|  |  |
| --- | --- |
| **Name of the student** | **T.Keerthana** |
| **Roll No** | **243** |
| **USN** | **01FE22BEC073** |
| **Div** | **B Div** |

**Part- A**

|  |
| --- |
| **Variables and Data Types** |
| 1.What is the difference between a variable and a data type in C programming? Provide examples to illustrate. |
| A:- A variable is a name given to a memory location where data is stored and can be manipulated during program execution.  A data type specifies the kind of data that a variable can hold.. It specifies the size and layout of the variable's memory and the range of values it can store.  Example: In C programming, int num declares a variable named num of type int, where int is the data type. |
| 2.Explain the concept of data types in C programming. Discuss the different types  of data types available in C |
| A:-C provides several built-in data types such as int, char, float, double, etc. These data types can be categorized as:  Basic data types (int, char, float, double)  Derived data types (arrays, pointers, structures, unions)  basic data types (int, char, float, double) are fundamental for storing simple values, while derived data types (arrays, pointers, structures, unions) enable complex data structures and memory management in C programming. |
| 3.How are variables declared and initialized in C programming? Provide  examples of variable declarations with different data types.  A:-Variables are declared by specifying the data type followed by the variable name.  Initialization assigns an initial value to the variable at the time of declaration.  Example:  int age; // Variable declaration  float pi = 3.14; |
| 4.Discuss the scope and lifetime of variables in C programming. What are global  and local variables  A:- Scope refers to the visibility of a variable within the program. Lifetime refers to the duration for which a variable exists in memory during program execution. Global variables have a global scope and exist throughout the program's execution. Local variables are confined to a specific scope and are only accessible within the block in which they are declared. |
| 5.Explain the concept of type casting in C programming. When is type casting  necessary, and how is it performed?  A:- Type casting in C programming involves the conversion of a value from one data type to another. It's necessary when operations involve different data types or when assigning a value of one type to a variable of another type. Type casting is performed by placing the desired data type in parentheses before the variable or expression to be converted. For example, `(float)x` would cast the integer variable `x` to a floating-point value. |
| 6.Describe the purpose and structure of a function prototype in C programming.  Why is it necessary to declare function prototypes?  A:-Purpose: A function prototype in C serves to declare the function before its actual implementation. It encompasses the function's identifier, return type, and parameter types.. This informs the compiler about the function's signature for type checking during compilation.  Structure: return\_type function\_name(parameter\_type1, parameter\_type2, ...);  Necessity: It's necessary to declare function prototypes to catch errors in function calls, such as passing wrong arguments. It allows defining functions in any order within or across source files. |
| 7.Explain the difference between call by value and call by reference in C  programming. Provide examples to illustrate both concepts.  A:-Call by Value: A copy of the parameter's value is passed to the function. Modifications inside the function don't affect the actual parameters.  Example:  void changeValue(int x) {  x = 10;  }  Call by Reference: The memory address of the parameter is passed. Changes inside the function affect the actual parameters.  Example:  void changeValue(int \*x) {  \*x = 10;  } |
| 8.Discuss the concept of recursion in C programming. Provide an example of a  recursive function and explain how it work  A:-Concept: Recursion is a technique where a function calls itself. It solves problems by breaking them into smaller, similar subproblems.  Example:  int factorial(int n) {  if (n == 0 || n == 1)  return 1;  else  return n \* factorial(n - 1);  }  Explanation: The factorial function calls itself with a smaller argument until reaching a base case. Each call contributes to the final result. |
| 9.Describe the purpose and usage of the ternary conditional operator (?:) in C  programming. Provide an example demonstrating its usage  A:-It's a conditional operator that evaluates an expression and returns one value if the expression is true and another value if the expression is false.  Syntax: condition ? value\_if\_true : value\_if\_false  Example:  int x = 10;  int result = (x > 5) ? 1 : 0; |
| 10.Discuss the bitwise operators available in C programming. Explain their usage  with suitable examples.  Bitwise operators perform operations on individual bits of integer operands.  Examples include AND (&), OR (|), XOR (^), left shift (<<), right shift (>>), etc.  Example:  int a = 5; // 101 in binary  int b = 3; // 011 in binary  int result = a & b; // Bitwise AND: 001 |
| 11.Explain the difference between the postfix and prefix increment operators (++)  in C programming. Provide examples to illustrate.  A:-Postfix increment (x++) increments the value of x after it's been used in the expression.  Prefix increment (++x) increments the value of x before it's used in the expression.  Example:  int x = 5;  int y = x++; // y will be 5, x will be 6  int z = ++x; // z will be 7, x will be 7 |
| 12.What is the significance of the logical AND (&&) and logical OR (||) operators  in C programming? How are they used in conditional expressions?  A:-The logical AND (&&) and logical OR (||) operators in C programming are crucial for constructing conditional expressions, allowing control of program flow based on multiple conditions.  && returns true only if both operands are true; otherwise, it returns false.  || returns true if at least one operand is true; otherwise, it returns false.  They are used in conditional statements like if, while, and for, specifying conditions that must be met or defining alternatives. Example:  if (x > 0 && y < 10) {  // Code block executes if both x > 0 and y < 10  }  if (grade == 'A' || grade == 'B') {  // Code block executes if grade is 'A' or 'B'  } |
| 13.Discuss the concept of operator precedence and associativity in C programming.  Provide examples to demonstrate how they affect expression evaluation.  A:-Operator precedence and associativity dictate the order of evaluation in expressions.  Precedence determines which operators are evaluated first.  Associativity defines the order of evaluation for operators of the same precedence.  Example:  int result1 = a + b \* c; // Precedence: multiplication (\*) is evaluated first  int result2 = a = b = c; // Associativity: right-to-left for assignment (=)  In result1, multiplication is evaluated before addition due to precedence. In result2, the assignment operator is evaluated right-to-left due to associativity. |
| 14.Describe the purpose and usage of the switch statement in C programming. How  does it differ from the if-else statement?  A:-The switch statement allows a program to evaluate an expression and execute different blocks of code based on its value.  It differs from the if-else statement by providing a more concise way to handle multiple conditions.  Example:  int choice;  switch (choice) {  case 1:  // Code block for choice 1  break;  case 2:  // Code block for choice 2  break;  default:  // Default code block  break;  } |
| 15.Explain the concept of nested control structures in C programming. Provide an  example demonstrating nested if-else statements.  A:-Nested control structures are control structures placed within other control structures.  Example:  if (condition1) {  if (condition2) {  // Code block  } else {  // Code block  }  } else {  // Code block  } |
| 16.Discuss the role of the break and continue statements in loop control in C  programming. Provide examples to illustrate their usage.  A:-The break statement is used to terminate the loop or switch statement and transfer control to the statement immediately following it.  The continue statement is used to skip the remaining code in the loop and move to the next iteration.  Example:for (int i = 0; i < 10; i++) {  if (i == 5) {  continue; // Skip iteration when i is 5  }  if (i == 8) {  break; // Exit the loop when i is 8  }  // Code block  } |
| 17.What are the advantages of using the for loop over the while loop in C  programming? Provide examples comparing the two.  A:-The for loop is typically used when the number of iterations is known before the loop starts.  It provides a compact way to write loops by combining initialization, condition, and increment/decrement in a single line.  Example:for (int i = 0; i < 5; i++) {  // Code block  } |
| 18.Explain the concept of short-circuit evaluation in C programming. How does it  affect the evaluation of logical expressions in if statements?  A:-Short-circuit evaluation is a mechanism where the evaluation of logical expressions stops as soon as the result is determined.  In an if statement with logical AND (&&) or logical OR (||) operators, if the left operand determines the result, the right operand is not evaluated.  Example:  if (x > 0 && y > 0) {  // Code block  }  In the above example, if x is not greater than 0, y > 0 won't be evaluated because the result of the expression is already determined to be false. |
| 19. What is the significance of the return statement in C programming? How are  values returned from functions?  A:-The return statement in C programming allows functions to provide a result back to the calling code. Values are returned using the return keyword, matching the function's declared return type. It exits the function and passes control back to the caller along with the returned value.  int add(int a, int b) {  return a + b; // Returns the sum of a and b  }  int result = add(3, 5); // Calls the add function and assigns its return value (8) to result |
| 20.Describe the role of function parameters and arguments in C programming.  How are function arguments passed to parameters  A:-Function parameters in C define the types and order of data expected by a function, while arguments are the actual values passed to the function. Arguments are passed to parameters using "pass by value," where the value of each argument is copied into the corresponding parameter.  #include <stdio.h>  // Function prototype with parameters  void greet(char name[]) {  printf("Hello, %s!\n", name);  }  int main() {  char myName[] = "John";  // Function call with arguments  greet(myName); // "myName" is passed as an argument to the "name" parameter  return 0;  } |
| 21.Explain the concept of arrays in C programming. How are arrays declared and  initialized?  A:-Arrays in C are a collection of elements of the same data type stored in contiguous memory locations. They provide a way to store multiple values of the same type under a single variable name.  Declaration and Initialization:  Declaration: type array\_name[size];  Initialization:  Static Initialization: type array\_name[size] = {value1, value2, ...};  Dynamic Initialization: type array\_name[size]; array\_name[index] = value; |
| 22.Discuss the difference between a one-dimensional array and a multi dimensional array in C programming.provide examples.  A:-Arrays:  One-dimensional Array: It's a linear collection of elements stored in a single row or column.  int arr1D[5] = {1, 2, 3, 4, 5};  Multi-dimensional Array: It's an array of arrays, organized in rows and columns.  int arr2D[2][3] = {{1, 2, 3}, {4, 5, 6}}; |
| 23.Describe the process of accessing array elements in C programming. How are  array indices used to access elements  A:-Process: Array elements are accessed using their indices.  Syntax: array\_name[index];  Indices: Start from 0 for the first element and go up to size - 1. |
| 24.What is the significance of the null character ('\0') in C strings? How is it used  to determine the end of a string  A:-Purpose: The null character ('\0') marks the end of a string in C. It indicates where the string's contents end in memory.  Usage: Functions that work with strings in C, like printf and strlen, use the null character to determine the string's length and where it ends. |
| 25.Explain the concept of dynamic memory allocation for arrays in C  programming. How are dynamic arrays allocated and deallocated  A:-Concept: Dynamic memory allocation allows the creation of arrays whose size is determined during runtime.  Functions: In C, malloc(), calloc(), and realloc() are used to allocate dynamic memory for arrays.  Allocation: ptr = (castType\*) malloc(size \* sizeof(type));  Deallocation: free(ptr);  Example:  int \*arr;  int size = 5;  arr = (int\*)malloc(size \* sizeof(int));  Note: Dynamic memory must be deallocated using free() to avoid memory leaks. |
| 26.Describe the purpose and usage of pointers in C programming. How are pointers  declared and initialized  A:-Purpose: Pointers are variables that store memory addresses. They are used to store addresses of variables or arrays, enabling dynamic memory allocation and efficient manipulation of data structures.  Usage: Pointers are extensively used in tasks like dynamic memory allocation, passing parameters by reference, and building complex data structures like linked lists, trees, etc.  Declaration and Initialization:  Declaration: type \*ptr\_name;  Initialization: type \*ptr\_name = &variable; |
| 27.Explain the concept of pointer arithmetic in C programming. Provide examples  to illustrate addition and subtraction operations on pointers  A:-Concept: Pointer arithmetic involves adding or subtracting an integer from a pointer, which moves it to a different memory location.  Examples:  int arr[5] = {1, 2, 3, 4, 5};  int \*ptr = arr;  ptr++; // Moves pointer to the next element in the array  ptr--; // Moves pointer back to the previous element  EX;  int arr[5] = {10, 20, 30, 40, 50};  int \*ptr = arr; // Points to the first element of the array  ptr = ptr + 2; // Moves pointer to the third element  printf("Value at new pointer position: %d\n", \*ptr); // Output: Value at new pointer position: 30 |
| 28.Discuss the difference between pass by value and pass by reference in function  arguments using pointers in C programming. Provide examples to illustrate both  approaches.  A:-Pass by Value:  - Function receives a copy of the argument's value.  - Modifications inside the function do not affect the original value.  - Example:  void incrementByValue(int x) {  x++;  }  int main() {  int num = 5;  incrementByValue(num);  // num remains 5  }  Pass by Reference using Pointers:  - Function receives the memory address of the argument.  - Modifications inside the function directly affect the original value.  - Example:  void incrementByReference(int \*x) {  (\*x)++;  }  int main() {  int num = 5;  incrementByReference(&num);  // num becomes 6  }  In pass by reference, we use pointers to pass the address of the variable, allowing direct modification of its value within the function. |
| 29.Describe the concept of NULL pointers in C programming. How are NULL  pointers used and checked for in programs?  A:In C programming, a NULL pointer is a pointer that does not point to any memory location. It is represented by the value 0 or by the macro NULL.  Usage of NULL Pointers  1. Initialization: NULL pointers are commonly used to initialize pointers before they are assigned a valid memory address.  2. ndication of Absence: They are used to indicate that a pointer does not currently point to a valid object or memory location.  Checking for NULL Pointers:  1. Equality Check: To check if a pointer is NULL, you simply compare it with NULL or the integer 0.  if (ptr == NULL) {  // Pointer is NULL  }  2. Dereferencing Safeguard: Before dereferencing a pointer, it's a good practice to check if it's NULL to avoid accessing invalid memory.  if (ptr != NULL) {  // Pointer is not NULL, safe to dereference  printf("Value at pointer: %d\n", \*ptr);  }  - NULL pointers are checked for equality with NULL or the integer 0 to determine if they are valid or not before dereferencing them. |
| 30.Explain the role of pointers in dynamic memory allocation in C programming.  How are pointers used to allocate and deallocate memory dynamically  A:-Pointers in Dynamic Memory Allocation in C:  In Cprogramming, pointers are vital for dynamic memory allocation, where memory is allocated and deallocated during program execution. Here's a concise overview:  1. Allocation of Memory:  - Pointers are used to request memory dynamically using functions like `malloc()`, `calloc()`, or `realloc()`.  int \*ptr;  ptr= (int \*)malloc(10 \* sizeof(int)); // Allocating memory for an array of 10 integer  2. Deallocation of Memory:  - Pointers are used to release dynamically allocated memory to avoid memory leaks.  free(ptr); // Deallocating the memory pointed to by ptR  Usage Example:  #include <stdio.h>  #include <stdlib.h>  int main() {  int \*ptr;  int size = 5;  // Dynamically allocate memory for an array of integers  ptr = (int \*)malloc(size \* sizeof(int));  if (ptr == NULL) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  // Use the dynamically allocated memory  for (int i = 0; i < size; i++) {  ptr[i] = i + 1;  }  // Print the values stored in the dynamically allocated memory  for (int i = 0; i < size; i++) {  printf("%d ", ptr[i]);  }  printf("\n");  // Free the dynamically allocated memory  free(ptr);  return 0;  }  In summary, pointers enable dynamic memory allocation in C, allowing programs to manage memory efficiently by requesting and releasing memory as needed during runtime. |
| 31.Discuss the concept of strings in C programming. How are strings represented  and manipulated in C  A:-In C programming, a NULL pointer is a pointer that doesn't point to any memory location. It's commonly used to signify that a pointer doesn't currently reference any valid object.  - Initialization: NULL pointers are often used to initialize pointers before assigning them valid memory addresses.  - Sentinel Values: They're used as sentinel values to indicate the end of data structures like linked lists or arrays.  - Error Handling: NULL pointers are checked to prevent dereferencing errors, avoiding crashes or undefined behavior.  - Checking for NULL Pointers:  - Prior to dereferencing a pointer, it's checked against NULL to ensure it's pointing to valid memory.  - Commonly done using an `if` statement or implicitly through operations that depend on pointer validity.  Example:  int \*ptr = NULL; // Initialization  if (ptr == NULL) {  // Handle the case where ptr is NULL  } else {  // Proceed with using ptr safely  }  By using NULL pointers and checking for them, C programs can handle memory safely and avoid unexpected crashes. |
| 32.Explain the difference between character arrays and string literals in C  programming. Provide examples to illustrate both concepts  A:-Certainly!  Character Arrays:  - Character arrays in C are sequences of characters stored in contiguous memory locations.  - Each element of the array holds a single character.  - They are mutable, meaning their contents can be modified after initialization.  - They must be terminated with a null character (`'\0'`) to indicate the end of the string.  - Example:  char tr1[10] = {'H', 'e', 'l', 'l', 'o', '\0'};  char str2[10];  strcpy(str2, "World"); // Using a string literal to initialize the array  String Literals:  - String literals are sequences of characters enclosed in double quotes.  - They are stored as null-terminated character arrays by the compiler.  - They are immutable, meaning their contents cannot be modified after compilation.  - They are automatically null-terminated by the compiler.  - Example:  char \*str3 = "Hello"; // Pointer to a string literal  char str4[] = "World"; // Character array initialized with a string literal  Key Differences:  - Mutability: Character arrays are mutable, allowing modifications to their contents. String literals are immutable and cannot be modified.  - Termination: Character arrays require explicit termination with a null character (`'\0'`). String literals are automatically null-terminated by the compiler.  - Usage: Character arrays are often used when mutable strings are required, while string literals are used for constant strings that do not need to be modified.  Understanding these differences is essential for efficient string handling and memory management in C programming. |
| 33.Describe common string manipulation functions available in the C standard  library. Provide examples of functions like strlen, strcpy, strcat, and strcmp.  A:-Sure, here's a more concise overview:  1. strlen():  - Description: Calculates the length of a string (excluding the null terminator).  - Prototype: `size\_t strlen(const char \*str);`  - Example:  char str[] = "Hello";  size\_t length = strlen(str); // length = 5  2. strcpy():  - Description: Copies the contents of one string to another.  - Prototype: `char \*strcpy(char \*dest, const char \*src);`  - Example:  char src[] = "Hello";  char dest[10];  strcpy(dest, src); // dest = "Hello"  3. strcat();  - Description: Appends the content of the source string to the destination string.  - Prototype: `char \*strcat(char \*dest, const char \*src);`  - Example  char dest[20] = "Hello";  char src[] = " World";  strcat(dest, src); // dest = "Hello World"  4. strcmp():  - Description: Compares two strings lexicographically.  - Prototype: `int strcmp(const char \*str1, const char \*str2);`  - Example:  char str1[] = "Hello";  char str2[] = "Hello";  int result = strcmp(str1, str2); // result = 0 (equal)  These functions are commonly used for basic string operations in C programming, offering efficient ways to handle and manipulate strings. |
| 34.Discuss the concept of string tokenization in C programming. How are strings  split into tokens using delimiter characters?  A:-String tokenization is the process of breaking a string into smaller pieces, known as tokens, based on specified delimiter characters. This concept is often used in C programming to parse strings and extract relevant information. Here's a more detailed explanation  1.Using `strtok()` FunctioN:  - C provides the `strtok()` function in the standard library for string tokenization.  - It takes two arguments: the string to be tokenized and a string containing delimiter characters.  - The function returns a pointer to the next token found in the string, or `NULL` if no more tokens are found.  2. Steps for String Tokenization:  - Call `strtok()` with the input string and delimiter string.  - The first call to `strtok()` returns a pointer to the first token found in the input string.  - Subsequent calls to `strtok()` with `NULL` as the first argument will continue tokenizing the string until no more tokens are found.  - Each call to `strtok()` modifies the input string by replacing delimiter characters with null characters (`'\0'`) to separate tokens.  3. Example:  #include <stdio.h>  #include <string.h>  int main() {  char str[] = "apple,banana,grape";  char \*token = strtok(str, ",");  while (token != NULL) {  printf("%s\n", token);  token = strtok(NULL, ",");  }  return 0;  }    Output:    apple  banana  grape  String tokenization is a fundamental technique for processing textual data, such as parsing command-line arguments, reading input from files, or extracting fields from structured data formats like CSV (comma-separated values). |
| 35.Explain the importance of null-terminated strings in C programming. How does  the null character ('\0') signify the end of a string  A:-Null-terminated strings are crucial in C programming because they serve as a fundamental way to represent and manipulate strings. The null character (`'\0'`) marks the end of a string, allowing C functions to determine the length of a string and perform string operations safely. Here's a brief overview of their importance:  1. String Length: Null-terminated strings enable functions like `strlen()` to determine the length of a string by iterating through its characters until the null character is encountered.  2. String Manipulation: Functions like `strcpy()`, `strcat()`, and `strcmp()` rely on the null character to perform string operations effectively and safely.  3. Memory Management: Null-terminated strings facilitate efficient memory management in C programs, allowing for the use of statically allocated character arrays or dynamically allocated memory for strings  4. Interoperability: Many C library functions and system calls expect null-terminated strings as input parameters, making them essential for interfacing with existing libraries and systems.  5. Compatibility: Null-terminated strings are widely used in C libraries, APIs, and protocols, ensuring compatibility and interoperability across different C programs.  Overall, null-terminated strings provide a simple and effective way to represent and manipulate textual data in C programming, making them a fundamental aspect of the language. |
| 36.Describe the purpose and usage of structures in C programming. How are  structures declared and accessed?  A:-Purpose and Usage of Structures in C Programming:  - Purpose: Structures allow you to group different data types under a single name. They are used to create complex data types that can hold multiple pieces of related information.  - Usage: Structures are commonly used to represent real-world entities or complex data types such as employees, students, records, etc. They provide a way to organize related data elements into a single unit.  Declaration and Access of Structures:  - Declaration: Structures are declared using the `struct` keyword followed by the structure tag and a list of member variables inside curly braces.  struct Person {  char name[50];  int age;  float height;  };  - Access: You can access structure members using the dot (`.`) operator.  struct Person person1;  strcpy(person1.name, "John");  person1.age = 30;  person1.height = 5.9;  Brief Explanation:  - \*\*Purpose\*\*: Structures help organize related data items into a single unit for easier management and manipulation.  - \*\*Usage\*\*: They are extensively used in applications where complex data structures are needed, such as in databases, file systems, and graphical user interfaces.  - \*\*Declaration\*\*: Structures are declared using the `struct` keyword followed by a name and a list of member variables inside curly braces.  - \*\*Access\*\*: Structure members are accessed using the dot (`.`) operator followed by the member name.  In essence, structures provide a way to group related data elements together under a single name, making it easier to manage and manipulate complex data in C programming. |
| 37.Discuss the concept of structure members in C programming. How are  individual members of a structure accessed and modified?  A:-Purpose and Usage of Structures in C Programming:  - Purpose. Structures allow you to group different data types under a single name. They are used to create complex data types that can hold multiple pieces of related information.  - Usage: Structures are commonly used to represent real-world entities or complex data types such as employees, students, records, etc. They provide a way to organize related data elements into a single unit.  Declaration and Access of Structures:  - Declaration: Structures are declared using the `struct` keyword followed by the structure tag and a list of member variables inside curly braces.  struct Person {  char name[50];  int age;  float height;  };  - Access: You can access structure members using the dot (`.`) operator.  struct Person person1;  strcpy(person1.name, "John");  person1.age = 30;  person1.height = 5.9;  Brief Explanation:  - Purpose: Structures help organize related data items into a single unit for easier management and manipulation.  - Usage: They are extensively used in applications where complex data structures are needed, such as in databases, file systems, and graphical user interfaces.  - Declaration: Structures are declared using the `struct` keyword followed by a name and a list of member variables inside curly braces.  - Access: Structure members are accessed using the dot (`.`) operator followed by the member name. |
| 38.Explain the difference between structures and unions in C programming. When  would you choose one over the other  A:-Structures vs Unions in C:  Structures:  - You use structures to group different kinds of data under one name.  - Each part of the structure has its own space in memory.  - You can access each part of the structure independently.  Unions:  - Unions also group different data types together.  - However, all parts of a union share the same memory space.  - Only one part of the union can be used at a time.When to Use Each:  - Structures are good when you need to store various types of data and access them separately.  - Unionsare useful when you want to save memory and only use one type of data at a time.  In simpler terms, structures let you keep different kinds of data organized, while unions let you save space by sharing memory between different types of data. Choose based on what you need for your program. |
| 39.Describe the concept of nested structures in C programming. How are structures  within structures defined and accessed?  A:-In C programming, nested structures refer to structures that are defined within other structures. This allows you to create complex data structures where one structure contains another as a member.  Definition of Nested Structures:  Nested structures are defined similarly to regular structures, but the member of a structure can itself be another structure. Here's how you define a nested structure:  struct Address {  char street[50];  char city[50];  char state[20];  int zip;  };  struct Person {  char name[50];  int age;  struct Address address; // Nested structure  };  In the above example, the Person structure contains a member named address, which is of type Address structure.  Accessing Nested Structure Members:  You can access members of nested structures using the dot (.) operator. Here's how you access members of a nested structure:  EX:  struct Person person1;  strcpy(person1.name, "John");  person1.age = 30;  strcpy(person1.address.street, "123 Main St");  strcpy(person1.address.city, "Anytown");  strcpy(person1.address.state, "CA");  person1.address.zip = 12345;  In this example, person1.address.street, person1.address.city, person1.address.state, and person1.address.zip are accessed using dot notation just like any other structure member.  Benefits of Nested Structures:  Organized Data: Nested structures help in organizing related data in a hierarchical manner.  Encapsulation: They encapsulate related data together, making the code easier to understand and maintain.  Code Reusability: Nested structures facilitate code reusability as you can define complex data structures using smaller, reusable structures. |
| 40. Discuss the concept of typedef in C programming. How is typedef used to define  custom data types, including structures and unions  A:-  In C, `typedef` is used to create custom names for existing data types, including structures and unions. It enhances code readability and simplifies complex declarations.  1. Creating Custom Data Types:  - Syntax: `typedef existing\_type new\_type;`  - Example:  typedef int INT;  typedef struct {  int day;  int month;  int year;  } Date;  2. Simplifying Declarations:  - Example;  typedef int (\*CompareFunction)(int, int);  3. Improving Readability:  - Example:  Date currentDate;  4. Structures and Unions:  - Example:  typedef struct {  char name[50];  int age;  } Person;    typedef union {  int num;  float value;  } Number;  . |
| 41.Explain the concept of file handling in C programming. How are files opened,  read from, and written to using standard file handling functions  A:-File Handling in C Programming:  When programming in C, you often need to work with files to read data from them or write data to them. This process is known as file handling. Here's how it works in simple terms:  1. File Pointer:  - To work with a file in C, you use something called a file pointer. Think of it as a special marker that points to where you are in the file.  2. File Modes:  - When you open a file, you can specify how you want to interact with it. For example, you can open it for reading, writing, or both.  3. Opening a File:  - To start working with a file, you need to open it. You do this using the `fopen()` function, which takes the file name and mode as arguments.  FILE \*filePointer;  filePointer = fopen("example.txt", "r")  4. Reading from a File:  - Once the file is open, you can read data from it using functions like `fscanf()` or `fgets()`.  char buffer[100];  fscanf(filePointer, "%s", buffer); // Read a string from the file  5. Writing to a File:  - You can also write data to a file using functions like `fprintf()` or `fputs()`.  fprintf(filePointer, "This is a line written to the file\n");  6. Closing a File:  - After you're done working with a file, it's important to close it using the `fclose()` function to free up system resources.  fclose(filePointer);  Example:  Suppose you have a file named "example.txt" containing some text. Here's how you would read from it using C:  #include <stdio.h>  int main() {  FILE \*filePointer;  char buffer[100];  filePointer = fopen("example.txt", "r");  if (filePointer == NULL) {  printf("File opening failed.");  return 1;  }  fscanf(filePointer, "%s", buffer);  printf("Data read from file: %s\n", buffer);  fclose(filePointer);  return 0;  }  In this example, the program opens "example.txt" for reading, reads a string from it, and then prints that string. Finally, the file is closed. |
| 42.Describe the role of file pointers in C programming. How are file pointers used  to navigate and manipulate files  A:-Role of File Pointers in C Programming:  In C, file pointers are like little guides that help us work with files. Here's how they work:  1. Opening Files  - When we open a file, we get a file pointer that helps us keep track of where we are in the file.  FILE \*filePointer;  filePointer = fopen("example.txt", "r");  2. Navigating within Files:  - We can move around the file using functions like `fseek()` or `rewind()`.  fseek(filePointer, 10, SEEK\_SET); // Move to 10th byte from the beginning  rewind(filePointer); // Move to the beginning  3. Reading and Writing Data:  - File pointers help us read data from files with functions like `fgets()` and write data to files with functions like `fprintf()`.  char buffer[100];  fgets(buffer, 100, filePointer); // Read a line from the file  fprintf(filePointer, "This is a line written to the file\n"); // Write a line  4.Closing Files:  - Finally, we need to close the file using `fclose()` when we're done with it.  fclose(filePointer);  In Short:  - File pointers guide us through files, helping us read, write, and move around efficiently.  - They're essential for managing files in C programs. |
| 43.Discuss the difference between text files and binary files in C programming.  How are they opened and processed differently  A:-Difference Between Text Files and Binary Files in C Programming:  Text Files:  - Human-Readable: Text files store data in a way that humans can easily understand, using character encoding like ASCII or UTF-8.  - Line-by-Line: They're processed line by line and opened in text mode (`"r"`, `"w"`, `"a"`, etc.).  - Examples: Documents, configuration files, source code.  Binary Files:  - Raw Data: Binary files store data as raw bytes, not meant for human consumption.  - Byte-by-Byte; They're processed byte by byte and opened in binary mode (`"rb"`, `"wb"`, `"ab"`, etc.).  - Examples: Complex data structures, like arrays or structures.  Opening and Processing:  - Text files are opened and processed differently from binary files due to their distinct formats and processing methods.  - Text files use functions like `fgets()` and `fprintf()` for processing, while binary files use `fread()` and `fwrite()`.  In short, text files are for human-readable data, while binary files are for raw data storage. They're opened and processed differently in C programs. |
| 44.Explain the purpose of file modes in C programming. Provide examples of  different file modes like "r", "w", "a", etc.  A:-Purpose of File Modes in C Programming:  File modes in C programming specify how a file should be opened and what operations are allowed on it. They control whether the file is opened for reading, writing, or appending. Here's a simple explanation with examples:  1. "r" (Read Mode):  - Opens the file for reading only.  FILE \*file = fopen("example.txt", "r");  2. "w" (Write Mode):  - Opens the file for writing only. If the file exists, it's truncated; if not, a new file is created.  FILE \*file = fopen("example.txt", "w");  3. "a" (Append Mode):  - Opens the file for writing only. Data is appended to the end of the file. If the file doesn't exist, a new file is created.  FILE \*file = fopen("example.txt", "a");  4. "r+" (Read/Write Mode):  - Opens the file for both reading and writing. The file must exist.  FILE \*file = fopen("example.txt", "r+");  5. "w+" (Read/Write Mode):  - Opens the file for both reading and writing. If the file exists, it's truncated; if not, a new file is created.  FILE \*file = fopen("example.txt", "w+");  6. "a+" (Read/Append Mode):  - Opens the file for both reading and appending. Data is appended to the end of the file. If the file doesn't exist, a new file is created.  FILE \*file = fopen("example.txt", "a+");  These modes provide flexibility in how files are opened and manipulated, allowing for various reading, writing, and appending operations. |
| 45.Describe error handling techniques in file operations in C programming. How  are errors detected and handled when working with files?  A:-Error Handling in C File Operations:  When working with files in C, it's crucial to handle errors properly. Here's how you can do it:  1. Check Return Values:  - Always check if file operations like `fopen()` succeed by examining their return values.  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  // Handle error  }  2. Use perror() for Message  - If an error occurs, use `perror()` to print a helpful error message.  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  perror("Error opening file");  exit(EXIT\_FAILURE);  }  3. Handle Specific Errors:  - Depending on the error, you may want to provide specific messages or actions.  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  if (errno == ENOENT) {  printf("File does not exist\n");  } else {  perror("Error opening file");  exit(EXIT\_FAILURE);  }  }  4. Cleanup Resources:  - Make sure to close the file and release resources if an error occurs.  c  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  perror("Error opening file");  exit(EXIT\_FAILURE);  }  // File processing code...  fclose(file);  5. Provide Clear Messages:  - Give meaningful error messages to users or log detailed information for debugging.  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  fprintf(stderr, "Error opening file: %s\n", strerror(errno));  exit(EXIT\_FAILURE);  } |

**Part- B**

|  |
| --- |
| 1. **Print Hello** |
| Code  #include <stdio.h>  int main()  {  printf("Hello!");  return 0;  } |
| Output |
| 2.Factorisl of a number |
| Code  #include <stdio.h>  int main()  {  int num, i;  unsigned long long factorial = 1;  printf("Enter a number: ");  scanf("%d", &num);  if (num < 0)  printf("Error! Factorial of a negative number doesn't exist");  else  {  for (i = 1; i <= num; ++i)  {  factorial \*= i;  }  printf("Factorial of %d = %llu", num, factorial);  }  return 0;  } |
| Output |
|  |
| 3.Number is prime or not |
| Code  #include <stdio.h>  int main()  {  int num, isPrime = 1;  printf("Enter a number: ");  scanf("%d", &num);  if (num <= 1)  {  isPrime = 0;  }  else  {  for (int i = 2; i \* i <= num; i++)  {  if (num % i == 0)  {  isPrime = 0;  break;  }  }  }  if (isPrime)  {  printf("%d is a prime number", num);  } else  {  printf("%d is not a prime number", num);  }  return 0;  } |
| Output |
|  |
| 4.Print the Fibonacci series for given limit. |
| Code  #include <stdio.h>  int main()  {  int limit, a = 0, b = 1, c;  printf("Enter the limit for Fibonacci series: ");  scanf("%d", &limit);  printf("Fibonacci Series: %d %d ", a, b);  for (int i=2; i<limit; ++i)  {  c = a + b;  printf("%d ", c);  a = b;  b = c;  }  printf("\n");  return 0;  } |
| Output |
|  |
| 1. Sum of digits |
| Code  #include <stdio.h>  int main() {  int num, sum = 0, remainder;  printf("Enter an integer: ");  scanf("%d", &num);  // Handle negative numbers (optional)  if (num < 0) {  num = -num; // Convert to positive for calculation  }  // Extract and add digits until the number becomes 0  while (num != 0) {  remainder = num % 10;  sum += remainder;  num /= 10;  }  printf("Sum of digits: %d\n", sum);  return 0;  } |
| output |
|  |
| 6.Reverse the digit. |
| Code  #include <stdio.h>  int main() {  int num, reversed = 0, remainder;  printf("Enter an integer: ");  scanf("%d", &num);  // Reverse the digits using a while loop  while (num != 0) {  remainder = num % 10;  reversed = reversed \* 10 + remainder;  num /= 10;  }  printf("Reversed number: %d\n", reversed);  return 0;  } |
| Output |
|  |
| 7.Check given number is palindrome or not. |
| Code  #include <stdio.h>  int main()  {  int num, revNum = 0, ogNum, rmdr;  printf("Enter a number: ");  scanf("%d", &num);  ogNum = num;  while (num > 0)  {  rmdr = num % 10;  revNum = revNum \* 10 + rmdr;  num /= 10;  }  if (ogNum == revNum)  {  printf("%d is a palindrome", ogNum);  }  else  {  printf("%d is not a palindrome", ogNum);  }  return 0;  } |
| Output |
|  |
| 8. c code for area of a circle. |
| Code  #include <stdio.h>  #define PI 3.14159 // Constant value for pi (can be replaced with a math library function)  int main() {  float radius, area;  printf("Enter the radius of the circle: ");  scanf("%f", &radius);  // Calculate area using pi and radius squared  area = PI \* radius \* radius;  printf("The area of the circle is: %.2f\n", area);  return 0;  } |
| output |
|  |
| 9. Implement a basic calculator with arithmetic operations. |
| Code  #include <stdio.h>  int main() {  char operator;  double num1, num2;  printf("Enter an operator (+, -, \*, /): ");  scanf(" %c", &operator);  printf("Enter two numbers: ");  scanf("%lf %lf", &num1, &num2);  switch (operator) {  case '+':  printf("%.2lf + %.2lf = %.2lf\n", num1, num2, num1 + num2);  break;  case '-':  printf("%.2lf - %.2lf = %.2lf\n", num1, num2, num1 - num2);  break;  case '\*':  printf("%.2lf \* %.2lf = %.2lf\n", num1, num2, num1 \* num2);  break;  case '/':  if (num2 == 0) {  printf("Error: Division by zero\n");  } else {  printf("%.2lf / %.2lf = %.2lf\n", num1, num2, num1 / num2);  }  break;  default:  printf("Invalid operator\n");  }  return 0;  } |
| Output |
|  |
| 10. addition of two arrays |
| Code  #include <stdio.h>  void addArrays(int arr1[], int arr2[], int result[], int size) {  for (int i = 0; i < size; i++) {  result[i] = arr1[i] + arr2[i];  }  }  int main() {  int size, arr1[100], arr2[100], result[100];  printf("Enter the size of the arrays (maximum 100): ");  scanf("%d", &size);  printf("Enter elements for the first array:\n");  for (int i = 0; i < size; i++) {  scanf("%d", &arr1[i]);  }  printf("Enter elements for the second array:\n");  for (int i = 0; i < size; i++) {  scanf("%d", &arr2[i]);  }  addArrays(arr1, arr2, result, size);  printf("Sum of the arrays:\n");  for (int i = 0; i < size; i++) {  printf("%d ", result[i]);  }  printf("\n");  return 0;  } |
| output |
|  |
| 11. concatenation of two strings. |
| Code  #include <stdio.h>  #include <string.h>  int main()  {  char string1[10], string2[10];  printf("Enter the first string: ");  scanf("%s", string1);  printf("Enter the second string: ");  scanf("%s", string2);  strcat(string1, string2);  printf("Concatenated string: %s\n", string1);  return 0;  } |
| output |
|  |
| 12.Search element in array using linear search. |
| Code  #include <stdio.h>  int linearSearch(int arr[], int size, int target) {  for (int i = 0; i < size; i++) {  if (arr[i] == target) {  return i; // Element found, return its index  }  }  return -1; // Element not found  }  int main() {  int arr[100], size, target;  printf("Enter the size of the array (maximum 100): ");  scanf("%d", &size);  printf("Enter elements of the array:\n");  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  printf("Enter the element to search: ");  scanf("%d", &target);  int index = linearSearch(arr, size, target);  if (index != -1) {  printf("Element found at index %d\n", index);  } else {  printf("Element not found in the array\n");  }  return 0;  } |
| output |
|  |
| 13. Search for an element in a sorted array using binary search |
| Code  #include <stdio.h>  int binarySearch(int arr[], int low, int high, int target) {  if (low > high) {  return -1; // Element not found if low exceeds high  }  int mid = low + (high - low) / 2; // Avoid overflow by calculating mid this way  if (arr[mid] == target) {  return mid; // Element found at the middle index  } else if (arr[mid] < target) {  return binarySearch(arr, mid + 1, high, target); // Search in the right half  } else {  return binarySearch(arr, low, mid - 1, target); // Search in the left half  }  }  int main() {  int arr[100], size, target;  printf("Enter the size of the sorted array (maximum 100): ");  scanf("%d", &size);  printf("Enter elements of the sorted array:\n");  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  printf("Enter the element to search: ");  scanf("%d", &target);  int index = binarySearch(arr, 0, size - 1, target);  if (index != -1) {  printf("Element found at index %d\n", index);  } else {  printf("Element not found in the array\n");  }  return 0;  } |
| output |
|  |
| 14. Sort an array using the selection sort algorithm |
| Code  #include <stdio.h>  void selectionSort(int arr[], int size) {  for (int i = 0; i < size - 1; i++) {  // Find the index of the minimum element in the unsorted subarray  int min\_idx = i;  for (int j = i + 1; j < size; j++) {  if (arr[j] < arr[min\_idx]) {  min\_idx = j;  }  }  // Swap the found minimum element with the first element of the unsorted subarray  if (min\_idx != i) {  int temp = arr[min\_idx];  arr[min\_idx] = arr[i];  arr[i] = temp;  }  }  }  int main() {  int arr[100], size;  printf("Enter the size of the array (maximum 100): ");  scanf("%d", &size);  printf("Enter elements of the array:\n");  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  printf("Sorting the array using selection sort...\n");  selectionSort(arr, size);  printf("Sorted array:\n");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| Output |
|  |
| 15. Sort an array using the bubble sort algorithm |
| Code  #include <stdio.h>  void bubbleSort(int arr[], int size) {  for (int i = 0; i < size - 1; i++) {  // Flag to check if any swaps occurred in the current pass  int swapped = 0;  for (int j = 0; j < size - i - 1; j++) {  if (arr[j] > arr[j + 1]) {  // Swap elements if they are in the wrong order  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  swapped = 1; // Set swapped flag if a swap occurred  }  }  // If no swaps occurred in the current pass, the array is already sorted  if (swapped == 0) {  break;  }  }  }  int main() {  int arr[100], size;  printf("Enter the size of the array (maximum 100): ");  scanf("%d", &size);  printf("Enter elements of the array:\n");  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  printf("Sorting the array using bubble sort...\n");  bubbleSort(arr, size);  printf("Sorted array:\n");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| Output |
|  |
| 16. Sort an array using the insertion sort algorithm. |
| Code  #include <stdio.h>  void insertionSort(int arr[], int size) {  for (int i = 1; i < size; i++) {  int key = arr[i]; // Save the current element as key  int j = i - 1;  // Shift elements greater than key one position ahead  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j--;  }  // Insert the key in its correct position  arr[j + 1] = key;  }  }  int main() {  int arr[100], size;  printf("Enter the size of the array (maximum 100): ");  scanf("%d", &size);  printf("Enter elements of the array:\n");  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  printf("Sorting the array using insertion sort...\n");  insertionSort(arr, size);  printf("Sorted array:\n");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| Output |
| z |
| 17.Multiplication of two matrix |
| Code  #include <stdio.h>  #define ROW\_A 3  #define COL\_A 3  #define ROW\_B 3  #define COL\_B 3  void matrixMultiplication(int A[][COL\_A], int B[][COL\_B], int C[][COL\_B]) {  for (int i = 0; i < ROW\_A; i++) {  for (int j = 0; j < COL\_B; j++) {  C[i][j] = 0;  for (int k = 0; k < COL\_A; k++) {  C[i][j] += A[i][k] \* B[k][j];  }  }  }  }  void displayMatrix(int matrix[][COL\_B], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  printf("%d\t", matrix[i][j]);  }  printf("\n");  }  }  int main() {  int A[ROW\_A][COL\_A] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };  int B[ROW\_B][COL\_B] = { {9, 8, 7}, {6, 5, 4}, {3, 2, 1} };  int C[ROW\_A][COL\_B];  matrixMultiplication(A, B, C);  printf("Matrix A:\n");  displayMatrix(A, ROW\_A, COL\_A);  printf("\nMatrix B:\n");  displayMatrix(B, ROW\_B, COL\_B);  printf("\nMatrix C (Result of multiplication):\n");  displayMatrix(C, ROW\_A, COL\_B);  return 0;  } |
| Output |
|  |

**Part- C**

|  |
| --- |
| 1.Implement basic operation like insertion at front using linked list. |
| Code #include <stdio.h>  #include <stdlib.h>  // Define the node structure  struct Node {  int data;  struct Node\* next;  };  // Function to insert a node at the beginning of the list  struct Node\* insertAtBeginning(struct Node\* head, int data) {  // Allocate memory for the new node  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  // Make the new node point to the current head  newNode->next = head;  // Update the head to point to the new node  head = newNode;  return head;  }  int main() {  struct Node\* head = NULL; // Initially empty list  // Insert elements at the beginning  head = insertAtBeginning(head, 5);  head = insertAtBeginning(head, 3);  head = insertAtBeginning(head, 1);  printf("Linked list after insertion: ");  // You can call a separate traversal function here to print the list  while (head != NULL) {  printf("%d ", head->data);  head = head->next;  }  printf("\n");  return 0;  } |
|  |
|  |
| 2. Implement basic operation like deletion at front using linked list. |
| Code #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  // Function to delete a node from the front of the linked list  void deleteFront(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is empty. Nothing to delete.\n");  return;  }  struct Node\* temp = \*head;  \*head = (\*head)->next;  free(temp);  printf("Node deleted from the front successfully.\n");  }  // Function to insert a node at the front of the linked list  void insertFront(struct Node\*\* head, int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->next = \*head;  \*head = newNode;  printf("Node inserted at the front successfully.\n");  }  // Function to print the linked list  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  // Inserting some nodes at the front  insertFront(&head, 3);  insertFront(&head, 7);  insertFront(&head, 9);  insertFront(&head, 5);  printf("Original list: ");  printList(head);  // Deleting a node from the front  deleteFront(&head);  printf("List after deletion: ");  printList(head);  return 0;  } |
| Output |
|  |
| 3. Implement basic operation like insertion at end using linked list. |
| Code  #include <stdio.h>  #include <stdlib.h>  // Define the node structure  struct Node {  int data;  struct Node\* next;  };  // Function to insert a node at the end of the list  void insertAtEnd(struct Node\*\* head\_ref, int data) {  // 1. Allocate memory for the new node  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->next = NULL;  // 2. Handle the case of an empty list  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  return;  }  // 3. Traverse the existing list to find the last node  struct Node\* last = \*head\_ref;  while (last->next != NULL) {  last = last->next;  }  // 4. The last node's 'next' pointer now points to the new node  last->next = newNode;  }  int main() {  struct Node\* head = NULL; // Initially empty list  // Insert elements at the end  insertAtEnd(&head, 5);  insertAtEnd(&head, 3);  insertAtEnd(&head, 1);  printf("Linked list after insertion: ");  // You can call a separate traversal function here to print the list  while (head != NULL) {  printf("%d ", head->data);  head = head->next;  }  printf("\n");  return 0;  } |
| Output |
|  |
| 4. Implement basic operation like deletion at end using linked list. |
| Code  #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to insert a node at the end of the linked list  void insertEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  return;  }  struct Node\* temp = \*head;  while (temp->next != NULL) {  temp = temp->next;  }  temp->next = newNode;  }  // Function to delete a node from the end of the linked list  void deleteEnd(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is empty. Nothing to delete.\n");  return;  }  if ((\*head)->next == NULL) {  free(\*head);  \*head = NULL;  printf("Node deleted from the end successfully.\n");  return;  }  struct Node\* secondLast = \*head;  while (secondLast->next->next != NULL) {  secondLast = secondLast->next;  }  free(secondLast->next);  secondLast->next = NULL;  printf("Node deleted from the end successfully.\n");  }  // Function to print the linked list  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  // Inserting some nodes at the end  insertEnd(&head, 3);  insertEnd(&head, 7);  insertEnd(&head, 9);  insertEnd(&head, 5);  printf("Original list: ");  printList(head);  // Deleting a node from the end  deleteEnd(&head);  printf("List after deletion: ");  printList(head);  return 0;  } |
| Output |
|  |
| 5. implementation for inserting a node at a specific position in a singly linked list |
| Code  #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to insert a node at a specific position in the linked list  void insertAtPosition(struct Node\*\* head, int data, int position) {  if (position < 1) {  printf("Invalid position. Position should be >= 1.\n");  return;  }    struct Node\* newNode = createNode(data);  if (position == 1) {  newNode->next = \*head;  \*head = newNode;  printf("Node inserted at position %d successfully.\n", position);  return;  }    struct Node\* temp = \*head;  for (int i = 1; i < position - 1 && temp != NULL; ++i) {  temp = temp->next;  }  if (temp == NULL) {  printf("Invalid position. Position exceeds the length of the list.\n");  return;  }    newNode->next = temp->next;  temp->next = newNode;  printf("Node inserted at position %d successfully.\n", position);  }  // Function to print the linked list  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  // Inserting some nodes at the end  insertAtPosition(&head, 3, 1); // Insert at position 1  insertAtPosition(&head, 7, 2); // Insert at position 2  insertAtPosition(&head, 9, 2); // Insert at position 2  insertAtPosition(&head, 5, 4); // Insert at position 4  insertAtPosition(&head, 11, 3); // Insert at position 3  printf("List after insertions: ");  printList(head);  return 0;  } |
| Output |
|  |
| 6. implementation for deleting a node at a specific position in a singly linked list in C: |
| Code #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to insert a node at the end of the linked list  void insertEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  return;  }  struct Node\* temp = \*head;  while (temp->next != NULL) {  temp = temp->next;  }  temp->next = newNode;  }  // Function to delete a node at a specific position in the linked list  void deleteAtPosition(struct Node\*\* head, int position) {  if (\*head == NULL) {  printf("List is empty. Nothing to delete.\n");  return;  }  struct Node\* temp = \*head;  if (position == 1) {  \*head = temp->next;  free(temp);  printf("Node deleted from position %d successfully.\n", position);  return;  }  for (int i = 1; temp != NULL && i < position - 1; i++) {  temp = temp->next;  }  if (temp == NULL || temp->next == NULL) {  printf("Invalid position. Position exceeds the length of the list.\n");  return;  }  struct Node\* nextNode = temp->next->next;  free(temp->next);  temp->next = nextNode;  printf("Node deleted from position %d successfully.\n", position);  }  // Function to print the linked list  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  // Inserting some nodes at the end  insertEnd(&head, 3);  insertEnd(&head, 7);  insertEnd(&head, 9);  insertEnd(&head, 5);  printf("Original list: ");  printList(head);  // Deleting a node at a specific position  int position = 3; // Position to delete  deleteAtPosition(&head, position);  printf("List after deletion at position %d: ", position);  printList(head);  return 0;  } |
| Output |
|  |
| 7. Implement combination of insertion and deletion at front using doubly linked list. |
| Code #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to insert a node at the beginning of the linked list  void insertFront(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  newNode->next = \*head;  \*head = newNode;  printf("Node inserted at the front successfully.\n");  }  // Function to delete a node from the front of the linked list  void deleteFront(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is empty. Nothing to delete.\n");  return;  }  struct Node\* temp = \*head;  \*head = (\*head)->next;  free(temp);  printf("Node deleted from the front successfully.\n");  }  // Function to print the linked list  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  // Inserting some nodes at the front  insertFront(&head, 3);  insertFront(&head, 7);  insertFront(&head, 9);  insertFront(&head, 5);  printf("Original list: ");  printList(head);  // Deleting a node from the front  deleteFront(&head);  printf("List after deletion at front: ");  printList(head);  return 0;  } |
| Output |
|  |
| 8. Array-based implementation on push operation of a stack |
| Code  #include <stdio.h>  #define MAX\_SIZE 100 // Define the maximum size of the stack  int stack[MAX\_SIZE];  int top = -1; // Initialize top to -1 for an empty stack  void push(int data) {  // Check if stack is full  if (top == MAX\_SIZE - 1) {  printf("Stack Overflow.\n");  return;  }  // Increment top and insert data  top++;  stack[top] = data;  printf("Element %d pushed to stack.\n", data);  }  int main() {  push(5);  push(3);  push(1);  // You can add code to print the stack contents or perform other operations  return 0;  } |
| Output |
|  |
| 9. **code for the array-based pop function in C** |
| Code  #include <stdio.h>  #define MAX\_SIZE 100 // Define the maximum size of the stack  int stack[MAX\_SIZE];  int top = -1; // Initialize top to -1 for an empty stack  // Function to pop an element from the stack  int pop() {  // Check if stack is empty  if (top == -1) {  printf("Stack Underflow.\n");  return -1; // Indicate error or a default value  }  // Decrement top and return the popped element  int data = stack[top];  top--;  printf("Element %d popped from stack.\n", data);  return data;  }  int main() {  // Push some elements to demonstrate pop  push(5);  push(3);  push(1);  printf("Popped element: %d\n", pop()); // Outputs 1  printf("Popped element: %d\n", pop()); // Outputs 3  printf("Popped element: %d\n", pop()); // Outputs 5  printf("Popped element: %d\n", pop()); // Causes underflow  return 0;  }  // Function for pushing elements (included for completeness)  void push(int data) {  // Check if stack is full  if (top == MAX\_SIZE - 1) {  printf("Stack Overflow.\n");  return;  }  // Increment top and insert data  top++;  stack[top] = data;  printf("Element %d pushed to stack.\n", data);  } |
| Output |
|  |
| 10. code to display the contents of a stack, assuming an array-based implementation |
| Code  #include <stdio.h>  #define MAX\_SIZE 100 // Define the maximum size of the stack  int stack[MAX\_SIZE];  int top = -1; // Initialize top to -1 for an empty stack  // Function to push an element onto the stack  void push(int data) {  // Check if stack is full (omitted for brevity)  if (top == MAX\_SIZE - 1) {  printf("Stack overflow. Cannot push element %d.\n", data);  return;  }  top++;  stack[top] = data;  printf("Element %d pushed to stack.\n", data);  }  // Function to pop an element from the stack  void pop() {  // Check if stack is empty  if (top == -1) {  printf("Stack underflow. Cannot pop element.\n");  return;  }  printf("Element %d popped from stack.\n", stack[top]);  top--;  }  // Function to display the contents of the stack  void display() {  // Check if stack is empty  if (top == -1) {  printf("Stack is empty.\n");  return;  }  // Iterate from top to bottom and print elements  printf("Stack contents:\n");  for (int i = top; i >= 0; i--) {  printf("%d ", stack[i]);  }  printf("\n");  }  int main() {  push(5);  push(3);  push(1);  display(); // Display the stack contents: 1 3 5  pop(); // Pop an element  display(); // Display the stack contents after pop: 1 3  return 0;  } |
| Output |
|  |
| 11.isEmplty function integrated into stack |
| Code  #include <stdio.h>  #define MAX\_SIZE 100 // Define the maximum size of the stack  int stack[MAX\_SIZE];  int top = -1; // Initialize top to -1 for an empty stack  // Function to check if the stack is empty  int isEmpty() {  return top == -1;  }  // Function to push an element onto the stack  void push(int data) {  // ... (implementation remains unchanged)  }  // Function to pop an element from the stack  int pop() {  // ... (implementation remains unchanged)  }  int main() {  if (isEmpty()) {  printf("Stack is initially empty.\n");  }  push(5);  push(3);  if (!isEmpty()) {  printf("Stack is not empty after pushing elements.\n");  }  // ... (more operations as needed)  return 0;  } |
| Output |
|  |
| 12. C code for a stack implementation using an array, including the peek operation. |
| Code  #include <stdio.h>  #define MAX\_SIZE 100 // Define the maximum size of the stack  int stack[MAX\_SIZE];  int top = -1; // Initialize top to -1 for an empty stack  // Function to push an element onto the stack  void push(int data) {  // Check if stack is full  if (top == MAX\_SIZE - 1) {  printf("Stack Overflow.\n");  return;  }  // Increment top and insert data  top++;  stack[top] = data;  printf("Element %d pushed to stack.\n", data);  }  // Function to pop an element from the stack  int pop() {  // Check if stack is empty  if (top == -1) {  printf("Stack Underflow.\n");  return -1; // Indicate error or a default value  }  // Decrement top and return the popped element  int data = stack[top];  top--;  printf("Element %d popped from stack.\n", data);  return data;  }  // Function to peek at the top element without removing it  int peek() {  // Check if stack is empty  if (top == -1) {  printf("Stack is empty.\n");  return -1; // Indicate error or a default value  }  // Return the top element without modifying the stack  return stack[top];  }  int main() {  push(5);  push(3);  push(1);  printf("Top element (peek): %d\n", peek()); // Outputs 1 (without popping)  pop();  printf("Top element (peek) after pop: %d\n", peek()); // Outputs 3 (without popping)  return 0;  } |
| Output |
|  |
| 13. Write a c code for basic implementation of stack using a linked list. |
| Code  #include <stdio.h>  #include <stdlib.h>  // Define a node structure for the linked list  struct Node {  int data;  struct Node\* next;  };  // Global variable to store the head of the linked list (top of the stack)  struct Node\* head = NULL;  // Function to push an element onto the stack  void push(int data) {  // Create a new node  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  // Set the new node's next pointer to the current head  newNode->next = head;  // Update the head to point to the new node  head = newNode;  printf("Element %d pushed to stack.\n", data);  }  // Function to pop an element from the stack  int pop() {  // Check if stack is empty  if (head == NULL) {  printf("Stack Underflow.\n");  return -1; // Indicate error or a default value  }  // Store the data from the head node  int data = head->data;  // Update the head to point to the next node  struct Node\* temp = head;  head = head->next;  // Free the memory of the popped node  free(temp);  printf("Element %d popped from stack.\n", data);  return data;  }  // Function to check if the stack is empty  int isEmpty() {  return head == NULL;  }  int main() {  push(5);  push(3);  push(1);  printf("Popped element: %d\n", pop());  printf("Popped element: %d\n", pop());  printf("Popped element: %d\n", pop());  // Stack is empty now  if (isEmpty()) {  printf("Stack is empty.\n");  }  return 0;  } |
| Output |
|  |
| 14. demonstrating a stack implementation in C to evaluate simple arithmetic expressions in postfix notation |
| Code  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100  int stack[MAX\_SIZE];  int top = -1;  int isDigit(char ch) {  return ch >= '0' && ch <= '9';  }  int precedence(char op) {  switch (op) {  case '+':  case '-':  return 1;  case '\*':  case '/':  return 2;  default:  return -1; // Invalid operator  }  }  void push(int data) {  if (top == MAX\_SIZE - 1) {  printf("Stack Overflow.\n");  return;  }  top++;  stack[top] = data;  }  int pop() {  if (top == -1) {  printf("Stack Underflow.\n");  return -1; // Indicate error or a default value  }  return stack[top--];  }  int peek() {  if (top == -1) {  printf("Stack is empty.\n");  return -1; // Indicate error or a default value  }  return stack[top];  }  int evaluatePostfix(char\* exp) {  int i = 0;  while (exp[i] != '\0') {  char ch = exp[i];  if (isDigit(ch)) {  int num = ch - '0';  push(num);  } else {  int val1 = pop();  int val2 = pop();  switch (ch) {  case '+':  push(val2 + val1);  break;  case '-':  push(val2 - val1);  break;  case '\*':  push(val2 \* val1);  break;  case '/':  if (val1 == 0) {  printf("Division by zero.\n");  return -1; // Indicate error  }  push(val2 / val1);  break;  default:  printf("Invalid operator.\n");  return -1; // Indicate error  }  }  i++;  }  // If there's only one element left, it's the result  if (top == 0) {  return pop();  } else {  printf("Invalid postfix expression.\n");  return -1; // Indicate error  }  }  int main() {  char exp[] = "231\*+"; // Evaluate 2 \* 3 + 1  int result = evaluatePostfix(exp);  if (result != -1) {  printf("Result: %d\n", result);  }  return 0;  } |
| Output |
|  |
| 15. **Enqueue** |
|  |
|  |
| 16.Front Dequeue |
|  |
|  |
| 17. **E**nqueue Front |
|  |
|  |
| 18..Enqueue Rare |
|  |
|  |
| 19.Sort Queue |
|  |
|  |
| 20..bubble sort |
|  |
|  |
| 21. Reverse the queue |
|  |
|  |
| 22..Implementation of Queue using stacks |
|  |
|  |
| 23.Circular Queue |
|  |
|  |
| 24.Priority queue |
|  |
|  |
| 25.Length of string using Queues |
|  |
|  |
| 26.Concatenated string using Queue. |
|  |
|  |
| 27.Armstrong number using Queue |
|  |
|  |
| 28.To Find the coupon winner Using Queues |
|  |
|  |
| 29.Fibonocci Series |
|  |
|  |
| 30.Series of perfect squares using queues |
|  |
|  |