it's cube and add to the sum.
    while (num != 0) {
        digit = num % 10;
        sum += digit * digit * digit; //hardcoded power of the digit as all are going to be 3 digit number
        num /= 10;
    }
  //our return statement checks if the sum is same as the number and returns true (1 in our case) if the number is a Armstrong number
    return (sum == originalNum);
}

int main() {
    int count = 0;

    printf("Armstrong numbers between 100 and 999:\n");

    for (int i = 100; i <= 999; i++) {
        if (isArmstrong(i)) {
            printf("%d ", i); // printing the Armstrong number
            count++;
            if (count >= 2) {
                break; // stop after finding at least 2 Armstrong numbers
            }
     printf("and ");
        }
    }

    printf("\n");

    return 0;
}
```

Attach screenshot of Debug(printf) viewer



1 Assessment Rubrics

Marks distribution

		LR2	LR5	LR9
In-lab	Task 1	10 points	5 points	10 points
	Task 2	20 points	5 points	
	Task 3	20 points	5 points	
	Task 4	20 points	5 points	
Total Marks 100				

Marks Obtained

		LR2	LR5	LR9
In-lab	Task 1			10 points
	Task 2			
	Task 3			
	Task 4			
Marks Obtained				