Name of the student	**Abhishekraddi G Muganur**
Roll No	**137**
USN	**01FE22BEI043**
Div	**G**

Part- A

Variables and Data Types:
1.What is the difference between a variable and a data type in C programming? Provide examples to illustrate.
A. data type defines the kind of data a variable can hold. It acts like a blueprint, specifying the range of values, operations allowed, and the amount of memory allocated for the variable. Common data types in C include (integers), float (floating-point numbers), char (characters), and double (double-precision floating-point numbers). Ex.int age;   Float pi;
2. Explain the concept of data types in C programming. Discuss the different types of data types available in C.
A. Data types in C act like blueprints, defining the kind of information variables can store. They specify the value range, allowed operations, and memory allocation for each variable. Common types include int (integers), float (decimals), char (characters), and double (high-precision decimals). Choosing the right data type ensures efficient memory usage and prevents errors in calculations.
3. How are variables declared and initialized in C programming? Provide examples of variable declarations with different data types
A. In C, you declare variables by specifying the data type and giving them a name. Here are some examples Int age; (stores integers), float pi = 3.14; (stores decimals with initial value),char initial = 'A'; (stores a single character),double distance; (stores high-precision decimals)
4. Discuss the scope and lifetime of variables in C programming. What are global and local variables?
A. Scope and lifetime define where and how long variables exist in C. Scope**:** Refers to the program area where a variable is accessible. It's like a zone where the variable's name is recognized. Lifetime**:** Determines how long the memory allocated for the variable remains valid during program execution. There are two main types: Global**:** Declared outside functions, accessible throughout the program (broader scope, program lifetime).

Local**:** Declared within functions, accessible only inside that function (limited scope, function execution lifetime).

5. Explain the concept of type casting in C programming. When is type casting necessary, and how is it performed?

A. Type casting in C lets you explicitly convert a variable from one data type to another. Imagine it as forcing a value into a different sized box.

It's useful in a few cases:

Assigning values between incompatible types**:** Going from int (whole numbers) to float (decimals) for calculations.

Function compatibility**:** Using a variable in a function expecting a different data type.

Casting is done using the cast operator (data_type) before the variable you want to convert. For example, (float)age converts age (int) to a float. Be cautious though, as casting can lead to data loss if the target type can't hold the original value's range.

Operators:

1.Describe the purpose and usage of the ternary conditional operator in C programming. Provide an example demonstrating its usage.

A. The ternary conditional operator, also called the conditional operator, offers a concise way to make decisions in C code. It acts like a shorthand if-else statement within a single expression.

condition: The expression to be evaluated.

Expression if true: The value returned if the condition is true.

Expression if false: The value returned if the condition is false.

Example:

2.Discuss the bitwise operators available in C programming. Explain their usage with suitable examples.

A. C programming offers bitwise operators that perform operations on individual bits of data. These operators work on integer data types.

Bitwise AND (&): Sets a bit to 1 only if the corresponding bits in both operands are 1. x & y: 1010 & 0110 = 0010

Bitwise OR (|): Sets a bit to 1 if at least one corresponding bit in either operand is 1. x | y: 1010 | 0110 = 1110

Bitwise XOR (^): Sets a bit to 1 if the corresponding bits in the operands are different. x ^ y: 1010 ^ 0110 = 1100

Left shift (<<): Shifts bits of the first operand to the left by the number of bits specified by the second operand. x << 2: 1010 << 2 = 101000

Right shift (>>): Shifts bits of the first operand to the right by the number of bits specified by the second operand. x >> 1: 1010 >> 1 = 0101

3. Explain the difference between the postfix and prefix increment operators (++) in C programming. Provide examples to illustrate.

A. In C, the prefix (++) and postfix (++) increment operators both increase a variable's value by 1, but the key difference lies in when the increment happens:

- Prefix **(++)**: Increments the variable first, then uses the new value.
  - Example: int x = 5; ++x; // x becomes 6 (increment happens before assignment)
- Postfix **(++):** Uses the variable's current value first, then increments it.
  - Example: int x = 5; int y = x++; // y becomes 5 (original value), then x becomes 6 (assignment uses original value, then increment happens)

4. What is the significance of the logical AND (&&) and logical OR (||) operators in C programming? How are they used in conditional expressions?

A. Logical AND (&&) and logical OR (||) operators are essential for combining conditions in C's conditional expressions. They determine the overall truth value based on the operands (conditions) being evaluated.

- Logical AND (&&): Represents "both conditions must be true". The expression returns true only if all operands evaluate to true. If the first operand is false, the second operand isn't even evaluated (short-circuit evaluation).
- Logical OR (||): Represents "at least one condition must be true". The expression returns true if any operand evaluates to true. Evaluation stops as soon as a true operand is encountered (short-circuit evaluation).

5. Discuss the concept of operator precedence and associativity in C programming. Provide examples to demonstrate how they affect expression evaluation.

A. Operator precedence and associativity are crucial concepts in C programming that dictate how expressions are evaluated.

- Precedence**:** Defines the order in which different operators are evaluated. Operators with higher precedence are evaluated first. Imagine it as a hierarchy, resolving high-priority operations before moving on.

- Associativity**:** Determines the order of evaluation for operators with the same precedence. It can be left-to-right or right-to-left. Think of it as a tiebreaker rule within the same precedence level.

Control Structures:

1. Describe the purpose and usage of the switch statement in C programming. How does it differ from the if-else statement?
A. The switch statement in C provides a multi-way branching mechanism for handling different cases based on a single variable's value. It acts like a more concise alternative to long chains of if-else statements.

1.Expression**:** The switch evaluates an expression (usually an integer or character).

2.Case Labels**:** Each case label represents a possible value the expression can hold.

3.Code Block**:** The code block associated with a matching case label is executed.

4.break (Optional): The break statement exits the switch after a matching case is executed, preventing fall-through to subsequent cases

Differences from if-else**:**

Clarity**:** switch can be more readable for handling multiple conditions based on a single variable.

Fall-through: By default, switch cases fall through to the next case if no break is present. if-else avoids this behaviour

Data Types**:** switch expressions are typically limited to integers or characters, while if-else can handle more complex conditions.

2.Explain the concept of nested control structures in C programming. Provide an example demonstrating nested if-else statements.

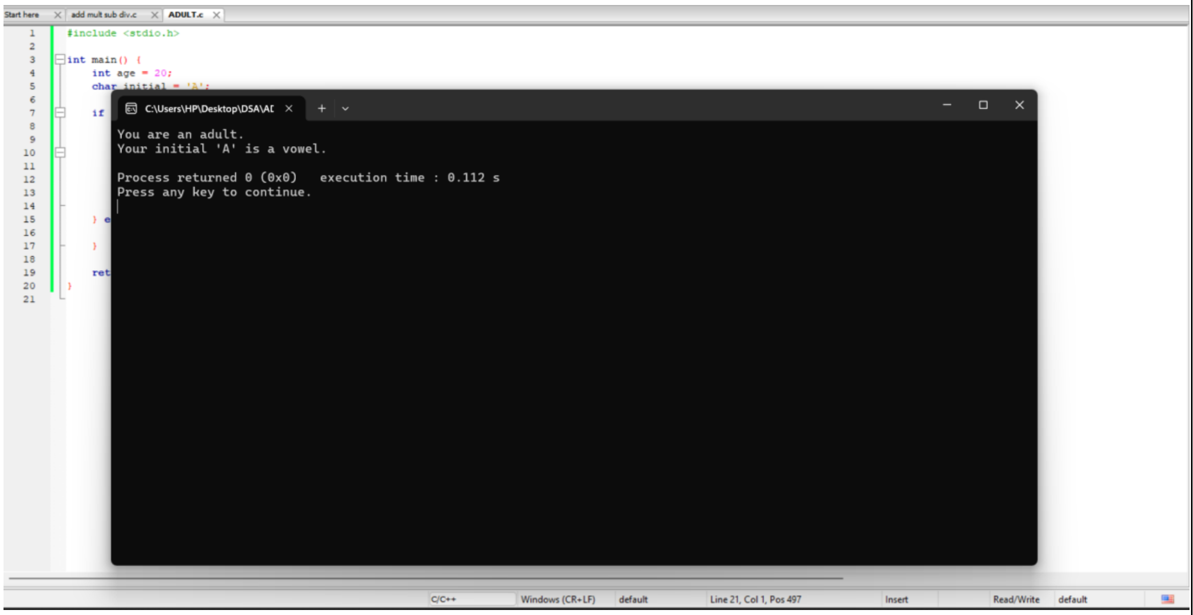
A. Nested control structures in C allow you to create complex decision-making logic by embedding one control structure within another. It's like creating layers of conditions.

Example (Nested if-else):

```
#include <stdio.h>



int main() {

    int age = 20;
    char initial = 'A';


    if (age >= 18) {
        printf("You are an adult.\n");


        if (initial == 'A' || initial == 'E' || initial == 'I' || initial == 'O' || initial == 'U') {
            printf("Your initial '%c' is a vowel.\n", initial);
        } else {
            printf("Your initial '%c' is a consonant.\n", initial);
        }
    } else {
        printf("You are not an adult.\n");
    }


    return 0;
}
```

Here, the outer if checks if age is 18 or more. If true, an inner if-else checks the vowel condition for the initial only if the outer condition is met. This demonstrates nesting if statements for a more intricate decision flow.

3. Discuss the role of the break and co



```
#include <stdio.h>

int main() {
    int age = 20;
    char initial = 'A';

    if
```

```
You are an adult.
Your initial 'A' is a vowel.

Process returned 0 (0x0)   execution time : 0.112 s
Press any key to continue.
```

ntinue statements in loop control in C programming. Provide examples to illustrate their usage.

A.Purpose of break: Terminates the loop entirely, exiting the loop's body as soon as it's encountered.

}

Purpose of continue**:** Skips the current iteration of the loop and jumps to the beginning of the next iteration. The remaining code within the current iteration is not executed.

4. What are the advantages of using the for loop over the while loop in C programming? Provide examples comparing the two.

A. Advantages

Clarity and Readability:

- for loops explicitly combine initialization, condition check, and increment/decrement in a single line, making the loop's purpose clearer at a glance.

Conciseness:

- This combined structure often leads to more concise code compared to while loops, especially when all three components (initialization, condition, increment) are }

5. Explain the concept of short-circuit evaluation in C programming. How does it affect the evaluation of logical expressions in if statements?

A. Short-circuit evaluation is an optimization technique used in C programming for logical operators (&& - AND, || - OR). It ensures expressions are evaluated only as far as necessary to determine the final outcome.

Benefits:

1.Improves efficiency by avoiding unnecessary calculations, especially when dealing with functions that might have side effects.

2.Can enhance code readability by allowing you to write conditions where the outcome of the second operand depends on the first.

Functions:

1. Describe the purpose and structure of a function prototype in C programming. Why is it necessary to declare function prototypes?

A. In C programming, a function prototype acts like a blueprint or an announcement for a function. It provides essential information about the function to the compiler before the actual function definition is encountered.

Importance:

- Type Checking: The compiler uses the prototype to ensure the function call matches the declared parameters and return type. This helps catch errors like passing incorrect data types or missing arguments during compilation.
- Code Organization: Prototypes improve code readability and maintainability by documenting function details in a separate header file. This allows you to use functions across multiple source files without worrying about the definition order.

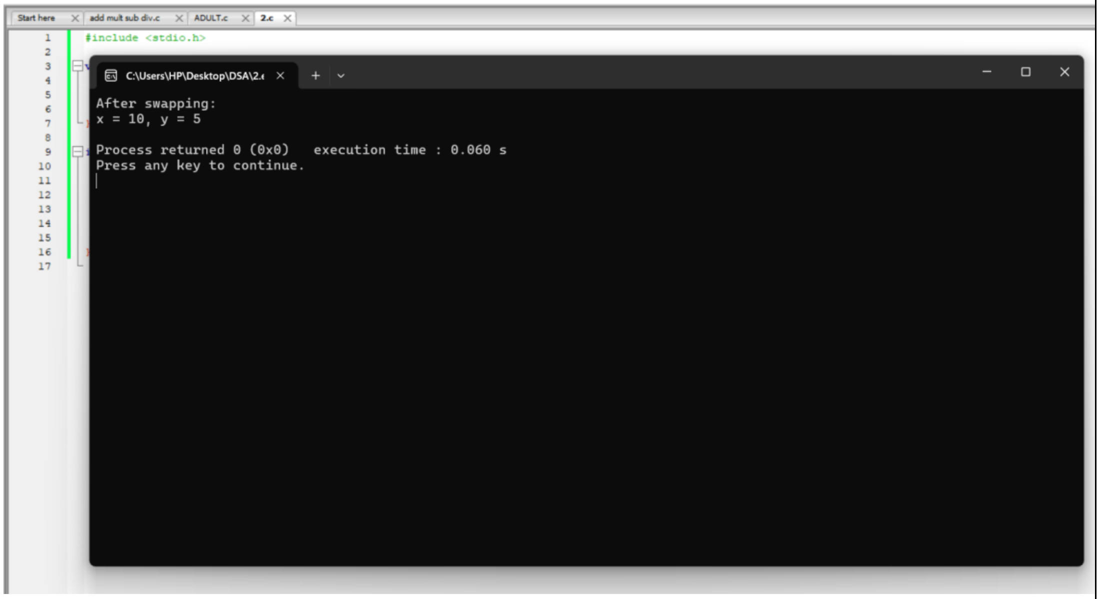
2. Explain the difference between call by value and call by reference in C programming. Provide examples to illustrate both concepts.

A. Call by Value:

- A copy of the actual argument's value is passed to the function.
- Changes made to the parameter inside the function do not affect the original variable in the calling code.
- This is the default mechanism in C for most data types (like int, float).

```
#include <stdio.h>

void swap(int *a, int *b) {

    int temp = *a;

    *a = *b;

    *b = temp;

}

    int main() {
```

```
    int x = 5, y = 10;

    swap(&x, &y);

    printf("After swapping:\n");

    printf("x = %d, y = %d\n", x, y);

   return 0;

}
```



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

```
After swapping:
x = 10, y = 5

Process returned 0 (0x0)   execution time : 0.060 s
Press any key to continue.
```

call by Reference:

- The address (memory location) of the actual argument is passed to the function.
- Modifications made to the parameter through the address directly affect the original variable.
- In C, you need to use pointers (*) to achieve call by reference

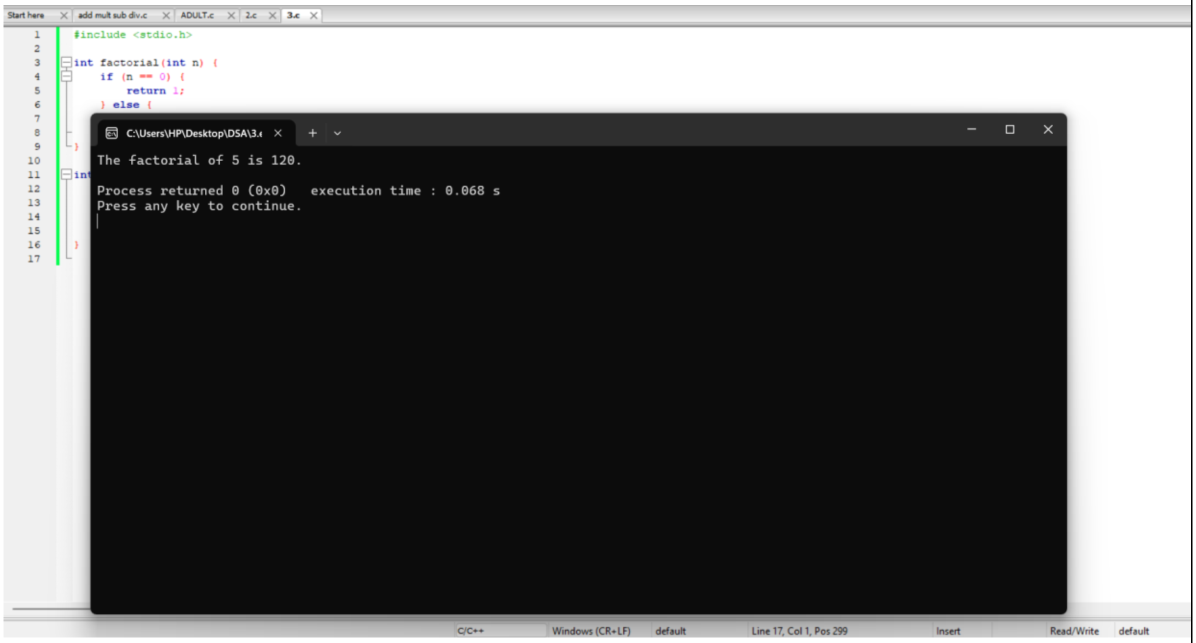
3.Discuss the concept of recursion in C programming. Provide an example of a recursive function and explain how it works.

A. Recursion in C programming involves a function calling itself. It's a technique for solving problems by breaking them down into smaller, self-similar subproblems.

Example: Factorial Function

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

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
int main() {
    int number = 5;
    int result = factorial(number);
    printf("The factorial of %d is %d.\n", number, result);
    return 0;
}
```



```
#include <stdio.h>

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
```

```
The factorial of 5 is 120.

Process returned 0 (0x0)   execution time : 0.068 s
Press any key to continue.
```

4. What is the significance of the return statement in C programming? How are values returned from functions?

A. The return statement in C programming plays a crucial role in function calls. It serves two key purposes:

1.Terminating Function Execution: It marks the end of the function's execution and instructs the program to return control back to the point where the function was called.

2.Returning Values (Optional): The return statement can optionally be used to send a value back from the function to the calling code. This value becomes the result of the function call.

Returning values
1.The return_type specifies the data type of the value being returned.

2.The value_to_return is the actual data you want to send back from the function.

5.Describe the role of function parameters and arguments in C programming. How are function arguments passed to parameters

A. In C programming, functions operate on data but often need input from the calling code. This is where function parameters and arguments come into play:

1.Parameters: These are variables declared within the function definition. They act like placeholders that receive the values passed during the function call. You specify their data types to indicate the kind of data the function expects.

Arguments**:** These are the actual values passed to the function when it's called. .They are listed within parentheses after the function name in the call statement.

Passing Arguments:

1.During a function call, arguments are matched with parameters based on their position in the call statement (not by name).

2.The value of the argument is copied (call by value) to the parameter's memory location within the function.

Arrays:

1. Explain the concept of arrays in C programming. How are arrays declared and initialized?

A. Arrays in C act like collections of elements of the same data type. Imagine them as a fixed-size box where each slot stores a value. You declare them by specifying the data type, array name, and size enclosed in square brackets []. Initialization (assigning values) can be done during declaration or later.

2. Discuss the difference between a one-dimensional array and a multidimensional array in C programming. Provide examples of both

A. One-dimensional arrays hold a single list of elements. Imagine a row of boxes. You declare them with data_type array_name[size]. Example: int numbers[5] = {1, 2, 3, 4, 5};

Multi-dimensional arrays (like 2D arrays) represent a grid-like structure with rows and columns. Think of a table. Declaration: data_type array_name[rows][columns]. Example: int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};

3. Describe the process of accessing array elements in C programming. How are array indices used to access elements?

A. To access elements in a C array, you use the array name followed by the element's index enclosed in square brackets []. The index starts from 0, so the first element has index 0, the second has index 1, and so on.

4. What is the significance of the null character ('\0') in C strings? How is it used to determine the end of a string?

A. In C strings, the null character (\0) acts as a special sentinel marking the string's end. It's a non-printable character with an ASCII value of 0. String functions in C rely on this null terminator to identify the string's boundaries, allowing them to process characters until they encounter the \0. This is essential because C strings are essentially character arrays without a built-in mechanism to store their length.

5. Explain the concept of dynamic memory allocation for arrays in C programming. How are dynamic arrays allocated and deallocated?

A. In C, arrays are typically fixed-size during declaration. Dynamic memory allocation offers flexibility to create arrays with sizes determined at runtime. Here's a basic overview

1.Functions like malloc (allocate memory) and calloc (allocate and initialize) are used to request memory blocks of a desired size (in bytes) based on the number of elements needed.

2.A pointer variable is used to store the memory address returned by malloc or calloc. This pointer acts like a reference to the dynamically allocated array.

3.After use, the allocated memory must be explicitly released using free to prevent memory leaks. This frees the memory block back to the system.

Pointers:

1.Describe the purpose and usage of pointers in C programming. How are pointers declared and initialized

A. In C programming, pointers are variables that store memory addresses. They act like signposts pointing to locations in memory where other data resides. Here's a breakdown (within 4 lines):

Purpose: Pointers offer several functionalities:

Dynamic memory allocation (discussed earlier).

Passing arguments to functions by reference (modifying the original variable).

Working with complex data structures like linked lists and trees.

Declaration and Initialization: Pointers are declared by specifying an asterisk (*) before the data type they point to, followed by the pointer variable name. Initialization can involve assigning the address of a variable (using the & operator) or NULL (indicating no memory allocated yet).

2. Explain the concept of pointer arithmetic in C programming. Provide examples to illustrate addition and subtraction operations on pointers.

A. Pointer arithmetic in C lets you perform calculations on memory addresses stored in pointers. It considers the data type the pointer points to when performing these operations.

Addition

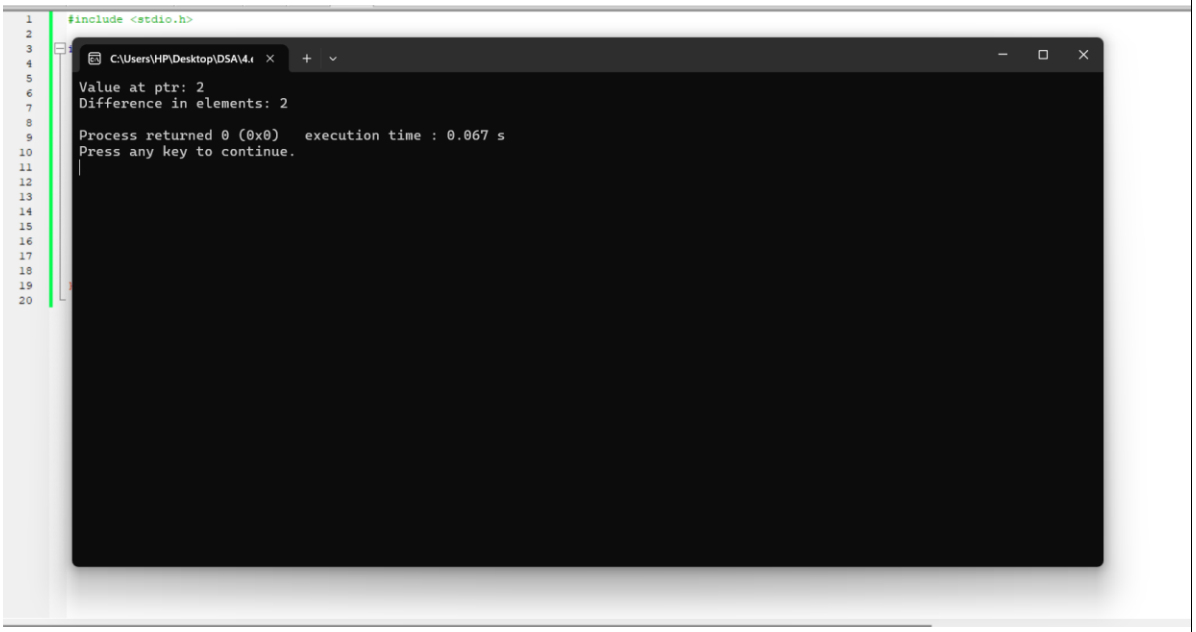
```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr; // ptr points to the first element (arr[0])

ptr++; // Moves ptr one element forward (points to arr[1])

printf("Value at ptr: %d\n", *ptr); // Output: 2 (value at arr[1])
```

Here, ptr++ increments the pointer by the size of an integer (usually 4 bytes), effectively moving it to the next element in the array.

Subtraction

```
int numbers[4] = {10, 20, 30, 40};
int *p1 = &numbers[2]; // p1 points to numbers[2]
int *p2 = numbers;   // p2 points to numbers[0]

int difference = p1 - p2; // Calculates the number of elements between p1 and p2

printf("Difference in elements: %d\n", difference); // Output: 2 (p1 points to the 3rd
element, 2 elements ahead of p2)
```

Subtracting two pointers pointing to the same array results in the number of elements between them. This is because the difference considers the size of the data type they point to.



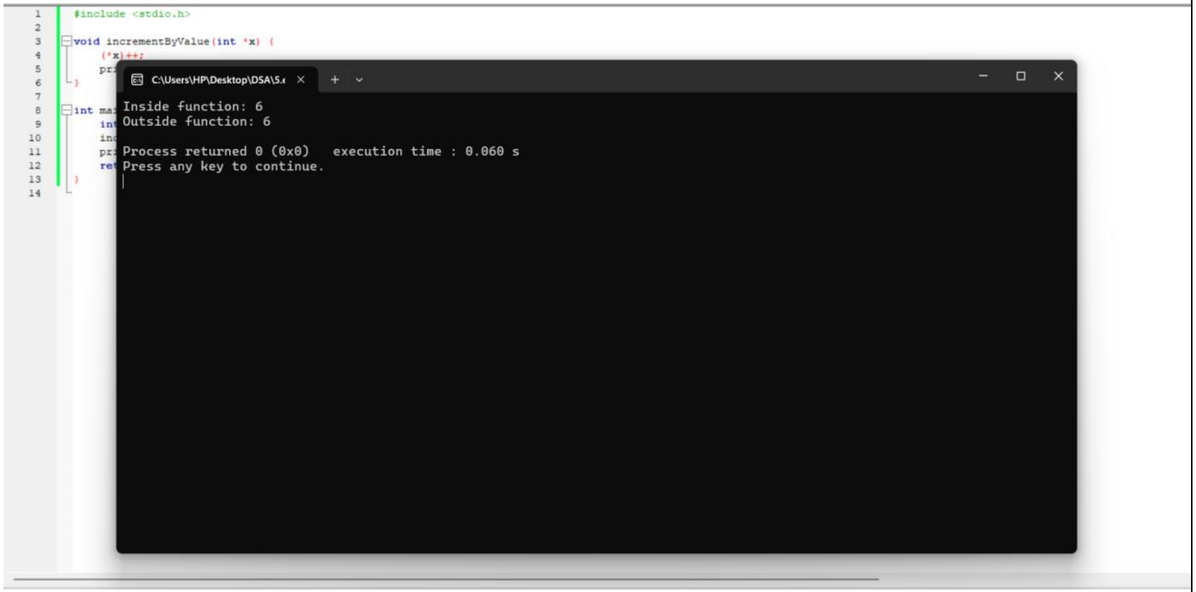
```
#include <stdio.h>
```

```
Value at ptr: 2
Difference in elements: 2

Process returned 0 (0x0)   execution time : 0.067 s
Press any key to continue.
```

3. Discuss the difference between pass by value and pass by reference in function arguments using pointers in C programming. Provide examples to illustrate both approaches

A. C primarily uses pass by value for function arguments. This means a copy of the argument's value is passed to the function. Changes made inside the function only affect the copy, not the original variable. For example, a function trying to increment an integer passed by value would only modify the copy.Pass by reference, achieved using pointers, allows functions to modify the original variable. In C, we pass the memory address of a variable (using the & operator) to the function. The function then works with the actual memory location of the variable. For instance, a function swapping two integers using pointers can directly change their values in memory.

```
#include <stdio.h>

void incrementByValue(int *x) {

    (*x)++;

    printf("Inside function: %d\n", *x);

}

int main() {

    int num = 5;

    incrementByValue(&num);

    printf("Outside function: %d\n", num);
    return 0;

}
```



```
#include <stdio.h>

void incrementByValue(int *x) {
    (*x)++;
    pr
}

int ma
    in
    in
    pr
    re
}
```

C:\Users\HP\Desktop\DSA\5.

```
Inside function: 6
Outside function: 6

Process returned 0 (0x0)   execution time : 0.060 s
Press any key to continue.
```

4. Describe the concept of NULL pointers in C programming. How are NULL pointers used and checked for in programs?

A. A null pointer in C is a special pointer variable that doesn't hold a valid memory address. It essentially points to "nothing." We use the NULL macro to assign a null pointer.Null pointers are used to indicate uninitialized pointers or the end of a linked list. We check for null pointers using the == operator before dereferencing (accessing the data they point to) to avoid program crashes.

5. Explain the role of pointers in dynamic memory allocation in C programming. How are pointers used to allocate and deallocate memory dynamically?
A. In dynamic memory allocation, pointers act as handles. Functions like malloc and calloc return pointers to the allocated memory block. These pointers allow us to access and manipulate the dynamically allocated data. We use free with the pointer to release the memory when it's no longer needed, preventing memory leaks.

STRINGS:

1. Discuss the concept of strings in C programming. How are strings represented and manipulated in C?
A. In C, strings are simply character arrays terminated by a null character (\0). This null terminator acts as a flag to mark the string's end. String manipulation functions like strcpy, strcat, and strlen work with pointers to manage these character arrays.

2. Explain the difference between character arrays and string literals in C programming. Provide examples to illustrate both concept
A. In C, character arrays and string literals are both used for storing text, but with key differences. A character array is a variable that holds multiple characters. You can declare its size and even modify the characters within it. For instance, char name[20] = "Alice"; creates a character array name that can hold up to 19 characters (excluding the null terminator). String literals, written in double quotes like "Hello", are fixed text defined

directly in your code. They are stored in read-only memory and their content cannot be changed.

3. Describe common string manipulation functions available in the C standard library. Provide examples of functions like strlen, strcpy, strcat, and strcmp
A. C provides a set of powerful tools in the <string.h> header for manipulating strings. These functions make working with text data more efficient. The strlen function calculates the length of a string, excluding the null terminator that marks the string's end. You can use strcpy to copy one string to another, but be cautious! strcpy doesn't check if the destination string has enough space, which can lead to program crashes. For safer copying, consider strncpy where you specify the maximum number of characters to copy. Similar caution applies to strcat which appends strings but can also cause buffer overflows. Safer alternatives include strncat with a limit on characters appended or pre-allocating enough space in the destination string. Finally, the strcmp function helps compare strings, returning 0 if they are identical.

4.Discuss the concept of string tokenization in C programming. How are strings split into tokens using delimiter characters?

A. String tokenization in C involves splitting a string into smaller substrings (tokens) based on separator characters (delimiters). The strtok function (from <string.h>) is commonly used for this purpose. It modifies the original string, keeping track of its position with subsequent calls.

5. Explain the importance of null-terminated strings in C programming. How does the null character signify the end of a string?
A. Null-terminated strings are fundamental in C for managing character arrays as strings. The null character (\0), with a value of zero, acts as a silent sentinel at the end of the string. Since arrays don't store their size, this null terminator allows functions to determine the string's length by iterating until they encounter the \0.

1. Describe the purpose and usage of structures in C programming. How are structures declared and accessed?

A. In C, structures group variables of potentially different data types under a single user-defined type. This allows us to represent real-world entities with various attributes. We declare structures using the struct keyword, specifying member names and data types. We access structure members using the dot (.) operator with the structure variable name.

2. Discuss the concept of structure members in C programming. How are individual members of a structure accessed and modified?

A. Structure members are individual variables of various data types bundled within a structure. They represent specific attributes of the entity the structure defines. We access and modify them using the dot (.) operator after the structure variable name followed by the member's name. For example, struct_name.member_name. This allows us to directly interact with specific data points within the structure.

3. Explain the difference between structures and unions in C programming. When would you choose one over the other?

A. In C programming, structures group different data types together under one name, allowing access to each member individually. Unions, on the other hand, share the same memory space for all members, allowing only one member to be active at a time. When you need to store and access multiple pieces of related data simultaneously. Choose unions when you need to conserve memory by storing different types of data in the same memory location, and only one type needs to be accessed at a given time.

4. Describe the concept of nested structures in C programming. How are structures within structures defined and accessed?

A. Nested structures allow you to create complex data hierarchies in C. One structure can contain another structure as a member, enabling you to model entities with composed attributes. You define nested structures by placing a structure declaration within another structure's member definition. To access members of a nested structure, you use the dot (.) operator chained with both the outer and inner structure member names.

5. Discuss the concept of typedef in C programming. How is typedef used to define custom data types, including structures and unions?

A. The typedef keyword in C lets you create aliases for existing data types, including structures and unions. This improves code readability by providing more meaningful names. You use typedef followed by the desired type name and the original data type. For structures and unions, you can typedef the entire structure/union declaration to create a new user-friendly type.

File handling:

1. Explain the concept of file handling in C programming. How are files opened, read from, and written to using standard file handling functions?

A. File handling in C allows programs to interact with files on the storage device. We use functions like fopen to open a file in a specific mode (read, write, append). Then, fread reads data from the file into a buffer, and fwrite writes data from a buffer to the file. Finally, fclose closes the file, releasing resources.

2. Describe the role of file pointers in C programming. How are file pointers used to navigate and manipulate files?

A.File pointers in C act as handles to open files. They store information about the file's state, including the current read/write position. We use these pointers with functions like fseek to move the pointer to specific locations within the file, enabling us to perform operations like random access reads or writes at desired positions.

3. Discuss the difference between text files and binary files in C programming. How are they opened and processed differently?

A. In C, text files store human-readable characters using ASCII encoding. Binary files store raw data (numbers, images, etc.) as sequences of bytes.(1).Text Files: Opened in text mode ("r", "w", "a") with functions like fgets (read) and fputs (write) that handle newline characters appropriately.(**2**)**.**Binary Files: Opened in binary mode ("rb", "wb", "ab") using fread and fwrite to read/write raw byte sequences without newline interpretation.

4. Explain the purpose of file modes in C programming. Provide examples of different file modes like "r", "w", "a", etc.

A. In C programming, file modes determine how files are opened and accessed. They dictate whether a file is opened for reading ("r"), writing ("w"), or appending ("a"). Additional modes include "r+" for reading and writing, "w+" for reading and writing, and "a+" for reading and appending

5. Describe error handling techniques in file operations in C programming. How are errors detected and handled when working with files?

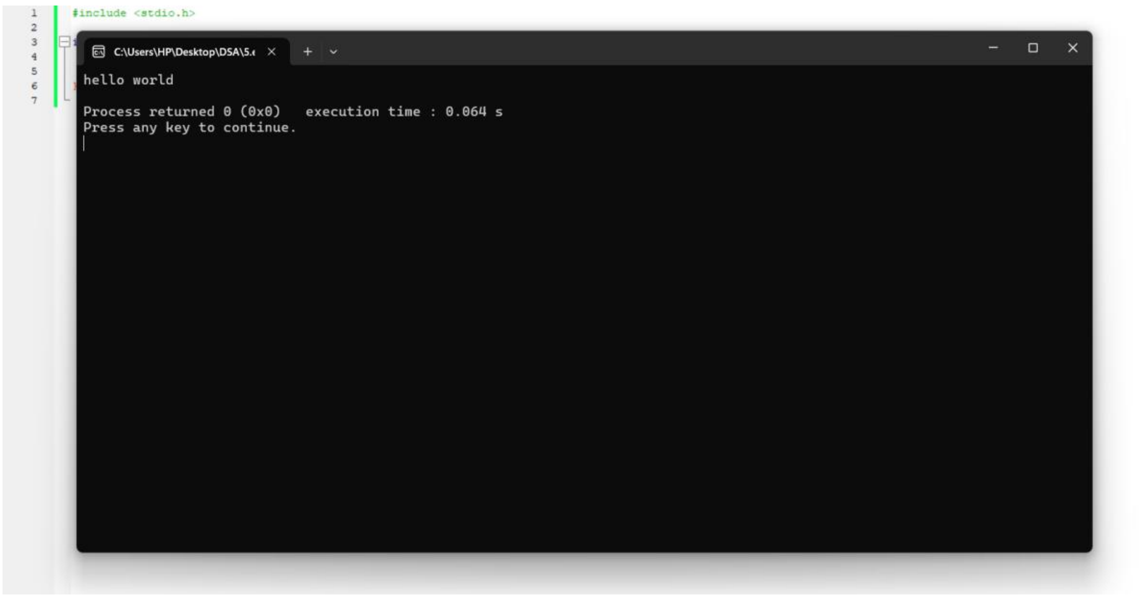
A. In C programming, errors in file operations are commonly detected through return values from functions like fopen(), fread(), fwrite(), etc. Error handling techniques include checking return codes for errors, using perror() to print error messages, and employing errno to identify specific error conditions. Additionally, file-related errors can be handled using techniques such as try-catch blocks or using macros for cleaner error handling code.

Part- B

1. Hello world

Code

```
#include<stdio.h>
main()
{
printf("hello world");
}
```



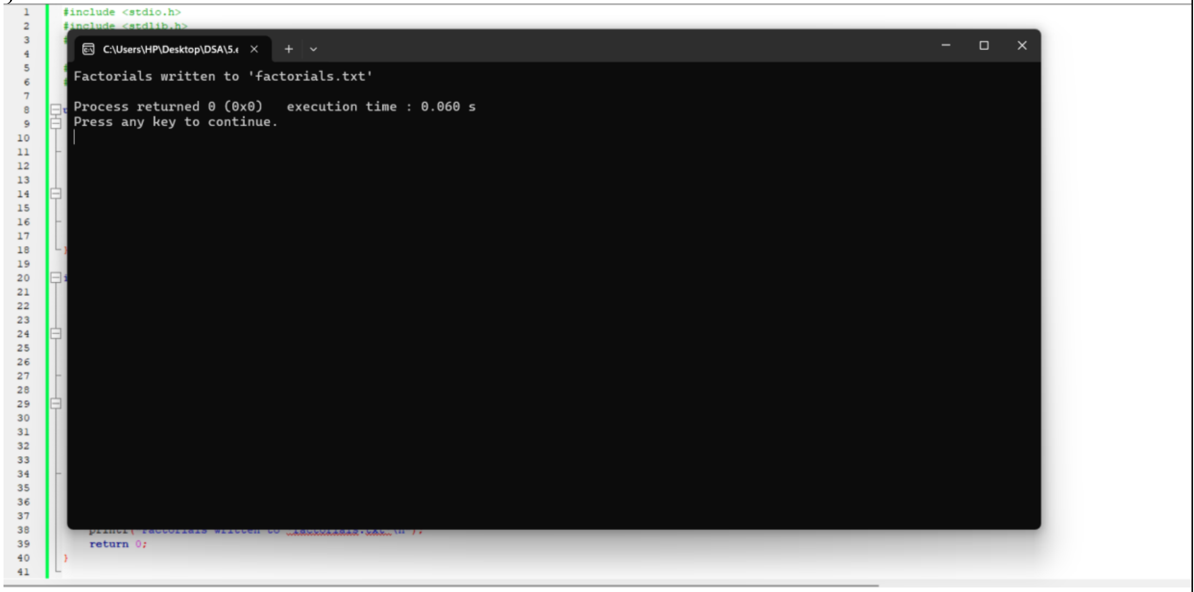
```
#include <stdio.h>
```

```
hello world

Process returned 0 (0x0)   execution time : 0.064 s
Press any key to continue.
```

2.Factorial

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

#define MAX_NUMBER 100
#define NUM_FACTORS 5

unsigned long long calculate_factorial(int n) {
    if (n == 0) {
        return 1;
    }

    unsigned long long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

```
int main() {
    srand(time(NULL));

    FILE *fp = fopen("factorials.txt", "w");
    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    for (int i = 0; i < NUM_FACTORS; i++) {
        int random_number = rand() % (MAX_NUMBER + 1);
        unsigned long long factorial = calculate_factorial(random_number);

        fprintf(fp, "Factorial of %d is %llu\n", random_number, factorial);
    }

    fclose(fp);

    printf("Factorials written to 'factorials.txt'\n");
    return 0;
}
```

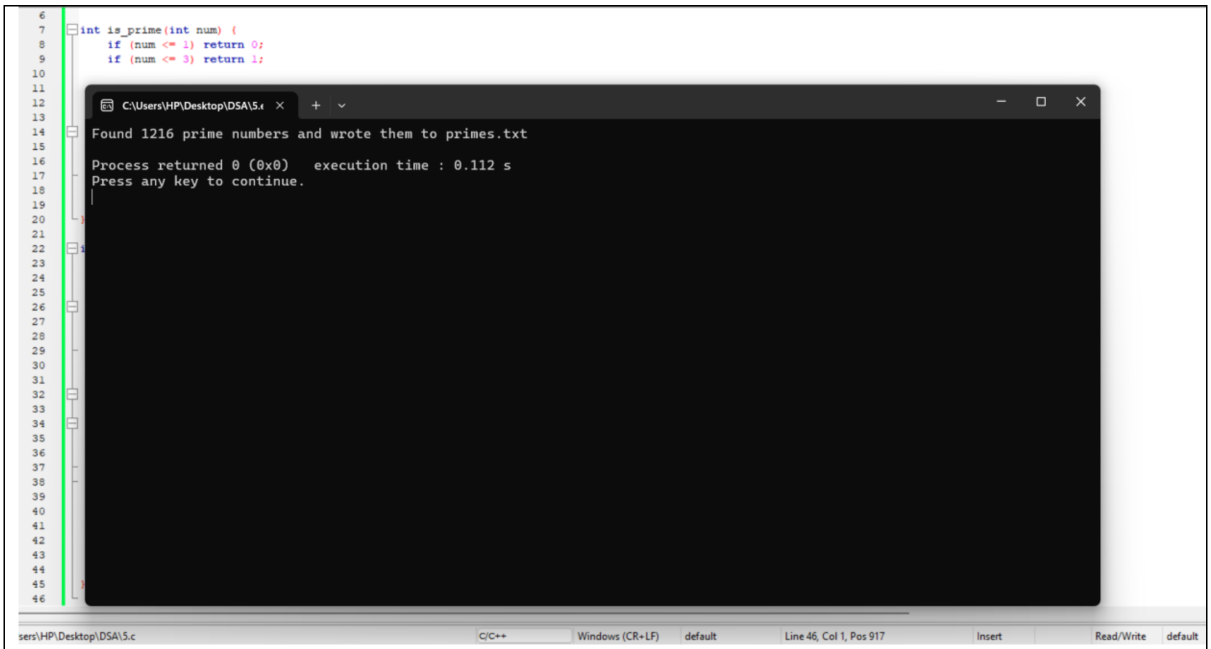


```
C:\Users\HP\Desktop\DSA\5.e

Factorials written to 'factorials.txt'

Process returned 0 (0x0)   execution time : 0.060 s
Press any key to continue.
```

3.Prime number

Code

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

#define MAX_NUMBER 10000

int is_prime(int num) {
    if (num <= 1) return 0;
    if (num <= 3) return 1;
    if (num % 2 == 0 || num % 3 == 0) return 0;

    int i = 5;
    while (i * i <= num) {
        if (num % i == 0 || num % (i + 2) == 0) return 0;
        i += 6;
    }

    return 1;
}

int main() {
    srand(time(NULL));

    FILE *fp = fopen("primes.txt", "w");
    if (fp == NULL) {
        perror("Error opening file");
        exit(1);
    }

    int num_primes = 0;
    for (int i = 0; i < MAX_NUMBER; i++) {
        int random_num = rand() % MAX_NUMBER + 1;
        if (is_prime(random_num)) {
            fprintf(fp, "%d\n", random_num);
            num_primes++;
        }
    }

    fclose(fp);

    printf("Found %d prime numbers and wrote them to primes.txt\n", num_primes);

    return 0;

}
```



```
int is_prime(int num) {
    if (num <= 1) return 0;
    if (num <= 3) return 1;
```

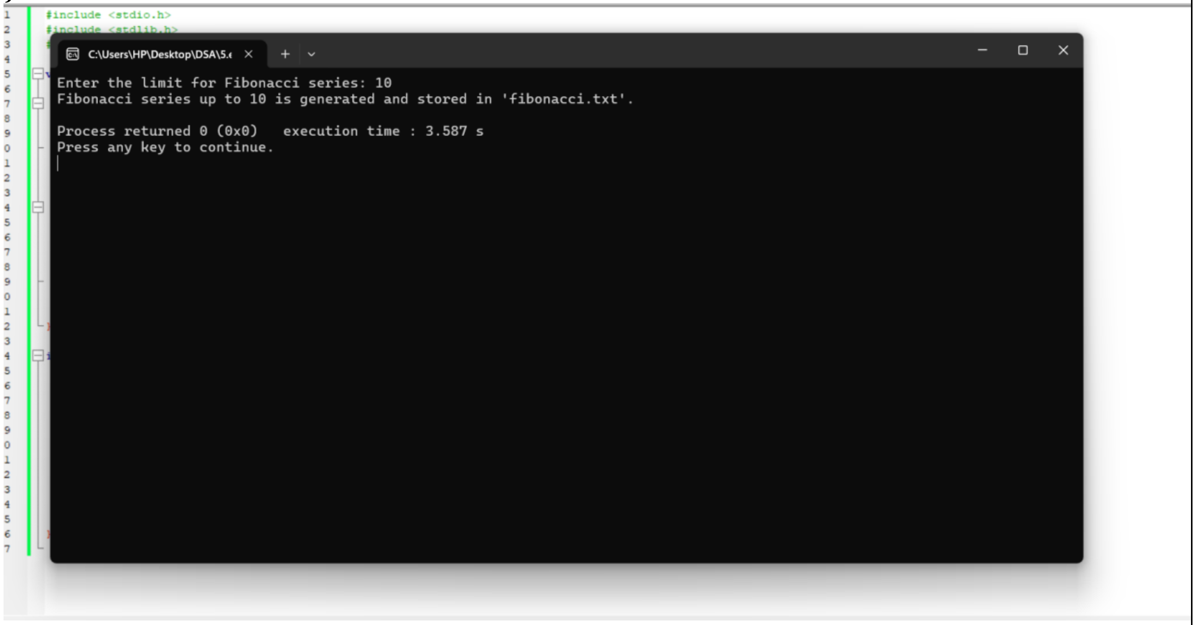
```
C:\Users\HP\Desktop\DSA\5.c
Found 1216 prime numbers and wrote them to primes.txt

Process returned 0 (0x0)   execution time : 0.112 s
Press any key to continue.
```

sers\HP\Desktop\DSA\5.c | C/C++ | Windows (CR+LF) | default | Line 46, Col 1, Pos 917 | Insert | Read/Write | default

4.Fibonacci Series

Code

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

void generateFibonacci(int limit, const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    srand(time(NULL));
    int prev = 0, current = 1, next;
    while (current <= limit) {
        fprintf(file, "%d\n", current);
        next = prev + current;
        prev = current;
        current = next;
    }

    fclose(file);
}

int main() {
    int limit;
    const char *filename = "fibonacci.txt";

    printf("Enter the limit for Fibonacci series: ");
    scanf("%d", &limit);

    generateFibonacci(limit, filename);

    printf("Fibonacci series up to %d is generated and stored in '%s'.\n", limit,
filename);

    return 0;
}
```

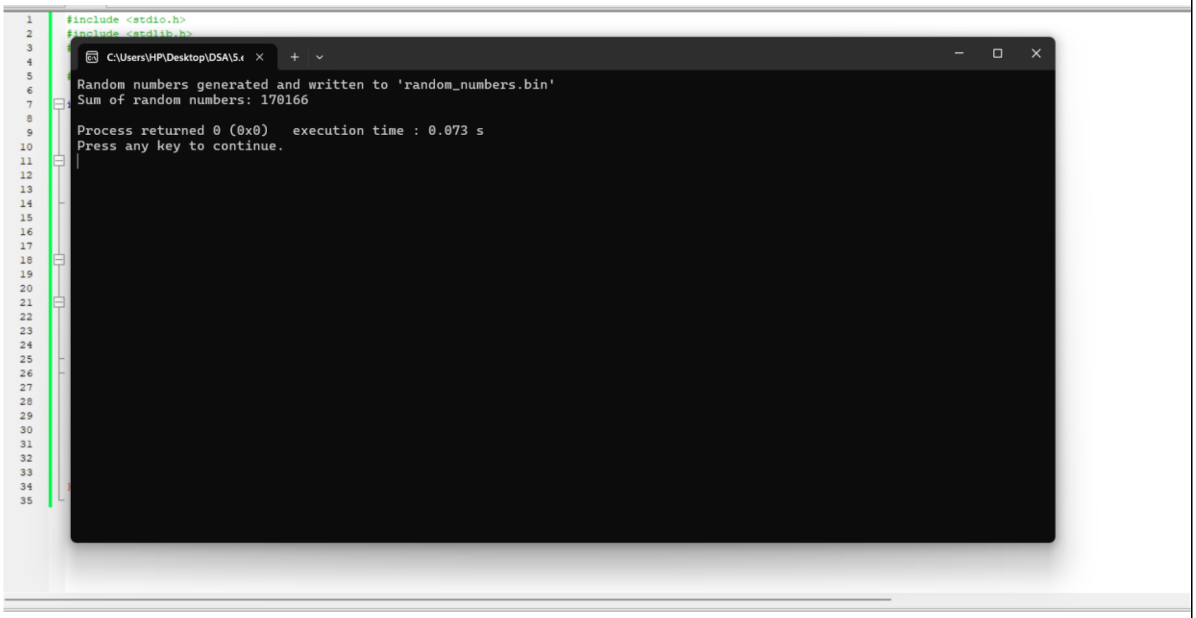


```
Enter the limit for Fibonacci series: 10
Fibonacci series up to 10 is generated and stored in 'fibonacci.txt'.

Process returned 0 (0x0)   execution time : 3.587 s
Press any key to continue.
```

5.Sum of digits

Code

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

#define NUM_RAND_NUMS 10

int main() {
    srand(time(NULL));

    FILE *fp = fopen("random_numbers.bin", "wb");
    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    int sum = 0;
    int num;
    for (int i = 0; i < NUM_RAND_NUMS; i++) {
        num = rand();
        sum += num;
        if (fwrite(&num, sizeof(int), 1, fp) != 1) {
            perror("Error writing to file");
            fclose(fp);
            return 1;
        }
    }

    fclose(fp);

    printf("Random numbers generated and written to 'random_numbers.bin'\n");
    printf("Sum of random numbers: %d\n", sum);

    return 0;
}
```



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

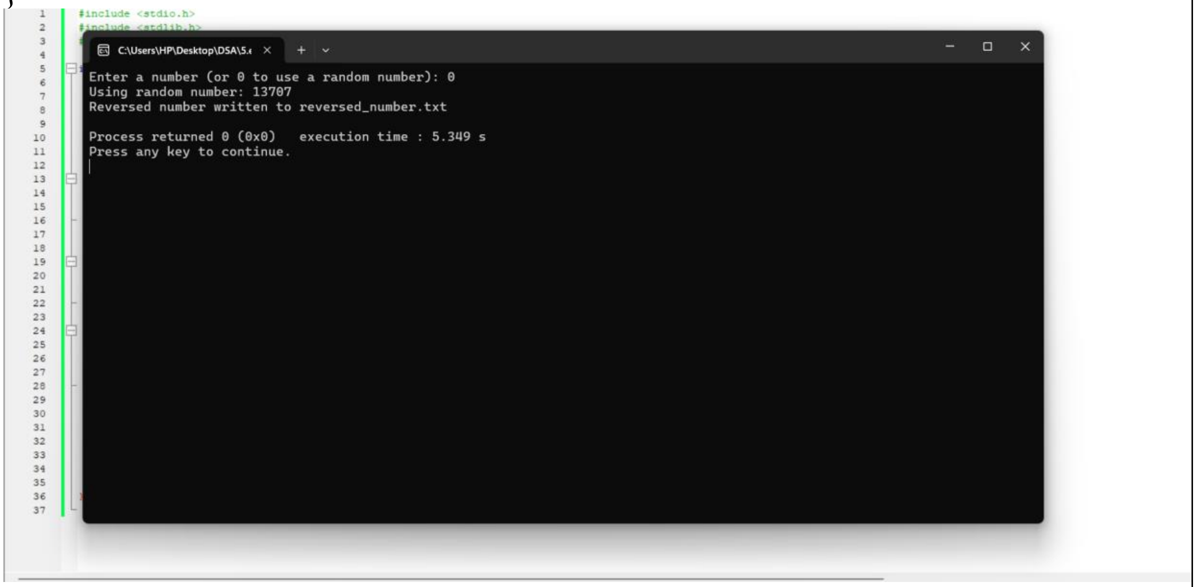
C:\Users\HP\Desktop\DSA\5.e

```
Random numbers generated and written to 'random_numbers.bin'
Sum of random numbers: 170166

Process returned 0 (0x0)   execution time : 0.073 s
Press any key to continue.
```

6. Reverse a Number

Code

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

int main() {
    int num, reversed = 0, digit;
    FILE *fp;
    srand(time(NULL));

    printf("Enter a number (or 0 to use a random number): ");
    scanf("%d", &num);
    if (num == 0) {
        num = rand() % 100000 + 1;
        printf("Using random number: %d\n", num);
    }

    fp = fopen("reversed_number.txt", "w");
    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    while (num != 0) {
        digit = num % 10;
        reversed = reversed * 10 + digit;
        num /= 10;
    }

    fprintf(fp, "%d\n", reversed);
    fclose(fp);

    printf("Reversed number written to reversed_number.txt\n");

    return 0;
}
```



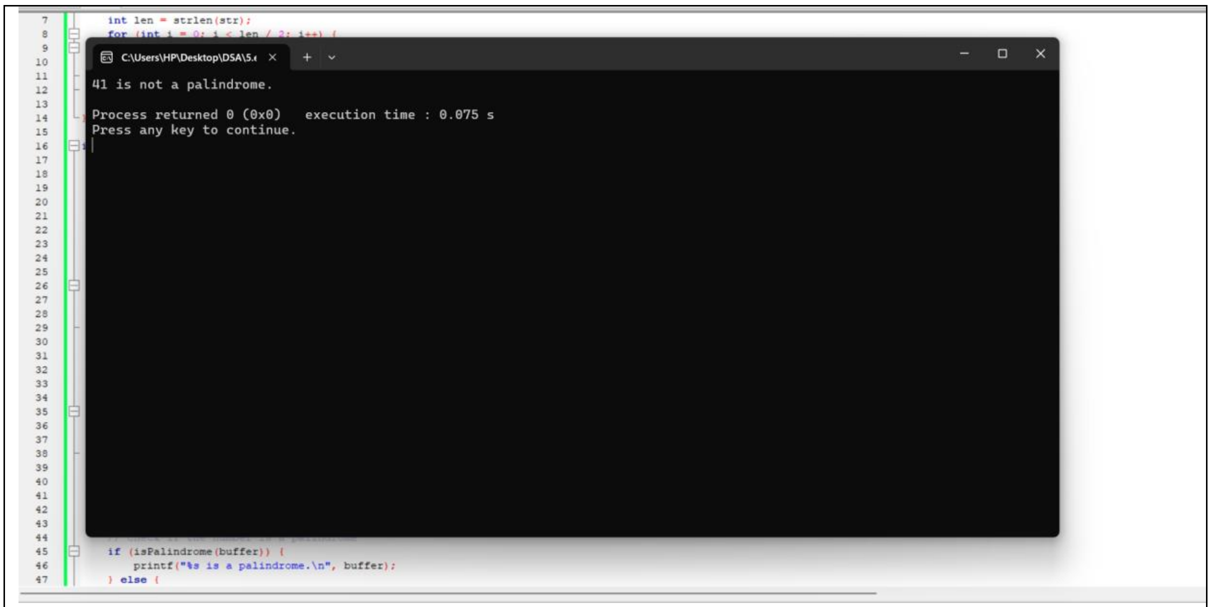
```
Enter a number (or 0 to use a random number): 0
Using random number: 13707
Reversed number written to reversed_number.txt

Process returned 0 (0x0)   execution time : 5.349 s
Press any key to continue.
```

7. Palindrome Check

Code

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

```
#include <string.h>

bool isPalindrome(const char *str) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {
        if (str[i] != str[len - i - 1]) {
            return false;
        }
    }
    return true;
}

int main() {
    // Generate a random number
    int num = rand() % 10000;

    // Convert the number to a string
    char num_str[20];
    sprintf(num_str, "%d", num);

    // Write the number to a file
    FILE *file = fopen("number.txt", "w");
    if (file == NULL) {
        perror("Error opening file for writing");
        return 1;
    }
    fprintf(file, "%s", num_str);
    fclose(file);

    // Read the number from the file
    file = fopen("number.txt", "r");
    if (file == NULL) {
        perror("Error opening file for reading");
        return 1;
    }

    char buffer[20];
    fscanf(file, "%s", buffer);
    fclose(file);

    // Check if the number is a palindrome
    if (isPalindrome(buffer)) {
        printf("%s is a palindrome.\n", buffer);
    } else {
        printf("%s is not a palindrome.\n", buffer);
    }

    return 0;
}
```



```
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {
```

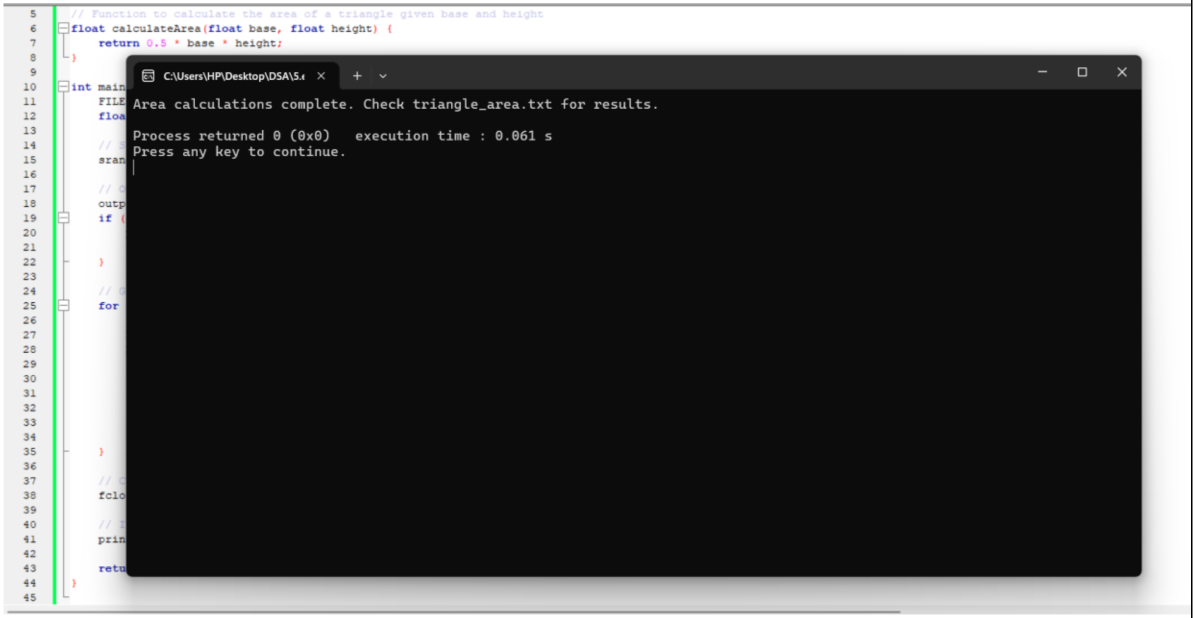
```
C:\Users\HP\Desktop\DSA\5.
41 is not a palindrome.

Process returned 0 (0x0)   execution time : 0.075 s
Press any key to continue.
```

```
    if (isPalindrome(buffer)) {
        printf("%s is a palindrome.\n", buffer);
    } else {
```

8. Area of Shapes

Code

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

// Function to calculate the area of a triangle given base and height
float calculateArea(float base, float height) {
    return 0.5 * base * height;
}

int main() {
    FILE *outputFile;
    float base, height, area;

    // Seed the random number generator with the current time
    srand(time(NULL));

    // Open the output file for writing
    outputFile = fopen("triangle_area.txt", "w");
    if (outputFile == NULL) {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }

    // Generate 10 random triangles and calculate their areas
    for (int i = 0; i < 10; i++) {
        // Generate random values for base and height (up to 99)
        base = (float)(rand() % 100);
        height = (float)(rand() % 100);
        // Calculate the area of the triangle
        area = calculateArea(base, height);

        // Write the triangle's information to the output file
        fprintf(outputFile, "Base: %.2f, Height: %.2f, Area: %.2f\n", base, height, area);
    }

    // Close the output file
    fclose(outputFile);

    // Inform the user that the area calculations are complete
    printf("Area calculations complete. Check triangle_area.txt for results.\n");

    return 0;
}
```



```
// Function to calculate the area of a triangle given base and height
float calculateArea(float base, float height) {
    return 0.5 * base * height;
}
```

```
Area calculations complete. Check triangle_area.txt for results.

Process returned 0 (0x0)   execution time : 0.061 s
Press any key to continue.
```

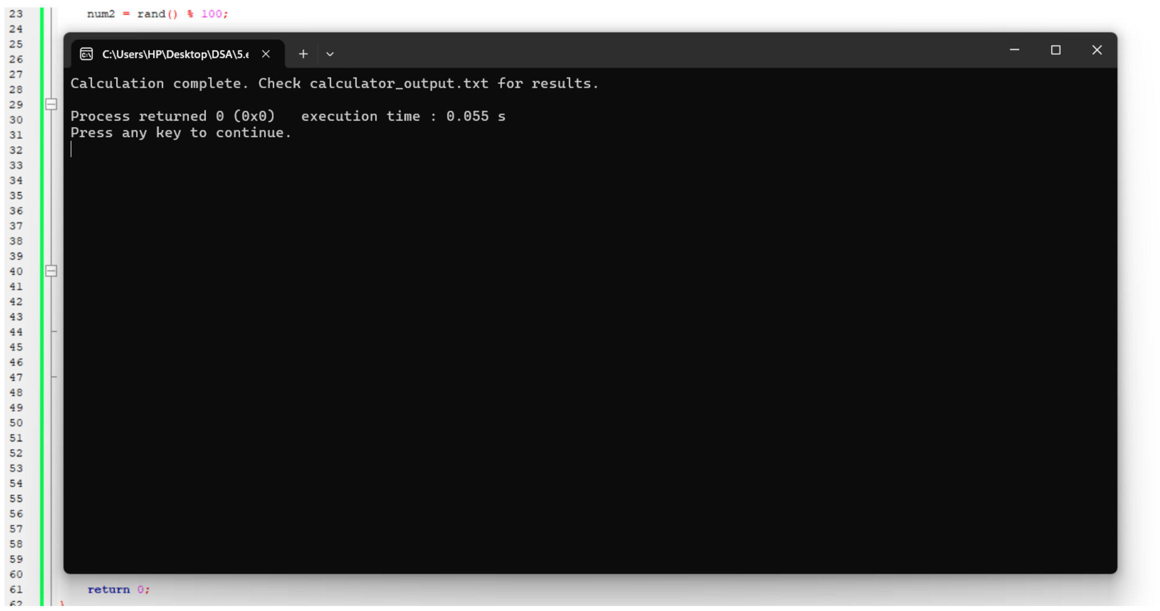
9. . Simple Calculator:

Code

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

int main() {
    FILE *outputFile;
    char operations[] = {'+', '-', '*', '/'};
    int num1, num2, result;
    char operation;

    // Seed the random number generator with the current time
```

```
srand(time(NULL));

// Open the output file for writing
outputFile = fopen("calculator_output.txt", "w");
if (outputFile == NULL) {
    perror("Error opening output file");
    exit(EXIT_FAILURE);
}

// Generate two random numbers
num1 = rand() % 100;
num2 = rand() % 100;

// Select a random operation
operation = operations[rand() % 4];

// Perform the selected operation
switch (operation) {
    case '+':
        result = num1 + num2;
        break;
    case '-':
        result = num1 - num2;
        break;
    case '*':
        result = num1 * num2;
        break;
    case '/':
        if (num2 == 0) {
            fprintf(outputFile, "Error: Division by zero\n");
            fclose(outputFile);
            exit(EXIT_FAILURE);
        }
        result = num1 / num2;
        break;
}

// Write the numbers, operation, and result to the output file
fprintf(outputFile, "Number 1: %d\n", num1);
fprintf(outputFile, "Number 2: %d\n", num2);
fprintf(outputFile, "Operation: %c\n", operation);
fprintf(outputFile, "Result: %d\n", result);

// Close the output file
fclose(outputFile);

// Inform the user that the calculation is complete
printf("Calculation complete. Check calculator_output.txt for results.\n");

return 0;
```

}



```
num2 = rand() % 100;
```

```
Calculation complete. Check calculator_output.txt for results.

Process returned 0 (0x0)   execution time : 0.055 s
Press any key to continue.
```

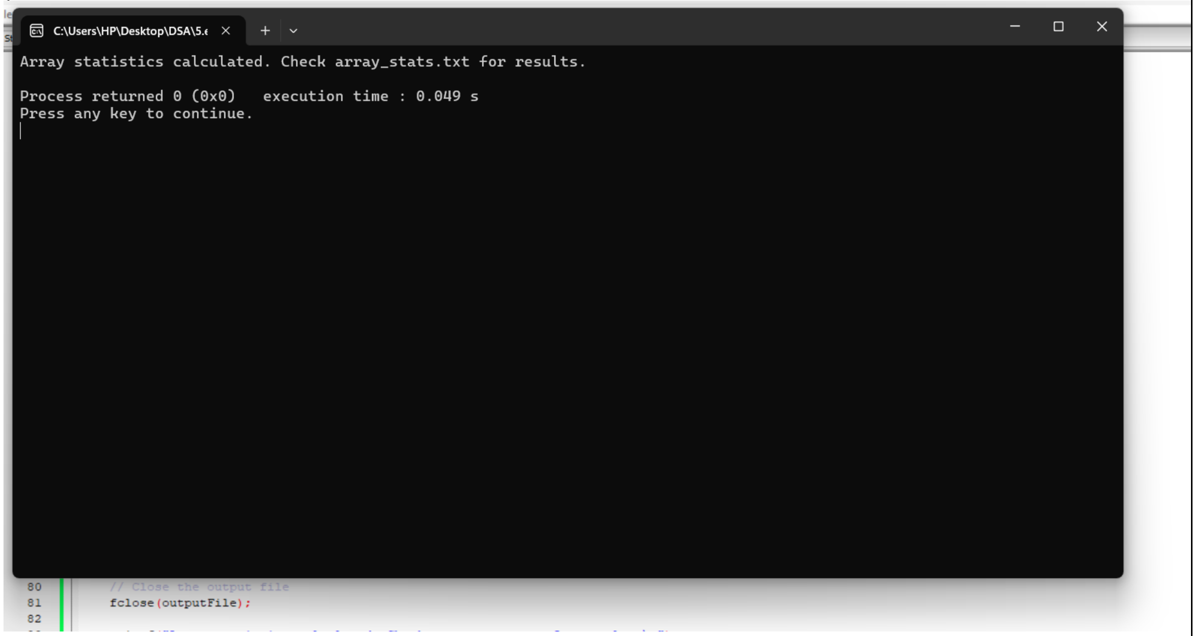
```
return 0;
```

10. Array Operations

Code

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

#define ARRAY_SIZE 10

// Function to generate random numbers and fill an array
void generateRandomNumbers(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 100;
    }
}

// Function to find the largest element in an array
int findLargest(int arr[], int size) {
    int largest = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > largest) {
            largest = arr[i];
        }
    }
    return largest;
}
```

```
// Function to find the smallest element in an array
int findSmallest(int arr[], int size) {
    int smallest = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] < smallest) {
            smallest = arr[i];
        }
    }
    return smallest;
}

// Function to find the sum of elements in an array
int findSum(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;
}

// Function to calculate the average of elements in an array
double findAverage(int sum, int size) {
    return (double) sum / size;
}

int main() {
    int numbers[ARRAY_SIZE];
    int largest, smallest, sum;
    double average;
    FILE *outputFile;

    srand(time(NULL));  // Seed the random number generator

    // Generate random numbers and fill the array
    generateRandomNumbers(numbers, ARRAY_SIZE);

    // Calculate array statistics
    largest = findLargest(numbers, ARRAY_SIZE);
    smallest = findSmallest(numbers, ARRAY_SIZE);
    sum = findSum(numbers, ARRAY_SIZE);
    average = findAverage(sum, ARRAY_SIZE);

    // Open the output file for writing
    outputFile = fopen("array_stats.txt", "w");
    if (outputFile == NULL) {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }

    // Write array statistics to the output file
    fprintf(outputFile, "Largest: %d\n", largest);
    fprintf(outputFile, "Smallest: %d\n", smallest);
    fprintf(outputFile, "Sum: %d\n", sum);
    fprintf(outputFile, "Average: %.2f\n", average);

    // Close the output file
    fclose(outputFile);

    printf("Array statistics calculated. Check array_stats.txt for results.\n");

    return 0;
}
```

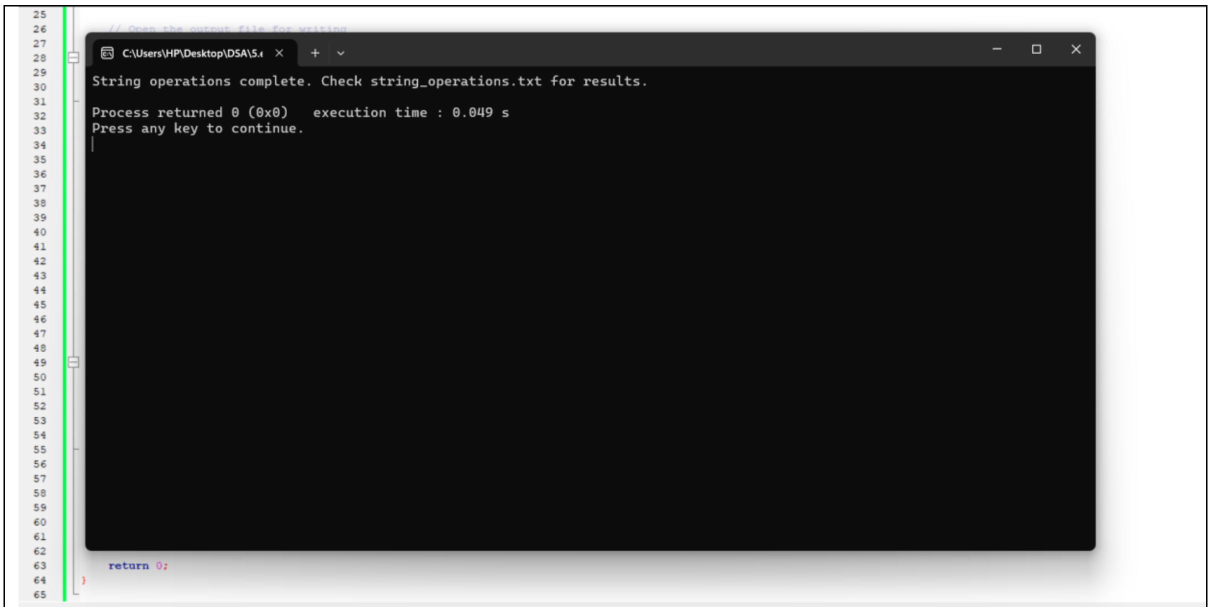


```
Array statistics calculated. Check array_stats.txt for results.

Process returned 0 (0x0)   execution time : 0.049 s
Press any key to continue.
```

11. String Operations

Code

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

#define MAX_STRING_LENGTH 100

// Function to generate a random string
void generateRandomString(char *str, int length) {
    const char charset[] =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

    for (int i = 0; i < length - 1; i++) {
        int index = rand() % (sizeof(charset) - 1);
        str[i] = charset[index];
    }
    str[length - 1] = '\0';  // Null-terminate the string
}

int main() {
    FILE *outputFile;
    char string1[MAX_STRING_LENGTH], string2[MAX_STRING_LENGTH],
concatenated[MAX_STRING_LENGTH * 2];

    // Seed the random number generator
    srand(time(NULL));

    // Open the output file for writing
    outputFile = fopen("string_operations.txt", "w");
    if (outputFile == NULL) {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }

    // Generate two random strings
    generateRandomString(string1, MAX_STRING_LENGTH);
    generateRandomString(string2, MAX_STRING_LENGTH);

    // Concatenate the two strings
    strcpy(concatenated, string1);
    strcat(concatenated, string2);
    fprintf(outputFile, "Concatenated string: %s\n", concatenated);

    // Copy the first string
    char copied[MAX_STRING_LENGTH];
    strcpy(copied, string1);
    fprintf(outputFile, "Copied string: %s\n", copied);

    // Compare the two strings
    int comparisonResult = strcmp(string1, string2);
    if (comparisonResult < 0) {
        fprintf(outputFile, "%s is lexicographically smaller than %s\n", string1, string2);
    } else if (comparisonResult > 0) {
        fprintf(outputFile, "%s is lexicographically greater than %s\n", string1, string2);
    } else {
        fprintf(outputFile, "%s is lexicographically equal to %s\n", string1, string2);
    }

    // Close the output file
    fclose(outputFile);

    // Inform the user that string operations are complete
    printf("String operations complete. Check string_operations.txt for results.\n");

    return 0;
}
```



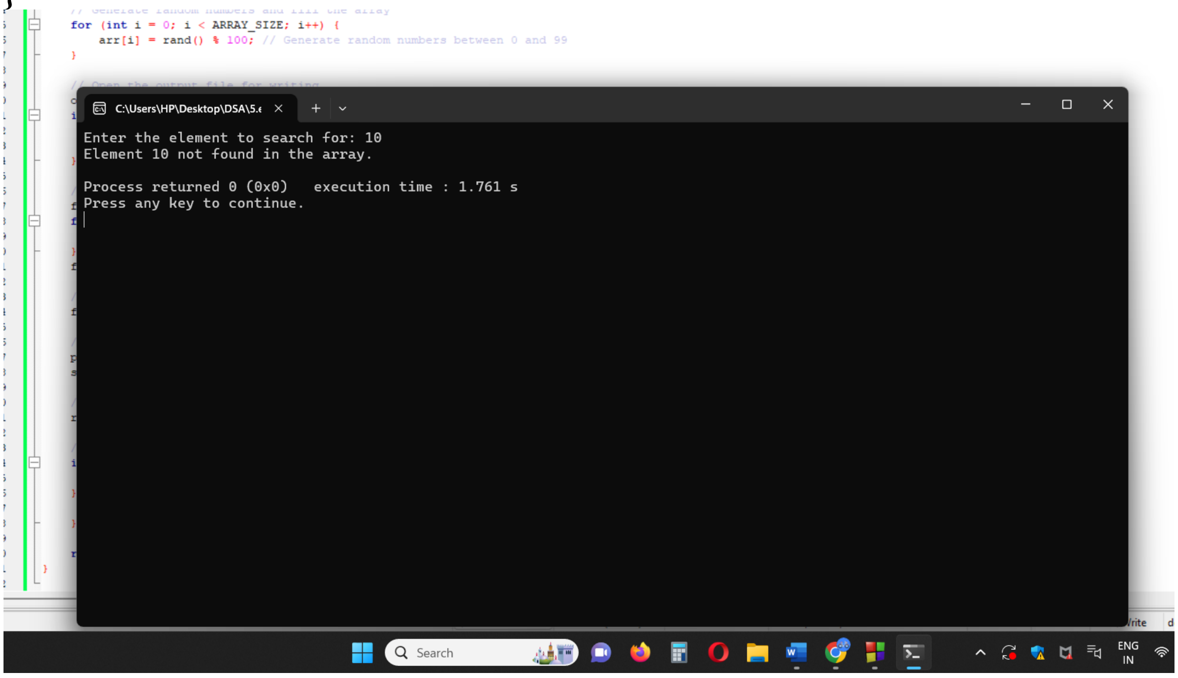
```
String operations complete. Check string_operations.txt for results.

Process returned 0 (0x0)   execution time : 0.049 s
Press any key to continue.
```

```
    return 0;
}
```

12.Linear search

Code

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

#define ARRAY_SIZE 10

// Function to perform linear search in an array
int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i; // Return the index if key is found
        }
    }
    return -1; // Return -1 if key is not found
}

int main() {
    FILE *outputFile;
    int arr[ARRAY_SIZE];
    int searchKey, result;
    srand(time(NULL)); // Seed the random number generator

    // Generate random numbers and fill the array
    for (int i = 0; i < ARRAY_SIZE; i++) {
        arr[i] = rand() % 100; // Generate random numbers between 0 and 99
    }

    // Open the output file for writing
    outputFile = fopen("array_elements.txt", "w");
    if (outputFile == NULL) {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }

    // Write the array elements to the output file
    fprintf(outputFile, "Array elements:\n");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        fprintf(outputFile, "%d ", arr[i]);
    }
    fprintf(outputFile, "\n");

    // Close the output file
    fclose(outputFile);

    // Prompt the user to enter the element to search for
    printf("Enter the element to search for: ");
    scanf("%d", &searchKey);

    // Perform linear search
    result = linearSearch(arr, ARRAY_SIZE, searchKey);

    // Print the result
    if (result != -1) {
        printf("Element %d found at index %d.\n", searchKey, result);
    } else {
        printf("Element %d not found in the array.\n", searchKey);
    }

    return 0;
}
```



```
    for (int i = 0; i < ARRAY_SIZE; i++) {
        arr[i] = rand() % 100; // Generate random numbers between 0 and 99
    }
```

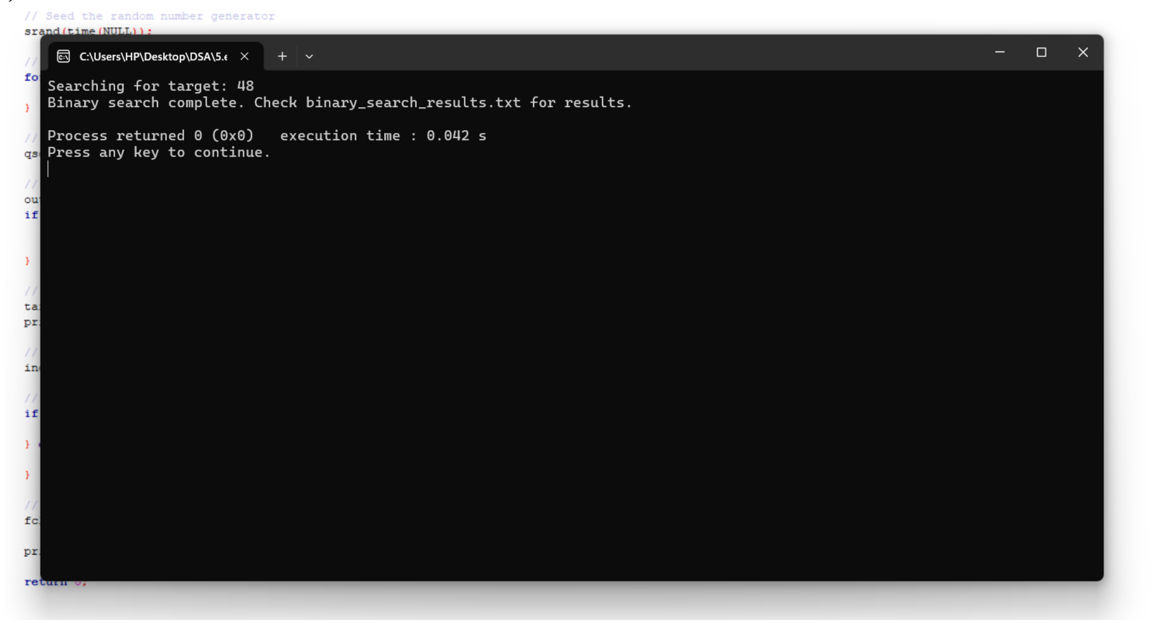
C:\Users\HP\Desktop\DSA\5.e

```
Enter the element to search for: 10
Element 10 not found in the array.

Process returned 0 (0x0)   execution time : 1.761 s
Press any key to continue.
```

13.Binary search

code

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

#define ARRAY_SIZE 10

// Comparison function for qsort
int cmpfunc(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}

// Binary search function
int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid; // Return the index if target is found
        }
        if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1; // Return -1 if target is not found
}

int main() {
    int arr[ARRAY_SIZE];
    int target, index;
    FILE *outputFile;

    // Seed the random number generator
    srand(time(NULL));

    // Generate random numbers and fill the array
    for (int i = 0; i < ARRAY_SIZE; i++) {
        arr[i] = rand() % 100; // Generate random numbers between 0 and 99
    }

    // Sort the array
    qsort(arr, ARRAY_SIZE, sizeof(int), cmpfunc);
    // Open the output file for writing
    outputFile = fopen("binary_search_results.txt", "w");
    if (outputFile == NULL) {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }

    // Choose a random target to search for
    target = rand() % 100; // Generate a random target between 0 and 99
    printf("Searching for target: %d\n", target);

    // Perform binary search
    index = binarySearch(arr, ARRAY_SIZE, target);

    // Write the search result to the output file
    if (index != -1) {
        fprintf(outputFile, "Target %d found at index %d in the sorted array.\n", target, index);
    } else {
        fprintf(outputFile, "Target %d not found in the sorted array.\n", target);
    }

    // Close the output file
    fclose(outputFile);

    printf("Binary search complete. Check binary_search_results.txt for results.\n");

    return 0;
}
```



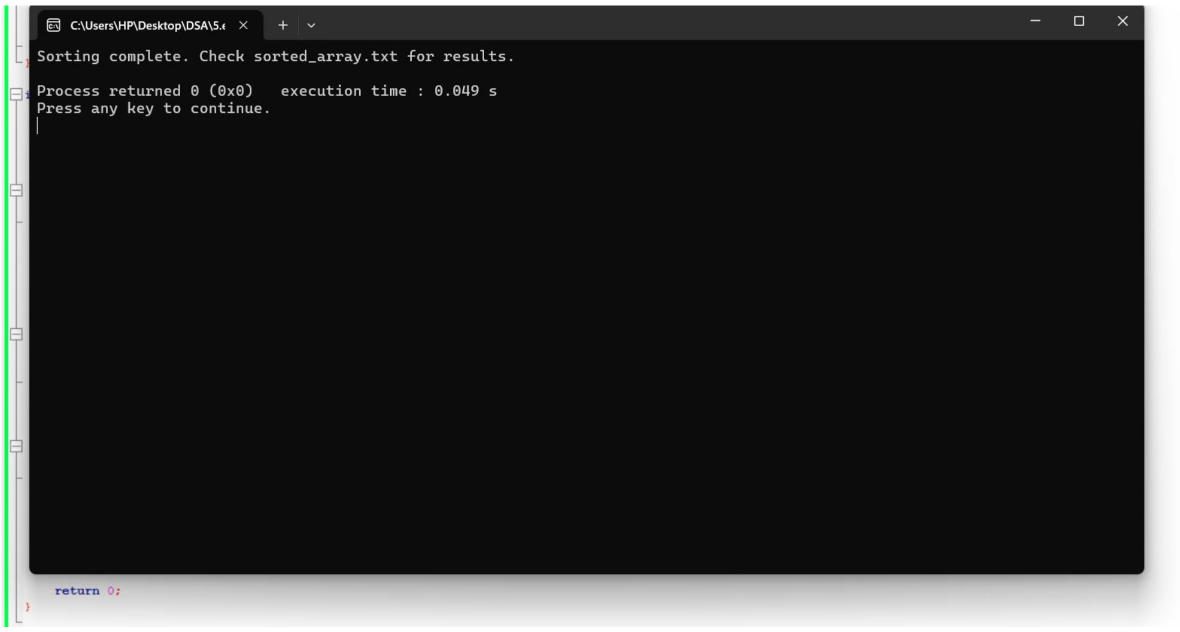
```
// Seed the random number generator
srand(time(NULL));
```

```
Searching for target: 48
Binary search complete. Check binary_search_results.txt for results.

Process returned 0 (0x0)   execution time : 0.042 s
Press any key to continue.
```

14. Selection Sort

Code

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

#define ARRAY_SIZE 10

// Function to perform selection sort
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }

        // Swap arr[i] with the smallest element
        int temp = arr[i];
        arr[i] = arr[min_index];
        arr[min_index] = temp;
    }
}

int main() {
    FILE *outputFile;
    int arr[ARRAY_SIZE];
    srand(time(NULL));

    // Generate random numbers and fill the array
    for (int i = 0; i < ARRAY_SIZE; i++) {
        arr[i] = rand() % 100;
    }

    // Perform selection sort
    selectionSort(arr, ARRAY_SIZE);

    // Open the output file for writing
    outputFile = fopen("sorted_array.txt", "w");
    if (outputFile == NULL) {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }

    // Write the sorted array to the output file
    fprintf(outputFile, "Sorted Array:\n");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        fprintf(outputFile, "%d\n", arr[i]);
    }

    // Close the output file
    fclose(outputFile);
    printf("Sorting complete. Check sorted_array.txt for results.\n");

    return 0;

}
```



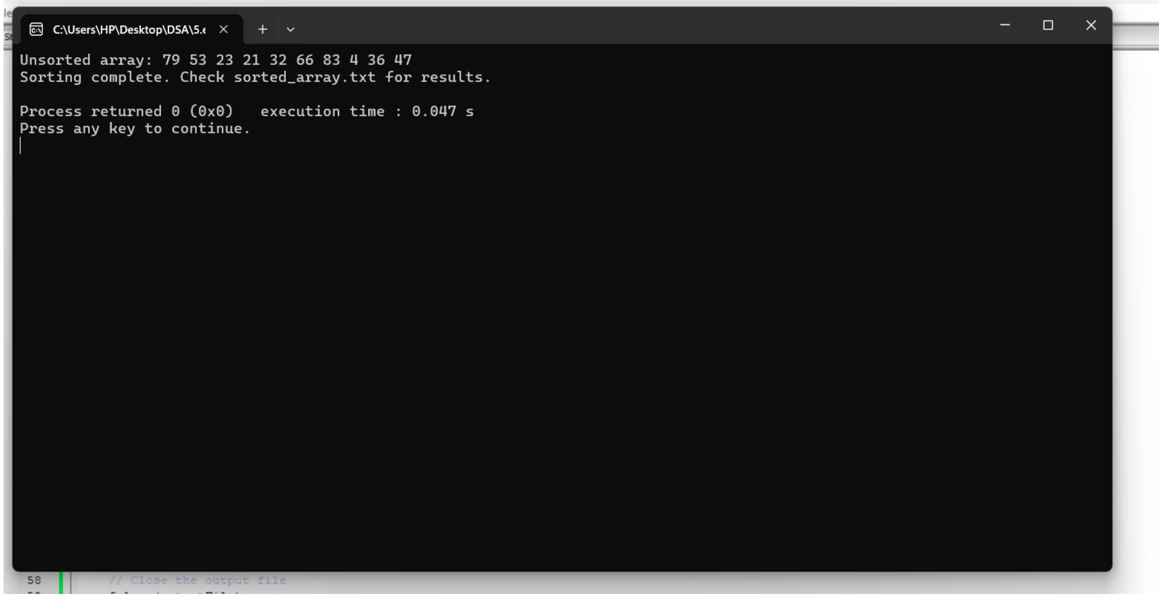
```
Sorting complete. Check sorted_array.txt for results.

Process returned 0 (0x0)   execution time : 0.049 s
Press any key to continue.
```

15.Bubble sort

Code

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

#define ARRAY_SIZE 10

// Function to perform bubble sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] with arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```
// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    FILE *outputFile;
    int n = ARRAY_SIZE; // Number of elements in the array
    int arr[n];

    srand(time(NULL)); // Seed the random number generator
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100; // Generate random numbers between 0 and 99
    }

    printf("Unsorted array: ");
    printArray(arr, n);

    // Perform bubble sort
    bubbleSort(arr, n);

    // Open the output file for writing
    outputFile = fopen("sorted_array.txt", "w");
    if (outputFile == NULL) {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }

    // Write the sorted array to the output file
    fprintf(outputFile, "Sorted array: ");
    for (int i = 0; i < n; i++) {
        fprintf(outputFile, "%d ", arr[i]);
    }

    // Close the output file
    fclose(outputFile);

    printf("Sorting complete. Check sorted_array.txt for results.\n");

    return 0;
}
```



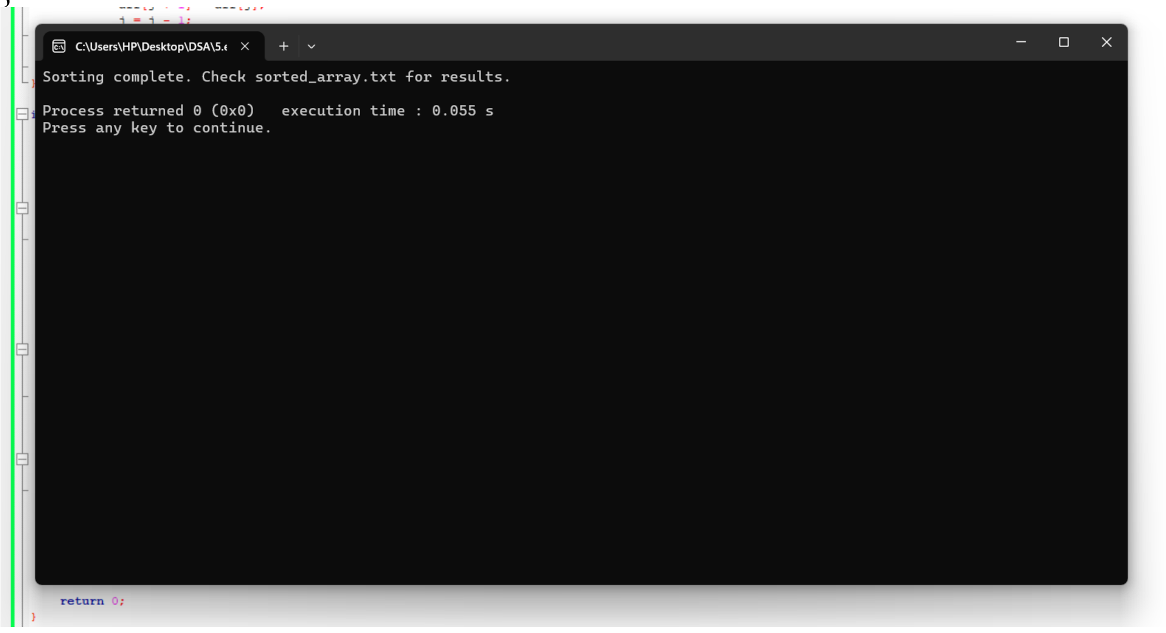
```
Unsorted array: 79 53 23 21 32 66 83 4 36 47
Sorting complete. Check sorted_array.txt for results.

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

16.Insertion sort

Code

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

#define ARRAY_SIZE 10

// Function to perform insertion sort
void insertionSort(int arr[], int n) {
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}

int main() {
   FILE *outputFile;
   int arr[ARRAY_SIZE]; // Array to store random numbers
   srand(time(NULL)); // Seed the random number generator

   // Generate random numbers and fill the array
   for (int i = 0; i < ARRAY_SIZE; i++) {
      arr[i] = rand() % 100; // Generate random numbers between 0 and 99
```

```
    }

    // Perform insertion sort
    insertionSort(arr, ARRAY_SIZE);

    // Open the output file for writing
    outputFile = fopen("sorted_array.txt", "w");
    if (outputFile == NULL) {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }

    // Write the sorted array to the output file
    fprintf(outputFile, "Sorted Array:\n");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        fprintf(outputFile, "%d ", arr[i]);
    }

    // Close the output file
    fclose(outputFile);

    printf("Sorting complete. Check sorted_array.txt for results.\n");

    return 0;
}
```



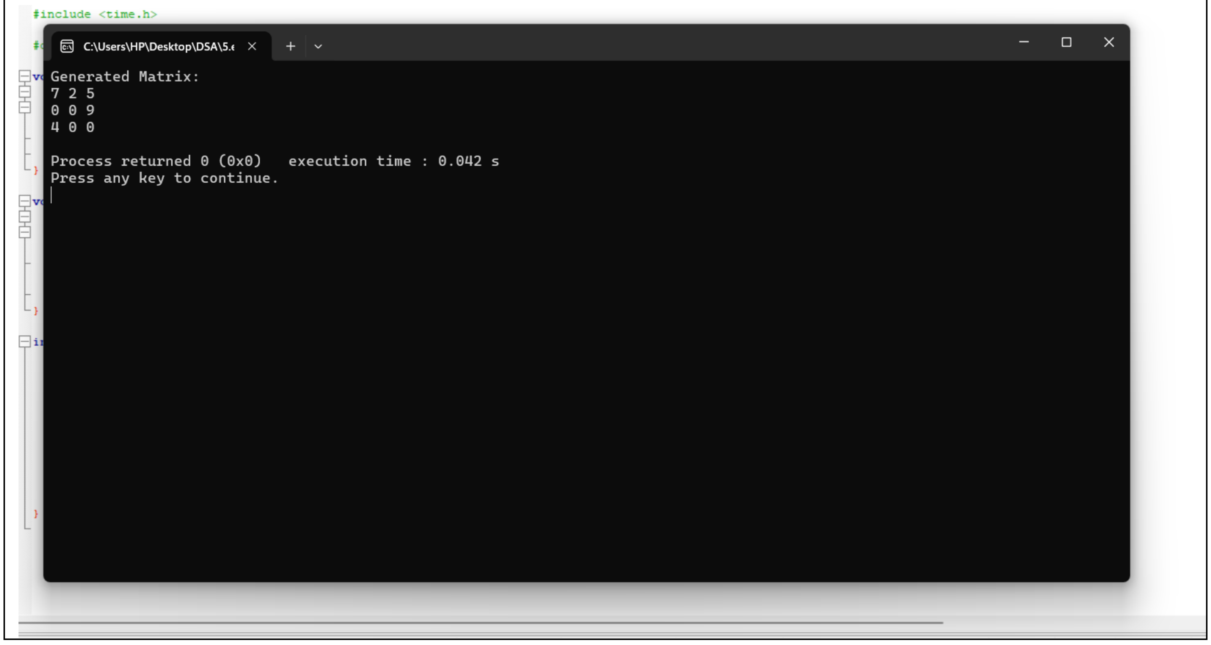
```
Sorting complete. Check sorted_array.txt for results.

Process returned 0 (0x0)   execution time : 0.055 s
Press any key to continue.
```

17. Matrix Operations

Code

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

```
#include <time.h>

#define SIZE 3

void generateRandomMatrix(int matrix[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            matrix[i][j] = rand() % 10;
        }
    }
}

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

int main() {
    srand(time(NULL));

    int matrix[SIZE][SIZE];

    generateRandomMatrix(matrix);

    printf("Generated Matrix:\n");
    printMatrix(matrix);

    return 0;
}
```

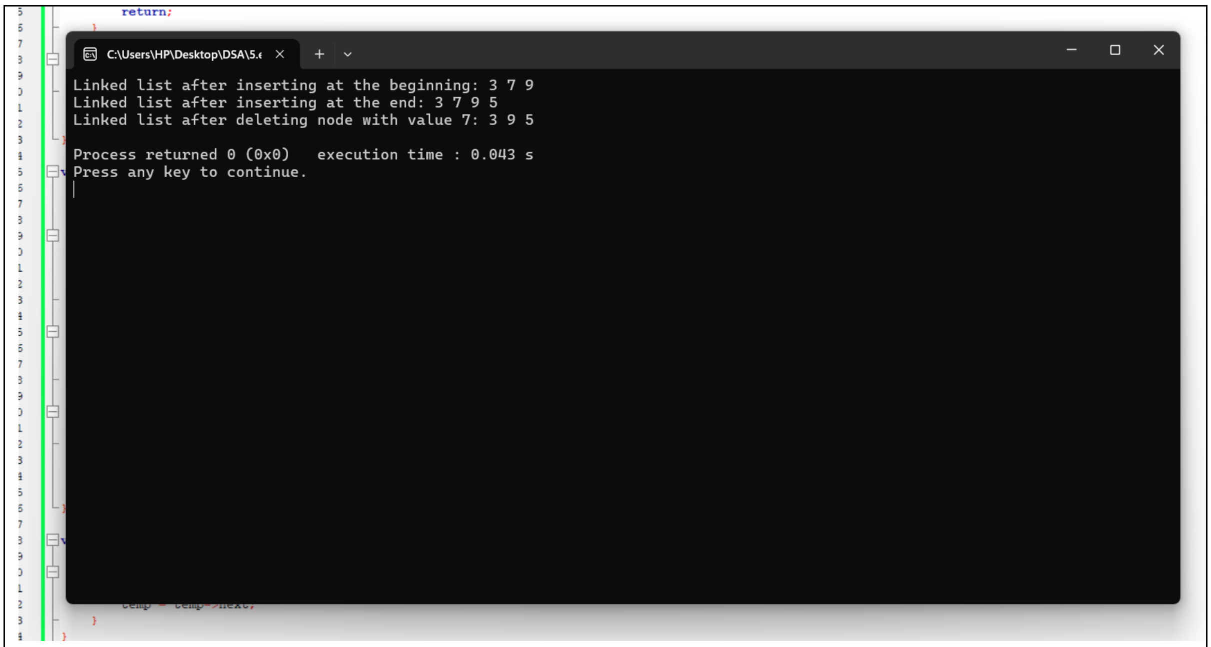


```
Generated Matrix:
7 2 5
0 0 9
4 0 0

Process returned 0 (0x0)   execution time : 0.042 s
Press any key to continue.
```

Part- C

- **Basic operations**

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

struct Node {
    int data;
    struct Node* next;
};

void insertAtBeginning(struct Node** head, int newData);
void insertAtEnd(struct Node** head, int newData);
void deleteNode(struct Node** head, int key);
void printList(struct Node* head);

int main() {
    struct Node* head = NULL;

    insertAtBeginning(&head, 9);
    insertAtBeginning(&head, 7);
    insertAtBeginning(&head, 3);

    printf("Linked list after inserting at the beginning: ");
    printList(head);
    printf("\n");

    insertAtEnd(&head, 5);

    printf("Linked list after inserting at the end: ");
    printList(head);
    printf("\n");

    deleteNode(&head, 7);

    printf("Linked list after deleting node with value 7: ");
    printList(head);
    printf("\n");

    return 0;
}

void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = *head;
    *head = newNode;
}

void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    struct Node* last = *head;
    newNode->data = newData;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    while (last->next != NULL) {
        last = last->next;
    }

    last->next = newNode;
}

void deleteNode(struct Node** head, int key) {
    struct Node* temp = *head;
    struct Node* prev = NULL;

    if (temp != NULL && temp->data == key) {
        *head = temp->next;
        free(temp);
        return;
    }

    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        return;
    }

    prev->next = temp->next;
    free(temp);
}

void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```

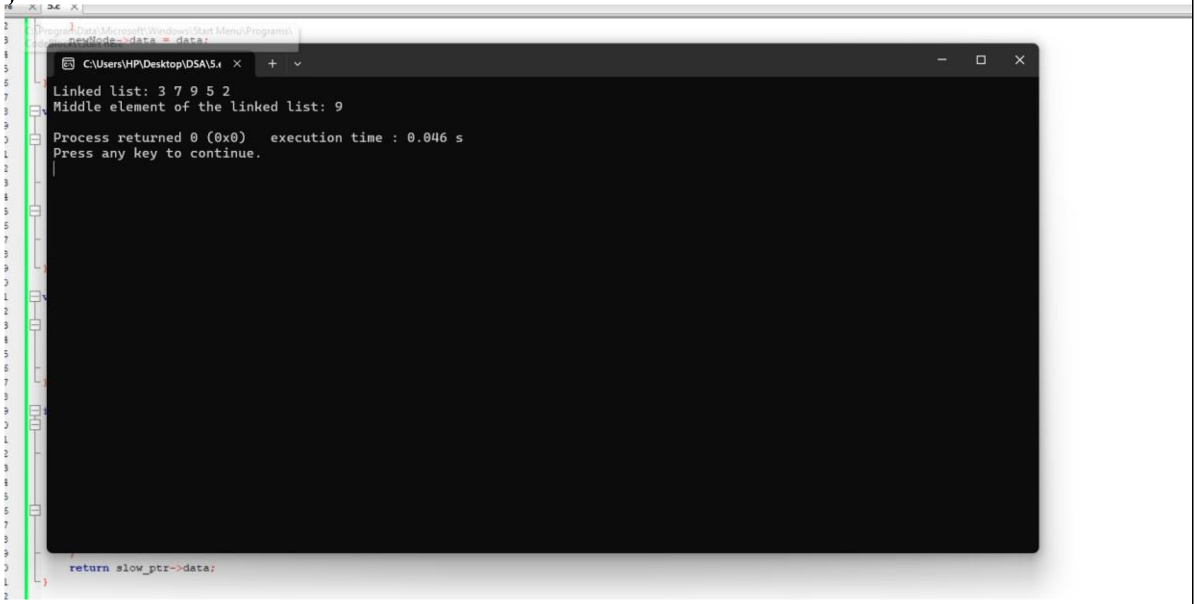


```
Linked list after inserting at the beginning: 3 7 9
Linked list after inserting at the end: 3 7 9 5
Linked list after deleting node with value 7: 3 9 5

Process returned 0 (0x0)   execution time : 0.043 s
Press any key to continue.
```

- **Finding middle element of linklist**

Code

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data);
void insertAtEnd(struct Node** head, int newData);
void printList(struct Node* head);
int findMiddle(struct Node* head);

int main() {
    struct Node* head = NULL;

    insertAtEnd(&head, 3);
    insertAtEnd(&head, 7);
    insertAtEnd(&head, 9);
    insertAtEnd(&head, 5);
    insertAtEnd(&head, 2);

    printf("Linked list: ");
    printList(head);
    printf("\n");
    int middle = findMiddle(head);
    if (middle != -1) {
        printf("Middle element of the linked list: %d\n", middle);
    } else {
        printf("Linked list is empty.\n");
    }

    return 0;
}

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = createNode(newData);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* last = *head;
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = newNode;
}

void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

int findMiddle(struct Node* head) {
    if (head == NULL) {
        return -1;
    }
    struct Node *slow_ptr = head;
    struct Node *fast_ptr = head;

    while (fast_ptr != NULL && fast_ptr->next != NULL) {
        fast_ptr = fast_ptr->next->next;
        slow_ptr = slow_ptr->next;
    }
    return slow_ptr->data;
}
```

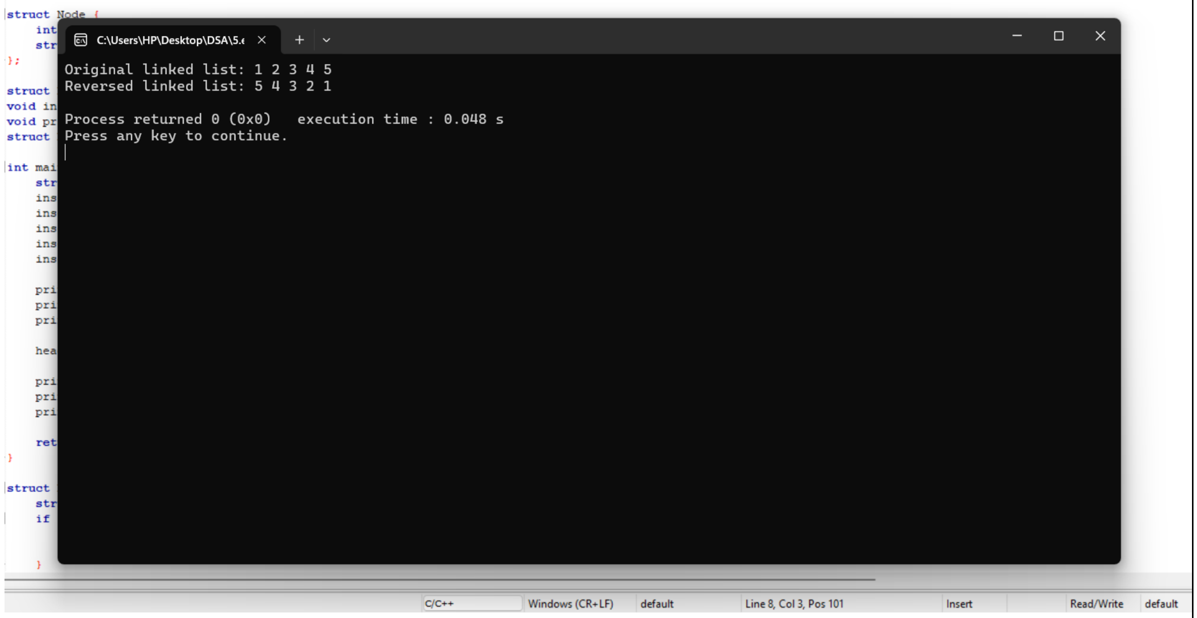


```
Linked list: 3 7 9 5 2
Middle element of the linked list: 9

Process returned 0 (0x0)   execution time : 0.046 s
Press any key to continue.
```

- **Reversing the link list**

Code

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data);
void insertAtEnd(struct Node** head, int newData);
void printList(struct Node* head);
struct Node* reverseList(struct Node* head);

int main() {
    struct Node* head = NULL;
    insertAtEnd(&head, 1);
    insertAtEnd(&head, 2);
    insertAtEnd(&head, 3);
    insertAtEnd(&head, 4);
    insertAtEnd(&head, 5);

    printf("Original linked list: ");
    printList(head);
    printf("\n");

    head = reverseList(head);
    printf("Reversed linked list: ");
    printList(head);
    printf("\n");

    return 0;
}

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = createNode(newData);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* last = *head;
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = newNode;
}

void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

struct Node* reverseList(struct Node* head) {
    struct Node* prev = NULL;
    struct Node* current = head;
    struct Node* next = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev;
}
```

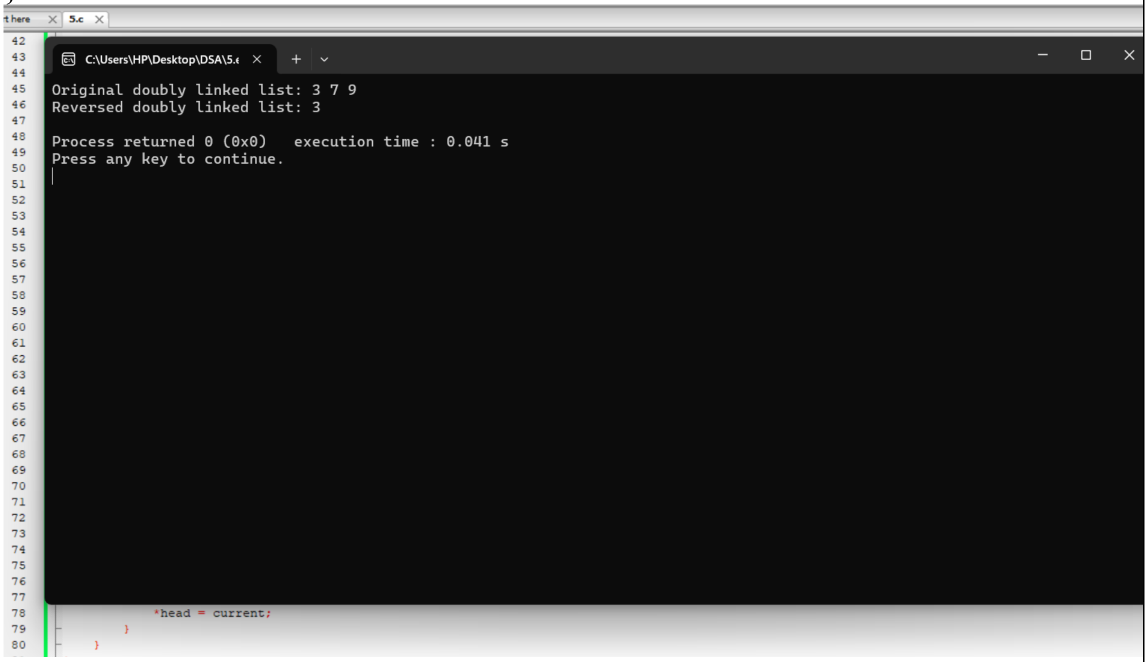


```
C:\Users\HP\Desktop\DSA\5.c
Original linked list: 1 2 3 4 5
Reversed linked list: 5 4 3 2 1

Process returned 0 (0x0)   execution time : 0.048 s
Press any key to continue.
```

C/C++ | Windows (CR+LF) | default | Line 8, Col 3, Pos 101 | Insert | Read/Write | default

- **Reverse doubly link list**

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

struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

struct Node* createNode(int data);
void insertAtEnd(struct Node** head, int newData);
void printList(struct Node* head);
void reverseList(struct Node** head);

int main() {
    struct Node* head = NULL;
    insertAtEnd(&head, 3);
    insertAtEnd(&head, 7);
    insertAtEnd(&head, 9);

    printf("Original doubly linked list: ");
    printList(head);
    printf("\n");

    reverseList(&head);

    printf("Reversed doubly linked list: ");
    printList(head);
    printf("\n");

    return 0;
}

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = createNode(newData);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* last = *head;
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = newNode;
    newNode->prev = last;
}

void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

void reverseList(struct Node** head) {
    struct Node* current = *head;
    struct Node* temp = NULL;

    while (current != NULL) {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
        if (current != NULL && current->prev == NULL) {
            *head = current;
        }
    }
}
```



```
Original doubly linked list: 3 7 9
Reversed doubly linked list: 3

Process returned 0 (0x0)   execution time : 0.041 s
Press any key to continue.
```

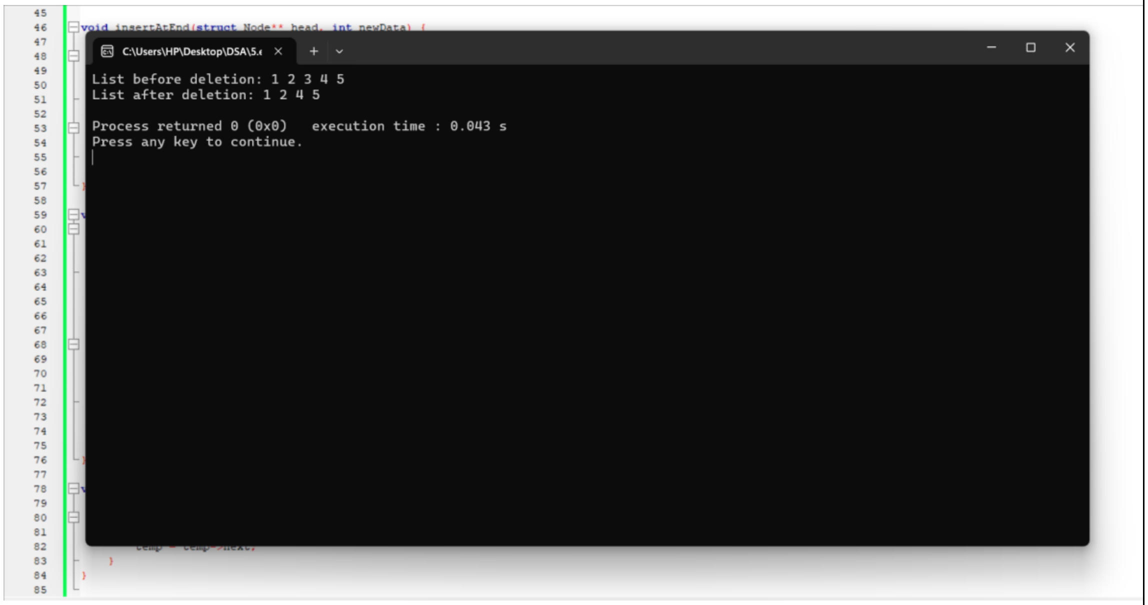
```
            *head = current;
        }
    }
```

- **Rotate the link list**

Code

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data);
void insertAtEnd(struct Node** head, int newData);
void deleteMiddleNode(struct Node** head);
void printList(struct Node* head);

int main() {
    struct Node* head = NULL;
    insertAtEnd(&head, 1);
    insertAtEnd(&head, 2);
    insertAtEnd(&head, 3);
    insertAtEnd(&head, 4);
    insertAtEnd(&head, 5);

    printf("List before deletion: ");
    printList(head);
    printf("\n");
    deleteMiddleNode(&head);

    printf("List after deletion: ");
    printList(head);
    printf("\n");

    return 0;
}

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = createNode(newData);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* last = *head;
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = newNode;
}

void deleteMiddleNode(struct Node** head) {
    if (*head == NULL || (*head)->next == NULL) {
        printf("List is empty or contains only one node.\n");
        return;
    }
    struct Node* slowPtr = *head;
    struct Node* fastPtr = *head;
    struct Node* prev = NULL;

    while (fastPtr != NULL && fastPtr->next != NULL) {
        fastPtr = fastPtr->next->next;
        prev = slowPtr;
        slowPtr = slowPtr->next;
    }

    prev->next = slowPtr->next;
    free(slowPtr);
}
```

```
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```

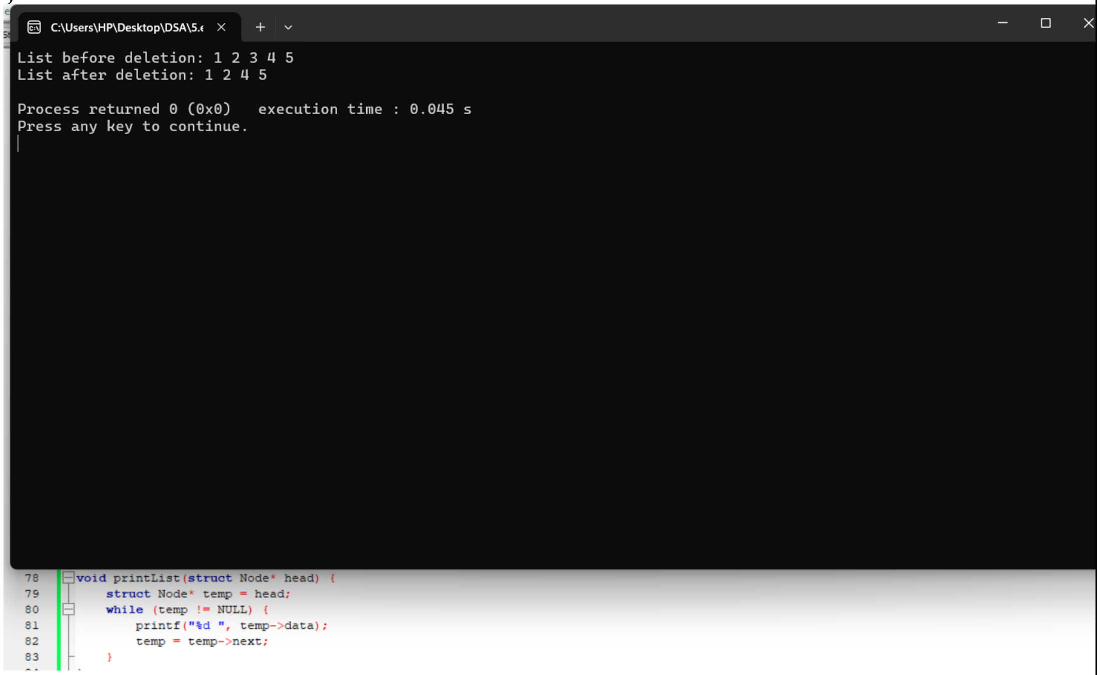


```
List before deletion: 1 2 3 4 5
List after deletion: 1 2 4 5

Process returned 0 (0x0)   execution time : 0.043 s
Press any key to continue.
```

- **Delete middle of link list**

  Code

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data);
void insertAtEnd(struct Node** head, int newData);
void deleteMiddleNode(struct Node** head);
void printList(struct Node* head);

int main() {
    struct Node* head = NULL;
    insertAtEnd(&head, 1);
    insertAtEnd(&head, 2);
    insertAtEnd(&head, 3);
    insertAtEnd(&head, 4);
    insertAtEnd(&head, 5);

    printf("List before deletion: ");
    printList(head);
    printf("\n");

    deleteMiddleNode(&head);

    printf("List after deletion: ");
    printList(head);
    printf("\n");

    return 0;
}

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = createNode(newData);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* last = *head;
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = newNode;
}

void deleteMiddleNode(struct Node** head) {
    if (*head == NULL || (*head)->next == NULL) {
        printf("List is empty or contains only one node.\n");
        return;
    }
    struct Node* slowPtr = *head;
    struct Node* fastPtr = *head;
    struct Node* prev = NULL;

    while (fastPtr != NULL && fastPtr->next != NULL) {
        fastPtr = fastPtr->next->next;
        prev = slowPtr;
        slowPtr = slowPtr->next;
    }

    prev->next = slowPtr->next;
    free(slowPtr);
}

void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```



C:\Users\HP\Desktop\DSA\5.e

```
List before deletion: 1 2 3 4 5
List after deletion: 1 2 4 5

Process returned 0 (0x0)   execution time : 0.045 s
Press any key to continue.
```

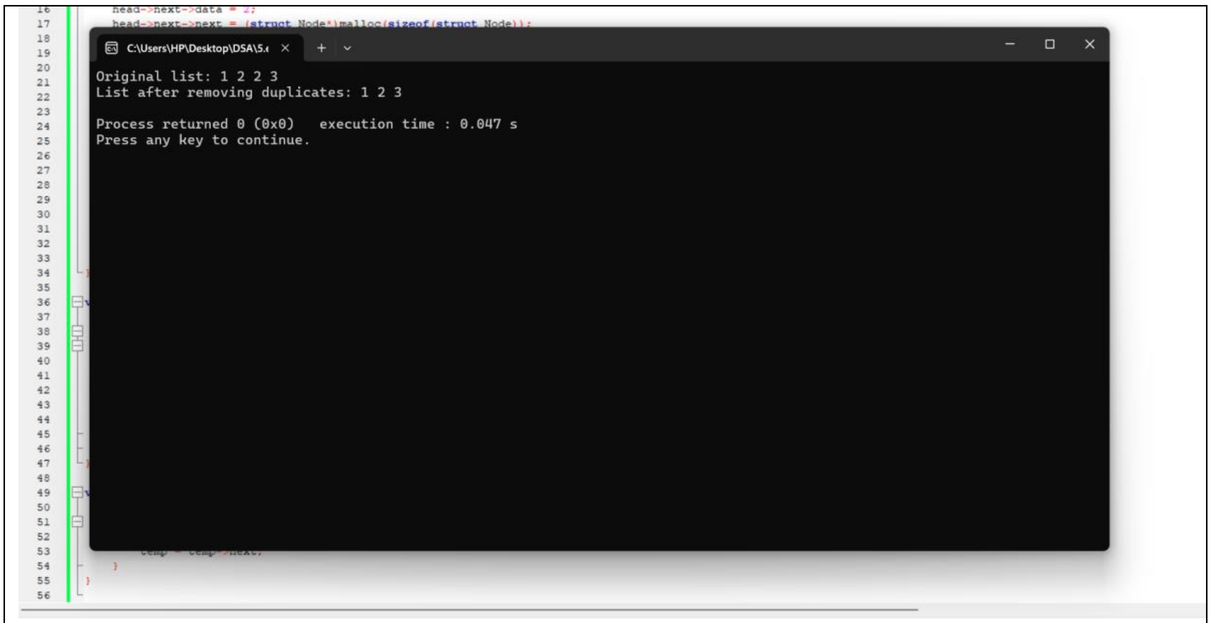
```
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
```

- **Remove duplicate element from sorted Linked List**

Code

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

struct Node {
    int data;
    struct Node* next;
};
```

```
void removeDuplicates(struct Node* head);
void printList(struct Node* head);

int main() {
    struct Node* head = (struct Node*)malloc(sizeof(struct Node));
    head->data = 1;
    head->next = (struct Node*)malloc(sizeof(struct Node));
    head->next->data = 2;
    head->next->next = (struct Node*)malloc(sizeof(struct Node));
    head->next->next->data = 2;
    head->next->next->next = (struct Node*)malloc(sizeof(struct Node));
    head->next->next->next->data = 3;
    head->next->next->next->next = NULL;

    printf("Original list: ");
    printList(head);
    printf("\n");

    removeDuplicates(head);

    printf("List after removing duplicates: ");
    printList(head);
    printf("\n");

    return 0;
}

void removeDuplicates(struct Node* head) {
    struct Node* current = head;
    while (current != NULL && current->next != NULL) {
        if (current->data == current->next->data) {
            struct Node* temp = current->next;
            current->next = current->next->next;
            free(temp);
        } else {
            current = current->next;
        }
    }
}

void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```

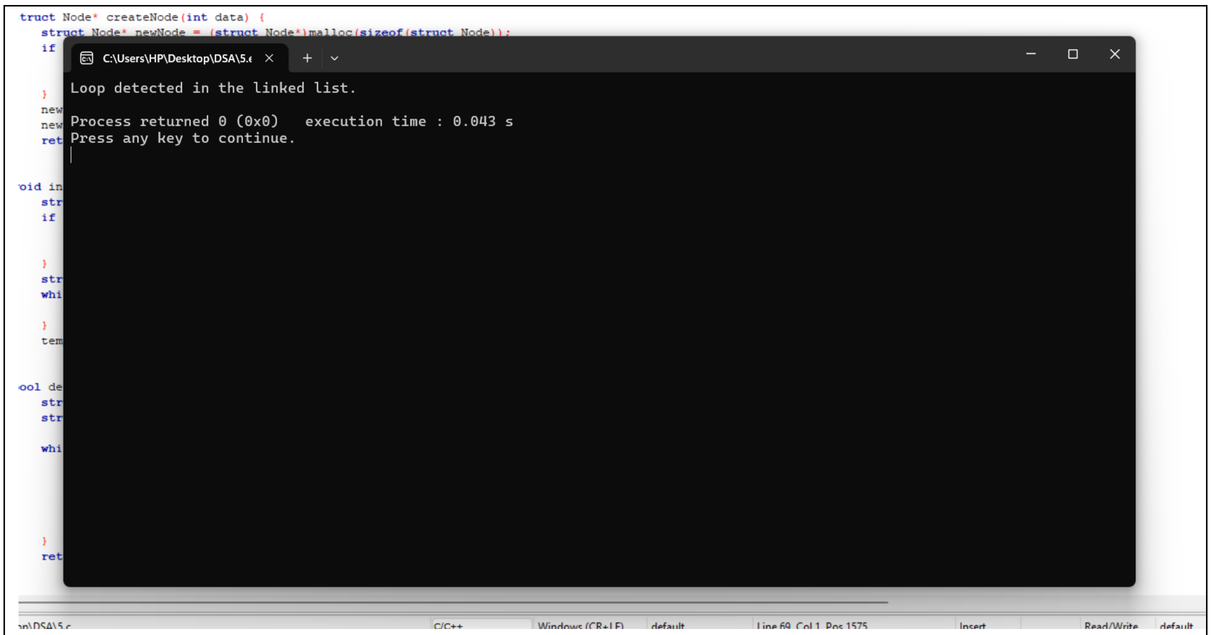


```
Original list: 1 2 2 3
List after removing duplicates: 1 2 3

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

- **Detect loop or cycle in a linked list**

Code

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data);
void insertAtEnd(struct Node** head, int data);
bool detectLoop(struct Node* head);

int main() {
    struct Node* head = NULL;

    insertAtEnd(&head, 1);
    insertAtEnd(&head, 2);
    insertAtEnd(&head, 3);
    insertAtEnd(&head, 4);
    head->next->next->next->next = head->next;

    if (detectLoop(head)) {
        printf("Loop detected in the linked list.\n");
    } else {
        printf("No loop detected in the linked list.\n");
    }

    return 0;
}

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

bool detectLoop(struct Node* head) {
    struct Node* slow = head;
    struct Node* fast = head;

    while (slow != NULL && fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {
            return true;
        }
    }
    return false;
}
```



```
truct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

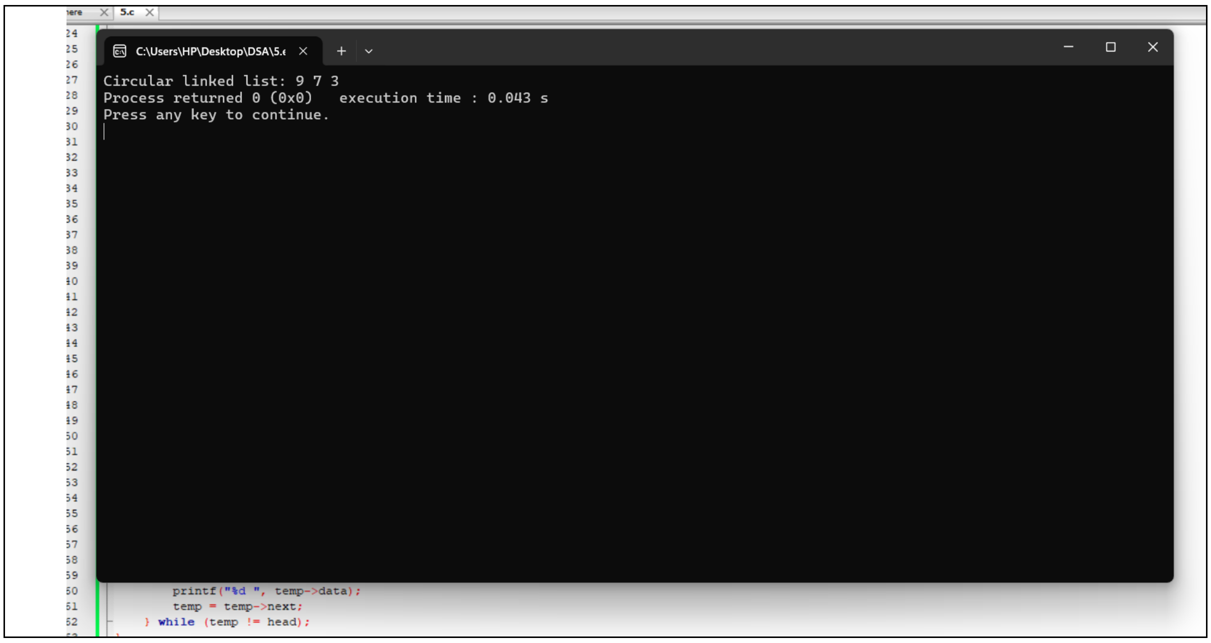
C:\Users\HP\Desktop\DSA\5.e

```
Loop detected in the linked list.

Process returned 0 (0x0)   execution time : 0.043 s
Press any key to continue.
```

- **Traversal of Circular Linked List**

Code

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data);
void insertAtEnd(struct Node** head, int newData);
void traverseCircularLinkedList(struct Node* head);

int main() {
    struct Node* head = NULL;

    insertAtEnd(&head, 3);
    insertAtEnd(&head, 7);
    insertAtEnd(&head, 9);

    printf("Circular linked list: ");
    traverseCircularLinkedList(head);

    return 0;
}

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = createNode(newData);
    if (*head == NULL) {
        *head = newNode;
        newNode->next = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = *head;
        *head = newNode; // Update head to point to the new node
    }
}

void traverseCircularLinkedList(struct Node* head) {
    if (head == NULL) {
        printf("Circular linked list is empty\n");
        return;
    }
    struct Node* temp = head;
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);
}
```



```
Circular linked list: 9 7 3
Process returned 0 (0x0)   execution time : 0.043 s
Press any key to continue.
```

```
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);
```