|  |  |
| --- | --- |
| **Name of the student** | **ANIRUDDHA ONKAR** |
| **Roll No** | **554** |
| **USN** | **01FE22BEC298** |
| **Div** | **E** |

**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. |
| Ans. ****Data Type****:   * A data type in C programming defines the type of data that a variable can hold. * It specifies the range of values that a variable of that type can store and the operations that can be performed on it. * Common data types in C include int, float, char, double, and pointers. * Ex ; * int age; // Declaration of a variable 'age' of type int * float weight; // Declaration of a variable 'weight' of type float * char grade; // Declaration of a variable 'grade' of type char   ****Variable****:   * A variable in C programming is a named storage location in memory that can hold a value of a specific data type. * Variables are used to store and manipulate data during program execution. * Each variable has a unique identifier (name) that is used to access its value.   int age; // Declaration of a variable 'age' of type int  float weight; // Declaration of a variable 'weight' of type float  char grade; // Declaration of a variable 'grade' of type char  age = 25; // Assigning a value to the variable 'age'  weight = 65.5; // Assigning a value to the variable 'weight'  grade = 'A'; // Assigning a value to the variable 'grade' |
| 2.Explain the concept of data types in C programming. Discuss the different types  of data types available in C.  Ans: C programming, data types are used to define the type of data that a variable can hold. They specify the range of values that a variable can store and the operations that can be performed on it. Data types are essential for declaring variables and specifying the memory layout for storing data.  ****int****: Represents integer values. It typically occupies 4 bytes of memory and can store whole numbers within a certain range .  int num = 10;  ****char****: Represents single characters. It typically occupies 1 byte of memory and can store characters from the ASCII character set,  char letter = 'A';  ****float****: Represents floating-point numbers (real numbers). It typically occupies 4 bytes of memory and provides single-precision floating-point values.  float pi = 3.14;  ****double****: Represents double-precision floating-point numbers. It typically occupies 8 bytes of memory and provides higher precision compared to float.  double height = 5.8; |
| 3] How are variables declared and initialized in C programming? Provide  examples of variable declarations with different data types.  Ans;  ****Declaration****:   * To declare a variable, you specify its data type followed by the variable name. * The data type determines the type of values the variable can store, and the variable name is used to identify the variable. * Syntax: data\_type variable\_name   Example:  1]int age; // Declaration of an integer variable named 'age'  2]float weight; // Declaration of a floating-point variable named 'weight'  3]char grade; // Declaration of a character variable named 'grade'  ****Initialization****:   * Initialization is the process of assigning an initial value to a variable at the time of declaration. * It involves using the assignment operator (**=**) to assign a value to the variable. * Examples: * int age = 25; // Declaration and initialization of 'age' with value 25 * float weight = 65.5; // Declaration and initialization of 'weight' with value 65.5 * char grade = 'A'; // Declaration and initialization of 'grade' with value 'A' |
| 4]Discuss the scope and lifetime of variables in C programming. What are global  and local variables?  ****Scope of Variables****:   * + The scope of a variable refers to the region of code within which the variable is accessible.   + Variables can have either global scope or local scope.   + Global variables are accessible from anywhere within the program, whereas local variables are accessible only within the block or function in which they are declared.   + Scope determines where a variable can be accessed and used.   ****Lifetime of Variables****:   * + The lifetime of a variable refers to the duration for which the variable exists in memory.   + Variables can have static lifetime or automatic (or dynamic) lifetime.   + Variables with static lifetime persist throughout the entire execution of the program. They are allocated memory when the program starts and deallocated memory when the program terminates.   + Variables with automatic lifetime are created when the block or function in which they are declared is entered, and they are destroyed when the block or function exits. They are typically stored on the stack.   ****Global Variables****:   * + Global variables are declared outside of any function or block, typically at the top of the file or in a header file.   + They have global scope, meaning they are accessible from anywhere in the program, including all functions.   + Global variables have static lifetime, meaning they persist for the entire duration of the program's execution. |
| 5]Explain the concept of type casting in C programming. When is type casting  necessary, and how is it performed? |
| Ans] In C programming, type casting refers to the conversion of one data type into another. This is necessary when you want to assign a value of one data type to a variable of another data type or when you want to perform operations involving different data types. Type casting helps in ensuring that the data is properly interpreted and manipulated according to the desired format.  Example:  float f = 3.14;  int i;  i = (int)f;  printf("Value of i: %d\n", i); // Output: Value of i: 3 |
| Operators: 1]Describe the purpose and usage of the ternary conditional operator (?:) in C programming. Provide an example demonstrating its usage. |
| Ans:The ternary conditional operator (?:) is a conditional operator in C programming that provides a concise way to express conditional statements. It's often used as a shorthand for simple if-else statements. The operator takes three operands:   1. A condition to be evaluated. 2. An expression to be evaluated if the condition is true. 3. An expression to be evaluated if the condition is false. 4. Syntax:   condition ? expression1 : expression2  An example demonstrating the usage of the ternary conditional operator:  #include <stdio.h>  int main() {  int num = 10;  int result;  // Using ternary conditional operator to assign value based on condition  result = (num > 0) ? 1 : -1;  printf("Result: %d\n", result);  return 0;  }  Here the result :1 is printed |
| 2]Discuss the bitwise operators available in C programming. Explain their usage  with suitable examples. |
| Ans]In C programming, bitwise operators are used to perform operations on individual bits of integer operands. These operators manipulate the binary representation of numbers at the bit level. There are six bitwise operators available in C:  ****Bitwise AND (&)****: Performs a bitwise AND operation between corresponding bits of two operands. The result is 1 if both bits are 1; otherwise, it's 0.  ****Bitwise OR (|)****: Performs a bitwise OR operation between corresponding bits of two operands. The result is 1 if at least one of the bits is 1; otherwise, it's 0.  ****Bitwise XOR (^)****: Performs a bitwise XOR (exclusive OR) operation between corresponding bits of two operands. The result is 1 if the bits are different; otherwise, it's 0.  ****Bitwise NOT (~)****: Performs a bitwise NOT operation on each bit of the operand, resulting in the one's complement of the operand.  ****Left Shift (<<)****: Shifts the bits of the left operand to the left by a specified number of positions, with zeros shifted in from the right.  ****Right Shift (>>)****: Shifts the bits of the left operand to the right by a specified number of positions. For unsigned integers, zeros are shifted in from the left. For signed integers, the behavior depends on the implementation-defined sign extension or zero-fill.  #include <stdio.h>  int main() {  unsigned int a = 10; // Binary: 1010  unsigned int b = 6; // Binary: 0110  unsigned int result;  // Bitwise AND  result = a & b; // result = 1010 & 0110 = 0010 (Decimal: 2)  printf("Bitwise AND: %u\n", result);  // Bitwise OR  result = a | b; // result = 1010 | 0110 = 1110 (Decimal: 14)  printf("Bitwise OR: %u\n", result);  // Bitwise XOR  result = a ^ b; // result = 1010 ^ 0110 = 1100 (Decimal: 12)  printf("Bitwise XOR: %u\n", result);  // Bitwise NOT  result = ~a; // result = ~1010 = 0101 (Decimal: 4294967285)  printf("Bitwise NOT of a: %u\n", result);  // Left Shift  result = a << 2; // result = 101000 (Decimal: 40)  printf("Left Shift of a by 2: %u\n", result);  // Right Shift  result = a >> 1; // result = 0101 (Decimal: 5)  printf("Right Shift of a by 1: %u\n", result);  return 0;  } |
| 3]Explain the difference between the postfix and prefix increment operators (++)  in C programming. Provide examples to illustrate. |
| Ans:In C programming, both the postfix (x++) and prefix (++x) increment operators are used to increment the value of a variable by 1. However, there's a fundamental difference in how they behave, particularly in terms of when the increment operation takes place.  ****Postfix Increment (x++)****:   * + In the postfix increment operator, the current value of the variable is used in the expression where the operator is applied, and then the variable's value is incremented.   + The increment operation occurs after the value of the variable is used in the current expression.   + After the increment, the updated value is stored back into the variable.   Example:  int x = 5;  int result = x++; // Postfix increment  printf("Result: %d\n", result); // Output: 5  printf("x: %d\n", x);  Output: 6  ****Prefix Increment (++x)****:   * In the prefix increment operator, the variable's value is incremented first, and then the updated value is used in the expression where the operator is applied. * The increment operation occurs before the value of the variable is used in the current expression.   Example:  int y = 5;  int result = ++y; // Prefix increment  printf("Result: %d\n", result); // Output: 6  printf("y: %d\n", y);  Output: 6 |
| 4]What is the significance of the logical AND (&&) and logical OR (||) operators in C programming? How are they used in conditional expressions? |
| Ans]In C programming, the logical AND (**&&**) and logical OR (**||**) operators are used to perform logical operations on boolean expressions. These operators are often used in conditional statements to control the flow of program execution based on certain conditions.  ****Logical AND (**&&**)****:   * + The logical AND operator (**&&**) returns true if both of its operands evaluate to true; otherwise, it returns false.   + It short-circuits the evaluation process, meaning that if the left operand evaluates to false, the right operand is not evaluated because the overall result will be false regardless of its value.   Example:  int x = 5;  int y = 10;  if (x > 0 && y < 20) {  printf("Both conditions are true\n");  }  In this example, the condition **x > 0 && y < 20** will evaluate to true because both **x > 0** and **y < 20** are true.  ****Logical OR (**||**)****:   * The logical OR operator (**||**) returns true if at least one of its operands evaluates to true; it returns false only if both operands are false. * Similar to logical AND, it also short-circuits the evaluation process. If the left operand evaluates to true, the right operand is not evaluated because the overall result will be true regardless of its value.   Example:  int a = 5;  int b = 10;  if (a == 5 || b == 20) {  printf("At least one condition is true\n");  }  In this example, the condition **a == 5 || b == 20** will evaluate to true because **a == 5** is true. |
| 5]Discuss the concept of operator precedence and associativity in C programming.  Provide examples to demonstrate how they affect expression evaluation. |
| Ans:In C programming, operator precedence and associativity are important concepts that determine the order in which operators are evaluated within an expression. These concepts ensure that expressions are evaluated correctly and consistently according to predefined rules.  ****Operator Precedence****: Operator precedence refers to the priority assigned to different operators in an expression. Operators with higher precedence are evaluated before those with lower precedence. If operators have the same precedence, the associativity of the operators determines the order of evaluation.  For example, in the expression **a + b \* c**, the multiplication operator **\*** has a higher precedence than the addition operator **+**, so **b \* c** is evaluated first, followed by the addition of **a**.  Here's a list of some common operators in C programming along with their precedence from highest to lowest:   * + Postfix increment/decrement (++/--), Function call (), Array subscripting []   + Prefix increment/decrement (++/--), Unary plus/minus (+/-), Logical NOT (!), Bitwise NOT (~), Sizeof, Cast ()   + Multiplication (\*), Division (/), Modulus (%)   + Addition (+), Subtraction (-)   + Bitwise left shift (<<), Bitwise right shift (>>)   + Relational operators (<, <=, >, >=)   + Equality operators (==, !=)   + Bitwise AND (&)   + Bitwise XOR (^)   + Bitwise OR (|)   + Logical AND (&&)   + Logical OR (||)   + Conditional operator (?:)   + Assignment operators (=, +=, -=, \*=, /=, %=, <<=, >>=, &=, ^=, |=   ****Operator Associativity****: Operator associativity determines the order in which operators of the same precedence are evaluated. It can be either left-to-right (left associativity) or right-to-left (right associativity).  For example, in the expression **a = b = c**, the assignment operator **=** has right-to-left associativity, so **b = c** is evaluated first, followed by **a = b**.  Most binary operators, such as addition **+** and multiplication **\***, have left-to-right associativity, meaning they are evaluated from left to right. However, some operators, like the assignment operator **=**, have right-to-left associativity.  #include <stdio.h>  int main() {  int a = 5, b = 10, c = 2;  int result = a + b \* c; // Operator precedence: Multiplication (\*) has higher precedence than Addition (+)  // Multiplication is evaluated first: b \* c = 10 \* 2 = 20  // Then addition is performed: a + 20 = 5 + 20 = 25  printf("Result: %d\n", result); // Output: Result: 25  return 0;  } |
| Control Structures: |
| 1] The switch statement in C programming is a control flow statement that allows you to execute one of many possible blocks of code based on the value of an expression. It provides an alternative to using multiple if-else statements when you have several conditions to check against the same variable. **Purpose and Usage of switch statement:** ****Purpose****:   * + The primary purpose of the switch statement is to simplify code readability and maintainability when dealing with multiple possible values of a variable.   + It provides a structured and efficient way to handle multiple conditional branches.   ****Usage****:  The syntax of the switch statement is as follows:   * + switch (expression) {   + case constant1:   + // code block 1   + break;   + case constant2:   + // code block 2   + break;   + ...   + default:   + // default code block   + }  **Difference from if-else statement:** ****Syntax****:   * + The switch statement is designed specifically for testing the value of a single variable against multiple possible values, whereas if-else statements are more versatile and can handle a wider range of conditions.   ****Multiple Conditions****:   * + The switch statement is well-suited for cases where you have multiple possible values to test against the same variable.   + If-else statements, on the other hand, are more flexible and can handle complex conditions involving multiple variables and relational operators.   ****Execution****:   * + In a switch statement, only one block of code corresponding to the matching case label is executed, providing more efficient execution compared to if-else chains where multiple conditions might need to be evaluated sequentially.   + In if-else chains, multiple conditions are evaluated sequentially until a true condition is found and its associated block of code is executed.   ****Default Case****:   * + The switch statement provides a default case that is executed when none of the case labels match the expression.   + If-else statements can also include a catch-all else block to handle cases not covered by previous conditions. |
| 2]Explain the concept of nested control structures in C programming. Provide an  example demonstrating nested if-else statements. |
| Ans]Nested control structures in C programming refer to the practice of using control structures within other control structures. This means placing one control structure (such as an if statement or loop) inside another control structure.  Nested control structures allow for more complex decision-making and looping scenarios, where certain conditions or iterations depend on the outcomes of other conditions or iterations.  One common example of nested control structures involves using nested if-else statements, where one if-else statement is nested within another.  Here's an example demonstrating nested if-else statements:  #include <stdio.h>  int main() {  int num = 10;  // Outer if-else statement  if (num > 0) {  printf("Number is positive\n");  // Nested if-else statement  if (num % 2 == 0) {  printf("Number is even\n");  } else {  printf("Number is odd\n");  }  } else {  printf("Number is not positive\n");  }  return 0;  } |
| 3]Discuss the role of the break and continue statements in loop control in C  programming. Provide examples to illustrate their usage. |
| Ans:In C programming, the **break** and **continue** statements are used to control the flow of execution within loops, such as for loops, while loops, and do-while loops. **1. break Statement:** The **break** statement is used to exit the loop immediately when encountered, regardless of the loop's condition. It allows you to prematurely terminate the loop execution based on a certain condition.  ****Usage of** break **statement:****   * It is commonly used to exit the loop when a specific condition is met. * It helps in avoiding unnecessary iterations when further processing is not required.   ****Example:****  #include <stdio.h>  int main() {  int i;  // Print numbers from 1 to 10, but exit loop when i reaches 5  for (i = 1; i <= 10; i++) {  if (i == 5) {  break; // Exit loop when i is 5  }  printf("%d ", i);  }  return 0;  }  Output:  1 2 3 4 **2. continue Statement:** The **continue** statement is used to skip the current iteration of the loop and proceed to the next iteration. It allows you to bypass certain iterations based on a specific condition without terminating the loop entirely.  ****Usage of** continue **statement:****   * It is used to skip certain iterations based on a condition. * It helps in avoiding the execution of certain code blocks within the loop for specific cases.   ****Example:****  #include <stdio.h>  int main() {  int i;  // Print even numbers from 1 to 10 using continue statement  for (i = 1; i <= 10; i++) {  if (i % 2 != 0) {  continue; // Skip iteration if i is odd  }  printf("%d ", i);  }  return 0;  }  Output:  2 4 6 8 10 |
| 4]What are the advantages of using the for loop over the while loop in C  programming? Provide examples comparing the two |
| In C programming, both the **for** loop and the **while** loop are used for iteration, but each has its advantages and situations where it is more appropriate to use. **Advantages of for loop over while loop:** ****Initialization, Condition, and Update in One Line****:   * + The **for** loop allows you to initialize loop control variables, specify loop continuation condition, and update loop control variables all in one line. This makes it more compact and easier to understand when the loop control logic is straightforward.   ****Compactness and Readability****:   * + For loops tend to be more compact and concise, making them more readable, especially for simple iteration scenarios.   ****Loop Counter Management****:   * + For loops are particularly useful when you need to iterate a specific number of times, as they provide a built-in loop counter that can be easily managed within the loop structure.  **Comparison Example:**Example using **for** loop; #include <stdio.h>  int main() {  // Using for loop to print numbers from 1 to 5  printf("Using for loop:\n");  for (int i = 1; i <= 5; i++) {  printf("%d ", i);  }  printf("\n");  return 0;  }  #include <stdio.h>  int main() {  // Using while loop to print numbers from 1 to 5  printf("Using while loop:\n");  int i = 1;  while (i <= 5) {  printf("%d ", i);  i++;  }  printf("\n");  return 0;  }  In both examples, we achieve the same result of printing numbers from 1 to 5. However, the **for** loop provides a more concise and readable way to achieve this task, especially when the loop counter is explicitly defined and managed within the loop structure. |
| 5]Explain the concept of short-circuit evaluation in C programming. How does it affect the evaluation of logical expressions in if statements? |
| Ans] Short-circuit evaluation is a feature in C programming (and many other programming languages) where the evaluation of a logical expression stops as soon as the result of the entire expression is known based on the evaluation of a portion of the expression. In other words, if the result of an expression can be determined by evaluating only a part of it, the remaining part of the expression is not evaluated.  Short-circuit evaluation is primarily used with logical AND (**&&**) and logical OR (**||**) operators. **Short-circuit evaluation with logical AND (&&):**  * When using the logical AND operator (**&&**), if the left operand evaluates to false, the overall result of the expression will be false regardless of the value of the right operand. Therefore, the right operand is not evaluated, and the evaluation of the expression is short-circuited. * If the left operand evaluates to true, then the right operand is evaluated to determine the overall result of the expression.  **Short-circuit evaluation with logical OR (||):**  * When using the logical OR operator (**||**), if the left operand evaluates to true, the overall result of the expression will be true regardless of the value of the right operand. Therefore, the right operand is not evaluated, and the evaluation of the expression is short-circuited. * If the left operand evaluates to false, then the right operand is evaluated to determine the overall result of the expression.  **Impact on if statements:** In if statements (and other conditional expressions), short-circuit evaluation can be leveraged to improve performance and avoid unnecessary evaluations. If the outcome of an if condition can be determined based on the evaluation of only one part of the condition, the remaining part is not evaluated.  **Ex;**  **#include <stdio.h>**  **int main() {**  **int a = 5, b = 0;**  **// Using short-circuit evaluation with logical AND operator**  **if (b != 0 && a / b > 2) {**  **printf("This line will not be executed\n");**  **}**  **// Using short-circuit evaluation with logical OR operator**  **if (b == 0 || a / b > 2) {**  **printf("This line will be executed\n");**  **}**  **return 0;**  **}**  }  }  In this example:   * In the first if statement, short-circuit evaluation prevents the division by zero error because the left operand **b != 0** evaluates to false, and the right operand is not evaluated. * In the second if statement, short-circuit evaluation prevents the division by zero error because the left operand **b == 0** evaluates to true, so the right operand is not evaluated, and the overall result of the expression is true. |
| • Functions: |
| 1]Describe the purpose and structure of a function prototype in C programming. Why is it necessary to declare function prototypes? |
| Ans]In C programming, a function prototype serves as a declaration of a function before its actual implementation. It provides the compiler with essential information about the function, such as its name, return type, and parameters. Function prototypes are typically declared at the beginning of a file or in a header file. **Purpose of function prototypes:** ****Provide Compiler Information****:   * + Function prototypes inform the compiler about the existence, return type, and parameters of functions that will be defined later in the code. This enables the compiler to perform type checking and detect errors in function calls.   ****Enable Forward Declaration****:   * + Function prototypes allow you to use a function before its actual definition in the code. This is especially useful when functions call each other, or when functions are defined in separate files.   ****Improve Readability and Maintainability****:   * + Function prototypes improve code readability by providing a summary of the functions available in a program. They serve as documentation for other programmers, indicating what functions are available and how they are used.  **Structure of a function prototype:** The structure of a function prototype follows this general format  return\_type function\_name(parameter\_list);   * **return\_type**: Specifies the data type of the value returned by the function. If the function does not return a value, the return type is **void**. * **function\_name**: The name of the function. * **parameter\_list**: Specifies the data types and names of parameters passed to the function. If the function takes no parameters, the parameter list is left empty or specified as **void**.   #include <stdio.h>  // Function prototype declaration  int add(int a, int b);  int main() {  int result = add(3, 4); // Function call  printf("Result: %d\n", result);  return 0;  }  // Function definition  int add(int a, int b) {  return a + b;  }   1. Explain the difference between call by value and call by reference in C programming. Provide examples to illustrate both concepts.   In C programming, a function prototype serves as a declaration of a function before its actual implementation. It provides the compiler with essential information about the function, such as its name, return type, and parameters. Function prototypes are typically declared at the beginning of a file or in a header file. **Purpose of function prototypes:** ****Provide Compiler Information****:   * + Function prototypes inform the compiler about the existence, return type, and parameters of functions that will be defined later in the code. This enables the compiler to perform type checking and detect errors in function calls.   ****Enable Forward Declaration****:   * + Function prototypes allow you to use a function before its actual definition in the code. This is especially useful when functions call each other, or when functions are defined in separate files.   ****Improve Readability and Maintainability****:   * + Function prototypes improve code readability by providing a summary of the functions available in a program. They serve as documentation for other programmers, indicating what functions are available and how they are used.  **Structure of a function prototype:** The structure of a function prototype follows this general format:  scssCopy code  return\_type function\_name(parameter\_list);   * **return\_type**: Specifies the data type of the value returned by the function. If the function does not return a value, the return type is **void**. * **function\_name**: The name of the function. * **parameter\_list**: Specifies the data types and names of parameters passed to the function. If the function takes no parameters, the parameter list is left empty or specified as **void**.  **Example of a function prototype:** #include <stdio.h>  // Function prototype  void increment(int x);  int main() {  int num = 10;  // Call the function with num as argument  increment(num);  // The value of num remains unchanged  printf("Value of num after function call: %d\n", num); // Output: Value of num after function call: 10  return 0;  }  // Function definition  void increment(int x) {  x++; // Increment the parameter x  }  In this example:   * The function prototype **int add(int a, int b);** informs the compiler about the existence, return type, and parameters of the **add** function before its actual implementation. * The function **add** is defined later in the code, after it has been called in the **main** functios  **Call by Reference:** **In call by reference, the memory address of the argument is passed to the function, allowing the function to directly access and modify the original data stored at that memory address. Changes made to the parameter within the function affect the original argument's value outside the function.**   * #include <stdio.h> * // Function prototype * void increment(int \*ptr); * int main() { * int num = 10; * // Call the function with the address of num as argument * increment(&num); * // The value of num |
| 3]Discuss the concept of recursion in C programming. Provide an example of a recursive function and explain how it works. |
| ecursion is a powerful programming technique in which a function calls itself directly or indirectly to solve a problem. It allows solving complex problems by breaking them down into smaller, more manageable subproblems. Recursion is widely used in C programming, especially in algorithms dealing with tree-like structures, sorting, searching, and mathematical problems. **Structure of a recursive function:** A recursive function typically consists of two parts:   1. ****Base case****: The base case defines the simplest scenario where the function does not need to call itself anymore. It serves as the termination condition for the recursion. 2. ****Recursive case****: The recursive case defines how the function calls itself with smaller input or subproblems, eventually leading to the base case.  **Example of a recursive function:** Let's consider an example of a recursive function to calculate the factorial of a non-negative integer.  #include <stdio.h>  // Function prototype  int factorial(int n);  int main() {  int n = 5;  printf("Factorial of %d = %d\n", n, factorial(n));  return 0;  }  // Recursive function to calculate factorial  int factorial(int n) {  // Base case: factorial of 0 or 1 is 1  if (n == 0 || n == 1) {  return 1;  } else {  // Recursive case: n! = n \* (n-1)!  return n \* factorial(n - 1);  }  }  ****Output:****  ****Factorial of 5 = 120**** |
| **Explanation of how the recursive function works:**  1. In the **main()** function, **factorial(5)** is called. 2. Inside the **factorial()** function:    * When **n** is 5, it's not the base case, so the function computes **5 \* factorial(4)**.    * **factorial(4)** further calls **4 \* factorial(3)**.    * This process continues until **factorial(1)** is reached, which returns 1 (base case). 3. Then, each pending multiplication operation is resolved:    * **4 \* factorial(3)** = **4 \* 6 = 24**    * **5 \* factorial(4)** = **5 \* 24 = 120** 4. Finally, the result (**120**) is returned to the **main()** function. |
| 4]What is the significance of the return statement in C programming? How are  values returned from functions? |
| Ans]In C programming, the **return** statement is used to terminate the execution of a function and return a value (if applicable) to the calling code. The **return** statement is significant as it allows functions to produce output or results that can be used by the rest of the program. **Significance of the return statement:** ****Function Termination****:   * + The **return** statement signals the end of a function's execution and returns control to the calling code.   + Once a **return** statement is encountered, the function stops executing, and subsequent statements in the function are not executed.   ****Returning Values****:   * + Functions can return values to the calling code using the **return** statement. These values can be of any data type, including integers, floating-point numbers, characters, pointers, or structures.   + Returning values from functions allows them to produce output or results that can be utilized by other parts of the program.   ****Error Handling****:   * + Functions can use the **return** statement to indicate error conditions or exceptional situations. For example, a function might return an error code or a special value to indicate failure.  **How values are returned from functions:** ****Returning Values****:   * + To return a value from a function, the **return** statement is followed by the expression or variable whose value is to be returned.   + The data type of the expression or variable being returned must match the return type specified in the function declaration.   + For example, if a function is declared to return an integer, the **return** statement should return an expression of type integer.   ****Function Return Type****:   * + The return type of a function specifies the data type of the value returned by the function.   + If a function is declared with a return type other than **void**, it must contain at least one **return** statement that returns a value of the specified type.   + If a function is declared with a return type of **void**, it does not return a value, and the **return** statement can be omitted or used without an expression.   #include <stdio.h>  // Function declaration with return type int  int add(int a, int b);  int main() {  int result;  // Function call  result = add(3, 4);  // Output the result returned by the function  printf("Result: %d\n", result);  return 0;  }  // Function definition  int add(int a, int b) {  int sum = a + b;  return sum; // Return the sum to the calling code  } |
| 5]Describe the role of function parameters and arguments in C programming. How are function arguments passed to parameters? |
| In C programming, function parameters and arguments play a crucial role in defining the input and output of functions. They allow functions to accept input values, perform operations on them, and optionally return results. **Function Parameters:** Function parameters are variables declared in the function declaration or definition. They represent the input values that the function expects to receive when it is called. Parameters are placeholders for the actual values, and they define the data types and names of the values that the function will operate on. **Function Arguments:** Function arguments are the actual values passed to a function when it is called. These values are provided by the caller of the function and are used as inputs for the function's operations. Arguments are the concrete values that replace the parameters when the function is executed. **Passing Function Arguments to Parameters:** In C programming, function arguments can be passed to function parameters in two ways:   1. ****Pass by Value****:    * In pass by value, a copy of the argument's value is passed to the function parameter.    * Changes made to the parameter inside the function do not affect the original argument.    * Pass by value is the default method of passing arguments in C. 2. ****Pass by Reference (or Address)****:    * In pass by reference, the address (or reference) of the argument is passed to the function parameter.    * This allows the function to directly access and modify the original argument.    * Pass by reference is achieved by using pointers as function parameters.  **Example:** #include <stdio.h>  // Function prototype  void printValues(int x, int \*ptr);  int main() {  int num1 = 10;  int num2 = 20;  // Function call with pass by value and pass by reference  printValues(num1, &num2);  // Output the original and modified values  printf("Original num1: %d\n", num1);  printf("Modified num2: %d\n", num2);  return 0;  }  // Function definition  void printValues(int x, int \*ptr) {  x = 100; // Modifies the local copy of num1 (pass by value)  \*ptr = 200; // Modifies the value at the address pointed by ptr (pass by reference)  }  ****Output:****  ****Original num1: 10****  ****Modified num2: 200****  In this example:   * **num1** is passed to the function **printValues** by value, so changes made to **x** inside the function do not affect **num1**. * **num2** is passed to the function **printValues** by reference (using a pointer), so changes made to **\*ptr** inside the function directly modify **num2**.   窗体顶端  窗体底端 |
| • Arrays: |
| 1]Explain the concept of arrays in C programming. How are arrays declared and initialized? |
| Ans]In C programming, an array is a collection of elements of the same data type that are stored in contiguous memory locations. Arrays provide a convenient way to store and manipulate multiple values of the same type under a single name. Each element in an array can be accessed using its index, which represents its position in the array. **Declaration of Arrays:** Arrays are declared using the following syntax:  data\_type array\_name[array\_size];   * **data\_type**: Specifies the data type of the elements in the array. * **array\_name**: Represents the name of the array. * **array\_size**: Indicates the number of elements the array can hold.  **Initialization of Arrays:** Arrays can be initialized at the time of declaration or later using assignment statements. There are several ways to initialize arrays:  ****Static Initialization****:   * + Static initialization involves specifying the initial values of the array elements at the time of declaration.   + The syntax for static initialization is as follows:   data\_type array\_name[array\_size] = {value1, value2, ..., valueN};  Example:  int numbers[5] = {1, 2, 3, 4, 5};  ****Partial Initialization****:   * It is also possible to initialize only a part of the array, leaving the rest of the elements uninitialized. In this case, the unspecified elements are automatically initialized to zero (for numeric types) or NULL (for pointer types). * Example:   int numbers[5] = {1, 2}; // Initializes the first two elements to 1 and 2; the remaining elements are set to 0  ****Dynamic Initialization****:   * Arrays can be initialized dynamically using loops or input from the user. * Example:   int numbers[5];  for (int i = 0; i < 5; i++) {  numbers[i] = i + 1; // Initializes each element of the array with consecutive numbers starting from 1  } **Accessing Array Elements:** Array elements are accessed using square brackets **[]** with the index of the desired element inside the brackets. Array indices start from 0 and go up to **array\_size - 1**.  Example:  int numbers[5] = {1, 2, 3, 4, 5};  printf("Element at index 2: %d\n", numbers[2]); // Output: Element at index 2: 3 |
| 2]Discuss the difference between a one-dimensional array and a multi  dimensional array in C programming. Provide examples of both. |
| **One-Dimensional Array:** A one-dimensional array is a linear collection of elements stored in a single row or column. It represents a simple list of elements accessed using a single index.  ****Example:****  #include <stdio.h>  int main() {  // Declaration and initialization of a one-dimensional array  int arr1D[5] = {1, 2, 3, 4, 5};  // Accessing elements of the one-dimensional array  for (int i = 0; i < 5; i++) {  printf("arr1D[%d] = %d\n", i, arr1D[i]);  }  return 0;  }  ****Output:****  ****arr1D[0] = 1****  ****arr1D[1] = 2****  ****arr1D[2] = 3****  ****arr1D[3] = 4****  ****arr1D[4] = 5**** **Multi-Dimensional Array:** A multi-dimensional array is an array with more than one dimension. It represents a table or grid of elements organized in rows and columns, or in higher dimensions.  ****Example:****  ****#include <stdio.h>****  ****int main() {****  ****// Declaration and initialization of a multi-dimensional array (2D array)****  ****int arr2D[3][3] = {****  ****{1, 2, 3},****  ****{4, 5, 6},****  ****{7, 8, 9}****  ****};****  ****// Accessing elements of the multi-dimensional array (2D array)****  ****for (int i = 0; i < 3; i++) {****  ****for (int j = 0; j < 3; j++) {****  ****printf("arr2D[%d][%d] = %d\n", i, j, arr2D[i][j]);****  ****}****  ****}****  ****return 0;****  ****}****  ****Output:****  ****arr2D[0][0] = 1****  ****arr2D[0][1] = 2****  ****arr2D[0][2] = 3****  ****arr2D[1][0] = 4****  ****arr2D[1][1] = 5****  ****arr2D[1][2] = 6****  ****arr2D[2][0] = 7****  ****arr2D[2][1] = 8****  ****arr2D[2][2] = 9**** |
| 3] Describe the process of accessing array elements in C programming. How are array indices used to access elements? |
| **Accessing Elements:** Array elements are accessed using square brackets **[]** with the index of the desired element inside the brackets. For example:  int value = numbers[2]; // Retrieves the value of the third element of the 'numbers' array **Example:** #include <stdio.h>  int main() {  int numbers[5] = {10, 20, 30, 40, 50}; // Array declaration and initialization  // Accessing and printing each element of the array  for (int i = 0; i < 5; i++) {  printf("numbers[%d] = %d\n", i, numbers[i]);  }  return 0;  }  ****Output:****  ****numbers[0] = 10****  ****numbers[1] = 20****  ****numbers[2] = 30****  ****numbers[3] = 40****  ****numbers[4] = 50**** |
| ****4]****What is the significance of the null character ('\0') in C strings? How is it used to determine the end of a string? |
| ****Ans]****In C programming, strings are represented as arrays of characters terminated by a null character ('\0'). The null character is a special character with the ASCII value of 0. Its significance lies in its role as a string terminator.  When you create a string in C, you define an array of characters. For example:  ****char myString[10] = "Hello";****  Here, **myString** is an array of characters containing the characters 'H', 'e', 'l', 'l', 'o', and then implicitly terminated with a null character ('\0'). It's important to note that C strings are not like strings in other programming languages that have a built-in length property. Instead, they rely on the null character to determine the end of the string.  Functions like **strlen()** (string length) or **printf()** use the null character to determine the end of the string. For instance, **strlen()** scans the characters in the string until it encounters the null character, and then returns the number of characters scanned up to that point.  Here's an example of how the null character is used to determine the end of a string:  ****#include <stdio.h>****  ****#include <string.h>****  ****int main() {****  ****char myString[10] = "Hello"; // Automatically terminated with '\0'****  ****// Print the length of the string****  ****printf("Length of myString: %zu\n", strlen(myString));****  ****return 0;****  ****}****  In this example, **strlen()** will return 5 because it counts all characters in **myString** until it reaches the null character. Without the null character, functions like **strlen()** would not be able to determine where the string ends, leading to undefined behavior or incorrect results. |
| ****5]****Explain the concept of dynamic memory allocation for arrays in C  programming. How are dynamic arrays allocated and deallocated? |
| ****Ans]****Dynamic memory allocation in C allows you to allocate memory at runtime, as opposed to static memory allocation where memory is allocated at compile time. Dynamic memory allocation is particularly useful when you need to create arrays whose size is not known until runtime or when you want to manage memory more flexibly.  In C programming, dynamic memory allocation for arrays typically involves three key functions: **malloc()**, **calloc()**, and **realloc()** for allocation, and **free()** for deallocation.  ****malloc()****: It stands for memory allocation. It allocates a block of memory of a specified size in bytes and returns a pointer to the beginning of the block. The syntax is:  void \*malloc(size\_t size);  Here, **size** is the number of bytes to allocate. It returns a void pointer (void \*) to the allocated memory block if successful, or NULL if it fails to allocate memory.   1. ****calloc()****: It stands for contiguous allocation. It allocates a block of memory for an array of elements, each of which has a size of **size** bytes, and initializes all bits to zero. The syntax is:   void \*calloc(size\_t num\_elements, size\_t size);  Here, **num\_elements** is the number of elements to allocate memory for, and **size** is the size of each element in bytes. It returns a void pointer to the allocated memory block if successful, or NULL if it fails to allocate memory.   1. ****realloc()****: It reallocates memory for an existing memory block. It can be used to resize a previously allocated memory block, either increasing or decreasing its size. The syntax is:   void \*realloc(void \*ptr, size\_t size);  ****free()****: It deallocates the memory block previously allocated by **malloc()**, **calloc()**, or **realloc()**. The syntax is:  void free(void \*ptr);  ere, **ptr** is a pointer to the memory block to be deallocated.  Here's a simple example demonstrating dynamic memory allocation and deallocation for an array of integers:  #include <stdio.h>  #include <stdlib.h>  int main() {  int \*dynamicArray;  int size = 5;  // Allocate memory for the array  dynamicArray = (int \*)malloc(size \* sizeof(int));  if (dynamicArray == NULL) {  printf("Memory allocation failed\n");  return 1;  }  // Initialize the array  for (int i = 0; i < size; i++) {  dynamicArray[i] = i \* 2;  }  // Print the array  printf("Array elements: ");  for (int i = 0; i < size; i++) {  printf("%d ", dynamicArray[i]);  }  printf("\n");  // Deallocate memory  free(dynamicArray);  return 0;  } |
| • Pointers: |
| ****1]****Describe the purpose and usage of pointers in C programming. How are pointers  declared and initialized? |
| ****Ans]****Pointers in C are variables that store memory addresses. They are powerful tools that enable efficient memory management, dynamic memory allocation, and manipulation of data structures like arrays, strings, and linked lists. Pointers allow you to access and manipulate data indirectly by referencing the memory location rather than the actual value. **Purpose of Pointers:**  1. ****Dynamic Memory Allocation****: Pointers are commonly used to allocate memory dynamically at runtime using functions like **malloc()**, **calloc()**, and **realloc()**. 2. ****Passing Parameters****: Pointers enable passing parameters by reference to functions, allowing functions to modify the original variables. 3. ****Working with Arrays and Strings****: Pointers provide a convenient way to access individual elements of arrays and characters in strings. 4. ****Manipulating Data Structures****: Pointers facilitate the creation and manipulation of complex data structures like linked lists, trees, and graphs. 5. ****Efficient Memory Access****: By directly accessing memory addresses, pointers can lead to more efficient memory usage and faster program execution.  **Declaring and Initializing Pointers:** In C, pointers are declared using the **\*** (asterisk) symbol followed by the data type they point to. Pointers must be initialized before they can be used to store memory addresses. Initialization can be done by assigning the address of a variable or by using dynamic memory allocation functions.  Here's the syntax for declaring and initializing pointers:  <datatype> \*<pointer\_variable\_name>;  int \*ptr; // Declaration of a pointer to an integer  int num = 10;  ptr = &num; // Assigning the address of 'num' to the pointer 'ptr'  Initialization:  int num = 10;  ptr = &num; // Assigning the address of 'num' to the pointer 'ptr'  Alternatively, declaration and initialization can be combined:  int \*ptr = &num; // Declaration and initialization in a single statement;  Example;  #include <stdio.h>  int main() {  int num = 10;  int \*ptr = &num; // Pointer initialized with the address of 'num'  printf("Value of num: %d\n", num);  printf("Address of num: %p\n", (void \*)&num); // Printing the address of 'num'  printf("Value stored at the address pointed by ptr: %d\n", \*ptr); // Dereferencing the pointer to access the value  return 0;  } |
| ****2]****Explain the concept of pointer arithmetic in C programming. Provide examples to illustrate addition and subtraction operations on pointers. |
| Ans]Pointer arithmetic in C involves performing arithmetic operations on pointers to manipulate memory addresses. When you perform arithmetic operations on pointers, they are adjusted based on the size of the data type they point to. This allows you to navigate through arrays or iterate over memory blocks efficiently. **Addition and Subtraction Operations on Pointers:**  1. ****Addition****: Adding an integer value to a pointer moves the pointer forward by a certain number of elements. 2. ****Subtraction****: Subtracting an integer value from a pointer moves the pointer backward by a certain number of elements.  **Example:** Consider an array of integers and a pointer pointing to its first element. We can perform pointer arithmetic to navigate through the array.  #include <stdio.h>  int main() {  int arr[] = {10, 20, 30, 40, 50};  int \*ptr = arr; // Pointer points to the first element of the array  printf("Value at ptr: %d\n", \*ptr); // Prints the value at the current pointer position (10)  // Move the pointer to the next element using pointer arithmetic  ptr++;  printf("Value at ptr after increment: %d\n", \*ptr); // Prints the value at the next position (20)  // Move the pointer two positions forward  ptr += 2;  printf("Value at ptr after adding 2: %d\n", \*ptr); // Prints the value at the position two elements ahead (40)  // Move the pointer one position backward  ptr--;  printf("Value at ptr after subtracting 1: %d\n", \*ptr); // Prints the value at the position one element before (30)  return 0;  }  Value at ptr: 10  Value at ptr after increment: 20  Value at ptr after adding 2: 40  Value at ptr after subtracting 1: 30 |
| ****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 |
| ****Ans]****In C programming, function arguments can be passed either by value or by reference using pointers. Understanding the difference between these two approaches is crucial for efficient memory usage and understanding how modifications to variables inside functions affect the original data. **Pass by Value:** When passing arguments by value, a copy of the variable's value is passed to the function. Any modifications made to the parameter inside the function do not affect the original variable outside the function.  Example:  ****#include <stdio.h>****  ****// Function prototype****  ****void increment(int x);****  ****int main() {****  ****int num = 10;****  ****printf("Before function call: %d\n", num);****  ****// Passing 'num' by value****  ****increment(num);****  ****printf("After function call: %d\n", num); // 'num' remains unchanged****  ****return 0;****  ****}****  ****// Function definition****  ****void increment(int x) {****  ****x++; // Incrementing the local copy of 'num'****  ****printf("Inside function: %d\n", x); // Prints the modified value****  ****}****  ****Before function call: 10****  ****Inside function: 11****  ****After function call: 10****  In this example, even though **num** is modified inside the **increment()** function, the original value of **num** remains unchanged because it was passed by value. **Pass by Reference (Using Pointers):** When passing arguments by reference using pointers, the memory address of the variable is passed to the function. This allows the function to directly manipulate the original variable, and any changes made inside the function affect the original variable outside the function.  Example:  #include <stdio.h>  // Function prototype  void incrementByRef(int \*x);  int main() {  int num = 10;  printf("Before function call: %d\n", num);  // Passing 'num' by reference  incrementByRef(&num);  printf("After function call: %d\n", num); // 'num' is modified  return 0;  }  // Function definition  void incrementByRef(int \*x) {  (\*x)++; // Incrementing the value pointed to by 'x'  printf("Inside function: %d\n", \*x); // Prints the modified value  }  Before function call: 10  Inside function: 11  After function call: 11 |
| ****4]****Describe the concept of NULL pointers in C programming. How are NULL  pointers used and checked for in programs? |
| ****Ans]****In C programming, a NULL pointer is a pointer that does not point to any memory location. It is typically used to indicate that a pointer is not currently pointing to a valid object. NULL pointers are commonly used as sentinel values to signify the end of a data structure, as error indicators, or to initialize pointers before they are assigned valid memory addresses. **Concept of NULL Pointers:**  * A NULL pointer is represented by the constant **NULL**, which is defined in the **<stddef.h>** header file. * It is defined as a macro that expands to an implementation-defined null pointer constant, which is typically represented as an integer constant zero (**0**). * Assigning **NULL** to a pointer means that the pointer does not point to any valid memory location. * Dereferencing a NULL pointer (attempting to access the value it points to) typically results in undefined behavior, often leading to program crashes or unexpected behavior. Hence, it's essential to check for NULL pointers before dereferencing them  **Usage and Checking for NULL Pointers:** ****Initialization****: Pointers are often initialized to NULL to indicate that they are not currently pointing to any valid memory location. This is especially important when dynamically allocating memory and when declaring pointers that may not be immediately assigned valid addresses.  int \*ptr = NULL; // Initialize pointer to NULL  ****Checking for NULL Pointers****: Before dereferencing a pointer, it's crucial to check if it is NULL to avoid undefined behavior. This is typically done using an **if** statement.  if (ptr != NULL) {  // Pointer is not NULL, safe to dereference  \*ptr = 10;  } else {  // Handle the case where pointer is NULL  printf("Pointer is NULL!\n");  }  ****Returning NULL from Functions****: Functions can return NULL to indicate failure or an inability to perform the desired operation. For example, memory allocation functions like **malloc()**, **calloc()**, and **realloc()** return NULL when they fail to allocate memory.  int \*createArray(int size) {  int \*arr = malloc(size \* sizeof(int));  if (arr == NULL) {  // Memory allocation failed, return NULL  return NULL;  }  return arr;  }  ****Error Handling****: NULL pointers are often used to handle error conditions or exceptional cases in programs. For example, if a function fails to open a file, it might return a NULL pointer. **Example:** #include <stdio.h>  #include <stdlib.h>  int main() {  int \*ptr = NULL; // Initializing pointer to NULL  if (ptr != NULL) {  // Attempting to dereference the pointer  \*ptr = 10;  printf("Value at ptr: %d\n", \*ptr);  } else {  printf("Pointer is NULL! Cannot dereference.\n");  }  // Example of returning NULL from a function  int \*arr = createArray(5);  if (arr == NULL) {  printf("Failed to allocate memory for array!\n");  } else {  printf("Memory allocation successful!\n");  }  return 0;  }  int \*createArray(int size) {  int \*arr = malloc(size \* sizeof(int));  return arr;  }  Pointer is NULL! Cannot dereference.  Failed to allocate memory for array! |
| ****5]****Explain the role of pointers in dynamic memory allocation in C programming. How are pointers used to allocate and deallocate memory dynamically |
| ****Ans]****In C programming, pointers play a crucial role in dynamic memory allocation. Dynamic memory allocation allows you to allocate memory at runtime, enabling flexibility in memory management and the creation of data structures whose size is not known at compile time. Pointers are used to manage dynamically allocated memory by storing the addresses of memory blocks allocated on the heap. **Role of Pointers in Dynamic Memory Allocation:** ****Allocation****: Pointers are used to allocate memory dynamically using functions like **malloc()**, **calloc()**, and **realloc()**. These functions return a pointer to the allocated memory block, which can then be assigned to a pointer variable.  ****Deallocation****: Pointers are also used to deallocate dynamically allocated memory using the **free()** function. Once memory is no longer needed, **free()** is called with the pointer to the allocated memory block, releasing the memory back to the system heap. **Allocation Using Pointers:** ****malloc()****: Allocates a block of memory of a specified size in bytes. It returns a pointer to the beginning of the allocated memory block.   * int \*ptr = (int \*)malloc(5 \* sizeof(int)); // Allocating memory for an array of 5 integers   ****calloc()****: Allocates a block of memory for an array of elements, initializes all bits to zero, and returns a pointer to the beginning of the allocated memory block.  int \*ptr = (int \*)calloc(5, sizeof(int)); // Allocating memory for an array of 5 integers  ****realloc()****: Reallocates memory for an existing memory block, either increasing or decreasing its size. It returns a pointer to the reallocated memory block.  int \*ptr = (int \*)realloc(ptr, 10 \* sizeof(int)); // Reallocating memory for an array of 10 integers **Deallocation Using Pointers:**  * ****free()****: Deallocates the memory block previously allocated by **malloc()**, **calloc()**, or **realloc()**.   free(ptr); // Deallocating memory **Example:** Here's an example illustrating dynamic memory allocation and deallocation using pointers:  #include <stdio.h>  #include <stdlib.h>  int main() {  // Allocation  int \*ptr = (int \*)malloc(5 \* sizeof(int)); // Allocate memory for an array of 5 integers  if (ptr == NULL) {  printf("Memory allocation failed!\n");  return 1;  }  // Initialization  for (int i = 0; i < 5; i++) {  ptr[i] = i \* 2; // Initialize array elements  }  // Print the array  printf("Dynamic array: ");  for (int i = 0; i < 5; i++) {  printf("%d ", ptr[i]);  }  printf("\n");  // Deallocation  free(ptr); // Deallocate memory  return 0;  } |
| • Strings: |
| ****1]****Discuss the concept of strings in C programming. How are strings represented  and manipulated in C? |
| ****Ans]****In C programming, a string is a sequence of characters stored in contiguous memory locations, terminated by a null character ('\0'). C does not have a built-in string data type like other programming languages. Instead, strings are represented as arrays of characters. This representation allows for flexibility and efficiency in handling strings. **Representation of Strings:** ****Character Arrays****: Strings are typically represented as arrays of characters. Each character in the array corresponds to a character in the string, with the null character ('\0') marking the end of the string.  ****Null-Terminated****: C strings are null-terminated, meaning the null character ('\0') is used to signify the end of the string. This allows functions to determine the length of the string by scanning characters until the null character is encountered. **Manipulation of Strings:** ****String Input****: Strings can be input using functions like **scanf()** or **gets()** (deprecated) for standard input or by directly assigning character arrays.  ****String Output****: Strings can be output using functions like **printf()** or **puts()** for standard output.  ****String Functions****: C provides a set of library functions for string manipulation defined in the **<string.h>** header. Some commonly used functions include:   * + **strlen()**: Returns the length of a string.   + **strcpy()**: Copies one string to another.   + **strcat()**: Concatenates two strings.   + **strcmp()**: Compares two strings.   + **strchr()**: Finds the first occurrence of a character in a string.   + **strstr()**: Finds the first occurrence of a substring in a string.  **Example:** ****#include <stdio.h>****  ****#include <string.h>****  ****int main() {****  ****char str1[20] = "Hello"; // Initializing a string****  ****char str2[20]; // Declaring another string****  ****// Input string****  ****printf("Enter a string: ");****  ****scanf("%s", str2); // Reading input string****  ****// Output strings****  ****printf("String 1: %s\n", str1);****  ****printf("String 2: %s\n", str2);****  ****// String manipulation****  ****printf("Length of String 1: %zu\n", strlen(str1));****  ****printf("Length of String 2: %zu\n", strlen(str2));****  ****strcat(str1, " World"); // Concatenating strings****  ****printf("Concatenated String 1: %s\n", str1);****  ****if (strcmp(str1, str2) == 0) { // Comparing strings****  ****printf("String 1 and String 2 are equal.\n");****  ****} else {****  ****printf("String 1 and String 2 are not equal.\n");****  ****}****  ****return 0;****  ****}****  ****Enter a string: Programming****  ****String 1: Hello****  ****String 2: Programming****  ****Length of String 1: 5****  ****Length of String 2: 11****  ****Concatenated String 1: Hello World****  ****String 1 and String 2 are not equal.**** |
| ****2]****Explain the difference between character arrays and string literals in C  programming. Provide examples to illustrate both concepts. |
| ****Ans]****In C programming, character arrays and string literals are both used to represent strings, but there are differences in their characteristics and usage. **Character Arrays:**  * Character arrays are sequences of characters stored in contiguous memory locations. * They are declared as arrays of characters, typically terminated by a null character ('\0') to mark the end of the string. * Character arrays provide flexibility in storing and manipulating strings, allowing dynamic modification of their contents.   Example:  #include <stdio.h>  int main() {  char \*str2 = "Hello"; // String literal assignment to a pointer  printf("String 2: %s\n", str2); // Outputting the string literal  return 0;  } **Differences:** ****Mutability****:   * + Character arrays can be modified directly because they are stored in writable memory. You can change individual characters or concatenate multiple strings.   + String literals, on the other hand, are immutable. They are stored in read-only memory, and attempting to modify them directly results in undefined behavior.   ****Storage Location****:   * + Character arrays are typically stored in writable memory, allowing for dynamic modification.   + String literals are stored in read-only memory, making them suitable for constant strings that do not need to be modified.   ****Initialization****:   * + Character arrays are initialized using array syntax and can be explicitly sized or left unsized (with the size determined by the number of characters and the null terminator).   + String literals are initialized using double quotes and are automatically null-terminated by the compiler.  **Example Illustrating Differences:** #include <stdio.h>  int main() {  // Character array  char str1[] = "Hello";  str1[0] = 'h'; // Modifying the first character  printf("Modified Character Array: %s\n", str1);  // String literal  char \*str2 = "Hello";  // Attempting to modify the string literal (undefined behavior)  // str2[0] = 'h';  printf("String Literal: %s\n", str2);  return 0;  }  Modified Character Array: hello  String Literal: Hello |
| ****3]****Describe common string manipulation functions available in the C standard  library. Provide examples of functions like strlen, strcpy, strcat, and strcmp. |
| ****Ans]**** Certainly! The C standard library **<string.h>** provides a set of functions for string manipulation. Here are some common string manipulation functions along with examples:   1. ****strlen()****:    * Returns the length of a string (excluding the null terminator).    * Prototype: **size\_t strlen(const char \*str);**   Example:  ****#include <stdio.h>****  ****#include <string.h>****  ****int main() {****  ****char str[] = "Hello";****  ****size\_t length = strlen(str);****  ****printf("Length of '%s': %zu\n", str, length);****  ****return 0;****  ****}****   1. ****strcpy()****:    * Copies the contents of one string to another.    * Prototype: **char \*strcpy(char \*dest, const char \*src);**   Example:  ****#include <stdio.h>****  ****#include <string.h>****  ****int main() {****  ****char src[] = "Hello";****  ****char dest[10];****  ****strcpy(dest, src);****  ****printf("Copied string: %s\n", dest);****  ****return 0;****  ****}****   1. ****strcat()****:    * Concatenates (appends) one string to another.    * Prototype: **char \*strcat(char \*dest, const char \*src);**   Example:  ****#include <stdio.h>****  ****#include <string.h>****  ****int main() {****  ****char str1[] = "Hello";****  ****char str2[] = "Hello";****  ****int result = strcmp(str1, str2);****  ****if (result == 0) {****  ****printf("Strings are equal.\n");****  ****} else if (result < 0) {****  ****printf("First string is less than second string.\n");****  ****} else {****  ****printf("First string is greater than second string.\n");****  ****}****  ****return 0;****  ****}**** |
| ****4]****Discuss the concept of string tokenization in C programming. How are strings  split into tokens using delimiter characters? |
| ****Ans]****String tokenization in C programming involves splitting a string into smaller parts called tokens based on delimiter characters. Delimiters are characters used to separate tokens within the string. Common delimiters include whitespace characters (space, tab, newline), punctuation marks (comma, semicolon), or any custom character chosen by the programmer. **Tokenization Process:** ****Initialization****: The process begins with the initialization of a pointer to the string to be tokenized.  ****Token Extraction****: The string is scanned, and tokens are extracted one at a time. Tokens are sequences of characters separated by delimiter characters.  ****Token Processing****: Each token extracted can be processed individually based on the application's requirements.  ****Loop Termination****: The process continues until no more tokens are found in the string. **Functions for Tokenization:** In C programming, the **strtok()** function is commonly used for tokenization. It is declared in the **<string.h>** header file. The **strtok()** function modifies the original string by inserting null characters ('\0') to terminate each token. It returns a pointer to the first character of each token. Prototype: char \*strtok(char \*str, const char \*delimiters); Parameters:  * **str**: The string to be tokenized. For subsequent calls to **strtok()**, it should be set to NULL to continue tokenizing the same string. * **delimiters**: A string containing delimiter characters. Any of these characters act as separators between tokens.  **Example:** #include <stdio.h>  #include <string.h>  int main() {  char str[] = "Hello, World;Good,Morning";  const char delimiters[] = ",;"; // Delimiter characters  char \*token;  // Tokenization  token = strtok(str, delimiters); // Get the first token  while (token != NULL) {  printf("Token: %s\n", token);  token = strtok(NULL, delimiters); // Get the next token  }  return 0;  }  Token: Hello  Token: World  Token: Good  Token: Morning |
| 5]Explain the importance of null-terminated strings in C programming. How does the null character ('\0') signify the end of a string?  In C programming, null-terminated strings are essential for representing and working with strings. A null-terminated string is a sequence of characters stored in memory, terminated by a null character (**'\0'**). The null character serves as a sentinel value to mark the end of the string. Understanding the importance of null-terminated strings is crucial for efficient string manipulation in C programming. **Importance of Null-Terminated Strings:** ****String Length****: Null-terminated strings allow for efficient determination of string length. By scanning the characters in the string until the null character is encountered, functions can determine the length of the string without needing an explicit length parameter.  ****String Operations****: Most standard string manipulation functions in C, such as **strcpy()**, **strlen()**, **strcmp()**, etc., rely on null-terminated strings. These functions iterate over characters until the null character is found to perform their respective operations.  ****Memory Efficiency****: Null-terminated strings are memory-efficient. They require only one additional byte of memory to store the null terminator, compared to other methods that require storing the length of the string separately.  ****Compatibility****: Null-terminated strings are compatible with standard C library functions and widely used in C programming. Understanding null-terminated strings allows developers to leverage the rich set of string manipulation functions available in the standard library. **Significance of the Null Character ('\0'):**  * The null character (**'\0'**) is a special character with an ASCII value of 0. * In null-terminated strings, the null character signifies the end of the string. * When the null character is encountered while processing a string, it indicates to functions that the string has ended, allowing them to stop further processing. * By convention, null-terminated strings are constructed with the null character placed immediately after the last character of the string.  **Example:****#include <stdio.h>****int main() {** **char str[] = "Hello"; // Null-terminated string** **printf("String: %s\n", str);** **// Finding length of the string** **int length = 0;** **while (str[length] != '\0') {** **length++;** **}** **printf("Length of the string: %d\n", length);** **return 0;****}****String: Hello****Length of the string: 5** |
| Structures and Unions: |
| ****1]****Describe the purpose and usage of structures in C programming. How are  structures declared and accessed?  In C programming, structures (structs) are used to group together related data items of different types under a single name. Structures allow you to create custom data types that can represent complex entities, such as a person, a car, a book, etc. Each element within a structure is called a member or a field. **Purpose of Structures:** ****Organizing Data****: Structures provide a way to organize related data items into a single unit, making it easier to manage and manipulate complex data.  ****Modularity****: Structures promote modularity by encapsulating related data and operations into a single entity. This makes code more organized and easier to maintain.  ****Abstraction****: Structures enable abstraction by hiding the implementation details of complex data structures and providing a simple interface to interact with them.  ****Passing Complex Data****: Structures are useful for passing complex data between functions, as they allow multiple values to be passed as a single argument. **Declaration and Usage of Structures:**Declaration: To declare a structure, you use the **struct** keyword followed by the structure tag (name) and a list of members enclosed in curly braces **{}**. Each member is defined with a data type and a name. |
| ****2]****Explain the difference between structures and unions in C programming. When  would you choose one over the other?  In C programming, structure members are the individual elements or fields that make up a structure. Structures allow you to group together related data items of different types under a single name. Each member within a structure has its own data type and name, and they can be accessed and modified using the dot (**.**) operator. **Structure Members:** ****Data Type****: Each member of a structure can have its own data type, which can be any valid C data type, including primitive types (int, float, char, etc.), arrays, pointers, or even other structures.  ****Name****: Each member is given a unique name within the structure, which allows you to refer to it when accessing or modifying its value. **Accessing and Modifying Structure Members:** To access or modify the members of a structure, you use the dot (**.**) operator followed by the member name. This allows you to interact with individual members of a structure variable. Syntax:**Example:** #include <stdio.h>  #include <string.h>  // Declaration of a structure  struct Person {  char name[50];  int age;  float height;  };  int main() {  // Declaration of a structure variable  struct Person person1;  // Accessing and modifying structure members  strcpy(person1.name, "John"); // Modifying the 'name' member  person1.age = 30; // Modifying the 'age' member  person1.height = 5.9; // Modifying the 'height' member  // Printing structure members  printf("Name: %s\n", person1.name);  printf("Age: %d\n", person1.age);  printf("Height: %.2f\n", person1.height);  return 0;  }  Name: John  Age: 30  Height: 5.90 |
| ****3]**** |
| ****Ans]**Structures:** ****Members****: In a structure, each member has its own memory location, and the size of the structure is the sum of the sizes of its members.  ****Usage****: Structures are used to group together related data items of different types under a single name. They are suitable for representing objects or entities that have multiple properties.  ****Accessing Members****: Each member of a structure is accessed independently using the dot (**.**) operator.  ****Example****:  ****struct Person {****  ****char name[50];****  ****int age;****  ****float height;****  ****};**** **Unions:** ****Members****: In a union, all members share the same memory location, and the size of the union is the size of its largest member.  ****Usage****: Unions are used when you need to store different types of data in the same memory location. They are suitable for situations where you only need to access one member at a time, and the memory allocated for the union needs to be as small as possible.  ****Accessing Members****: Only one member of a union can be accessed at a time. Accessing a member of a union is similar to accessing a member of a structure using the dot (**.**) operator.  ****Example****:  ****union Value {****  ****int intValue;****  ****float floatValue;****  ****char stringValue[20];****  ****};**** **Differences:**  1. ****Memory Allocation****:    * Structures allocate separate memory locations for each member, while unions allocate a single memory location that is shared among all members. 2. ****Size****:    * The size of a structure is the sum of the sizes of its members, while the size of a union is the size of its largest member. 3. ****Usage****:    * Structures are used when you need to store multiple related data items, each with its own memory location.    * Unions are used when you need to store different types of data in the same memory location and access only one member at a time.  **When to Choose One Over the Other:**  * ****Choose Structures****:   + When you need to store multiple related data items that have different meanings or purposes.   + When you need to access multiple members of the data structure simultaneously. * ****Choose Unions****:   + When you need to store different types of data in the same memory location and only access one member at a time.   + When you need to conserve memory and ensure that the size of the data structure is as small as possible. |
| ****4]****Describe the concept of nested structures in C programming. How are structures within structures defined and accessed? |
| ****Ans]**** In C programming, nested structures refer to structures that are defined within other structures. This allows for the creation of hierarchical data structures where a structure member itself is another structure. Nested structures enable the representation of complex relationships between different data elements. **Definition of Nested Structures:** Nested structures are defined by declaring a structure within the definition of another structure. The inner structure can have its own members, including other structures, arrays, or primitive data types. **Accessing Members of Nested Structures:** Accessing members of nested structures is done using the dot (**.**) operator for each level of the structure hierarchy.  ****struct OuterStruct outer;****  ****// Accessing members of the outer structure****  ****outer.outerMember = 10;****  ****// Accessing members of the inner structure****  ****outer.inner.innerMember = 20;****  ****struct OuterStruct outer;****  ****// Accessing members of the outer structure****  ****outer.outerMember = 10;****  ****// Accessing members of the inner structure****  ****outer.inner.innerMember = 20;****  ****#include <stdio.h>****  ****// Definition of nested structures****  ****struct Date {****  ****int day;****  ****int month;****  ****int year;****  ****};****  ****struct Person {****  ****char name[50];****  ****struct Date birthDate;****  ****};****  ****int main() {****  ****// Declaration of a variable of type struct Person****  ****struct Person person;****  ****// Assigning values to members of nested structures****  ****strcpy(person.name, "John");****  ****person.birthDate.day = 15;****  ****person.birthDate.month = 6;****  ****person.birthDate.year = 1990;****  ****// Accessing and printing members of nested structures****  ****printf("Name: %s\n", person.name);****  ****printf("Birth Date: %d/%d/%d\n", person.birthDate.day, person.birthDate.month, person.birthDate.year);****  ****return 0;****  ****}****  ****#include <stdio.h>****  ****// Definition of nested structures****  ****struct Date {****  ****int day;****  ****int month;****  ****int year;****  ****};****  ****struct Person {****  ****char name[50];****  ****struct Date birthDate;****  ****};****  ****int main() {****  ****// Declaration of a variable of type struct Person****  ****struct Person person;****  ****// Assigning values to members of nested structures****  ****strcpy(person.name, "John");****  ****person.birthDate.day = 15;****  ****person.birthDate.month = 6;****  ****person.birthDate.year = 1990;****  ****// Accessing and printing members of nested structures****  ****printf("Name: %s\n", person.name);****  ****printf("Birth Date: %d/%d/%d\n", person.birthDate.day, person.birthDate.month, person.birthDate.year);****  ****return 0;****  ****}****  ****Name: John****  ****Birth Date: 15/6/1990**** |
| ****5]****Discuss the concept of typedef in C programming. How is typedef used to define  custom data types, including structures and unions? |
| ****Ans]****In C programming, the **typedef** keyword is used to create custom data types, including aliases for existing data types, structures, and unions. **typedef** allows you to define a new name for an existing data type, making the code more readable and easier to understand. **Syntax for typedef:** ****typedef existing\_type new\_type\_name;****   * **existing\_type**: The existing data type for which you want to create an alias. * **new\_type\_name**: The new name (alias) you want to define for the existing data type.  **Uses of typedef:** ****Creating Aliases for Existing Data Types****:  ****typedef int integer; // Creating an alias 'integer' for the data type 'int'****  ****typedef float real; // Creating an alias 'real' for the data type 'float'****  ****typedef struct {****  ****int day;****  ****int month;****  ****int year;****  ****} Date; // Defining a custom data type 'Date' for a structure****  ****Defining Custom Data Types****:  typedef struct {  int day;  int month;  int year;  } Date; // Defining a custom data type 'Date' for a structure  ****Simplifying Complex Data Declarations****:  c  typedef struct Person {  char name[50];  int age;  } Person; // Defining a custom data type 'Person' for a structure and its alias  ****Creating Aliases for Pointers****:  typedef int\* IntPtr; // Creating an alias 'IntPtr' for a pointer to an integer  typedef char\* StringPtr; // Creating an alias 'StringPtr' for a pointer to a character **Example:** #include <stdio.h>  // Creating aliases for existing data types  typedef int integer;  typedef float real;  // Defining a custom data type 'Date' for a structure  typedef struct {  int day;  int month;  int year;  } Date;  // Defining a custom data type 'Person' for a structure and its alias  typedef struct Person {  char name[50];  int age;  } Person;  int main() {  // Using aliases for existing data types  integer num1 = 10;  real num2 = 3.14;  // Using the custom data type 'Date'  Date today = {23, 3, 2024};  printf("Today's Date: %d/%d/%d\n", today.day, today.month, today.year);  // Using the custom data type 'Person'  Person person1 = {"John", 30};  printf("Name: %s, Age: %d\n", person1.name, person1.age);  return 0;  }  Today's Date: 23/3/2024  Name: John, Age: 30 |
| 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? |
| ****Ans]****File handling in C programming involves performing operations on files, such as reading from and writing to them. C provides standard library functions for file handling, which allow you to perform various operations on files, including opening, closing, reading, and writing. **Basic File Handling Operations:** ****Opening Files****:   * + Before you can perform any operations on a file, you must open it using the **fopen()** function, which returns a file pointer (**FILE\***) that represents the opened file.   ****FILE \*fp;****  ****fp = fopen("filename.txt", "mode");****  ****if (fp == NULL) {****  ****// Handle error****  ****}****  ****Closing Files****:   * After you have finished working with a file, you should close it using the **fclose()** function to release the associated resources.   ****fclose(fp);****  ****Reading from Files****:   * To read data from a file, you can use functions like **fscanf()**, **fgets()**, or **fread()**.   ****char buffer[100];****  ****fscanf(fp, "%s", buffer);****  ****Writing to Files****:   * To write data to a file, you can use functions like **fprintf()**, **fputs()**, or **fwrite()**.   ****fprintf(fp, "Hello, world!\n");**** **File Opening Modes:** When opening a file, you must specify the mode in which you want to open the file. The mode determines whether the file will be opened for reading, writing, or both, and whether the file will be created if it does not exist.   * ****"r"****: Open for reading. The file must exist. * ****"w"****: Open for writing. If the file exists, its contents are overwritten. If the file does not exist, it is created. * ****"a"****: Open for appending. Data is written to the end of the file. If the file does not exist, it is created. * ****"r+"****: Open for reading and writing. The file must exist. * ****"w+"****: Open for reading and writing. If the file exists, its contents are overwritten. If the file does not exist, it is created. * ****"a+"****: Open for reading and appending. Data can be read from and written to the file. If the file does not exist, it is created.  **Example:** ****#include <stdio.h>****  ****int main() {****  ****FILE \*fp;****  ****char buffer[100];****  ****// Opening a file for writing****  ****fp = fopen("example.txt", "w");****  ****if (fp == NULL) {****  ****printf("Error opening file!\n");****  ****return 1;****  ****}****  ****// Writing data to the file****  ****fprintf(fp, "Hello, world!\n");****  ****fclose(fp);****  ****// Opening the same file for reading****  ****fp = fopen("example.txt", "r");****  ****if (fp == NULL) {****  ****printf("Error opening file!\n");****  ****return 1;****  ****}****  ****// Reading data from the file****  ****fscanf(fp, "%s", buffer);****  ****printf("Data read from file: %s\n", buffer);****  ****fclose(fp);****  ****return 0;****  ****}****  ****Data read from file: Hello,**** |
| ****2]****Describe the role of file pointers in C programming. How are file pointers used  to navigate and manipulate files? |
| ****Ans]**** In C programming, file pointers (**FILE\***) are used to manage and manipulate files. A file pointer is a special data type that represents a connection to a file. It allows you to perform various operations on files, such as reading from and writing to them, as well as navigating through their contents. **Role of File Pointers:** ****Opening Files****: File pointers are used to open files using the **fopen()** function. The **fopen()** function returns a file pointer that is used to perform operations on the opened file.  ****Navigating Through Files****: File pointers are used to move the file position indicator, which determines the current position within the file. This allows you to read from or write to specific locations within the file.  ****Reading from Files****: File pointers are used to read data from files using functions like **fscanf()**, **fgets()**, or **fread()**. These functions use the file pointer to determine the current position within the file from which to read data.  ****Writing to Files****: File pointers are used to write data to files using functions like **fprintf()**, **fputs()**, or **fwrite()**. These functions use the file pointer to determine the current position within the file at which to write data.  ****Closing Files****: File pointers are used to close files using the **fclose()** function. Closing a file releases any resources associated with the file and ensures that any pending data is written to disk. **Example:** ****#include <stdio.h>****  ****int main() {****  ****FILE \*fp;****  ****char buffer[100];****  ****// Opening a file for reading****  ****fp = fopen("example.txt", "r");****  ****if (fp == NULL) {****  ****printf("Error opening file!\n");****  ****return 1;****  ****}****  ****// Moving the file pointer to the beginning of the file****  ****fseek(fp, 0, SEEK\_SET);****  ****// Reading data from the file****  ****fscanf(fp, "%s", buffer);****  ****printf("Data read from file: %s\n", buffer);****  ****// Closing the file****  ****fclose(fp);****  ****return 0;****  ****}**** |
| ****2]****Describe the role of file pointers in C programming. How are file pointers used to navigate and manipulate files? |
| ****Ans]**** In C programming, file pointers (**FILE\***) are used to manage and manipulate files. A file pointer is a special data type that represents a connection to a file. It allows you to perform various operations on files, such as reading from and writing to them, as well as navigating through their contents. **Role of File Pointers:** ****Opening Files****: File pointers are used to open files using the **fopen()** function. The **fopen()** function returns a file pointer that is used to perform operations on the opened file.  ****Navigating Through Files****: File pointers are used to move the file position indicator, which determines the current position within the file. This allows you to read from or write to specific locations within the file.  ****Reading from Files****: File pointers are used to read data from files using functions like **fscanf()**, **fgets()**, or **fread()**. These functions use the file pointer to determine the current position within the file from which to read data.  ****Writing to Files****: File pointers are used to write data to files using functions like **fprintf()**, **fputs()**, or **fwrite()**. These functions use the file pointer to determine the current position within the file at which to write data.  ****Closing Files****: File pointers are used to close files using the **fclose()** function. Closing a file releases any resources associated with the file and ensures that any pending data is written to disk. **Example:****#include <stdio.h>****int main() {** **FILE \*fp;** **char buffer[100];** **// Opening a file for reading** **fp = fopen("example.txt", "r");** **if (fp == NULL) {** **printf("Error opening file!\n");** **return 1;** **}** **// Moving the file pointer to the beginning of the file** **fseek(fp, 0, SEEK\_SET);** **// Reading data from the file** **fscanf(fp, "%s", buffer);** **printf("Data read from file: %s\n", buffer);** **// Closing the file** **fclose(fp);** **return 0;****}** |
| ****3]**** |
| In C programming, text files and binary files are two different types of files used to store data, and they are handled differently due to their distinct characteristics. **Text Files:** ****Human Readable****: Text files contain data that is stored in a human-readable format. Characters in text files represent textual data, and they can include letters, numbers, symbols, and control characters like newline (**\n**) and carriage return (**\r**).  ****Interpretation of Data****: Data in text files is interpreted as characters. When reading from or writing to a text file, the data is treated as characters, and any textual representation can be easily viewed and understood by humans.  ****Examples****: Examples of text files include **.txt**, **.c**, **.html**, **.csv**, and other files containing textual data. **Binary Files:** ****Binary Representation****: Binary files store data in a binary (non-text) format, consisting of sequences of bytes that can represent any type of data, including numbers, characters, structures, or even executable code. The data in binary files is not intended to be human-readable.  ****Raw Data Storage****: Binary files store data exactly as it is represented in memory, without any additional formatting or interpretation. This allows for efficient storage and retrieval of data in its raw form.  ****Examples****: Examples of binary files include image files (e.g., **.jpg**, **.png**), audio files (e.g., **.mp3**, **.wav**), video files (e.g., **.mp4**, **.avi**), database files, executable files (e.g., **.exe**), and more. **Opening and Processing:**Text Files:  * Text files are typically opened and processed using standard text file handling functions such as **fopen()**, **fclose()**, **fscanf()**, **fprintf()**, **fgets()**, and **fputs()**. * Data in text files is read and written as sequences of characters using functions designed to work with text data.  Binary Files:  * Binary files are opened and processed using similar file handling functions, but they often require different modes when opening the file (e.g., **"rb"** for reading and **"wb"** for writing). * Data in binary files is read and written using functions like **fread()** and **fwrite()**, which work with raw binary data represented as byte sequences.  **Example (Text File):** ****#include <stdio.h>****  ****int main() {****  ****FILE \*fp;****  ****char buffer[100];****  ****// Opening a text file for reading****  ****fp = fopen("example.txt", "r");****  ****if (fp == NULL) {****  ****printf("Error opening file!\n");****  ****return 1;****  ****}****  ****// Reading data from the text file****  ****fscanf(fp, "%s", buffer);****  ****printf("Data read from text file: %s\n", buffer);****  ****// Closing the text file****  ****fclose(fp);****  ****return 0;****  ****}****    ****#include <stdio.h>****  ****int main() {****  ****FILE \*fp;****  ****int number;****  ****// Opening a binary file for reading****  ****fp = fopen("data.bin", "rb");****  ****if (fp == NULL) {****  ****printf("Error opening file!\n");****  ****return 1;****  ****}****  ****// Reading data from the binary file****  ****fread(&number, sizeof(int), 1, fp);****  ****printf("Data read from binary file: %d\n", number);****  ****// Closing the binary file****  ****fclose(fp);****  ****return 0;****  ****}**** |
| ****4]****Explain the purpose of file modes in C programming. Provide examples of different file modes like "r", "w", "a", etc. |
| ****Ans]**** In C programming, file modes specify the intended operations that can be performed on a file when it is opened. The file mode is passed as a parameter to the **fopen()** function when opening a file. Each file mode determines whether the file will be opened for reading, writing, appending, or a combination of these operations. Understanding file modes is crucial for correctly accessing and manipulating files in C programs.  Here are some common file modes used in C programming:  ****"r" (Read)****:   * + Opens the file for reading.   + The file must exist; otherwise, **fopen()** will return **NULL**.   + The file pointer is positioned at the beginning of the file.   ****FILE \*fp = fopen("example.txt", "r");****  ****"w" (Write)****:   * Opens the file for writing. * If the file exists, its contents are truncated (deleted), and the file is opened as an empty file. * If the file does not exist, a new file is created. * The file pointer is positioned at the beginning of the file.   ****FILE \*fp = fopen("example.txt", "w");****  ****"a" (Append)****:   * Opens the file for appending. * If the file exists, new data is appended to the end of the file. * If the file does not exist, a new file is created. * The file pointer is positioned at the end of the file.   ****FILE \*fp = fopen("example.txt", "a");****  ****"r+" (Read and Write)****:   * Opens the file for both reading and writing. * The file must exist; otherwise, **fopen()** will return **NULL**. * The file pointer is positioned at the beginning of the file.   ****FILE \*fp = fopen("example.txt", "r+");****  ****w+" (Read and Write)****:   * Opens the file for both reading and writing. * If the file exists, its contents are truncated (deleted), and the file is opened as an empty file. * If the file does not exist, a new file is created. * The file pointer is positioned at the beginning of the file.   ****FILE \*fp = fopen("example.txt", "w+");****  ****"a+" (Read and Append)****:   * Opens the file for both reading and appending. * If the file exists, new data is appended to the end of the file. * If the file does not exist, a new file is created. * The file pointer is positioned at the end of the file * FILE \*fp = fopen("example.txt", "a+"); |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Part- B**

|  |
| --- |
| 1. **Hello World:** Print "Hello, World!" to the console. |
| Code  #include<stdio.h>  int main ()  {  printf("hellow world");  } |
| Output  WhatsApp Image 2024-03-23 at 23.18.05_b13633d1 |
| 2] **Factorial:** Calculate the factorial of a given number. |
| Code;#include<stdio.h>  int main ()  {  int x;  int pro=1,i;  scanf("%d",&x);  for(i=1;i<=x;i++)  {  pro=pro\*i;  }  printf("%d",pro);  }  WhatsApp Image 2024-03-23 at 23.54.17_457037c8 |
| 3] **Prime Numbers:** Determine whether a given number is prime |
| ans]#include <stdio.h>  #include <math.h>  int isPrime(int num) {  if (num < 2) {  return 0;  }  for (int i = 2; i <= sqrt(num); i++) {  if (num % i == 0) {  return 0; // Not a prime number  }  }  return 1; // Prime number  }  int main() {  int number;  printf("Enter a number: ");  scanf("%d", &number);  if (isPrime(number)) {  printf("%d is a prime number.\n", number);  } else {  printf("%d is not a prime number.\n", number);  }  return 0;  }  WhatsApp Image 2024-03-24 at 00.16.58_98506d19 |
| 4]4. **Fibonacci Series:** Generate the Fibonacci series up to a certain limit |
| #include <stdio.h>  void generateFibonacci(int limit) {  int prev = 0, current = 1, next;  printf("Fibonacci Series up to %d:\n", limit);  printf("%d, %d, ", prev, current);  while (1) {  next = prev + current;  if (next > limit) {  break;  }  printf("%d, ", next);  prev = current;  current = next;  }  }  int main() {  int limit;  printf("Enter the limit for Fibonacci series: ");  scanf("%d", &limit);  generateFibonacci(limit);  return 0;  }  WhatsApp Image 2024-03-24 at 00.36.40_16a6df77 |
| 5]#include<stdio.h>  int main()  {  int a,b;  printf("enter the a and b value ");  scanf("%d%d",&a,&b);  int c=0;  c=a+b;  printf("%d",c);  }  WhatsApp Image 2024-03-24 at 00.46.57_60486735 |
|  |
| 6] 6. **Reverse a Number:** Reverse the digits of a given number  #include <stdio.h>  int reverseNumber(int num) {  int reversedNumber = 0;  while (num != 0) {  int lastDigit = num % 10;  reversedNumber = reversedNumber \* 10 + lastDigit;  num /= 10;  }  return reversedNumber;  }  int main() {  int number;  // Input the number from the user  printf("Enter a number: ");  scanf("%d", &number);  // Reverse the number  int reversed = reverseNumber(number);  // Print the reversed number  printf("Reversed number: %d\n", reversed);  return 0;  } |
| 7]7. **Palindrome Check:** Check if a given number or string is a palindrome  #include <stdio.h>  #include <string.h>  // Function to check if a number is a palindrome  int isPalindromeNumber(int num) {  int reversed = 0, original = num;  // Reverse the number  while (num != 0) {  int digit = num % 10;  reversed = reversed \* 10 + digit;  num /= 10;  }  // Check if the original number is equal to its reversed version  return original == reversed;  }  // Function to check if a string is a palindrome  int isPalindromeString(char \*str) {  int length = strlen(str);  int i, j;  // Compare characters from beginning and end  for (i = 0, j = length - 1; i < j; i++, j--) {  if (str[i] != str[j]) {  return 0; // Not a palindrome  }  }  return 1; // Palindrome  }  int main() {  int choice;  printf("Enter 1 to check a number, 2 to check a string: ");  scanf("%d", &choice);  if (choice == 1) {  int number;  printf("Enter a number: ");  scanf("%d", &number);  if (isPalindromeNumber(number)) {  printf("%d is a palindrome.\n", number);  } else {  printf("%d is not a palindrome.\n", number);  }  } else if (choice == 2) {  char str[100];  printf("Enter a string: ");  scanf("%s", str);  if (isPalindromeString(str)) {  printf("%s is a palindrome.\n", str);  } else {  printf("%s is not a palindrome.\n", str);  }  } else {  printf("Invalid choice.\n");  }  return 0;  }  WhatsApp Image 2024-03-24 at 01.13.53_babe8133 |
| 8]8. **Area of Shapes:** Calculate the area of shapes like rectangle, triangle, and circle. |
| Ans] |
| Code;#include <stdio.h>  #include <math.h>  // Function to calculate the area of a rectangle  float areaRectangle(float length, float width) {  return length \* width;  }  // Function to calculate the area of a triangle  float areaTriangle(float base, float height) {  return 0.5 \* base \* height;  }  // Function to calculate the area of a circle  float areaCircle(float radius) {  return M\_PI \* radius \* radius;  }  int main() {  while(1)  {  int choice;  printf("Enter 1 for rectangle, 2 for triangle, 3 for circle: ");  scanf("%d", &choice);  if (choice == 1) {  float length, width;  printf("Enter length and width of the rectangle: ");  scanf("%f %f", &length, &width);  printf("Area of the rectangle: %.2f\n", areaRectangle(length, width));  } else if (choice == 2) {  float base, height;  printf("Enter base and height of the triangle: ");  scanf("%f %f", &base, &height);  printf("Area of the triangle: %.2f\n", areaTriangle(base, height));  } else if (choice == 3) {  float radius;  printf("Enter radius of the circle: ");  scanf("%f", &radius);  printf("Area of the circle: %.2f\n", areaCircle(radius));  } else {  printf("Invalid choice.\n");  }  }  return 0;  } |
| 9]9. **Simple Calculator:** Implement a basic calculator with arithmetic operations. |
| #include <stdio.h>  int main() {  char operator;  double num1, num2;  // Input operator and operands from user  printf("Enter operator (+, -, \*, /): ");  scanf(" %c", &operator);  printf("Enter two operands: ");  scanf("%lf %lf", &num1, &num2);  // Perform arithmetic operation based on the operator  switch (operator) {  case '+':  printf("Result: %.2lf + %.2lf = %.2lf\n", num1, num2, num1 + num2);  break;  case '-':  printf("Result: %.2lf - %.2lf = %.2lf\n", num1, num2, num1 - num2);  break;  case '\*':  printf("Result: %.2lf \* %.2lf = %.2lf\n", num1, num2, num1 \* num2);  break;  case '/':  // Check if second operand is not zero to avoid division by zero error  if (num2 != 0) {  printf("Result: %.2lf / %.2lf = %.2lf\n", num1, num2, num1 / num2);  } else {  printf("Error: Division by zero\n");  }  break;  default:  printf("Error: Invalid operator\n");  }  return 0;  } |
| 10]10. **Array Operations:** Perform operations like finding the largest/smallest element, sum,  and average of an array. |
| Code]#include <stdio.h>  #define MAX\_SIZE 100  int main() {  int arr[MAX\_SIZE];  int size, i;  int sum = 0;  int max, min;  printf("Enter size of the array (up to %d): ", MAX\_SIZE);  scanf("%d", &size);  printf("Enter elements of the array: ");  for (i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  // Initialize max and min with the first element of the array  max = min = arr[0];  // Calculate sum, maximum, and minimum  for (i = 0; i < size; i++) {  sum += arr[i];  if (arr[i] > max) {  max = arr[i];  }  if (arr[i] < min) {  min = arr[i];  }  }  // Calculate average  double average = (double)sum / size;  // Print results  printf("Sum of elements: %d\n", sum);  printf("Average of elements: %.2f\n", average);  printf("Largest element: %d\n", max);  printf("Smallest element: %d\n", min);  return 0;  } |
| 11]11. **String Operations:** Manipulate strings such as concatenation, copying, and  comparison.  #include <stdio.h>  #include <string.h>  #define MAX\_LENGTH 100  int main() {  char str1[MAX\_LENGTH], str2[MAX\_LENGTH];  // Input two strings from the user  printf("Enter the first string: ");  fgets(str1, MAX\_LENGTH, stdin);  printf("Enter the second string: ");  fgets(str2, MAX\_LENGTH, stdin);  // Remove newline characters from input  str1[strcspn(str1, "\n")] = '\0';  str2[strcspn(str2, "\n")] = '\0';  // Concatenate strings  strcat(str1, str2);  printf("Concatenated string: %s\n", str1);  // Copy strings  char copiedString[MAX\_LENGTH];  strcpy(copiedString, str1);  printf("Copied string: %s\n", copiedString);  // Compare strings  int comparisonResult = strcmp(str1, str2);  if (comparisonResult == 0) {  printf("Strings are equal.\n");  } else if (comparisonResult < 0) {  printf("First string is lexicographically less than second string.\n");  } else {  printf("First string is lexicographically greater than second string.\n");  }  return 0;  } |
| 12]12. **Linear Search:** Search for an element in an array using linear search.  #include <stdio.h>  #define MAX\_SIZE 100  // Function to perform linear search  int linearSearch(int arr[], int size, int key) {  for (int i = 0; i < size; i++) {  if (arr[i] == key) {  return i;  }  }  return -1;  }  int main() {  int arr[MAX\_SIZE];  int size, key;  // Input size of the array  printf("Enter size of the array (up to %d): ", MAX\_SIZE);  scanf("%d", &size);  // Input elements of the array  printf("Enter elements of the array: ");  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  // Input key to search  printf("Enter element to search: ");  scanf("%d", &key);  // Perform linear search  int index = linearSearch(arr, size, key);  // Check if the element is found or not  if (index != -1) {  printf("Element found at index %d.\n", index);  } else {  printf("Element not found in the array.\n");  }  return 0;  } |
| 13]13. **Binary Search:** Search for an element in a sorted array using binary search. |
| #include <stdio.h>  // Function to perform binary search  int binarySearch(int arr[], int size, int key) {  int left = 0;  int right = size - 1;  while (left <= right) {  int mid = left + (right - left) / 2;  // If key is present at the middle  if (arr[mid] == key) {  return mid;  }  // If key is greater, ignore left half  else if (arr[mid] < key) {  left = mid + 1;  }  // If key is smaller, ignore right half  else {  right = mid - 1;  }  }  // Element is not present in array  return -1;  }  int main() {  int arr[] = {2, 5, 7, 10, 14, 18, 20};  int size = sizeof(arr) / sizeof(arr[0]);  int key;  // Input key to search  printf("Enter element to search: ");  scanf("%d", &key);  // Perform binary search  int index = binarySearch(arr, size, key);  // Check if the element is found or not  if (index != -1) {  printf("Element found at index %d.\n", index);  } else {  printf("Element not found in the array.\n");  }  return 0;  } |
| 14]14. **Selection Sort:** Sort an array using the selection sort algorithm.  Code;#include <stdio.h>  // Function to perform selection sort  void selectionSort(int arr[], int size) {  for (int i = 0; i < size - 1; i++) {  // Find the minimum element in the unsorted part of the array  int min\_index = i;  for (int j = i + 1; j < size; j++) {  if (arr[j] < arr[min\_index]) {  min\_index = j;  }  }  // Swap the found minimum element with the first element  if (min\_index != i) {  int temp = arr[i];  arr[i] = arr[min\_index];  arr[min\_index] = temp;  }  }  }  int main() {  int arr[] = {64, 25, 12, 22, 11};  int size = sizeof(arr) / sizeof(arr[0]);  // Print the original array  printf("Original array: ");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  // Perform selection sort  selectionSort(arr, size);  // Print the sorted array  printf("Sorted array: ");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| 16. **Insertion Sort:** Sort an array using the insertion sort algorithm.   1. #include <stdio.h>   // Function to perform insertion sort  void insertionSort(int arr[], int size) {  int i, key, j;  for (i = 1; i < size; i++) {  key = arr[i];  j = i - 1;  // Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j = j - 1;  }  arr[j + 1] = key;  }  }  int main() {  int arr[] = {64, 25, 12, 22, 11};  int size = sizeof(arr) / sizeof(arr[0]);  // Print the original array  printf("Original array: ");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  // Perform insertion sort  insertionSort(arr, size);  // Print the sorted array  printf("Sorted array: ");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| 17. **Matrix Operations:** Perform matrix addition, subtraction, multiplication, and  transpose.  Code;  #include <stdio.h>  #define MAX\_SIZE 10  // Function to perform matrix addition  void matrixAddition(int mat1[][MAX\_SIZE], int mat2[][MAX\_SIZE], int result[][MAX\_SIZE], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  result[i][j] = mat1[i][j] + mat2[i][j];  }  }  }  // Function to perform matrix subtraction  void matrixSubtraction(int mat1[][MAX\_SIZE], int mat2[][MAX\_SIZE], int result[][MAX\_SIZE], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  result[i][j] = mat1[i][j] - mat2[i][j];  }  }  }  // Function to perform matrix multiplication  void matrixMultiplication(int mat1[][MAX\_SIZE], int mat2[][MAX\_SIZE], int result[][MAX\_SIZE], int rows1, int cols1, int cols2) {  for (int i = 0; i < rows1; i++) {  for (int j = 0; j < cols2; j++) {  result[i][j] = 0;  for (int k = 0; k < cols1; k++) {  result[i][j] += mat1[i][k] \* mat2[k][j];  }  }  }  }  // Function to perform matrix transpose  void matrixTranspose(int mat[][MAX\_SIZE], int transpose[][MAX\_SIZE], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  transpose[j][i] = mat[i][j];  }  }  }  // Function to display matrix  void displayMatrix(int mat[][MAX\_SIZE], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  printf("%d ", mat[i][j]);  }  printf("\n");  }  }  int main() {  int mat1[MAX\_SIZE][MAX\_SIZE], mat2[MAX\_SIZE][MAX\_SIZE], result[MAX\_SIZE][MAX\_SIZE], transpose[MAX\_SIZE][MAX\_SIZE];  int rows1, cols1, rows2, cols2;  // Input dimensions of matrix 1  printf("Enter number of rows and columns of matrix 1: ");  scanf("%d %d", &rows1, &cols1);  // Input elements of matrix 1  printf("Enter elements of matrix 1:\n");  for (int i = 0; i < rows1; i++) {  for (int j = 0; j < cols1; j++) {  scanf("%d", &mat1[i][j]);  }  }  // Input dimensions of matrix 2  printf("Enter number of rows and columns of matrix 2: ");  scanf("%d %d", &rows2, &cols2);  // Input elements of matrix 2  printf("Enter elements of matrix 2:\n");  for (int i = 0; i < rows2; i++) {  for (int j = 0; j < cols2; j++) {  scanf("%d", &mat2[i][j]);  }  }  // Perform addition if dimensions are compatible  if (rows1 == rows2 && cols1 == cols2) {  printf("Matrix Addition:\n");  matrixAddition(mat1, mat2, result, rows1, cols1);  displayMatrix(result, rows1, cols1);  } else {  printf("Matrix addition not possible: Dimensions are not compatible.\n");  }  // Perform subtraction if dimensions are compatible  if (rows1 == rows2 && cols1 == cols2) {  printf("Matrix Subtraction:\n");  matrixSubtraction(mat1, mat2, result, rows1, cols1);  displayMatrix(result, rows1, cols1);  } else {  printf("Matrix subtraction not possible: Dimensions are not compatible.\n");  }  // Perform multiplication if dimensions are compatible  if (cols1 == rows2) {  printf("Matrix Multiplication:\n");  matrixMultiplication(mat1, mat2, result, rows1, cols1, cols2);  displayMatrix(result, rows1, cols2);  } else {  printf("Matrix multiplication not possible: Dimensions are not compatible.\n");  }  // Perform transpose of matrix 1  printf("Transpose of Matrix 1:\n");  matrixTranspose(mat1, transpose, rows1, cols1);  displayMatrix(transpose, cols1, rows1);  return 0;  } |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Part- C**

|  |
| --- |
| **1.** |
| Code1 |
| 2]code    3]    4]     |  | | --- | | 18. **Title of the program:** Implement basic operations like insertion, deletion, and traversal in a linked list | | Code:  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {  // Allocate memory for new node  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));  if (new\_node == NULL) {  printf("Memory allocation failed.\n");  return;  }    new\_node->data = new\_data;  // Link new node to the current head  new\_node->next = \*head\_ref;  // Update head to point to new node  \*head\_ref = new\_node;  }  void deleteNode(struct Node\*\* head\_ref, int key) {  struct Node\* temp = \*head\_ref;  struct Node\* prev = NULL;    if (temp != NULL && temp->data == key) {  \*head\_ref = temp->next; // Change head  free(temp); // Free old head  return;  }  while (temp != NULL && temp->data != key) {  prev = temp;  temp = temp->next;  }  if (temp == NULL)  return;  prev->next = temp->next;  free(temp); // Free memory  }  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d ", node->data);  node = node->next;  }  printf("\n");  }  int main() {  // Initialize an empty linked list  struct Node\* head = NULL;    insertAtBeginning(&head, 7);  insertAtBeginning(&head, 5);  insertAtBeginning(&head, 3);  insertAtBeginning(&head, 1);  printf("Initial linked list: ");  printList(head);  deleteNode(&head, 3);  printf("Linked list after deletion: ");  printList(head);  return 0;  } |  |  | | --- | | Outpu | | 19. **Title of the program:** Implement basic stack operations like push, pop, display and peek | | Code:  #include <stdio.h>  #include <stdlib.h>  struct StackNode {  int data;  struct StackNode\* next;  };  struct StackNode\* createNode(int data) {  struct StackNode\* newNode = (struct StackNode\*)malloc(sizeof(struct StackNode));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  int isEmpty(struct StackNode\* root) {  return (root == NULL);  }  void push(struct StackNode\*\* root, int data) {  struct StackNode\* newNode = createNode(data);  newNode->next = \*root;  \*root = newNode;  printf("%d pushed to stack.\n", data);  }  int pop(struct StackNode\*\* root) {  if (isEmpty(\*root)) {  printf("Stack underflow.\n");  exit(EXIT\_FAILURE);  }  struct StackNode\* temp = \*root;  \*root = (\*root)->next;  int popped = temp->data;  free(temp);  return popped;  }  void display(struct StackNode\* root) {  if (isEmpty(root)) {  printf("Stack is empty.\n");  return;  }  printf("Stack contents: ");  while (root != NULL) {  printf("%d ", root->data);  root = root->next;  }  printf("\n");  }  int peek(struct StackNode\* root) {  if (isEmpty(root)) {  printf("Stack is empty.\n");  exit(EXIT\_FAILURE);  }  return root->data;  }  int main() {  struct StackNode\* root = NULL;  // Push some elements onto the stack  push(&root, 10);  push(&root, 20);  push(&root, 30);    display(root);    printf("Top element of stack: %d\n", peek(root));    printf("%d popped from stack.\n", pop(&root));  printf("%d popped from stack.\n", pop(&root));  display(root);  return 0;  }  Output: | | 20. **Title of the program:**  Circular Queue, Double-ended Queue, Priority Queue | | Code:  Circular Queue:  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 5  // Circular Queue structure  struct CircularQueue {  int items[MAX\_SIZE];  int front, rear;  };  // Initialize circular queue  void initCircularQueue(struct CircularQueue\* queue) {  queue->front = -1;  queue->rear = -1;  }  // Check if circular queue is full  int isFull(struct CircularQueue\* queue) {  return ((queue->rear + 1) % MAX\_SIZE == queue->front);  }  // Check if circular queue is empty  int isEmpty(struct CircularQueue\* queue) {  return (queue->front == -1);  }  // Enqueue an element into circular queue  void enqueue(struct CircularQueue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full.\n");  return;  }  if (isEmpty(queue))  queue->front = 0;  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  }  // Dequeue an element from circular queue  int dequeue(struct CircularQueue\* queue) {  int value;  if (isEmpty(queue)) {  printf("Queue is empty.\n");  exit(EXIT\_FAILURE);  }  value = queue->items[queue->front];  if (queue->front == queue->rear)  queue->front = queue->rear = -1;  else  queue->front = (queue->front + 1) % MAX\_SIZE;  return value;  }  // Display circular queue elements  void display(struct CircularQueue\* queue) {  int i;  if (isEmpty(queue)) {  printf("Queue is empty.\n");  return;  }  printf("Queue elements: ");  for (i = queue->front; i != queue->rear; i = (i + 1) % MAX\_SIZE)  printf("%d ", queue->items[i]);  printf("%d\n", queue->items[i]);  }  // Main function  int main() {  struct CircularQueue queue;  initCircularQueue(&queue);  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 30);  enqueue(&queue, 40);  display(&queue);  printf("Dequeued element: %d\n", dequeue(&queue));  printf("Dequeued element: %d\n", dequeue(&queue));  display(&queue);  enqueue(&queue, 50);  enqueue(&queue, 60);  display(&queue);  return 0;  }  Double-ended Queue (Deque):  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 5  // Deque structure  struct Deque {  int items[MAX\_SIZE];  int front, rear;  };  // Initialize deque  void initDeque(struct Deque\* deque) {  deque->front = -1;  deque->rear = -1;  }  // Check if deque is full  int isFull(struct Deque\* deque) {  return ((deque->front == 0 && deque->rear == MAX\_SIZE - 1) || (deque->front == deque->rear + 1));  }  // Check if deque is empty  int isEmpty(struct Deque\* deque) {  return (deque->front == -1);  }  // Insert element at front of deque  void insertFront(struct Deque\* deque, int value) {  if (isFull(deque)) {  printf("Deque is full.\n");  return;  }  if (deque->front == -1) {  deque->front = 0;  deque->rear = 0;  } else if (deque->front == 0)  deque->front = MAX\_SIZE - 1;  else  deque->front = deque->front - 1;  deque->items[deque->front] = value;  }  // Insert element at rear of deque  void insertRear(struct Deque\* deque, int value) {  if (isFull(deque)) {  printf("Deque is full.\n");  return;  }  if (deque->front == -1) {  deque->front = 0;  deque->rear = 0;  } else if (deque->rear == MAX\_SIZE - 1)  deque->rear = 0;  else  deque->rear = deque->rear + 1;  deque->items[deque->rear] = value;  }  // Delete element from front of deque  int deleteFront(struct Deque\* deque) {  int value;  if (isEmpty(deque)) {  printf("Deque is empty.\n");  exit(EXIT\_FAILURE);  }  value = deque->items[deque->front];  if (deque->front == deque->rear)  deque->front = deque->rear = -1;  else if (deque->front == MAX\_SIZE - 1)  deque->front = 0;  else  deque->front = deque->front + 1;  return value;  }  // Delete element from rear of deque  int deleteRear(struct Deque\* deque) {  int value;  if (isEmpty(deque)) {  printf("Deque is empty.\n");  exit(EXIT\_FAILURE);  }  value = deque->items[deque->rear];  if (deque->front == deque->rear)  deque->front = deque->rear = -1;  else if (deque->rear == 0)  deque->rear = MAX\_SIZE - 1;  else  deque->rear = deque->rear - 1;  return value;  }  // Display deque elements  void display(struct Deque\* deque) {  int i;  if (isEmpty(deque)) {  printf("Deque is empty.\n");  return;  }  printf("Deque elements: ");  if (deque->front <= deque->rear) {  for (i = deque->front; i <= deque->rear; i++)  printf("%d ", deque->items[i]);  } else {  for (i = deque->front; i < MAX\_SIZE; i++)  printf("%d ", deque->items[i]);  for (i = 0; i <= deque->rear; i++)  printf("%d ", deque->items[i]);  }  printf("\n");  }  // Main function  int main() {  struct Deque deque;  initDeque(&deque);  insertFront(&deque, 10);  insertRear(&deque, 20);  insertRear(&deque, 30);  insertFront(&deque, 40);  display(&deque);  printf("Deleted element from front: %d\n", deleteFront(&deque));  printf("Deleted element from rear: %d\n", deleteRear(&deque));  display(&deque);  return 0;  }  Priority Queue:  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 5  // Priority Queue structure  struct PriorityQueue {  int items[MAX\_SIZE];  int front, rear;  };  // Initialize priority queue  void initPriorityQueue(struct PriorityQueue\* queue) {  queue->front = -1;  queue->rear = -1;  }  // Check if priority queue is full  int isFull(struct PriorityQueue\* queue) {  return (queue->rear == MAX\_SIZE - 1);  }  // Check if priority queue is empty  int isEmpty(struct PriorityQueue\* queue) {  return (queue->front == -1);  }  // Enqueue an element into priority queue  void enqueue(struct PriorityQueue\* queue, int value) {  int i;  if (isFull(queue)) {  printf("Queue is full.\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  queue->rear = 0;  queue->items[queue->rear] = value;  } else {  for (i = queue->rear; i >= 0; i--) {  if (value > queue->items[i])  queue->items[i + 1] = queue->items[i];  else  break;  }  queue->items[i + 1] = value;  queue->rear++;  }  }  // Dequeue an element from priority queue  int dequeue(struct PriorityQueue\* queue) {  int value;  if (isEmpty(queue)) {  printf("Queue is empty.\n");  exit(EXIT\_FAILURE);  }  value = queue->items[queue->front];  if (queue->front == queue->rear)  queue->front = queue->rear = -1;  else  queue->front++;  return value;  }  // Display priority queue elements  void display(struct PriorityQueue\* queue) {  int i;  if (isEmpty(queue)) {  printf("Queue is empty.\n");  return;  }  printf("Priority Queue elements: ");  for (i = queue->front; i <= queue->rear; i++)  printf("%d ", queue->items[i]);  printf("\n");  }  // Main function  int main() {  struct PriorityQueue queue;  initPriorityQueue(&queue);  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 15);  enqueue(&queue, 30);  enqueue(&queue, 25);  display(&queue);  printf("Dequeued element: %d\n", dequeue(&queue));  printf("Dequeued element: %d\n", dequeue(&queue));  display(&queue);  return 0;  }  Output: |   21]  Code;#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node  {  double data;  struct node \*r\_link,\*l\_link;  };  struct node\* creat\_node()  {  struct node \*newnode=(struct node\*)malloc(sizeof(struct node));  if(newnode==NULL)  {  printf("hi");  }  else  {  scanf("%lf",&newnode->data);  newnode->r\_link=NULL;  newnode->l\_link=NULL;  }  return newnode;  }  struct node\* insert\_end(struct node \*head)  {  struct node \*newnode=creat\_node();  if(head==NULL)  {  head=newnode;  }  else  {  struct node \*cur=head;  while(cur->l\_link!=NULL)  {  cur=cur->l\_link;  }  cur->l\_link=newnode;  newnode->r\_link=cur;  }  return head;  }  void display\_f(struct node \*head)  {  struct node \*cur=NULL;  if (head==NULL)  {  printf("hi");  }  else  {  cur=head;  while(cur!=NULL)  {  printf("%0.2lf ",cur->data);  cur=cur->l\_link;  }  }  }  void display\_r(struct node \*head)  {  struct node \*cur=NULL;  if(head==NULL)  {  printf("hu");  }  else  {  cur=head;  while(cur->l\_link!=NULL)  {  cur=cur->l\_link;  }  while(cur!=NULL)  {  printf("%0.2lf ",cur->data);  cur=cur->r\_link;  }  }  }  int main() {  struct node \*head=NULL;  int n;  scanf("%d",&n);  if(n>=1)  {  for(int i=0;i<n;i++)  {  head=insert\_end(head);  }  printf("DLL in forward direction:\n");  display\_f(head);  printf("\n");  printf("\nDLL in reverse direction:\n");  display\_r(head);  }  else  {  printf("N should be positive.\n");  }  return 0;  }  22]#include <stdio.h>  #include <stdlib.h>  struct node  {  char data;  struct node \*l\_link, \*r\_link;  };  struct node\* create()  {  struct node \*newnode = (struct node \*)malloc(sizeof(struct node));  if (newnode == NULL)  {  printf("Memory allocation failed.");  exit(1);  }  else  {  scanf(" %c", &newnode->data);  newnode->l\_link = NULL;  newnode->r\_link = NULL;  }  return newnode;  }  struct node\* insert\_front(struct node \*head)  {  struct node \*newnode = create();  if (head == NULL)  {  head = newnode;  head->l\_link = head;  head->r\_link = head;  }  else  {  newnode->r\_link = head;  newnode->l\_link = head->l\_link;  head->l\_link->r\_link = newnode;  head->l\_link = newnode;  head = newnode;  }  return head;  }  struct node\* delete\_front(struct node \*head)  {  if (head == NULL)  {  printf("empty\n");  }  else if (head->r\_link == head) // Only one node in the list  {  free(head);  head = NULL;  }  else  {  struct node \*temp = head;  head = head->r\_link;  head->l\_link = temp->l\_link;  temp->l\_link->r\_link = head;  free(temp);  }  return head;  }  struct node\* delete\_end(struct node \*head)  {  if (head == NULL)  {  printf("empty\n");  }  else if (head->r\_link == head) // Only one node in the list  {  free(head);  head = NULL;  }  else  {  struct node \*end\_node = head->l\_link;  head->l\_link = end\_node->l\_link;  end\_node->l\_link->r\_link = head;  free(end\_node);  }  return head;  }  void display\_f(struct node \*head)  {  if (head == NULL)  {  printf("empty\n");  return;  }  struct node \*cur = head;  do  {  printf("%c ", cur->data);  cur = cur->r\_link;  } while (cur != head);  printf("\n");  }  int main() {  struct node \*head = NULL;  int n, h;  scanf("%d", &n);  if (n > 1)  {  for (int i = 0; i < n; i++)  {  head = insert\_front(head);  }  printf("DLL before deletion:\n");  display\_f(head);  scanf("%d", &h);  for (int i = 0; i < h; i++)  {  head = delete\_front(head);  }  for (int i = 0; i < h; i++)  {  head = delete\_end(head);  }  printf("\n");  printf("DLL after deletion:\n");  display\_f(head);  }  else  {  printf("N should be positive.\n");  }  return 0;  }  23]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node  {  int data;  struct node \*r\_link,\*l\_link;  };  struct node\* creat\_node()  {  struct node \*newnode=(struct node \*)malloc(sizeof(struct node));  if(newnode==NULL)  {  printf("memmoedkcm");  }  else{  scanf("%d",&newnode->data);  newnode->r\_link=NULL;  newnode->l\_link=NULL;  }  return newnode;  }  struct node\* insert\_end(struct node \*head)  {  struct node \*newnode = creat\_node();  struct node \*cur=NULL;  if(head==NULL)  head=newnode;  else  {  cur=head;  while(cur->r\_link!=NULL)  cur=cur->r\_link;  cur->r\_link=newnode;  newnode->l\_link=cur;  }  return head;  }  int count\_(struct node \*head)  {  int count=0;  struct node \*cur=NULL;  cur=head;  while(cur!=NULL)  {  count++;  cur=cur->r\_link;  }  return count;  }  struct node\* insert\_pos(struct node \*head,int pos)  {  struct node \*newn = creat\_node();  int count=count\_(head);  struct node \*cur,\*next;  if(head==NULL)  head=newn;  else if (pos<=1)  {  newn->r\_link=head;  head->l\_link=newn;  head=newn;  }  else if(pos>1&&pos<=count)  {  cur=head;  for(int i=1;i<pos-1;i++)  cur=cur->r\_link;  next=cur->r\_link;  newn->r\_link=cur->r\_link;  cur->r\_link=newn;  newn->l\_link=cur;  next->l\_link=newn;  }  else if(pos>=count+1)  {  cur=head;  while(cur->r\_link!=NULL)  cur=cur->r\_link;  cur->r\_link=newn;  newn->l\_link=cur;  }  return head;  }  struct node\* delete\_pos(struct node \*head,int pos)  {  struct node \*cur=NULL,\*temp=NULL;  int count = count\_(head);  if(head==NULL)  return NULL;  else if(head->r\_link==NULL&&pos==1)  {  free(head);  head=NULL;  }  else if(pos<=1&&pos>count)  {  return head;  }  else if(pos==count)  {  struct node \*prev=NULL;  cur=head;  while(cur->r\_link!=NULL)  cur=cur->r\_link;  prev=cur->l\_link;  free(cur);  prev->r\_link=NULL;  }  else if(pos>1&&pos<count)  {  cur=head;  for(int i=1;i<pos;i++)  {  cur=cur->r\_link;  }  struct node \*prev=cur->l\_link;  temp=cur->r\_link;  prev->r\_link=cur->r\_link;  temp->l\_link=prev;  free(cur);  }  return head;  }  void display(struct node \*head)  {  struct node\* cur=NULL;  if(head==NULL)  printf("NULL");  else  {  cur=head;  while(cur!=NULL)  {  printf("%d ",cur->data);  cur=cur->r\_link;  }  }  }  int main() {  struct node \*head=NULL;  int n,k,l;  scanf("%d",&n);  if(n>0)  {  for(int i=0;i<n;i++)  head=insert\_end(head);  scanf("%d",&k);  head=insert\_pos(head,k);  scanf("%d",&l);  display(head);  printf("\n\n");  head=delete\_pos(head,l);  display(head);  }  else  {  printf("invalis inptu");  }  /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/  return 0;  }    24]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node  {  char name[20];  int year;  struct node \*r\_link,\*l\_link;    };  struct node\* creat\_node()  {  struct node newnode=(struct node)malloc(sizeof(struct node));  if(newnode==NULL)  {  printf("hi");  }  else  {  scanf("%s%d",newnode->name,&newnode->year);  newnode->r\_link=NULL;  newnode->l\_link=NULL;  }  return newnode;  }  struct node\* insert\_end(struct node \*head)  {  struct node \*newnode=creat\_node();  if(head==NULL)  {  head=newnode;  }  else  {  struct node \*cur=head;  while(cur->l\_link!=NULL)  {  cur=cur->l\_link;  }  cur->l\_link=newnode;  newnode->r\_link=cur;  }  return head;  }  void display\_f(struct node \*head)  {  struct node \*cur=NULL;  if (head==NULL)  {  printf("hi");  }  else  {  cur=head;  while(cur!=NULL)  {  printf("%s %d \n",cur->name,cur->year);  cur=cur->l\_link;  }    }    }  void find(struct node \*head,int date)  {  struct node \*cur=NULL;  int count=0;  if(head==NULL)  {  printf("empty");  }  else  {  cur=head;  while(cur!=NULL)  {  if(cur->year==date)  {  printf("%s %d \n",cur->name,cur->year);  count++;  }  cur=cur->l\_link;  }  if(count==0)  {  printf("No web series in %d\n",date);  }  printf("\n");    printf("%d",count);  }  }  int main() {  struct node \*head=NULL;  int n,date;  scanf("%d",&n);  if(n>=1)  {  for(int i=0;i<n;i++)  {  head=insert\_end(head);  }  display\_f(head);  printf("\n");  scanf("%d",&date);  find(head,date);    }    return 0;  }    25]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node  {  int j;  float amt;  struct node \*r\_link,\*l\_link;  };  struct node\* creat\_node()  {  struct node \*newnode=(struct node\*)malloc(sizeof(struct node));  if(newnode==NULL)  {  printf("hi");  }  else  {  scanf("%d %f",&newnode->j,&newnode->amt);  newnode->r\_link=NULL;  newnode->l\_link=NULL;  }  return newnode;  }  struct node\* insert\_end(struct node \*head)  {  struct node \*newnode=creat\_node();  if(head==NULL)  {  head=newnode;  }  else  {  struct node \*cur=head;  while(cur->l\_link!=NULL)  {  cur=cur->l\_link;  }  cur->l\_link=newnode;  newnode->r\_link=cur;  }  return head;  }  void delete\_f(struct node \*head)  {  struct node \*cur = NULL;  float count=0;  if (head == NULL)  {  printf("List is empty.");  }  else if (head->l\_link == NULL)  {  count=count+head->amt;  free(head);  head = NULL;  }  else  {  cur = head;  head = head->l\_link;  head->r\_link = NULL;  count=count+cur->amt;  free(cur);  }  struct node \*prev = NULL;  if (head == NULL)  {  printf("List is empty.");  }  else if (head->l\_link == NULL)  {  free(head);  count=count+head->amt;  head = NULL;  }  else  {  cur = head;  while (cur->l\_link != NULL)  {  prev = cur;  cur = cur->l\_link;  }  prev->l\_link = NULL;  count=count+cur->amt;  free(cur);  }  if (head==NULL)  {  printf("hi");  }  else  {  cur=head;  while(cur!=NULL)  {  printf("%d %0.2f \n",cur->j,cur->amt);  cur=cur->l\_link;  }  }  printf("Total loot = %0.2f",count);  }  int main() {  struct node \*head=NULL;  int n;  scanf("%d",&n);  if(n>=5)  {  for(int i=0;i<n;i++)  {  head=insert\_end(head);  }  delete\_f(head);  }  else  {  printf("Kaalia cannot loot houses\n");  }  return 0;  }    26]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node  {  char name[20], gender[20];  int age;  struct node \*r\_link, \*l\_link;  };  struct node\* create\_node()  {  struct node \*newnode = (struct node\*)malloc(sizeof(struct node));  if (newnode == NULL)  {  printf("Memory allocation failed!");  return NULL;  }  else  {  scanf("%s%s%d", newnode->name, newnode->gender, &newnode->age);  newnode->r\_link = NULL;  newnode->l\_link = NULL;  }  return newnode;  }  struct node\* insert\_end(struct node \*head)  {  struct node \*newnode = create\_node();  if (head == NULL)  {  head = newnode;  }  else  {  struct node \*cur = head;  while (cur->l\_link != NULL)  {  cur = cur->l\_link;  }  cur->l\_link = newnode;  newnode->r\_link = cur;  }  return head;  }  struct node\* insert\_pos(struct node \*head)  {  struct node \*newnode = create\_node();  struct node \*cur=NULL,\*prev=NULL;  if(head->age==newnode->age)  {  head->r\_link = newnode;  newnode->l\_link = head;  head = newnode;  }  else  {  cur=head;  while(cur!=NULL)  {  if(cur->age==newnode->age)  {  cur->r\_link=newnode;  newnode->l\_link=cur;  prev->l\_link=newnode;  newnode->r\_link=prev;  }  prev=cur;  cur=cur->l\_link;  }  }  return head;  }  void display(struct node \*head)  {  struct node \*cur = NULL;  if (head == NULL)  {  printf("List is empty!");  }  else  {  cur = head;  while (cur != NULL)  {  printf("%s %s %d \n", cur->name, cur->gender, cur->age);  cur = cur->l\_link;  }  }  }  int main()  {  struct node \*head = NULL;  int n, g,count=0;  char s[]="female";  scanf("%d", &n);  scanf("%d", &g);  if (n >= 5)  {  for (int i = 0; i < n; i++)  {  head = insert\_end(head);  }  for(int i=0;i<g;i++)  {  head=insert\_pos(head);  }  display(head);  if(head==NULL)  {  printf("null");  }  else  {  struct node \*cur=head;  while(cur!=NULL)  {  if(strcmp(cur->gender,s)==0)  {  break;  }  count++;  cur=cur->l\_link;  }  printf("%d",count+1);  }  }  else  {  printf("Soldiers cannot go on a war\n");  }  return 0;  }    27]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node  {  int data;  struct node \*l\_link,\*r\_link;  };  struct node\* creat\_node()  {  struct node newnode=(struct node)malloc(sizeof(struct node));  if(newnode==NULL)  {  printf("hi");  }  else  {  scanf("%d",&newnode->data);  newnode->r\_link=newnode;  newnode->l\_link=newnode;  }  return newnode;  }  struct node\* insert\_front(struct node \*head)  {  struct node \*newnode=creat\_node();  struct node \*last=NULL;  if(head==NULL)  {  head=newnode;  }  else  {  last=head->l\_link;  newnode->r\_link=head;  head->l\_link=newnode;  last->r\_link=newnode;  newnode->l\_link=last;  head=newnode;  }  return head;  }  void display(struct node \*head) {  struct node \*cur = NULL, \*last = NULL;  if (head == NULL) {  printf("empty\n");  } else {  last = head->l\_link;  cur = last;  do {  printf("%d ", cur->data); // Added a space after %d for better readability  cur = cur->l\_link;  }  while (cur!=last);  }  }  void display\_f(struct node \*head)  {  struct node \*cur=NULL;  if(head==NULL)  {  printf("empty\n");  }  else  {  cur=head;  do  {  printf("%d ",cur->data);  cur=cur->r\_link;  }  while(cur!=head);    }  }  int main() {  struct node \*head=NULL;  int n;  scanf("%d",&n);  if(n>=1)  {  for(int i=0;i<n;i++)  {  head=insert\_front(head);  }  printf("CDLL in forward direction is:\n");  display\_f(head);  printf("\n");  printf("\n");  printf("CDLL in reverse direction is:\n");  display(head);  }  else  {  printf("Invalid input\n");  }    /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/  return 0;  }    28]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node  {  double data;  struct node \*left\_link, \*right\_link;  };  struct node\* create\_node()  {  struct node \*newn = NULL;  newn = (struct node\*)malloc(sizeof(struct node));  if(newn != NULL)  {  scanf("%lf", &newn->data);  newn->left\_link = newn;  newn->right\_link = newn;  }  return newn;  }  struct node\* insert\_end\_cdll(struct node \*head)  {  struct node \*last = NULL, \*newn = NULL;  newn = create\_node();  if(head == NULL)  head = newn;  else  {  last = head->left\_link;  last->right\_link = newn;  newn->left\_link = last;  newn->right\_link = head;  head->left\_link = newn;  }  return head;  }  struct node\* delete\_end\_cdll(struct node \*head)  {  struct node \*last = NULL, \*prev = NULL;  if(head == NULL)  return NULL;  else if(head->right\_link == head)  {  free(head);  head = NULL;  }  else  {  last = head->left\_link;  prev = last->left\_link;  prev->right\_link = head;  head->left\_link = prev;  free(last);  }  return head;  }  struct node\* delete\_front\_cdll(struct node \*head)  {  struct node \*temp = NULL, \*last = NULL;  if(head == NULL)  return NULL;  else if(head->right\_link == head)  {  free(head);  head = NULL;  }  else  {  last = head->left\_link;  temp = head;  head = head->right\_link;  last->right\_link = head;  head->left\_link = last;  free(temp);  }  return head;  }  void display\_cdll(struct node \*head)  {  struct node \*cur = NULL;  if(head == NULL)  printf("Empty.\n");  else  {  cur = head;  do  {  printf("%.3lf ", cur->data);  cur = cur->right\_link;  }while(cur != head);  }  }  int main() {  /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/    struct node \*head = NULL;  int i, j, n, k;  scanf("%d", &n);  if(n >= 0 && n<= 100)  {  for(i=0; i<n; i++)  head = insert\_end\_cdll(head);  printf("CDLL is:\n");  display\_cdll(head);  scanf("%d", &k);  if(k<= 100 && k>= 0)  {  for(j=0; j<k; j++)  {  head = delete\_end\_cdll(head);  head = delete\_front\_cdll(head);  }  printf("\n\nCDLL after deletions is:\n");  display\_cdll(head);  }  }  else  printf("N should be positive.\n");    return 0;  }    29]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  typedef struct node  {  char data;  struct node \*r\_link,\*l\_link;  }\*N;  N create\_node()  {  N newn=(N)malloc(sizeof(struct node));  if(newn==NULL)  return NULL;  else  {  scanf(" %c",&newn->data);  newn->r\_link=newn;  newn->l\_link=newn;  }  return newn;  }  N insert\_front(N head,N newn)  {  // N newn=create\_node();  if(head==NULL)  head=newn;  else  {  N last=head->l\_link;  head->l\_link=newn;  newn->r\_link=head;  newn->l\_link=last;  last->r\_link=newn;  head=newn;  }  return head;  }  int count\_nodes(N head)  {  int count=0;  N cur=head;  if(head==NULL)  return count;  else  {  do  {  count++;  cur=cur->r\_link;  }  while(cur!=head);  }  return count;  }  N insert\_end(N head,N newn)  {  N last=NULL;  // N newn=create\_node();  if(head==NULL)  head=newn;  else  {  last=head->l\_link;    last->r\_link=newn;  newn->l\_link=last;  newn->r\_link=head;  head->l\_link=newn;  }  return head;  }  N insert\_pos(N head,N newn,int pos)  {  N cur=NULL,prev=NULL;  //N newn=create\_node();  int count=count\_nodes(head);  if(head==NULL)  {  head=newn;  }  else if(pos<=1)  {  head=insert\_front(head,newn);  }  else if(pos>count)  {  head=insert\_end(head,newn);  }  else if(pos>1&&pos<=count)  {  cur=head;  for(int i=0;i<pos-1;i++)  cur=cur->r\_link;  prev=cur->l\_link;    prev->r\_link=newn;  newn->l\_link=prev;  newn->r\_link=cur;  cur->l\_link=newn;  }  return head;  }  N delete\_pos(N head,int pos)  {  N cur=NULL,last=NULL,temp=NULL,prev=NULL;  int count =count\_nodes(head);  if(head==NULL)  return NULL;  else if(head->r\_link==head&&pos==1)  {  free(head);  head=NULL;  }  else if(pos==1)  {  N last = head->l\_link;    temp=head;  head=head->r\_link;  head->l\_link=last;  last->r\_link=head;  free(temp);  }  else if(pos==count)  {  last=head->l\_link;    prev=last->l\_link;  prev->r\_link=head;  head->l\_link=prev;  free(last);  }  else if(pos>1&&pos<count)  {  cur=head;  for(int i=1;i<pos;i++)  {  cur=cur->r\_link;  }  prev=cur->l\_link;  temp=cur->r\_link;  prev->r\_link=cur->r\_link;  temp->l\_link=prev;  free(cur);  }  return head;  }  void display(N head)  {  N cur=NULL;  if(head==NULL)  printf("Empty.\n");  else  {  cur=head;  do  {  printf("%c ",cur->data);  cur=cur->r\_link;  }  while(cur!=head);  }  }  int main() {  N head=NULL;  int n,k,l;  scanf("%d",&n);  if(n>0)  {  for(int i=0;i<n;i++)  {  N newn=create\_node();  head=insert\_front(head,newn);  }  printf("CDLL is:\n");  display(head);  scanf("%d",&k);  N newn=create\_node();  head=insert\_pos(head,newn,k);  printf("\n\nCDLL after insertion is:\n");  display(head);  scanf("%d",&l);  head=delete\_pos(head,l);  printf("\n\nCDLL after deletion is:\n");  display(head);  }  else  printf("Invalid input");  /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/  return 0;  }    30]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node  {  char song[20],singer[15];  float time;  int year;  struct node \*l\_link,\*r\_link;  };  struct node \*create()  {  struct node newnode=(struct node)malloc(sizeof(struct node));  if(newnode==NULL)  printf("Memory not allocated\n");  else  {  scanf("%s%s%f%d",newnode->song,newnode->singer,&newnode->time,&newnode->year);  newnode->r\_link=newnode;  newnode->l\_link=newnode;  }  return newnode;  }  struct node\* insert\_end(struct node \*head)  {  struct node \*last=NULL;  struct node \*newnode=create();  if(head==NULL)  head=newnode;  else  {  last=head->l\_link;  last->r\_link=newnode;  newnode->l\_link=last;  newnode->r\_link=head;  head->l\_link=newnode;  }  return head;  }  void display\_singer(struct node \*head,char key[15])  {  struct node \*cur=NULL;  int status=0;  if(head==NULL)  printf("List is empty\n");  else  {  cur=head;  do  {  if(strcmp(cur->singer,key)==0)  {  printf("%s %s %.2f %d\n",cur->song,cur->singer,cur->time,cur->year);  status=1;  }  cur=cur->r\_link;  }while(cur!=head);  if(status==0)  printf("No songs of %s to display\n",key);  }  }  void display(struct node \*head)  {  struct node \*cur=NULL;  if(head==NULL)  printf("List is empty\n");  else  {  cur=head;  do  {  printf("%s %s %.2f %d\n",cur->song,cur->singer,cur->time,cur->year);  cur=cur->r\_link;  }while(cur!=head);  }  }  struct node \*delete(struct node \*head,char s\_key[15])  {  struct node \*cur,\*prev,\*next,\*last;  int status=0;  if(head==NULL)  return NULL;  else if(head->r\_link==head&&strcmp(s\_key,head->song)==0)  {  free(head);  head=NULL;  status=1;  }  else if(strcmp(s\_key,head->song)==0)  {  cur=head;  last=head->l\_link;  head=head->r\_link;  free(cur);  last->r\_link=head;  head->l\_link=last;  status=1;  }  else  {  cur=head;  do  {  if(strcmp(s\_key,cur->song)==0)  break;  cur=cur->r\_link;  }while(cur!=head);  if(cur==head)  status=0;  else  {  printf("Deleted %s %s %.2f %d\n",cur->song,cur->singer,cur->time,cur->year);  status=1;  prev=cur->l\_link;  next=cur->r\_link;  prev->r\_link=next;  next->l\_link=prev;  free(cur);  }  if(status==0)  printf("No song %s to delete\n",s\_key);  }  return head;  }    int main() {  int n;  struct node \*head=NULL;  char key[15],s\_key[15];  scanf("%d",&n);  if(n>0)  {  for(int i=0;i<n;i++)  head=insert\_end(head);  scanf("%s%s",key,s\_key);  display(head);  printf("\n");  display\_singer(head,key);  printf("\n");  head=delete(head,s\_key);  }  return 0;  }  31]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node  {  char www[30],comp[15];  int year;  struct node \*l\_link,\*r\_link;  };  struct node \*create()  {  struct node \*newnode=(struct node\*)malloc(sizeof(struct node));  if(newnode==NULL)  printf("Memory not allocated\n");  else  {  scanf("%s%s%d",newnode->www,newnode->comp,&newnode->year);  newnode->r\_link=NULL;  newnode->l\_link=NULL;  }  return newnode;  }  struct node\* insert\_end(struct node \*head)  {  struct node \*newnode=create();  if(head==NULL)  {  head=newnode;  }  else  {  struct node \*cur=head;  while(cur->l\_link!=NULL)  {  cur=cur->l\_link;  }  cur->l\_link=newnode;  newnode->r\_link=cur;  }  return head;  }  void dislay(struct node \*head,int k,int d)  {  struct node \*cur=NULL;  if(head==NULL)  {  printf("null");  }  else{  cur=head;  for(int i=0;i<k;i++)  {  cur=cur->l\_link;    }  printf("%s %s %d\n\n",cur->www,cur->comp,cur->year);  for(int i=0;i<d;i++)  {  cur=cur->r\_link;  }  printf("%s %s %d\n\n",cur->www,cur->comp,cur->year);  }  }  int main() {  struct node \*head=NULL;  int n,d,k;  scanf("%d",&n);  if(n>1)  {  for(int i=0;i<n;i++)  {  head=insert\_end(head);  }  printf("%s %s %d\n\n",head->www,head->comp,head->year);  scanf("%d",&d);  scanf("%d",&k);  dislay(head,d,k);    }    return 0;  }  32]  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  struct node  {  char name[20];  int edition;  float price;  struct node \*next, \*prev;  };  struct node \*create()  {  struct node \*newnode = (struct node \*)malloc(sizeof(struct node));  if (newnode == NULL)  printf("Memory not allocated\n");  else  {  scanf("%s %d %f", newnode->name, &newnode->edition, &newnode->price);  newnode->next = NULL;  newnode->prev = NULL;  }  return newnode;  }  struct node \*insert\_end(struct node \*head)  {  struct node \*newnode = create();  if (head == NULL)  {  head = newnode;  head->next = head;  head->prev = head;  }  else  {  newnode->next = head;  newnode->prev = head->prev;  head->prev->next = newnode;  head->prev = newnode;  }  return head;  }  void calculate\_cost(struct node \*head)  {  struct node \*cur = head;  float sume = 0, sumo = 0;  int count = 1; // Start count at 1 (for 1-indexing)  if (head != NULL)  {  do  {  if (count % 2 == 0) // Even positions are for the second subject  {  sumo += cur->price;  }  else // Odd positions are for the first subject  {  sume += cur->price;  }  cur = cur->next;  count++;  } while (cur != head);  }  printf("Total cost of first subject books: Rs. %.2f\n", sume);  printf("\nTotal cost of second subject books: Rs. %.2f\n", sumo);  }  int main()  {  struct node \*head = NULL;  int n;  scanf("%d", &n);  if (n > 1)  {  for (int i = 0; i < n; i++)  {  head = insert\_end(head);  }  calculate\_cost(head);  // Free allocated memory  if (head != NULL)  {  struct node \*cur = head->next;  while (cur != head)  {  struct node \*temp = cur;  cur = cur->next;  free(temp);  }  free(head);  }  }  else  {  printf("Invalid input");  }  return 0;  }  33]  #include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  #define SIZE 10  struct stack  {  int num[SIZE];  int top;    };  void push(struct stack \*sptr,int data)  {  if(sptr->top==SIZE-1)  {  printf("stack is overflow");  }  else  {  sptr->top++;  sptr->num[sptr->top]=data;  }  }  int pop(struct stack \*sptr)  {  int data;  if(sptr->top==-1)  {  return -1;  }  else  {  data=sptr->num[sptr->top];  sptr->top--;  return data;  }  }  void display(struct stack \*sptr)  {  if(sptr->top==-1)  {  printf("Stack Empty.\n");  }  else  {  for(int i=sptr->top;i>=0;i--)  {  printf("%d\n",sptr->num[i]);  }  }  }  int peek(struct stack \*sptr)  {  if(sptr->top==-1)  {  return -1;  }  else{  int data=sptr->num[sptr->top];  return data;  }  }  int main() {  struct stack sp;  struct stack \*sptr=&sp;  sptr->top=-1;  int n,j,k,data;  scanf("%d",&n);  if(n>0)  {  for(int i=0;i<n;i++)  {  scanf("%d",&data);  push(sptr, data);  }  scanf("%d",&j);  for(int i=0;i<j;i++)  {  scanf("%d",&data);  push(sptr,data);  }  printf("Stack after PUSH operations is:\n");  display(sptr);  printf("\n");  scanf("%d",&k);  for(int i=0;i<k;i++)  {  data=pop(sptr);  printf("Popped data is: %d\n",data);  }  printf("\n");  if(data==-1)  {  printf("stack empty\n");  }  printf("Stack after POP operations is:\n");  display(sptr);  printf("\n");  printf("Top data of stack is:\n");  data=peek(sptr);    if(data==-1)  {  printf("Stack Empty.\n");  }  else  {  printf("%d",data);  }  }  else  {  printf("Invalid Stack size");  }        /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/  return 0;  }  34}#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  #define SIZE 10  struct stack  {  int num[SIZE];  int top;    };  void push(struct stack \*sptr,int data)  {  if(sptr->top==SIZE-1)  {  printf("stack is overflow");  }  else  {  sptr->top++;  sptr->num[sptr->top]=data;  }  }  int pop(struct stack \*sptr)  {  int data;  if(sptr->top==-1)  {  return -1;  }  else  {  data=sptr->num[sptr->top];  sptr->top--;  return data;  }  }  void display(struct stack \*sptr)  {  if(sptr->top==-1)  {  printf("Stack Empty.\n");  }  else  {  for(int i=sptr->top;i>=0;i--)  {  printf("%d\n",sptr->num[i]);  }  }  }  void hu(struct stack \*sptr)  {  if(sptr->top==-1)  {  printf("stack Empty");  }  else  {  int count=0,sount=0,dount=0,fount=0;  for(int i=sptr->top;i>=0;i--)  {  if(sptr->num[i]==5)  {  count++;  }  else if(sptr->num[i]==1)  {  sount++;  }  else if(sptr->num[i]==2)  {  dount++;  }  else if(sptr->num[i]==10)  {  fount++;  }    }  printf("Denomination 1 coins: %d\n",sount);  printf("Denomination 2 coins: %d\n",dount);  printf("Denomination 5 coins: %d\n",count);  printf("Denomination 10 coins: %d\n",fount);  }  }  int main() {  struct stack sp;  struct stack \*sptr=&sp;  sptr->top=-1;  int n,data;  scanf("%d",&n);  if(n>0)  {  for(int i=0;i<n;i++)  {  scanf("%d",&data);  push(sptr, data);  }  printf("Coins in Stack are:\n");  display(sptr);  printf("\n");  hu(sptr);  }  else  {  printf("No coins.");  }  /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/  return 0;  }  35]#include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100  struct Stack {  int top;  int array[MAX\_SIZE];  };  void initStack(struct Stack\* stack) {  stack->top = -1;  }  int isEmpty(struct Stack\* stack) {  return stack->top == -1;  }  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1;  }  void push(struct Stack\* stack, int item) {  if (isFull(stack)) {  printf("Stack overflow\n");  return;  }  stack->array[++stack->top] = item;  }  int pop(struct Stack\* stack) {  if (isEmpty(stack)) {  printf("Stack underflow\n");  return -1;  }  return stack->array[stack->top--];  }  void reverseAndCheckPalindrome(int number) {  struct Stack stack;  initStack(&stack);  int originalNumber = number;  // Push each digit onto the stack  while (number != 0) {  int digit = number % 10;  push(&stack, digit);  number /= 10;  }  printf("Stack with digits is:\n");  // Pop and print each digit from the stack  while (!isEmpty(&stack)) {  int digit = pop(&stack);  printf("%d\n", digit);  }  // Check if the original number is a palindrome  int reversedNumber = 0;  number = originalNumber;  while (number != 0) {  int digit = number % 10;  reversedNumber = reversedNumber \* 10 + digit;  number /= 10;  }  printf("\n");  if (originalNumber == reversedNumber) {  printf("%d is a palindrome integer.\n", originalNumber);  } else {  printf("%d is not a palindrome integer.\n", originalNumber);    }  }  int main() {  int number;    scanf("%d", &number);  if(number>0)  {  reverseAndCheckPalindrome(number);  }  else  {  printf("Invalid input.");  }  return 0;  }  36]#include <stdio.h>  #include <stdbool.h>  #include <string.h>  #define MAX\_SIZE 100  struct Stack {  int top;  char arr[MAX\_SIZE];  };  void push(struct Stack\* stack, char item) {  if (stack->top == MAX\_SIZE - 1) {  printf("Stack Overflow\n");  return;  }  stack->arr[++stack->top] = item;  }  char pop(struct Stack\* stack) {  if (stack->top == -1) {  printf("Stack Underflow\n");  return '\0';  }  return stack->arr[stack->top--];  }  bool isEmpty(struct Stack\* stack) {  return stack->top == -1;  }  void reverseAndCheckPalindrome(char\* string) {  struct Stack stack;  stack.top = -1;  int len = strlen(string);  char reversedString[MAX\_SIZE];  int i;  // Push each character onto the stack  for (i = 0; i < len; i++) {  push(&stack, string[i]);  }  // Pop each character from the stack and print  for (i = 0; i < len; i++) {  char charFromStack = pop(&stack);  reversedString[i] = charFromStack;  printf("%c\n", charFromStack);  }  reversedString[i] = '\0';  printf("\n");  // Check if the reversed string is a palindrome  if (strcmp(string, reversedString) == 0) {  printf("%s is a palindrome string.\n", string);  } else {  printf("%s is not a palindrome string.\n", string);  }  }  int main() {  char string[MAX\_SIZE];  scanf("%s", string);  printf("Stack of characters is:\n");  reverseAndCheckPalindrome(string);  return 0;  }  37]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  #define SIZE 20  struct stack  {  char num[SIZE];  int top;  };  void push(struct stack \*sptr,int data)  {  if(sptr->top==SIZE-1)  {  printf("stack is empty\n");  }  else  {  sptr->top++;  sptr->num[sptr->top]=data;  }  }  int pop(struct stack \*sptr)  {  int data;  if(sptr->top==-1)  {  return 0;  }  else  {  data=sptr->num[sptr->top];  sptr->top--;  return data;  }  }  int validation(struct stack \*sptr,char exp[])  {  char ch;  int i=0;  while(exp[i]!='\0')  {  if(exp[i]=='('||exp[i]=='{'||exp[i]=='[')  push(sptr,exp[i]);  else if(exp[i]==')')  {  if(sptr->top==-1)  return 0;  ch=pop(sptr);  if(ch!='(')  return 0;  }  else if(exp[i]=='}')  {  if(sptr->top==-1)  return 0;  ch=pop(sptr);  if(ch!='{')  return 0;  }  else if(exp[i]==']')  {  if(sptr->top==-1)  return 0;  ch=pop(sptr);  if(ch!='[')  return 0;  }  i++;  }  if(sptr->top==-1)  {  return 1;  }  else  {  return 0;  }  }  int main() {  struct stack c;  struct stack \*sptr=&c;  sptr->top=-1;  char exp[SIZE];  scanf("%s",exp);  int f=validation(sptr,exp);  if(f==1)  {  printf("Valid expression.\n");  }  else  {  printf("Invalid expression.\n");  }  /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/  return 0;  }  38]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  #include <ctype.h>  #define SIZE 25  struct stack  {  char data[SIZE];  int top;  };  void push(struct stack \*sptr,char opr)  {  if(sptr->top == SIZE -1)  printf("Stack overflow.\n");  else  sptr->data[++sptr->top]=opr;  }  char pop(struct stack \*sptr)  {  if(sptr->top==-1)  return '\0';  else  return (sptr->data[sptr->top--]);  }  char peek(struct stack \*sptr)  {  if(sptr->top == -1)  return '\0';  else  return (sptr->data[sptr->top]);  }  int precedence(char ch)  {  switch(ch)  {  case '#': return 0;  break;  case '(': return 1;  break;  case '+':  case '-': return 2;  break;  case '\*':  case '/':  case '%': return 3;  break;  case '^': return 4;  break;  }  return -1;  }  void conversion(struct stack \*sptr,char infix[SIZE])  {  char postfix[SIZE],ch,temp;  int i=0,j=0;  push(sptr, '#');  while((ch = infix[i++]) != '\0')  {  if(ch == '(')  push(sptr,ch);  else if(isalnum(ch)) // operand  postfix[j++] = ch;  else if(ch == ')')  {  while(sptr->data[sptr->top] != '(')  postfix[j++] = pop(sptr);  //one time pop( psrenthesis  temp=pop(sptr);    }  else //operator  {  while (precedence(peek(sptr))>=precedence(ch))  postfix[j++]=pop(sptr);  //one time  push(sptr,ch);  }    }//end of while  while(sptr->data[sptr->top] != '#')  postfix[j++]=pop(sptr);  postfix[j]='\0';  printf("Given Infix Expression: %s\n",infix);  printf("Postfix Expression: %s\n",postfix);    }  int main()  {  struct stack st,\*sptr;  sptr = &st;  char infix[SIZE];  sptr->top=-1;  scanf("%s",infix);  conversion(sptr, infix);  return 0;  }  39]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  #include<ctype.h>  #define SIZE 20  struct stack  {  int data[SIZE];  int top;  };  void push(struct stack \*sptr ,int operand)  {  if(sptr->top==SIZE-1)  printf("stack verflow");  else  sptr->data[++sptr->top]=operand;  }  int pop(struct stack\*sptr)  {  if(sptr->top==-1)  return -1;  else  return (sptr->data[sptr->top--]);  }  void evaluate(struct stack \*sptr,char postfix[SIZE])  {  char ch;  int i=0,op1,op2;  while((ch=postfix[i++])!='\0')  {  if(isdigit(ch))//return 1 if true else return 0  push(sptr,ch-'0');  else  {  op2=pop(sptr);  op1=pop(sptr);    switch(ch)  {  case '+':push(sptr ,(op1+op2));  break;  case '-':push(sptr,(op1-op2));  break;  case '\*':push(sptr,(op1\*op2));  break;  case '/':push(sptr,(op1/op2));  break;  case '^':push(sptr,pow(op1,op2));  }//end of swicth  }  }//end of while loop  printf("Given postfix Expression: %s\n",postfix);  printf("Result after Evaluation: %d",pop(sptr));  }  int main()  { struct stack\*sptr,st;  sptr=&st;  sptr->top=-1;  char postfix[SIZE];  scanf("%s",postfix);  evaluate(sptr,postfix);  return 0;  }  40]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node {  char car;  struct node \*link;  };  struct node\* create\_node() {  struct node \*newnode = (struct node\*)malloc(sizeof(struct node));  if (newnode == NULL) {  printf("Memory not allocated.\n");  return NULL;  } else {  scanf(" %c", &newnode->car);  newnode->link = NULL;  }  return newnode;  }  // Push operation, inserts at the front  struct node\* push(struct node \*top, struct node \*newnode) {  if (top == NULL)  top = newnode;  else {  newnode->link = top;  top = newnode;  }  return top;  }  // Pop operation, deletes from the front  struct node\* pop(struct node \*top) {  struct node \*cur = NULL;  if (top == NULL)  return NULL;  else if (top->link == NULL) {  free(top);  top = NULL;  } else {  cur = top;  top = top->link;  free(cur);  }  return top;  }  int validate\_cars(struct node \*top, int n) {  char cars[20];  int i = 0;    // Copy cars from stack to array  while (n--) {  cars[i++] = top->car;  top = pop(top);  }    // Compare array elements with remaining stack elements  while (top != NULL) {  if (cars[--i] != top->car) {  return 0; // Invalid parking  }  top = pop(top);  }    return 1; // Valid parking  }  int main() {  struct node \*top = NULL, \*newnode = NULL;  int n, status;  scanf("%d", &n);    // Read cars and push into the stack  for (int i = 0; i < 2 \* n; i++) {  newnode = create\_node();  top = push(top, newnode);  }    status = validate\_cars(top, n);    if (status == 1)  printf("Valid parking\n");  else  printf("Invalid parking\n");    return 0;  }  41]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node {  int num;  struct node \*link;  };  struct node\* create\_node() {  struct node \*newnode = (struct node\*)malloc(sizeof(struct node));  if (newnode == NULL) {  printf("Memory not allocated.\n");  return NULL;  } else {  scanf("%d", &newnode->num);  newnode->link = NULL;  }  return newnode;  }  // Push operation, inserts at the front  struct node\* push(struct node \*top, struct node \*newnode) {  if (top == NULL)  top = newnode;  else {  newnode->link = top;  top = newnode;  }  return top;  }  // Pop operation, deletes from the front  struct node\* pop(struct node \*top) {  struct node \*cur = NULL;  if (top == NULL)  return NULL;  else if (top->link == NULL) {  free(top);  top = NULL;  } else {  cur = top;  top = top->link;  free(cur);  }  return top;  }  void check\_luckydraw(struct node \*top, int k) {  int cw = 0, cl = 0;  while (top != NULL) {  if (top->num % k == 0) {  printf("W ");  cw++;  } else {  printf("L ");  cl++;  }  top = pop(top);  }  printf("\nNo of winners=%d\n", cw);  printf("No of losers=%d\n", cl);  }  int main() {  struct node \*top = NULL, \*newnode = NULL;  int n, k;  scanf("%d%d", &n, &k);  if (n > 0) {  for (int i = 0; i < n; i++) {  newnode = create\_node();  top = push(top, newnode);  }  check\_luckydraw(top, k);  } else {  printf("Invalid number of customers\n");  }  return 0;  }  42]#include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  #define SIZE 10  struct queues  {  int num[SIZE];  int front, rear;  };  void inque(struct queues \*sptr, int data)  {  if (sptr->rear == SIZE - 1)  {  printf("Queue overflow\n");  }  else  {  sptr->rear++;  sptr->num[sptr->rear] = data;  }  }  int deque(struct queues \*sptr)  {  if (sptr->front == sptr->rear)  {  return 0;  }  else  {  sptr->front++;  return sptr->num[sptr->front];  }  }  void display(struct queues \*sptr)  {  if (sptr->front == sptr->rear)  {  printf("Queue Empty\n");  }  else  {  for (int i = sptr->front + 1; i <= sptr->rear; i++)  {  printf("%d ", sptr->num[i]);  }  printf("\n");  }  }  int main()  {  struct queues sp;  struct queues \*sptr = &sp;  sptr->rear = -1;  sptr->front = -1;  int n, g, data = 0;  scanf("%d", &n);  if (n >= 1)  {  for (int i = 0; i < n; i++)  {  scanf("%d", &data);  inque(sptr, data);  }  scanf("%d", &g);  for (int i = 0; i < g; i++)  {  data = deque(sptr);  if (data > 1)  {  printf("Dequeued %d\n", data);  }  else  {  printf("Queue underflow\n");  }  }  printf("Queue after dequeue is:\n");  display(sptr);  }  else  {  printf("Cannot perform operations\n");  }  return 0;  } |