C Programs of last 6 years

Program 1. Write a program in c to illustrate the caparison of structure variables.
Ans.

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

// Define a structure for a person
struct Person {
   char name[50];
   int age;
};

int main() {
   // Declare two structure variables of type Person
   struct Person person1, person2;

   // Initialize person1
   strcpy(person1.name, "Bhavy");
   person1.age = 25;

   // Initialize person2
   strcpy(person2.name, "Doremon");
   person2.age = 30;

   // Compare the ages of person1 and person2
   if (person1.age > person2.age) {
      printf("%s is eligible to marry %s\n", person1.name, person2.name);
   } else if (person1.age < person2.age) {
      printf("%s is currently not eligible for marry, So do it latter %s\n",
person2.name, person1.name);
   } else {
      printf("%s and %s are of the same age\n", person1.name, person2.name);
   }

   return 0;
}
```

Program 2. Write a program in C using pointer to read in an array of integers and print its elements in reverse order.

Ans.

```
#include <stdio.h>

int main() {
   int n;

   printf("Enter the number of elements in the array: ");
   scanf("%d", &n);

   int arr[n]; // Declare an array of size n

   printf("Enter %d integers:\n", n);
   // Read integers into the array using pointer arithmetic
   for (int *ptr = arr; ptr < arr + n; ptr++) {
      scanf("%d", ptr);
   }

   printf("The elements of the array in reverse order are:\n");
   // Print elements in reverse order using pointer arithmetic
   for (int *ptr = arr + n - 1; ptr >= arr; ptr--) {
      printf("%d ", *ptr);
   }
   printf("\n");

   return 0;
}
```

Program 3. Write a program in C to illustrate the use of structure pointers.
Ans.

```
#include <stdio.h>

// Define a structure for a student
struct Student {
   char name[50];
   int age;
```

```
    float marks;
};

int main() {
    // Declare a structure variable of type Student
    struct Student student1;

    // Declare a pointer to a structure of type Student
    struct Student *ptr;

    // Assign the address of student1 to ptr
    ptr = &student1;

    // Access and modify structure members using the pointer
    printf("Enter student's name: ");
    scanf("%s", ptr->name); // Accessing structure member using '->'

    printf("Enter student's age: ");
    scanf("%d", &ptr->age);

    printf("Enter student's marks: ");
    scanf("%f", &ptr->marks);

    // Print the details of the student using the pointer
    printf("\nStudent Details:\n");
    printf("Name: %s\n", ptr->name);
    printf("Age: %d\n", ptr->age);
    printf("Marks: %.2f\n", ptr->marks);

    return 0;
}
```

Program 4. Write a program in C which will read a text and count all occurrences of a particular word.
Ans.

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

int main() {
    char text[1000], word[100];
    int count = 0;

    printf("Enter a text: ");
    fgets(text, sizeof(text), stdin);

    printf("Enter the word to count: ");
    scanf("%s", word);

    // Convert both the text and word to lowercase for case-insensitive comparison
    strlwr(text);
    strlwr(word);

    // Count occurrences of the word in the text
    char *ptr = strstr(text, word);
    while (ptr != NULL) {
        count++;
        ptr = strstr(ptr + 1, word);
    }

    printf("The word '%s' occurs %d times in the text.\n", word, count);

    return 0;
}
```

Program 5. Write a function in C using pointers to add two matrix and to return the resultant matrix to the calling function.
Ans.

```
#include <stdio.h>

// Function to add two matrices
void addMatrices(int mat1[2][2], int mat2[2][2], int result[2][2]) {
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            result[i][j] = mat1[i][j] + mat2[i][j];
        }
    }
}

int main() {
    int mat1[2][2] = {{1, 2}, {3, 4}};
    int mat2[2][2] = {{5, 6}, {7, 8}};
    int result[2][2];

    // Add the matrices
    addMatrices(mat1, mat2, result);

    // Print the result matrix
    printf("Resultant Matrix:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Program 6. Write a program in c to interchange second element with last element of an array.
Ans.

```
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5}; // Sample array
    int n = 5; // size of the array

    if (n >= 2) {
        // Interchange the second element with the last element
        int temp = arr[1];
        arr[1] = arr[n - 1];
        arr[n - 1] = temp;

        // Print the modified array
        printf("Modified Array: ");
        for (int i = 0; i < n; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
    } else {
        printf("Array should have at least 2 elements to interchange.\n");
    }

    return 0;
}
```

Program 7. Write a program in c to find concatenate of two string using pointers without Library function.
Ans.

```
#include <stdio.h>

int main() {
    char str1[100], str2[100], *ptr;

    // Input the first string
    printf("Enter the first string: ");
    scanf("%s", str1);

    // Input the second string
    printf("Enter the second string: ");
    scanf("%s", str2);

    // Find the end of the first string
    ptr = str1;
    while (*ptr != '\0') {
        ptr++;
    }

    // Concatenate the second string to the first string
    while (*str2 != '\0') {
        *ptr = *str2;
        ptr++;
        str2++;
    }
    *ptr = '\0'; // Add null terminator to the end

    // Print the concatenated string
    printf("Concatenated string: %s\n", str1);
    return 0;
}
```

Program 8. Write a program using pointers to search a value from an array.
Ans.

```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = 5; // size of array
    int value, index = -1; // Initialize index to -1

    printf("Enter the value to search: ");
    scanf("%d", &value);

    // Search for the value in the array
    for (int i = 0; i < n; i++) {
        if (arr[i] == value) {
            index = i; // Store the index if the value is found
            break; // Break the loop once the value is found
        }
    }

    // Check if the value is found or not
    if (index != -1) {
        printf("Value %d found at index %d.\n", value, index);
    } else {
        printf("Value %d not found in the array.\n", value);
    }

    return 0;
}
```

Program 9.In a class there are 5 students. Each student is supposed to appear in 3 tests. Write a program using 2-D array to print.
(i) the marks obtained by each student in different subjects.
(ii) total marks and average obtained by each student.

Ans.

```
#include <stdio.h>

int main() {
   // Define constants for number of students and tests
   const int NUM_STUDENTS = 5;
   const int NUM_TESTS = 3;

   // Declare a 2-D array to store marks for each student and test
   int marks[NUM_STUDENTS][NUM_TESTS];

   // Input marks for each student and test
   for (int i = 0; i < NUM_STUDENTS; i++) {
      printf("Enter marks for student %d:\n", i + 1);
      for (int j = 0; j < NUM_TESTS; j++) {
         printf("Test %d: ", j + 1);
         scanf("%d", &marks[i][j]);
      }
   }

   // Print marks obtained by each student in different subjects
   printf("\nMarks obtained by each student in different subjects:\n");
   for (int i = 0; i < NUM_STUDENTS; i++) {
      printf("Student %d:", i + 1);
      for (int j = 0; j < NUM_TESTS; j++) {
         printf(" %d", marks[i][j]);
      }
      printf("\n");
   }

   // Calculate total marks and average obtained by each student
   printf("\nTotal marks and average obtained by each student:\n");
   for (int i = 0; i < NUM_STUDENTS; i++) {
      int total = 0;
      for (int j = 0; j < NUM_TESTS; j++) {
         total += marks[i][j];
      }
      float average = (float)total / NUM_TESTS;
      printf("Student %d - Total: %d, Average: %.2f\n", i + 1, total, average);
   }
```

```
    return 0;
}
```

Program 10. Write C program to copy the contents of one file into another file.
Ans.

```
#include <stdio.h>

int main() {
    char sourceFileName[100], destinationFileName[100];
    char ch;

    // Input the name of the source file
    printf("Enter the name of the source file: ");
    scanf("%s", sourceFileName);

    // Input the name of the destination file
    printf("Enter the name of the destination file: ");
    scanf("%s", destinationFileName);

    // Open the source file in read mode
    FILE *sourceFile = fopen(sourceFileName, "r");
    if (sourceFile == NULL) {
        printf("Error opening the source file.\n");
        return 1;
    }

    // Open the destination file in write mode
    FILE *destinationFile = fopen(destinationFileName, "w");
    if (destinationFile == NULL) {
        printf("Error opening the destination file.\n");
        fclose(sourceFile);
        return 1;
    }

    // Copy contents of source file to destination file
    while ((ch = fgetc(sourceFile)) != EOF) {
        fputc(ch, destinationFile);
    }
    // Close both files
    fclose(sourceFile);
    fclose(destinationFile);

    printf("File copied successfully.\n");

    return 0;
}
```

Program 11. Write a program to explain the use of structure with function.
Ans.

```
#include <stdio.h>

// Define a structure
struct Person {
    char name[50];
    int age;
};

// Function to display details of a person
void displayPerson(struct Person p) {
    printf("Name: %s\n", p.name);
    printf("Age: %d\n", p.age);
}

int main() {
    // Declare and initialize a structure variable
    struct Person person1 = {"Bhavy", 18};

    // Call the function to display details of the person
    displayPerson(person1);

    return 0;
}
```

Program 12. Write a C program to find the reverse of each word Of a string (how are you : output Woh era uoy)
Ans.

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

int main() {
   char str[100];
   int start = 0, end = 0;

   // Input the string
   printf("Enter a string: ");
   fgets(str, sizeof(str), stdin);
   str[strlen(str) - 1] = '\0'; // Remove the newline character from fgets

   // Reverse each word in the string
   for (int i = 0; i <= strlen(str); i++) {
      if (str[i] == ' ' || str[i] == '\0') {
         for (int j = i - 1; j >= start; j--) {
            printf("%c", str[j]);
         }
         printf(" ");
         start = i + 1;
      }
   }

   return 0;
}
```

Program 13. Write a program to read a text file, convert all the lower case characters into upper case.
Ans.

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

int main() {
   FILE *inputFile, *outputFile;
   char inputFileName[100], outputFileName[100];
   char ch;

   // Input the name of the input file
   printf("Enter the name of the input file: ");
```

```
    scanf("%s", inputFileName);

    // Input the name of the output file
    printf("Enter the name of the output file: ");
    scanf("%s", outputFileName);

    // Open the input file for reading
    inputFile = fopen(inputFileName, "r");
    if (inputFile == NULL) {
        printf("Error opening the input file.\n");
        return 1;
    }

    // Open the output file for writing
    outputFile = fopen(outputFileName, "w");
    if (outputFile == NULL) {
        printf("Error opening the output file.\n");
        fclose(inputFile);
        return 1;
    }

    // Read each character from the input file, convert to uppercase, and write to
output file
    while ((ch = fgetc(inputFile)) != EOF) {
        if (islower(ch)) {
            ch = toupper(ch);
        }
        fputc(ch, outputFile);
    }

    // Close both files
    fclose(inputFile);
    fclose(outputFile);

    printf("Conversion completed successfully.\n");

    return 0;
}
```

Program 14. Write a program to swap two numbers using pointer with structure.
Ans.

```
#include <stdio.h>

// Define a structure to hold two integers
struct Numbers {
    int num1;
    int num2;
};

// Function to swap two numbers using pointers
void swapNumbers(struct Numbers *nums) {
    int temp = nums->num1;
    nums->num1 = nums->num2;
    nums->num2 = temp;
}

int main() {
    struct Numbers numbers;

    // Input two numbers
    printf("Enter two numbers: ");
    scanf("%d %d", &numbers.num1, &numbers.num2);

    // Print the numbers before swapping
    printf("Before swapping: num1 = %d, num2 = %d\n", numbers.num1,
numbers.num2);

    // Swap the numbers using pointers
    swapNumbers(&numbers);

    // Print the numbers after swapping
    printf("After swapping: num1 = %d, num2 = %d\n", numbers.num1,
numbers.num2);

    return 0;
}
```

Program 15. Write a program to pre-multiply a matrix by its transpose.
Ans.

```
#include <stdio.h>

int main() {
    int matrix[2][2] = {{1, 2}, {3, 4}};
    int result[2][2] = {0}; // Initialize result matrix with zeros

    printf("Original Matrix:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    // Transpose the matrix
    int transpose[2][2];
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            transpose[i][j] = matrix[j][i];
        }
    }

    printf("\nTranspose Matrix:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", transpose[i][j]);
        }
        printf("\n");
    }

    // Multiply the original matrix by its transpose
    printf("\nResultant Matrix (Original matrix multiplied by its transpose):\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                result[i][j] += matrix[i][k] * transpose[k][j];
            }
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Program 16. Write a function that receive a sorted array of integers and an integer value, and inserts the value in its correct place.
Ans.

```
#include <stdio.h>

void insertInSortedArray(int arr[], int *size, int value) {
    int i = (*size) - 1;
    while (i >= 0 && arr[i] > value) {
        arr[i + 1] = arr[i--];
    }
    arr[i + 1] = value;
    (*size)++;
}

int main() {
    int arr[100], size, value;
    printf("Enter the size of the sorted array: ");
    scanf("%d", &size);
    printf("Enter the elements of the sorted array: ");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Enter the value to be inserted: ");
    scanf("%d", &value);
    insertInSortedArray(arr, &size, value);
    printf("Sorted array after insertion: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Program 17. Write a program in C to print following pattern :
2 3 4 5 6 7

```
3 4 5 6 7
4 5 6 7
5 6 7
6 7
7
```

Ans.

```c
#include <stdio.h>
int main() {
int rows = 6; // Number of rows in the pattern
for (int i = 2; i <= 7; i++) {
for (int j = i; j <= 7; j++) {
printf("%d ", j);
}
printf("\n");
}
return 0;
}
```

Program 18. Write a program, which read your name from the key board and outputs a list of ASCII codes, which represent your name.

Ans.

```c
#include <stdio.h>
#include <string.h>
void main() {
char name[50];
int i;
// Input name from the user
printf("Enter your name: ");
scanf("%s", name);
// Output ASCII codes of the name
printf("ASCII codes representing your name:\n");
for (i = 0; i < strlen(name); i++) {
printf("%d ", name[i]);
}
printf("\n");
}
```

C Language Important Question by Bhavy Sharma

Q1. Explain the need of array variable?

Ans. In C programming, an array is a collection of elements of the same data type stored at contiguous memory locations. Arrays are essential because they allow us to store multiple values of the same type under a single name, making it easier to manage and access related data.

> **Efficient Storage:** Arrays allow for the efficient storage of multiple elements of the same data type in contiguous memory locations. This leads to better memory utilization and faster access times.
>
> **Sequential Access:** Arrays enable sequential access to elements using index values, simplifying operations such as traversal, searching, and sorting. This sequential access pattern facilitates various data processing tasks efficiently.
>
> **Grouping Data:** Arrays are essential for grouping related data items under a single name. For instance, they can store the grades of students in a class, the temperatures recorded over time, or the elements of a mathematical vector. This grouping of data enhances organization and readability in code, making it easier to manage and manipulate related data sets.

Q2. Distinguish between automatic(local scoped) and static(global scoped) variables.

Ans.

Automatic Variables:

- **Automatic variables have a limited scope, typically within the block or function where they are declared.**
- **They are created when the block or function is entered and destroyed when it exits.**
- **Memory for automatic variables is allocated from the stack.**

Static Variables:

- Static variables have a global scope if declared outside a function or a local scope if declared within a function, but they retain their values across function calls.
- They are initialized only once, and their value persists throughout the program's execution.
- Memory for static variables is allocated from the data segment of the program's memory.

Q3. What is meant by array of structures?

Ans. An array of structures refers to a data structure in programming where each element of the array is itself a structure.

<u>Combination of Data Types:</u> An array of structures allows programmers to combine different data types into a single data structure. Each element of the array holds a complete set of data, organized using a structure.

<u>Efficient Organization:</u> This data structure is useful when dealing with related but distinct pieces of information. For example, in a program tracking student records, each element of the array can represent a student's information, such as their name, roll number, and grades, organized within a Structure.

<u>Simplified Access and Manipulation:</u> With an array of structures, accessing and manipulating related data becomes more straightforward. Programmers can use array indexing to access individual elements and structure members to retrieve or modify specific data fields. This simplifies tasks such as searching, sorting, and updating data within the array.

Q4. What is a pointer? How is a pointer initialized?

Ans.

<u>Pointer Definition:</u>

- A pointer is a variable that stores the memory address of another variable. It "points to" the location of a data element in the computer's memory.

<u>Initialization of Pointers:</u>

- Pointers are initialized by assigning them the memory address of another variable using the address-of operator (`&`) in C programming.
- For example, to initialize a pointer `ptr` to point to the memory address of a variable `x`, which is integer type you would write `int *ptr = &x;`

Q5. Describe the use and limitations of function getc.

Ans.

Use of `getc` Function:

- The `getc` function is used to read a single character from a specified input stream, such as the standard input (`stdin`) or a file stream (`FILE*`).
- It returns the next character from the input stream as an integer representation of the character's ASCII value.
- This function is commonly used when input is needed character by character, such as parsing text files, processing user input, or implementing custom input handling mechanisms.

Limitation of `getc` Function:

- Single Character Reading: The primary limitation of the `getc` function is that it reads only one character at a time. This can be inefficient for tasks requiring reading larger chunks of data, such as reading entire lines or blocks of text.
- Performance Consideration: Since `getc` reads characters one at a time and may involve disk I/O operations for file streams, it can be slower compared to reading larger blocks of data using alternative functions like `fgets` or `fread`.
- End-of-File Handling: `getc` returns a special value (`EOF`) when it encounters the end of the input stream, indicating that no more characters are available. Handling end-of-file conditions appropriately is crucial to prevent errors or infinite loops in programs using `getc`.

Q6. What is a data structure? Why is an array called a data structure? Write a program to read a matrix of size mxn and print its transpose.

Ans.

Data Structure:

A data structure is a way of organizing and storing data in a computer's memory in such a way that it can be efficiently accessed and modified. It defines the relationships between the data elements and facilitates operations such as insertion, deletion, traversal, and searching. Data structures are fundamental for efficient algorithm design and play a crucial role in the development of software systems.

Array as a Data Structure:

An array is called a data structure because it represents a collection of elements stored in contiguous memory locations. It provides a structured way to organize and access multiple elements of the same data type using a single name. Arrays offer efficient storage and retrieval mechanisms, allowing for easy manipulation of data elements through index-based access. Additionally, arrays support various operations such as sorting, searching, and traversal, making them a fundamental building block for implementing more complex data structures.

Program to Print Matrix Transpose:

```
#include <stdio.h>
#include <conio.h>
void main() {
   int m, n, i, j;

   // Input dimensions of the matrix
   printf("Enter the number of rows and columns of the matrix: ");
   scanf("%d %d", &m, &n);

   // Declare matrix of size m x n
   int matrix[m][n];

   // Input elements of the matrix
   printf("Enter the elements of the matrix:\n");
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++) {
         scanf("%d", &matrix[i][j]);
      }
   }

   // Print the original matrix
   printf("Original Matrix:\n");
```

```
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    // Print the transpose of the matrix
    printf("Transpose Matrix:\n");
    for (j = 0; j < n; j++) {
        for (i = 0; i < m; i++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

Q7. Describe the three logical bitwise operators what is the purpose Of each? What types of operands are required by each of the logical bitwise operators?
Ans.

Logical Bitwise Operators:

Logical bitwise operators perform bitwise operations on the individual bits of integer operands. There are three logical bitwise operators: AND, OR, and XOR. Each operator serves a specific purpose and requires integer operands.

AND Operator (&):

- Purpose: The AND operator performs a bitwise AND operation between corresponding bits of two operands. It returns 1 if both bits are 1, otherwise returns 0.
- Example Purpose: The AND operator is commonly used to mask or clear specific bits within an integer by setting certain bits to 0 while preserving others.
- Operand Types: Requires two integer operands.

OR Operator (|):

- Purpose: The OR operator performs a bitwise OR operation between corresponding bits of two operands. It returns 1 if at least one of the bits is 1, otherwise returns 0.
- Example Purpose: The OR operator is often used to set or enable specific bits within an integer by setting certain bits to 1 while preserving others.
- Operand Types: Requires two integer operands.

XOR Operator (^):

- Purpose: The XOR (exclusive OR) operator performs a bitwise XOR operation between corresponding bits of two operands. It returns 1 if the bits are different (one is 1 and the other is 0), otherwise returns 0.
- Example Purpose: The XOR operator is useful for toggling or flipping specific bits within an integer, as it can toggle a bit from 0 to 1 or from 1 to 0.
- Operand Types: Requires two integer operands.

Q8. What is the relationship between an array name and a pointer? How is an array name interpreted when it appears as an argument to a function? How can a function return a pointer to its calling routine?

Ans.

Array Name and Pointer:

- **An array name in C programming is often interpreted as a pointer to the first element of the array. This means that when you use the array name in an expression, it behaves like a pointer pointing to the first element's memory location.**
- **For example, if `arr` is an array, then `arr` and `&arr[0]` represent the same memory address.**

Array Name as an Argument to a Function:

- **When an array name is passed as an argument to a function, it decays into a pointer to its first element.**
- **In other words, the function receives a pointer to the first element of the array, not a copy of the entire array.**
- **This allows functions to operate on arrays of varying sizes without having to know their exact sizes at compile time.**

Returning a Pointer from a Function:

- A function in C can return a pointer to its calling routine by declaring the return type of the function as a pointer type.
- It's crucial to ensure proper memory management, including deallocating memory when it's no longer needed, to prevent memory leaks.

Q9. Character string in C are automatically terminated by the null character. Explain how this feature helps in string manipulations.

Ans. In C programming, character strings are represented as arrays of characters terminated by a special character called the null character ('\0'). This null character marks the end of the string and is automatically appended by C when a string is declared using double quotes or when a string is manipulated using standard string functions.

String Length Determination:

- The presence of the null character at the end of a string allows for easy determination of the string's length. By iterating through the characters of the string until encountering the null character, the length of the string can be calculated efficiently.

Safe String Operations:

- String manipulation functions in C, such as `strcpy`, `strcat`, `strcmp`, etc., rely on the presence of the null character to perform operations safely.
- These functions ensure that they do not read or write beyond the boundaries of the string by stopping when they encounter the null character. This prevents buffer overflows and related security vulnerabilities.

Concatenation and Appending:

- When concatenating or appending strings, the null character ensures that the resulting string remains properly terminated. The null character is appended to the end of the concatenated string, preserving its integrity and allowing it to be treated as a valid string.

<u>String Comparison:</u>

- During string comparison operations, such as with `strcmp`, the presence of the null character allows for accurate comparison of strings. The comparison stops as soon as a difference is encountered, as both strings are properly terminated by null characters.

<u>Traversal and Access:</u>

- The null character allows for easy traversal and access of individual characters within a string. By iterating through the characters until the null character is reached, each character can be accessed or modified as needed.

Q10. main() is a user defined function. How does iS differ from other user- defined functions?

Ans. In C programming, the `main` function holds a special significance compared to other user-defined functions.

<u>Entry Point:</u>

- `main` is the entry point of a C program. When you run a C program, the operating system starts executing the code from the `main` function. It's like the starting point or the gateway to your program.

<u>Execution Control:</u>

- The `main` function controls the execution flow of the entire program. It's where your program begins its execution, and typically, it also signals the end of the program's execution. Anything you want your program to do, you'll likely organize it within or call it from `main`.

<u>Special Return Type:</u>

- Unlike other user-defined functions, `main` has a predefined return type of `int`. This return value serves as an indicator of the program's execution status. By convention, a return value of 0 indicates successful execution, while a non-zero value usually signifies an error or abnormal termination.

Command Line Arguments:

- `main` can accept command-line arguments, which allow you to pass inputs to your program when it starts running. These arguments can be used to customize the behavior of your program based on user input or external factors.

Interactions with the Operating System:

- Since `main` is the starting point of the program, it often interacts with the operating system for tasks such as input/output operations, memory allocation, file handling, and more. It serves as the interface between your program and the underlying system environment.

Unique Naming Convention:

- While `main` is a user-defined function, its name is reserved and has special significance. It must be named `main` for the program to compile and execute correctly. Unlike other user-defined functions, you cannot choose an arbitrary name for `main`.

Q11. Compare the working of the function strcat and strncat. Write a program, which read your name from the key board and outputs a list of ASCII codes, which represent your name.

Ans.

1. strcat Function:

- Purpose: `strcat` stands for "string concatenate." It concatenates (joins) two strings together, appending the characters from the source string onto the end of the destination string.
- Behavior: `strcat` takes two arguments - the destination string (`dest`) and the source string (`src`). It appends the characters from `src` to the end of `dest`. It stops when it encounters the null character ('\0') in `src` and then appends a null character to the end of the concatenated string.
- No Length Limitation: `strcat` does not have a built-in mechanism to limit the number of characters appended from `src`. Therefore, it may lead to buffer

overflow if the `dest` string does not have enough space to accommodate the concatenated string.

Example Program of strcat function :-

```
char dest[20] = "Hello";
char src[] = " World!";
strcat(dest, src);
// After concatenation, dest contains "Hello World!"
```

<u>2. strncat Function:</u>

- **Purpose: `strncat` stands for "string concatenate with a maximum length." It concatenates a specified number of characters from the source string onto the end of the destination string.**
- **Behavior: `strncat` takes three arguments - the destination string (`dest`), the source string (`src`), and the maximum number of characters to concatenate (`n`). It appends at most `n` characters from `src` to the end of `dest`. It also appends a null character to the end of the concatenated string, ensuring null termination.**
- **Length Limitation: `strncat` includes a parameter `n` that specifies the maximum number of characters to append from `src`. This prevents buffer overflow by limiting the length of the concatenated string.**

Example program of strncat function:-

```
char dest[20] = "Hello";
char src[] = " World!";
strncat(dest, src, 3);
// After concatenation, dest contains "Hello Wor"
```

<u>Program to Output ASCII Codes of Name:</u>

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

void main() {
    char name[50];
    int i;

    // Input name from the user
    printf("Enter your name: ");
    scanf("%s", name);
    // Output ASCII codes of the name
    printf("ASCII codes representing your name:\n");
    for (i = 0; i < strlen(name); i++) {
        printf("%d ", name[i]);
    }
    printf("\n");
}
```

Output :-

```
Enter your name: bhavy
ASCII codes representing your name:
98 104 97 118 121
```

Q12. What are the rules that govern the passing of arrays to function? Use recursive function calls to evaluate f(x) = $x^3/3! + x^5/5! - x^7/7! +$.....

Ans.

Rules Governing Passing Arrays to Functions:

Passing by Reference:

- When you pass an array to a function in C, you're actually passing a pointer to the first element of the array. This means that changes made to the array within the function will affect the original array in the calling function.

Array Size is Not Passed:

- C functions do not receive information about the size of the array being passed. It's the responsibility of the programmer to ensure that the function operates within the bounds of the array to avoid accessing out-of-bounds memory.

Array Decay:

- **When an array is passed to a function, it "decays" into a pointer to its first element. This means that the array name behaves like a pointer within the function, and you can use pointer arithmetic to access its elements.**

Passing Arrays of Different Sizes:

- **You can pass arrays of different sizes to the same function by using pointers and specifying the size of the array as a separate argument.**

Program to Evaluation of the Function f(x) = x^3/3! + x^5/5! - x^7/7! + ... using Recursive Function Calls:

```
#include <stdio.h>

// Function to calculate factorial
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}


int main() {
    double x;
    printf("Enter the value of x: ");
    scanf("%lf", &x);

    // Number of terms in the series
    int terms;
    printf("Enter the number of terms: ");
    scanf("%d", &terms);

    double result = 0;
    int sign = 1;
    int power = 3;
    for (int i = 1; i <= terms; i++) {
        result += sign * pow(x, power) / factorial(power);
        sign = -sign;
        power += 2;
    }

    printf("Result: %lf\n", result);

    return 0;
}
```

Q13. Explain the meantng and purpose of the following;
(i) Struct keyword
(ii) Typedef keyword
(iii) Size of operator
Ans.

1. Struct Keyword:

- Meaning: In C, the `struct` keyword is used to define a user-defined data type known as a structure. A structure allows you to group together different types of variables under a single name.
- Purpose: The purpose of the `struct` keyword is to create a composite data type that can hold related pieces of information. It enables you to organize and manage complex data structures by grouping together variables of different data types into a single entity. Structures are commonly used for representing records, entities, or objects in a program, making it easier to work with related data.

2. Typedef Keyword:

- Meaning: The `typedef` keyword in C is used to create an alias or a new name for an existing data type. It allows you to define custom names for data types to improve code readability and maintainability.
- Purpose: The purpose of the `typedef` keyword is to provide a way to define more descriptive and meaningful names for existing data types, making the code easier to understand and maintain. It also allows you to abstract away implementation details and create platform-independent code by defining custom data types with platform-specific data types.

3. Sizeof Operator:

- Meaning: The `sizeof` operator in C is used to determine the size, in bytes, of a data type or a variable. It returns the number of bytes occupied by an object or a type, including any padding bytes added by the compiler for alignment purposes.
- Purpose: The purpose of the `sizeof` operator is to provide a way to dynamically determine the size of data types or variables at compile time. It is commonly used in memory allocation, array manipulation, and low-level programming tasks where knowledge of the size of data is essential. By using the `sizeof` operator, you can write more flexible and portable code that adapts to changes in data types or platform requirements.

Q14. Write a function that receive a sorted array of integers and an integer value, and inserts the value in its correct place.
Ans.

```
#include <stdio.h>

void insertInSortedArray(int arr[], int *size, int value) {
   int i = *size - 1; // Index of last element in the array

   // Shift elements to the right until we find the correct position to insert the value
   while (i >= 0 && arr[i] > value) {
      arr[i + 1] = arr[i];
      i--;
   }

   // Insert the value at its correct position
   arr[i + 1] = value;

   // Increase the size of the array
   (*size)++;
}

int main() {
   int arr[100]; // Assuming maximum size of array is 100
   int size, value;
```

```
    // Input the size of the sorted array
    printf("Enter the size of the sorted array: ");
    scanf("%d", &size);

    // Input the elements of the sorted array
    printf("Enter the elements of the sorted array: ");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    // Input the value to be inserted
    printf("Enter the value to be inserted: ");
    scanf("%d", &value);

    // Call the function to insert the value in its correct place
    insertInSortedArray(arr, &size, value);

    // Output the updated sorted array
    printf("Sorted array after insertion: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Q15. Write various data type supported in C with example.

Ans. In c language there are various data types are supported to store different types of data. Which are discuss as follow :-

int: Used to store integer values.
Example :- int num = 10;

float: Used to store floating-point numbers.
Example :- float pi = 3.14;

char: Used to store single characters or small integers represented as characters.
Example :- char grade = 'A';

These are just a few examples of data types supported in C. There are other data types such as `double`, `long`, `short`, `unsigned`, etc., each with its own purpose and range of values.

Q16. What are headers files ? Why they are important ?

Ans. Header files in C are files that contain declarations of functions, data types, constants, and other entities that are used in a program. They typically have a `.h` extension and are included at the beginning of C source files using the `#include` preprocessor directive.

Q17. Write limitations of switch case Statement.

Ans. There are Number of limitations in the switch case statement in C language, Some of them are discuss as follow :-

Limited Expressiveness: Switch cases can only be used to compare the value of a variable against constant integral expressions. This restricts their expressiveness compared to conditional statements like if-else, which allow for more complex conditions.

Cannot Compare Floating-Point Numbers: Switch cases cannot be used to compare floating-point numbers or strings directly. They only work with integer or character constants.

No Range Checking: Switch cases cannot handle ranges of values directly. Each case must represent a specific value, so checking for ranges requires multiple case statements, which can make the code less readable and maintainable.

Q18. Differentiate between scope and lifetime of a variable.

Ans.

Scope of a Variable:

- **Definition: Scope refers to the region of a program where a variable is accessible and can be used.**
- **Duration: The scope of a variable begins at its declaration and ends at the end of the block in which it is declared.**

- **Visibility: Variables with block scope are only visible within the block in which they are declared. They cannot be accessed from outside that block.**
- **Example: In C, variables declared inside a function have block scope and are only accessible within that function.**

Lifetime of a Variable:

- **Definition: Lifetime refers to the duration for which a variable exists in memory during program execution.**
- **Duration: The lifetime of a variable begins when it is initialized or allocated and ends when it goes out of scope or is deallocated.**
- **Memory Management: The memory allocated for a variable is released when its lifetime ends, either automatically at the end of its scope (for automatic variables) or manually through deallocation (for dynamically allocated variables).**
- **Example: In C, local variables have a lifetime limited to the duration of the function call in which they are declared. Once the function returns, the memory allocated for these variables is reclaimed.**

Q19. Define preprocessor and its usage in programming.

Ans. Preprocessor:

- **Definition: The preprocessor is a program that processes source code before it is passed to the compiler. It performs text manipulation tasks specified by directives, such as `#include`, `#define` and so on.**
- **Usage:**

  Including Header Files: The `#include` directive is used to include header files in the source code, allowing access to external functions, data types, and constants.

  Macro Definitions: The `#define` directive is used to define macros, which are symbolic constants or code snippets that are replaced with their respective values during preprocessing.

Example :-

```
#include <stdio.h> // Includes the standard input/output header file
#define PI 3.14 // Defines a macro for the value of pi

int main() {
    int radius = 5;
    float area = PI * radius * radius; // Uses the PI macro
    printf("Area of the circle: %f\n", area);
    return 0;
}
```

Q20. Write short notes on following

(i) Enumerated Data Type

(ii) String

(iii) Macro Expansion

(iv) File Inclusion

Ans. (i) Enumerated Data Type:

- Definition: An enumerated data type (enum) is a user-defined data type in C that consists of a set of named integer constants, known as enumerators.
- Purpose: Enums provide a way to define a set of related named constants, making the code more readable and self-documenting. They help in writing cleaner and more maintainable code by giving meaningful names to integer values.

Example :-

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday};
enum Weekday today = Tuesday;
```

(ii) String:

- Definition: A string in C is a sequence of characters terminated by a null character ('\0'). It is represented as an array of characters.
- Purpose: Strings are widely used for representing and manipulating text data in C programs. They are essential for tasks such as input/output operations, text processing, and string manipulation.

Example :-

```
char str[] = "Hello, world!";
```

(iii) Macro Expansion:

- **Definition: Macro expansion refers to the process of replacing macro identifiers with their respective macro values during preprocessing.**
- **Purpose: Macros are used to define symbolic constants or code snippets that are replaced with their values wherever they are used in the program. Macro expansion helps in improving code readability, reducing redundancy, and enabling code modularity.**

Example :-

```
#define PI 3.14
float area = PI * radius * radius;
```

(iv) File Inclusion:

- **Definition: File inclusion is a preprocessing directive that allows including the contents of one file into another file during compilation.**
- **Purpose: File inclusion enables code reuse and modularity by allowing the use of header files containing declarations and definitions across multiple source files. It simplifies the process of managing and organizing code.**

Example :-

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

Q21. Write a program in C to print following pattern :

```
2 3 4 5 6 7
3 4 5 6 7
4 5 6 7
5 6 7
```

6 7
7

Ans.

```
#include <stdio.h>

int main() {
    int rows = 6; // Number of rows in the pattern

    for (int i = 2; i <= 7; i++) {
        for (int j = i; j <= 7; j++) {
            printf("%d ", j);
        }
        printf("\n");
    }

    return 0;
}
```

Q22. Given an array of 20 integers. Write program in C to search for a given integer in that array.

Ans.

```
#include <stdio.h>

int main() {
    int arr[20]; // Array of 20 integers
    int target, found = 0;

    // Input elements of the array
    printf("Enter 20 integers:\n");
    for (int i = 0; i < 20; i++) {
        scanf("%d", &arr[i]);
    }

    // Input the target integer to search
    printf("Enter the integer to search: ");
    scanf("%d", &target);

    // Search for the target integer in the array
    for (int i = 0; i < 20; i++) {
        if (arr[i] == target) {
            printf("Integer %d found at index %d.\n", target, i);
            found = 1;
            break;
        }
    }

    // Check if the target integer was not found
    if (!found) {
        printf("Integer %d not found in the array.\n", target);
    }

    return 0;
}
```

Q23. Write a program in C to multiply two matrics of Nx N dimension.

Ans.

```
#include <stdio.h>

int main() {
    int N;

    // Input the dimension of the matrices
    printf("Enter the dimension of the matrices: ");
    scanf("%d", &N);

    int mat1[N][N], mat2[N][N], result[N][N];

    // Input elements of the first matrix
    printf("Enter elements of the first matrix:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            scanf("%d", &mat1[i][j]);
        }
    }

    // Input elements of the second matrix
    printf("Enter elements of the second matrix:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            scanf("%d", &mat2[i][j]);
        }
    }

    // Multiply the matrices
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            result[i][j] = 0;
            for (int k = 0; k < N; k++) {
                result[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }

    // Print the result matrix
    printf("Result of matrix multiplication:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Q24. Define recursive function. Write a program to find the factorial Of a number With a recursive function.

Ans. A recursive function is a function that calls itself either directly or indirectly in order to solve a problem. It breaks down the problem into smaller, similar subproblems and calls itself to solve each subproblem. Recursive functions typically have a base case, which defines the simplest form of the problem and terminates the recursion.

Program to find the factorial of a number :-

```
#include <stdio.h>

// Recursive function to calculate factorial
int factorial(int n) {
    // Base case: factorial of 0 is 1
    if (n == 0) {
        return 1;
    }
    // Recursive case: factorial of n is n multiplied by factorial of (n-1)
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num;

    // Input the number
    printf("Enter a number: ");
    scanf("%d", &num);

    // Call the factorial function and print the result
    printf("Factorial of %d is %d\n", num, factorial(num));

    return 0;
}
```

Q25. Write a program to check whether a given number is Armstrong number or not.

Ans. An Armstrong number is a number that is equal to the sum of its own digits raised to the power of the number of digits. For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 153.

Program to find the Armstrong Number:-

```
#include <stdio.h>

int main() {
    int num, originalNum, remainder, result = 0, digits = 0;

    // Input the number from the user
    printf("Enter a number: ");
    scanf("%d", &num);

    originalNum = num;
    // Count the number of digits
    while (originalNum != 0) {
        originalNum /= 10;
        digits++;
    }

    originalNum = num;

    // Calculate sum of digits raised to the power of number of digits
    while (originalNum != 0) {
        remainder = originalNum % 10;
        result += remainder * remainder * remainder;
        originalNum /= 10;
    }

    // Check if the sum is equal to the original number
    if (result == num) {
        printf("%d is an Armstrong number.\n", num);
    } else {
        printf("%d is not an Armstrong number.\n", num);
    }

    return 0;
}
```

Q26. Differentiate between call by value and call by reference.
Ans. <u>Call by Value:</u>

In call by value:

- The value of the actual parameter (argument) is copied into the formal parameter of the function.
- Changes made to the formal parameter inside the function do not affect the actual parameter.
- It is simple and straightforward, but it requires more memory as it creates a separate copy of the parameter.
- It is suitable for small-sized variables or when the function does not need to modify the original value.

Example:

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 5, y = 10;
    swap(x, y); // x and y remain unchanged
    return 0;
}
```

<u>Call by Reference:</u>

In call by reference:

- The memory address (reference) of the actual parameter is passed to the formal parameter of the function.
- Changes made to the formal parameter inside the function affect the actual parameter because they refer to the same memory location.
- It eliminates the need for creating a separate copy of the parameter, thus saving memory.
- It is suitable for large-sized variables or when the function needs to modify the original value.

Example:

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 10;
    swap(&x, &y); // x and y are swapped
    return 0;
}
```

Q27. Differentiate between string and character array.

Ans. <u>String:</u>

> Definition: A string in C is a sequence of characters terminated by a null character ('\0').
> Declaration: Strings are declared using the `char` data type followed by square brackets [].
> Initialization: Strings can be initialized using double quotes "".
> Functions: Strings have built-in library functions for string manipulation, such as `strlen`, `strcpy`, `strcat`, etc.
> Termination: Strings are terminated by a null character ('\0'), which marks the end of the string.
> Example: `char str[] = "Hello";`

<u>Character Array:</u>

> Definition: A character array in C is a fixed-size array that stores characters.
> Declaration: Character arrays are declared using the `char` data type followed by square brackets [] with a specific size.
> Initialization: Character arrays can be initialized like any other array using curly braces {}.
> Functions: Character arrays do not have built-in library functions specifically for string manipulation. However, general array functions can be used.
> Termination: Character arrays are not terminated by a null character ('\0'). They can contain any sequence of characters up to their specified size.
> Example: `char arr[10] = {'H', 'e', 'l', 'l', 'o'};`

Q28. Write a C program to find the reverse of each word Of a string (how are you : output Woh era uoy)

Ans.

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

void reverseWords(char *str) {
   int start = 0;
   int end = 0;

   // Iterate through each character of the string
   while (str[end] != '\0') {
      // Find the start of each word
      while (str[start] == ' ') {
         start++;
```

```
        }
        end = start;

        // Find the end of each word
        while (str[end] != ' ' && str[end] != '\0') {
            end++;
        }

        // Reverse the word
        int left = start;
        int right = end - 1;
        while (left < right) {
            char temp = str[left];
            str[left] = str[right];
            str[right] = temp;
            left++;
            right--;
        }

        // Move to the next word
        start = end;
    }
}

int main() {
    char str[100];

    // Input the string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strlen(str) - 1] = '\0'; // Remove the newline character from fgets

    // Call the function to reverse words in the string
    reverseWords(str);

    // Print the modified string
    printf("Reversed words: %s\n", str);

    return 0;
}
```

Q29. What are three dimensional arrays ? How can you initialize them ?

Ans. Three-dimensional arrays in C are arrays that have three dimensions, allowing data to be stored in a cuboid-like structure. Each element in a three-dimensional array is identified by three indices: one for each dimension.

Initialization of Three-Dimensional Arrays:

Three-dimensional arrays can be initialized in various ways, including:

1. Direct Initialization:
```
int arr[2][3][4] = {
   {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
   {{13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}}
};
```

2. Using Nested Loops:
```
int arr[2][3][4];
for (int i = 0; i < 2; i++) {
   for (int j = 0; j < 3; j++) {
      for (int k = 0; k < 4; k++) {
         arr[i][j][k] = i * 12 + j * 4 + k + 1;
      }
   }
}
```

3. Using Assignments:
```
int arr[2][3][4];
arr[0][0][0] = 1;
arr[0][0][1] = 2;
// Continue assigning values for other elements
```

Thankuuu for Viewing…
I hope it will help you a lot.

I.P. College Campus-2, Bulandshahr
Department of Computer Science
Programming in C Internal Question Bank Solved

BY Bhavy Sharma

B.C.A – 202
Objective types questions:

Objective types questions:

1. The array name itself is a –
   (a) Last element (c) Address
   (b) First element (d) None of these
   Ans. Address

2. The value within[ ] in an array declaration specifies-
   (a) Size of the array (c) Largest permitted subscript value (b) Type of the array (d) (a) or (b)
   Ans. Size of the array

3. The 2-d array is the array of:
   a) 1-d array b)2-d array c) matrix d) none of these.
   Ans. Matrix

4. The following is a valid C string
   a) 'a' b)'C" c)"programming' d)"language"
   Ans. "language"

5. In char C[]="Hello", if 'H' is stored at 65470, then 'o' will be stored at-
(a) 65474 b) 65475 c) 65480 d) 65476
Ans. (a) 65474

6. The string is said to be collection of:
   a) Characters b)numbers c) words d) all
   Ans. All

7. stdin is a predefined --
   a)array b)pointer c)macro d)function
   Ans. macro

8. A structure is a collection of_data elements and an array is a collection of_data elements.
(a) Homogeneous / Heterogeneous (b) Consecutive / Homogeneous
(c) Similar / Frequent (d) Heterogeneous / Homogeneous
Ans. (d) Heterogeneous / Homogeneous

9. In Dynamic memory location, malloc() takes arguments and calloc() takes arguments.
   (a) 2,3 (b) 1,2 (c) 2,1 (d) 0,1
   Ans. (b) 1,2

10. Which operator is used by structure variables to access structure members :

a)& operator b) %d operator c)arrow operator d) dot operator

Ans. d) dot operator

11. perform comparison of two string up to a certain specified length.

(a) strcmp (b) strucmp() (c) strncmp() (d) stricmp()

Ans. (c) strncmp()

12. Which of the following is a bitwise operator?

(a) && (b) >>> (c) @ (d) ~

Ans. (d) ~

13. Connection of a file to a program is obtained by using a function-

(a) Fopen() (b) FOPEN() (c) fopen() (d) open()

Ans. (c) fopen()

14. Which of the following is not a preprocessor directive?

(a) #ifdef (b) #elseif (c) #undef (d) #define

Ans, (b) #elseif

15. Which modes occupies less space while working over files

a) Binary modes. b) Text modes. C) Sys modes. d) Both b & c.

Ans. a) Binary modes.

State true and false

1. The array elements did not share the same name and data type.

False

2. The #define directives are used to define a Macros**.**

True

3. NULL is a macro defined in header file conio.h.

False

4. In r+ mode we can do both read & write in a file.

True

5. A pointer can contain the address of another pointer

True

6. The default value of array element is 0.

True

7. If we place the value in array beyond its size the compiler shows error.

True

8. There is no difference in input a string either through gets() or scanf().

False

9. In C programming the char array and string both are different.

False

10. We can pass argument in main().

True

11. atoi()function is used to convert the string into int.

True

12. Define directives allow compiling a selected portion of the code.
True
13. Bitfield provides exact amount of bits required for storage of values.
False
14. Fopen() write an integer in the file.
False
15. To reposition the file pointer we use ftell() in file handling.
False

Short question answer

1. **Write a C program to read 10 integer numbers and print their average, minimum & maximum number.**

Ans.

```
#include <stdio.h>
void main() {
   int numbers[10];
   int i, sum = 0;
   int min, max;

   printf("Enter 10 integer numbers:\n");
   for (i = 0; i < 10; i++) {
      scanf("%d", &numbers[i]);
      sum += numbers[i];

   if (i == 0) {
         min = max = numbers[i];
      } else {
         if (numbers[i] < min) {
            min = numbers[i];
         }
         if (numbers[i] > max) {
            max = numbers[i];
         }
      }
   }

   // Calculate average
   float average = (float)sum / 10;

   // Print results
   printf("Average: %.2f\n", average);
   printf("Minimum number: %d\n", min);
   printf("Maximum number: %d\n", max);
}
```

2. **Distinguish between structure and union. Explain with suitable example and program.**

Ans.

Structure:

- **Memory Allocation:**
  - Each member of a structure is allocated its own memory space.
- **Data Storage:**
  - All members of a structure are stored in memory sequentially.
- **Size:**
  - The size of a structure is the sum of the sizes of all its members.
- **Usage:**
  - Structures are used when you want to group related data of different types together.

Union:

- **Memory Allocation:**
  - All members of a union share the same memory space.
- **Data Storage:**
  - Only one member of a union can be stored in memory at a time.
- **Size:**
  - The size of a union is equal to the size of its largest member.
- **Usage:**
  - Unions are used when you want to store different types of data in the same memory location.

```
#include <stdio.h>

struct Rectangle {
   int width;
   int height;
};

struct Circle {
   float radius;
};

union Shape {
   struct Rectangle rect;
   struct Circle cir;
};

void main() {
   union Shape shape;

   shape.rect.width = 5;
```

```
    shape.rect.height = 3;

    printf("Rectangle Width: %d\n", shape.rect.width);
    printf("Rectangle Height: %d\n", shape.rect.height);

    shape.cir.radius = 2.5;

    printf("Circle Radius: %.2f\n", shape.cir.radius);

}
```

3. Explain the importance of the #define preprocessor directive.

Ans. The `#define` preprocessor directive in C is used to define symbolic constants and macros.

1. **Symbolic Constants:** **`#define`** allows you to define symbolic constants, which are meaningful names given to constants in your code. For example:

```
#define PI 3.14159
```

2. **Code Maintainability:** By using `#define`, you can define important values or parameters in one place and easily change them later if needed. This improves code maintainability and reduces the risk of errors when making changes.

3. **Macro Functions:** You can define macros using `#define` to create code snippets that behave like functions. These macros can take arguments and perform certain operations. For example:

4. **Enhancing Readability:** `#define` can also be used to enhance code readability by giving descriptive names to complex or frequently used expressions. This makes the code easier to understand for other developers.

4. Write a program to illustrate the use of structure pointers.

Ans.

```
#include <stdio.h>

struct Student {
    char name[50];
    int age;
    float marks;
};
```

```
void main() {
    struct Student student1;
    struct Student *ptr;
    ptr = &student1;

    printf("Enter student name: ");
    scanf("%s", ptr->name);
    printf("Enter student age: ");
    scanf("%d", &ptr->age);
    printf("Enter student marks: ");
    scanf("%f", &ptr->marks);

    printf("\nStudent details:\n");
    printf("Name: %s\n", ptr->name);
    printf("Age: %d\n", ptr->age);
    printf("Marks: %.2f\n", ptr->marks);

}
```

5. What are merits & demerits of array in C language.
Ans.

Merits of Arrays in C:

1. **Efficient Storage:** Arrays provide a contiguous block of memory to store elements of the same data type, which allows for efficient memory usage.
2. **Random Access:** Elements of an array can be accessed directly using their indices, enabling fast and efficient random access to elements.
3. **Ease of Iteration:** Arrays facilitate easy iteration through elements using loops, making it convenient to process large sets of data.
4. **Simple Syntax:** Arrays in C have simple and straightforward syntax for declaration and initialization, making them easy to use.
5. **Fixed Size:** Once defined, the size of an array is fixed and cannot be changed during runtime, ensuring predictable memory usage and behavior.

Demerits of Arrays in C:

1. **Fixed Size Limitation:** The size of an array must be specified at the time of declaration, which limits flexibility and may lead to wasted memory if the actual size needed is unknown or variable.
2. **No Built-in Bounds Checking:** C arrays do not perform bounds checking, so accessing elements outside the bounds of an array can lead to undefined behavior, including

segmentation faults or memory corruption.

3. **Inflexible Memory Allocation:** Arrays in C are allocated statically or dynamically, but both approaches have limitations. Static allocation requires knowing the size at compile time, while dynamic allocation using `malloc()` may lead to memory leaks or fragmentation if not managed properly.
4. **Passing to Functions:** When passing an array to a function in C, only a pointer to the array's first element is passed, losing information about the array's size. Developers must often pass the array size as a separate parameter, which can be error-prone.
5. **Homogeneous Data Types:** Arrays can only store elements of the same data type, which may not be suitable for certain data structures or scenarios requiring heterogeneous data storage.
6. **No Built-in Functions:** C arrays do not have built-in functions for common operations like searching, sorting, or resizing. Developers must implement these functions manually or use libraries.

6. Explain the meaning and purpose of the following.
a) Typedef keyword b) Size of operator
Ans.

a) Typedef Keyword:

The typedef keyword in C is used to create an alias or a new name for existing data types. It allows programmers to define custom data types with meaningful names, which can improve code readability and maintainability. The syntax for typedef is as follows:

```
typedef existing_data_type new_data_type;
```

b) Sizeof Operator:

The sizeof operator in C is used to determine the size in bytes of a variable or a data type. It returns the size of its operand, which can be a variable, an expression, or a data type. The syntax for sizeof is as follows:

```
sizeof(expression);
```

Long question answers

1. Explain the use of bitwise operators in programming with suitable example.

Ans.

Bitwise Operators in C:

1. AND (`&`): Performs a bitwise AND operation.
2. OR (`|`): Performs a bitwise OR operation.
3. XOR (`^`): Performs a bitwise XOR operation.
4. NOT (`~`): Performs a bitwise NOT (inversion) operation.
5. Left Shift (`<<`): Shifts bits to the left.
6. Right Shift (`>>`): Shifts bits to the right.

1. Bitwise AND (`&`):

Performs an AND operation on each pair of corresponding bits of the operands. The result bit is `1` if both corresponding bits are `1`, otherwise it is `0`.

Example:-

```
#include <stdio.h>
void main() {
    unsigned char a = 12;  // 00001100 in binary
    unsigned char b = 10;  // 00001010 in binary
    unsigned char result = a & b; // 00001000 in binary (8 in decimal)
   printf("Result of %d & %d is %d\n", a, b, result);
   }
```

2. Bitwise OR (`|`):

Performs an OR operation on each pair of corresponding bits of the operands. The result bit is `1` if at least one of the corresponding bits is `1`, otherwise it is `0`.

Example:-

```
#include <stdio.h>
void main() {
    unsigned char a = 12;  // 00001100 in binary
    unsigned char b = 10;  // 00001010 in binary
    unsigned char result = a | b; // 00001110 in binary (14 in decimal)
    printf("Result of %d | %d is %d\n", a, b, result);
}
```

3. Bitwise XOR (^):

Performs an XOR operation on each pair of corresponding bits of the operands. The result bit is `1` if the corresponding bits are different, otherwise it is `0`.

Example:-

```
#include <stdio.h>
void main() {
   unsigned char a = 12;  // 00001100 in binary
   unsigned char b = 10;  // 00001010 in binary
   unsigned char result = a ^ b; // 00000110 in binary (6 in decimal)
   printf("Result of %d ^ %d is %d\n", a, b, result);
}
```

4. Bitwise NOT (~):

Inverts all the bits of the operand. Each `1` becomes `0` and each `0` becomes `1`.

Example:-

```
#include <stdio.h>
void main() {
   unsigned char a = 12;  // 00001100 in binary
   unsigned char result = ~a; // 11110011 in binary (243 in decimal, due to two's
complement representation in unsigned char)
   printf("Result of ~%d is %d\n", a, result);
}
```

5. Left Shift (<<):

Shifts the bits of the first operand to the left by the number of positions specified by the second operand. Zero bits are shifted in from the right.

Example:-

```
#include <stdio.h>
void main() {
   unsigned char a = 12;  // 00001100 in binary
   unsigned char result = a << 2; // 00110000 in binary (48 in decimal)
   printf("Result of %d << 2 is %d\n", a, result);
}
```

6. Right Shift (>>):

Shifts the bits of the first operand to the right by the number of positions specified by the second operand. Zero bits are shifted in from the left (for unsigned types).

Example:-

```
#include <stdio.h>
void main() {
   unsigned char a = 12;  // 00001100 in binary
   unsigned char result = a >> 2; // 00000011 in binary (3 in decimal)
   printf("Result of %d >> 2 is %d\n", a, result);

}
```

2. Write a short note on the following
a) strlen() b)strcpy() c)strcat() d)strcmp()
Ans.

a) `strlen()`

The `strlen()` function calculates the length of a null-terminated string, excluding the null character (`\0`) at the end.

Syntax :-
size_t strlen(const char *str);

Program:-

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

int main() {
   char myString[] = "Hello, World!";
   size_t length = strlen(myString);
   printf("Length of the string is: %zu\n", length);
   return 0;
}
```

b) `strcpy()`

The `strcpy()` function copies the null-terminated string pointed to by the source (`src`) to the destination (`dest`), including the null character.

Syntax:- char *strcpy(char *dest, const char *src);

Program :-

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

int main() {
   char src[] = "Hello, World!";
   char dest[50];
   strcpy(dest, src);
   printf("Copied string is: %s\n", dest);
   return 0;
}
```

c) `strcat()`

The `strcat()` function appends the null-terminated string pointed to by the source (`src`) to the end of the destination (`dest`). The initial null character of `src` overwrites the null character at the end of `dest`.

Syntax:- char *strcat(char *dest, const char *src);

Program :-

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

int main() {
   char dest[50] = "Hello, ";
   char src[] = "World!";
   strcat(dest, src);
   printf("Concatenated string is: %s\n", dest);
   return 0;
}
```

d) `strcmp()`

The `strcmp()` function compares two null-terminated strings lexicographically. It returns an integer indicating the relationship between the strings.

Syntax:- int strcmp(const char *str1, const char *str2);

Program:-

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

int main() {
   char str1[] = "Hello";
   char str2[] = "World";

   int result = strcmp(str1, str2);

   if (result < 0) {
      printf("%s is less than %s\n", str1, str2);
   } else if (result == 0) {
      printf("%s is equal to %s\n", str1, str2);
   } else {
      printf("%s is greater than %s\n", str1, str2);
   }

   return 0;
}
```

3. Discuss the purpose, syntax, example of different dynamic memory management functions.

Ans. Dynamic memory management in C allows programs to request and release memory at runtime, providing flexibility and efficient memory usage. The standard library provides several functions for dynamic memory management, including `malloc()`, `calloc()`, `realloc()`, and `free()`. Each serves a specific purpose and has its own syntax and use cases.

1. `malloc()`

Allocates a specified number of bytes of memory and returns a pointer to the first byte of the allocated space. The allocated memory is uninitialized.

Syntax:- void *malloc(size_t size);

Program:-

```
#include <stdio.h>

#include <stdlib.h>
int main() {
   int *ptr;
   int n = 5;
   ptr = (int *)malloc(n * sizeof(int));
   if (ptr == NULL) {
       printf("Memory allocation failed\n");
       return 1;
   }
   for (int i = 0; i < n; i++) {
       ptr[i] = i + 1;
       printf("%d ", ptr[i]);
   }
   free(ptr);
   return 0;
}
```

`calloc()`

Allocates memory for an array of elements, initializes all bytes to zero, and returns a pointer to the allocated memory.

Syntax:- void *calloc(size_t num, size_t size);

`realloc()`

Changes the size of the previously allocated memory block, possibly moving it to a new location.

Syntax:- void *realloc(void *ptr, size_t size);

`free()`

Deallocates the memory previously allocated by `malloc()`, `calloc()`, **or** `realloc()`, freeing it for future use.

Syntax:- void free(void *ptr);

4. Write a program to sort the elements of given array in descending order. 5. How can you read and write text files? Explain with an example.

Ans. Program to Sort Elements of an Array in Descending Order

```
#include <stdio.h>

void sortDescending(int arr[], int n) {
    int temp;
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] < arr[j+1]) {
                // Swap elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {5, 3, 8, 6, 2, 7, 4, 1, 9};
    int n = sizeof(arr) / sizeof(arr[0]);

    sortDescending(arr, n);

    printf("Sorted array in descending order:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Reading and Writing Text Files in C

Reading from and writing to text files in C involves using the standard input/output library (`stdio.h`) and its file handling functions.

Reading from a Text File

To read from a text file, you typically use `fopen()`, `fgets()`, `fscanf()`, or `fread()`, and `fclose()` functions.

Program :-

```
#include <stdio.h>

int main() {
   FILE *file;
   char line[100];

   // Open file for reading
   file = fopen("example.txt", "r");
   if (file == NULL) {
      printf("Error opening file.\n");
      return 1;
   }

   // Read each line from the file and print it
   while (fgets(line, sizeof(line), file) != NULL) {
      printf("%s", line);
   }

   // Close the file
   fclose(file);

   return 0;
}
```

Writing to a Text File

To write to a text file, you typically use `fopen()`, `fprintf()`, `fputs()`, or `fwrite()`, and `fclose()` functions.

Program:-

```
#include <stdio.h>

int main() {
    FILE *file;

    // Open file for writing
    file = fopen("output.txt", "w");
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    // Write to the file
    fprintf(file, "Hello, World!\n");
    fputs("This is a text file example.\n", file);

    // Close the file
    fclose(file);

    return 0;
}
```

6. Write a program to multiply two matrices of dimensions m*n.
Ans.

```
#include <stdio.h>

int main() {
    int mat1[2][2], mat2[2][2], result[2][2];

    // Input elements of the first matrix
    printf("Enter elements of the first matrix:\n");
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            scanf("%d", &mat1[i][j]);

    // Input elements of the second matrix
    printf("Enter elements of the second matrix:\n");
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            scanf("%d", &mat2[i][j]);

    // Multiply the matrices
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            result[i][j] = mat1[i][0] * mat2[0][j] + mat1[i][1] * mat2[1][j];
    // Print the resultant matrix
    printf("Resultant matrix:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++)
            printf("%d ", result[i][j]);
        printf("\n");
    }

    return 0;
}
```

C Language Notes by Bhavy Sharma

Que1. What is Variable?
Ans. A variable is like a storage box in programming. It's a way to give a name to a piece of information that you want to remember and use later in your code.

Example :- Imagine you have a box (variable) with a label (name), and you can put different things (values) inside it. For example, you might have a variable called "age" that stores the number 25. Later in your program, you can use the "age" variable to refer to the number 25 without having to remember the actual value.
How to Declare a Variable :-
int age = 25;

Que2. What is an interpreter?
Ans. An interpreter is a type of computer program that directly executes instructions written in a programming language without the need for a separate compilation step. It translates and executes the source code line by line in real-time.
An interpreter is like a language translator who listens to you speak one sentence at a time and immediately translates it, rather than waiting to hear the entire conversation before providing a translation.

Que3. Define Recursion?
Ans. Recursion is a concept where a function calls itself during its execution.
A recursive function is a function that solves a problem by calling itself with a smaller instance of the same problem. It keeps breaking the problem down into smaller and smaller parts until it reaches a point where the problem is so simple that it can be solved directly. It's a way of solving complex problems by tackling simpler versions of the same problem.

Que4. Write Four Preprocessor Directive?
Ans. Here are four preprocessor directive:

1. **#include:**
   Think of it like copying and pasting. When you write `#include <stdio.h>`**, this means you want to include the <stdio.h> file in your program This is often used to include libraries or header files.**

2. **#define:**

This is like giving a nickname to a piece of code. When you write `#define PI 3.14`**, it means whenever the compiler sees** `PI` **in your code, it replaces it with** `3.14`**. It's a way to make your code more readable.**

3. **#undef:**
   This is the opposite of `#define`**. It's used to undefine a previously defined macro. For example,** `#undef PI` **would undefine the previously defined** `PI`**.**

4. **#line:**
   It allows you to set the line number and filename for error messages and line directives. It's often used for debugging and code generation.

Que5. Write syntax and usage of Ternary Operator?
Ans. The ternary operator is a concise way to write an `if-else` **statement in a single line. It has the following syntax:**
Syntax : condition ? expression_if_true : expression_if_false;

Usage :-

```
#include <stdio.h>

Void main() {
   int number = 10;
   // Ternary operator to determine if the number is even or odd
   const char* result = (number % 2 == 0) ? "even" : "odd";

   printf("The number %d is %s.\n", number, result);
   }
```

Que6. What is the structure of the C Program? Define it.
Ans. The Structure of C Program are Following:-

1. **Documentation Section (Optional):**
   This Section contain comments given by Programmer. These comments can be single line or multiline comments. For single line comments we use two forward slash (//). To write Multiline comments we use /*(astrik)*/
   For Example :-

```
/* This is a simple C program
   Author: Bhavy Sharma
   Date: January 1, 2024
```

*/
2. Link Section:
   There are so many functions which are pre defined in C language. Such Functions are stored in the header file. Now if we want to use those predefined functions, we need to include their corresponding header file in our C programme. Now if we want to attach these header file in our C program we can Simply write this
   For Example :-
   #include <stdio.h>

3. Global Variable:
   If we have some communication variable which may be accessed by any function or global variable. We always keep such variables before the main function.
   For Example:
   ```
   #include <stdio.h>
   #include <conio.h>
   Int age = 26;   // Global Variable
   Void main{
           // Your Code
           }
   ```

4. Void main() :
   1. Every C program must have a main() Function.
   2. Every Function name always has parentheses in the end.
   3. Every C program always start execution from main
   4. We may have a void keyword before the main function which represents no return whenever that particular function is executed.
   5. Inside the main function we declare local variables and functions.

   The actual logic of the programme we write in the main function
   . A programmer may have n number of functions, Generally we write the function definition after main().

Que7. Why is time complexity an important issue?Explain.
Ans. Time complexity is an important issue in programming because it helps us understand and measure how fast an algorithm runs as the input size increases. In easy terms, it tells us how much time our program will take to complete its task.

Time complexity is important because :-
1. Efficiency:

Time complexity helps us identify the most efficient way to solve a problem. It allows programmers to choose algorithms that perform well, especially when dealing with large amounts of data.

2. Scalability:
   As you work with larger sets of data, some algorithms might become significantly slower. Time complexity helps us predict how an algorithm's performance will scale with increased input size. This is crucial when developing applications that need to handle diverse and growing datasets.

3. Resource Management:
   Different algorithms use different amounts of computer resources like memory and processing power. Time complexity analysis allows us to choose algorithms that are suitable for the available resources, preventing performance issues and ensuring the program runs smoothly.

4. Problem-solving Efficiency:
   For programmers, it's essential to solve problems in a reasonable amount of time. Time complexity guides the choice of algorithms, enabling developers to provide solutions that meet time constraints and user expectations.

Que8. What do you mean by call-by-value and call-by-reference?Explain in Brief.
Ans. Call-by-value and call-by-reference are two ways that programming languages pass arguments to functions.

1. Call-by-Value:
   In call-by-value, when you pass a variable to a function, you're essentially passing a copy of the variable's value. The function receives this copy and works with it. However, any changes made to the parameter inside the function do not affect the original variable outside the function.
   For Example :

```
void square(int num) {
 num = num * num;
}

void main() {
   int x = 5;
   square(x); // x is still 5 after this call
}
```

2. Call-by-Reference:
   In call-by-reference, when you pass a variable to a function, you are passing a reference or address of the variable, not its value. This means any changes made to the parameter inside the function directly affect the original variable outside the function.
   For Example :
```
void square(int *num) {
   *num = (*num) * (*num);
}

void main() {
   int x = 5;
   square(&x); // x is now 25 after this call
}
```

Que9. What is Array? Explain in Brief. How array is declared in C? Also Write limitation of array?
Ans. Array :
An array is like a set of pigeonholes where you can keep multiple values of the same type under one name. It's a way to organize data so that you can access it more conveniently.

Declaration of Array in C:
In C, you declare an array by specifying its type and size. Here's a simple example of an array of integers: int numbers[5]; // This declares an array named 'numbers' that can hold 5 integers.
You can also initialize the array with values:
int numbers[5] = {1, 2, 3, 4, 5}; // This creates an array with values 1, 2, 3, 4, 5.
To access elements in the array, you use an index:
int thirdNumber = numbers[2]; // This gets the value at index 2 (arrays start at 0), which is 3.

Limitations of Arrays:
1. Fixed Size:
   Once you declare an array, its size is fixed. You can't easily change it. If you initially say an array can hold 5 values, it will always hold 5 values.
2. Single Data Type:
   Arrays store elements of the same data type. If you declare an array of integers, you can only store integers.
3. Memory Usage:

Arrays allocate a fixed amount of memory even if you don't use all of it. This can be inefficient in terms of memory usage, especially if your program needs to handle varying amounts of data.

4. No Built-in Bounds Checking:
   C arrays don't automatically check if you're accessing elements within their bounds. If you go beyond the size of the array, you might access unintended memory locations, leading to unexpected behavior or errors.

Que10. Write note on :

1. Function
2. Header File
3. Flow Chart
4. Extern Storage Class

Ans. Here are Short Notes of :

1. Function:
   A function is like a mini-program inside your main program. It's a set of instructions that you can reuse whenever you need.
   For Example :
   ```
   // This is a function
   int addNumbers(int a, int b) {
      return a + b;
   }
   void main(){
   // Using the function
   int result = addNumbers(5, 3); // result is now 8
   printf(“The sum of Numbers is : %d”,result);
   }
   ```

2. Header File :
   In C programming all the pre-defined Function are specified in some files, called Header Files. All Header files have .h extension. We use include Directive to attach any header file with our C program.

3. Flow Chart:
   A flowchart is like a visual roadmap for your program. It uses shapes and arrows to show the steps and decision points in your code. It's helpful to plan out how your program should work before you start coding.

4. Extern Storage Class:

The `extern` storage class is used when a variable or function is declared in one file but defined in another.
For Example :

```
// File1.c
int globalVariable = 10;

// File2.c
extern int globalVariable; // This tells the compiler that globalVariable is defined somewhere else

int main() {
   // Now you can use globalVariable here
   return 0;
}
```

Programs

Que1. Write a program to find maximum of three number?
Ans.

```
#include <stdio.h>
void main() {
   int num1, num2, num3;
   printf("Enter three numbers: ");
   scanf("%d %d %d", &num1, &num2, &num3);

   int max = num1; // Assume num1 is the maximum
   if (num2 > max) {
      max = num2; // Update max if num2 is greater
   }
   if (num3 > max) {
      max = num3; // Update max if num3 is greater
   }
  printf("The maximum of %d, %d, and %d is: %d\n", num1, num2, num3, max);
}
```

Que2. Write a Program in C to calculate area of circle?
Ans.

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

```
void main() {
     float radius, area;
    printf("Enter the radius of the circle: ");
    scanf("%f", &radius);
    area = 3.14 * radius * radius;
    printf("The area of the circle with radius %.2f is: %.2f\n", radius, area);
}
```

Que3. Write a C program using while loop to calculate the factorial of a given Number?
Ans.

```
#include <stdio.h>

void main() {
    int number, factorial = 1;
    printf("Enter a number: ");
    scanf("%d", &number);
    while (number > 0) {
        factorial *= number;
        number--;
    }
    printf("The factorial is: %d\n", factorial);
}
```

This is Part 1
Wait for part 2 Coming soon...

String By Bhavy Sharma

In C, a string is a sequence of characters stored in an array. Each character in the array corresponds to one element in the string. Strings in C are terminated by a special character called the null terminator, represented by '\0'. This null terminator indicates the end of the string.

1. **<u>String Declaration</u>: Strings in C can be declared as character arrays. For example:- char str[10]; // Declares a string of size 10**

2. **<u>Initialization</u>: Strings can be initialized using double quotes. For example:- char str[] = "Hello";**

3. **<u>String Input</u>: You can use functions like `scanf()` or `gets()` to input strings from the user. For example:- scanf("%s", str); // Reads a string from the user and the second way to take input form the user is :-** `gets(str);`

4. **<u>String Output</u>: You can use `printf()` to output strings. For example:- printf("%s", str); // Prints the string.**

5. **String Functions: C provides several built-in functions to perform operations on strings, such as:**

- `strlen()`**: Returns the length of a string.**
- `strcpy()`**: Copies one string to another.**
- `strcat()`**: Concatenates two strings.**
- `strcmp()`**: Compares two strings.**

Now, we learn all the function perform with string with example and easy explanations.

1. **strlen() - String Length:**

- `strlen()` **returns the length of a string, i.e., the number of characters in the string excluding the null terminator ('\0').**

  For Example :-

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

  void main() {
  char str[] = "Hello";
  int length = strlen(str);
  printf("Length of the string: %d\n", length);
  }
  ```

  Output: `Length of the string: 5`

2. **strcpy() - String Copy:**

- `strcpy()` **copies the contents of one string to another.**

  For Example :-

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

  void main() {
  char source[] = "Hello";
  char destination[10]; // Make sure the destination array is large enough
  strcpy(destination, source);
  printf("Copied string: %s\n", destination);
  }
  ```

  Output: `Copied string: Hello`

3. **strcat() - String Concatenation:**

- `strcat()` appends (adds) one string to the end of another.

  For Example :-

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

```
void main() {
char str1[20] = "Hello";
char str2[] = " World";
strcat(str1, str2);
printf("Concatenated string: %s\n", str1);
}
```
Output: `Concatenated string: Hello World`

4. **strcmp() - String Comparison:**

- `strcmp()` **compares two strings lexicographically (dictionary order).**
- **It returns 0 if the strings are equal, a negative value if the first string is lexicographically less than the second, and a positive value if the first string is lexicographically greater than the second.**

For Example :-
```
#include <stdio.h>
#include <string.h>

void main() {
char str1[] = "apple";
char str2[] = "banana";
int result = strcmp(str1, str2);
if (result == 0)
        printf("The strings are equal.\n");
else if (result < 0)
        printf("'%s' comes before '%s'.\n", str1, str2);
else
        printf("'%s' comes after '%s'.\n", str1, str2);
}
```
Output: `apple comes before banana.`

C languages Notes By Bhavy Sharma

Dec – 2021 Paper Solved

Very Short Questions in 75 Words

Que1. Write the use of comments in C programming?
Ans. In C programming, comments are essential for enhancing code readability and explaining its logic. They serve as non-executable text that provides information about the code. Comments help programmers understand the purpose of specific lines or sections, making collaboration more efficient. Additionally, comments aid in debugging by allowing developers to temporarily disable code segments. Effective use of comments in C fosters clear communication and documentation, facilitating both individual and collaborative coding efforts.

Que2. What do you understand by associativity and precedence of operators?
Ans. In programming, associativity and precedence of operators dictate the order in which operations are performed. Precedence determines which operator takes precedence over others, while associativity defines the grouping of operators with the same precedence. For example, in an expression with multiple operators, precedence ensures that higher-precedence operators are evaluated first. Associativity comes into play when operators with the same precedence appear, determining the order of evaluation. Understanding both concepts is crucial for accurately predicting the outcome of complex expressions in programming languages.

Que3. What do you mean by Complexity of an algorithm?
Ans. The complexity of an algorithm measures its efficiency and performance. It is classified as time complexity and space complexity. Time complexity quantifies the amount of time an algorithm takes to complete based on the input size. Space complexity assesses the memory consumption. Big O notation is commonly used to express complexity. Lower complexity indicates better efficiency. Analyzing algorithmic complexity is crucial for choosing optimal solutions, particularly in large-scale computations or resource-constrained environments.

Que4. What is Scope of Variable?
Ans. The scope of a variable in programming defines the region of code where the variable can be accessed and modified. Variables can have local or global scope. Local scope confines a variable to a specific block or function, limiting its visibility outside. Global scope allows access throughout the entire program. Understanding variable scope is crucial to prevent unintended side effects, optimize memory usage, and maintain code modularity, enhancing the overall structure and efficiency of a program.

Que5. What do you mean by Storage classes in C?
Ans. In C programming, storage classes determine the scope, visibility, and lifetime of variables. The primary storage classes include auto, register, static, and extern. "Auto" variables have local scope and are automatically initialized. "Register" variables are stored in CPU registers for faster access. "Static" variables retain their values between function calls. "Extern" variables are declared outside functions, allowing multiple files to access them. Understanding storage classes is crucial for managing variable behavior and memory allocation in C programs.

Short Questions in 200 Words

Que1. Explain printf() and scanf() Function used in C Programming with the help of an Example?

Ans. The `printf()` and `scanf()` functions are fundamental in C programming for input and output operations. `printf()` is used to display output on the console, while `scanf()` is used to take input from the user.

Example :-

```
#include <stdio.h>

int main() {
   // Example of printf() for output
   printf("Hello, World!\n");

   // Example of scanf() for input
   int number;
```

```
    printf("Enter an integer: ");
    scanf("%d", &number);
    printf("You entered: %d\n", number);

    return 0;
}
```

In this example, `printf()` is employed to print the "Hello, World!" message to the console. The `\n` is a newline character for formatting.

The `scanf()` function is used to take an integer input from the user. The format specifier `%d` is used to specify the type of input expected (integer in this case). The `&` before the variable `number` is the address-of operator, indicating the location in memory where the input should be stored. Finally, the entered value is printed using `printf()`.

Que2. Explain the different steps to solve any problem?
Ans. <u>Problem Solving</u> :- Problem-solving is the process of finding solutions to difficult or complex issues. It involves analyzing a situation, identifying the problem, and devising a strategy to address and resolve it. Effective problem-solving requires critical thinking, creativity, and a systematic approach.

<u>Steps to solve any problem</u> :-

1. Define Problem:
   Defining the problem is the initial step in problem-solving. It involves clearly understanding and articulating the nature of the challenge or obstacle. This includes identifying the specific issues, determining the desired outcome, and establishing the criteria for a successful solution. Clarity in defining the problem sets the foundation for the subsequent steps in the problem-solving process.

2. Analyse Problem:
   After defining the problem, the next step is to analyze it in-depth. This involves breaking down the problem into its components, understanding the relationships between these components, and identifying any underlying causes or contributing factors. The goal is to gain a comprehensive understanding of the problem's complexity. Analyzing the problem helps in formulating a strategy to address its root causes and develop effective solutions.

3. Explore Solution:

   Once the problem is defined and analyzed, the exploration of potential solutions begins. This phase involves brainstorming, evaluating various options, and considering different approaches to solving the problem. It's essential to be creative during this step, exploring both conventional and innovative solutions. The aim is to generate a range of possible strategies that could effectively address the identified issues. Evaluation criteria, such as feasibility, cost-effectiveness, and impact, are often considered when exploring potential solutions.

Que3. Write a program in C to check whether the given number is prime or not?

Ans. Code is here :-

```
#include <stdio.h>
int main() {
   int num, i, flag = 0;
// Input from user
   printf("Enter a number: ");
   scanf("%d", &num);

   // Check for prime
   for (i = 2; i <= num / 2; ++i) {
      if (num % i == 0) {
         flag = 1;
         break;
      }
   }
   // Display the result
   if (num == 1) {
      printf("1 is not a prime number.\n");
   } else {
      if (flag == 0)
         printf("%d is a prime number.\n", num);
      else
         printf("%d is not a prime number.\n", num);
   }
   return 0;
}
```

Long Questions

Que1. Explain the logical and bitwise operator in C with the help of suitable example?

Ans.

Logical Operators:

Logical operators in C perform logical operations (AND, OR, NOT) on boolean values (0 for false, non-zero for true).

1. Logical AND (`&&`)
- Returns true if both operands are true.
- Example:

```
#include <stdio.h>
int main() {
   int a = 1, b = 0;
   if (a && b) {
      printf("Both conditions are true.\n");
   } else {
      printf("At least one condition is false.\n");
   }
   return 0;
}
```

Output: `At least one condition is false.`

2. Logical OR (`||`):
- Returns true if at least one operand is true.
- Example:

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

```
int main() {
    int a = 1, b = 0;

    if (a || b) {
        printf("At least one condition is true.\n");
    } else {
        printf("Both conditions are false.\n");
    }

    return 0;
}
```

Output: `At least one condition is true.`

3. Logical NOT (`!`):
- Returns true if the operand is false and vice versa.
- Example:

```
#include <stdio.h>

int main() {
    int a = 1;

    if (!a) {
        printf("The condition is false.\n");
    } else {
        printf("The condition is true.\n");
    }

    return 0;
}
```

Output: `The condition is false.`

Bitwise Operators:

Bitwise operators in C perform operations at the bit level.

1. Bitwise AND (`&`):
- Performs bitwise AND operation.
- Example:

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

```
int main() {
    int a = 5, b = 3;

    int result = a & b;

    printf("Result of bitwise AND: %d\n", result);

    return 0;
}
```

Output: `Result of bitwise AND: 1` (Binary: 0101 & 0011 = 0001)

2. Bitwise OR (`|`):
- Performs bitwise OR operation.
- Example:

```
#include <stdio.h>

int main() {
    int a = 5, b = 3;

    int result = a | b;

    printf("Result of bitwise OR: %d\n", result);

    return 0;
}
```

Output: `Result of bitwise OR: 7` (Binary: 0101 | 0011 = 0111)

3. Bitwise XOR (`^`):
- Performs bitwise XOR (exclusive OR) operation.
- Example:

```
#include <stdio.h>

int main() {
    int a = 5, b = 3;

    int result = a ^ b;

    printf("Result of bitwise XOR: %d\n", result);

    return 0;
}
```

Output: `Result of bitwise XOR: 6` **(Binary: 0101 ^ 0011 = 0110)**

4. Bitwise NOT (`~`):

- Performs bitwise NOT operation (complements the bits).
- **Example:**

```
#include <stdio.h>
int main() {
   int a = 5;
   int result = ~a;
   printf("Result of bitwise NOT: %d\n", result);
   return 0;
}
```

Output: `Result of bitwise NOT: -6` (Binary: ~0101 = 1010,

Que2. Discuss the following syntax and example :-

1.getch(), getchar(), putchar()

2. #include, #define

Ans. 1. getch():

- Syntax: `int getch(void);`
- Example:

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

int main() {
   char ch;
   printf("Press any key: ");
   ch = getch();
   printf("\nYou pressed: %c\n", ch);
   return 0;
}
```

2. getchar():

- Syntax: `int getchar(void);`
- Example:

```
#include <stdio.h>

int main() {
   char ch;
   printf("Enter a character: ");
   ch = getchar();
   printf("You entered: %c\n", ch);
   return 0;
}
```

3. putchar():

- Syntax: `int putchar(int char);`
- Example:

```
#include <stdio.h>
int main() {
   char ch = 'A';
   putchar(ch);
   return 0;
}
```

2.1. #include:

- The `#include` **directive is used to include the content of a file in a C program. It is typically used to include header files that contain declarations for functions or variables used in the program.**
- **Example:**

```
#include <stdio.h>
int main() {
   printf("Hello, World!\n");
   return 0;
}
```

2.2. #define:

- The `#define` **directive is used to create macros or symbolic constants in C.**
- **Example:**

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

```
#define PI 3.14159

int main() {
   double radius = 5.0;
   double area = PI * radius * radius;
   printf("Area of the circle: %lf\n", area);
   return 0;
}
```

Que3. Explain any three control structures in C with proper syntax and Example?

Ans. 1. `if` Statement:

The `if` statement allows you to conditionally execute a block of code.

Syntax:

```
if (condition) {
   // code to be executed if the condition is true
} else {
   // code to be executed if the condition is false
}
```

Example:

```
#include <stdio.h>

int main() {
   int num = 10;

   if (num > 0) {
      printf("The number is positive.\n");
   } else {
      printf("The number is non-positive.\n");
   }

   return 0;
}
```

2. `for` Loop:

The `for` loop is used for iterative execution of a block of code.

Syntax:

```
for (initialization; condition; update) {
    // code to be executed in each iteration
}
```

Example:

```
#include <stdio.h>

int main() {
    int i;

    for (i = 1; i <= 5; i++) {
        printf("%d ", i);
    }

    printf("\n");

    return 0;
}
```

3. `while` Loop:

The `while` loop is another looping construct in C, which repeats a block of code as long as the given condition is true.

Syntax:

```
while (condition) {
    // code to be executed as long as the condition is true
}
```

Example:

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

```
int main() {
   int count = 1;

   while (count <= 3) {
      printf("This is iteration number %d\n", count);
      count++;
   }

   return 0;
}
```

Que4. Discuss the following problem solving technique:-

1. Trail & Error
2. Divide and concur

Ans. **1. Trial and Error:**

Trial and Error is a problem-solving technique where various solutions are tried until the correct one is found. It involves systematically attempting different approaches, observing the outcomes, and learning from failures.

Steps:

Generate Hypotheses: Propose potential solutions or strategies.

Test Hypotheses: Implement the solutions and observe the results.

Analyse Outcomes: Assess the success or failure of each attempt.

Adjust and Retry: Modify the approach based on the observed outcomes and try again.

2. Divide and Conquer:

Divide and Conquer is a problem-solving technique that breaks down a problem into smaller, more manageable sub-problems. It involves recursively solving these sub-problems, combining their solutions, and ultimately solving the original problem.

Steps:

Divide: Break the problem into smaller, more manageable sub-problems.

Conquer: Solve each sub-problem independently, often through recursion.

Combine: Merge the solutions of sub-problems to obtain the solution for the original problem.

Que5. How do you call a function in main c progam? Explain with the help of C program of your choice?

Ans. Example

```
#include <stdio.h>

// Function declaration
void greetUser(char name[]);

int main() {
   // Call the greetUser function
   greetUser("Bhavy Sharma");

   return 0;
}

// Function definition
void greetUser(char name[]) {
   printf("Hello, %s! Welcome to the program.\n", name);
}
```

Function Declaration:

- The function `greetUser` is declared before the `main` function to inform the compiler about its existence. This is called a function prototype.
- `void greetUser(char name[]);` declares a function named `greetUser` that takes a character array (`char[]`) as an argument and returns `void` (no return value).

Function Definition:

- The actual implementation of the function is provided after the `main` function.
- `void greetUser(char name[])` defines the function. It prints a greeting message using the provided name.

Calling the Function in `main`:

- Inside the `main` function, we call the `greetUser` function with the argument "John" by writing `greetUser("Bhavy Sharma");`.

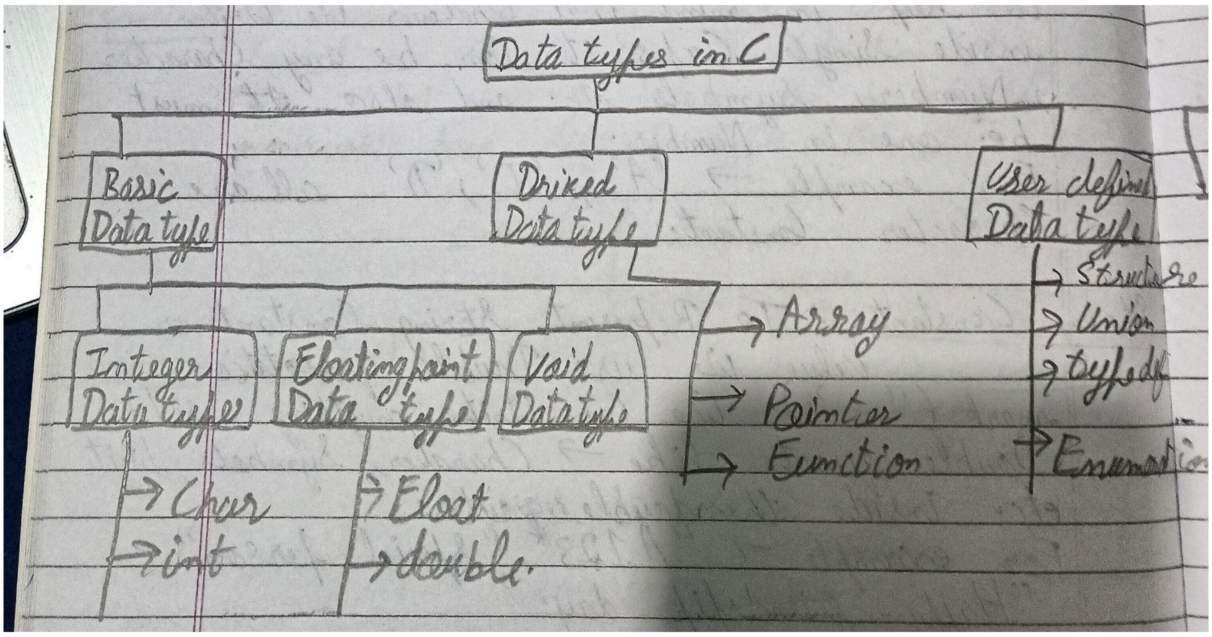
- This line invokes the `greetUser` function and passes the string "John" as the value of the `name` parameter.

Output:

- When you run this program, the output will be: `Hello, Bhavy Sharma! Welcome to the program.`

C language Part 2 By Bhavy Sharma

Que1. What is the need of data type in programming? Explain.
Ans. In C programming, data types are like labels for different types of information that a program can use. They tell the computer what kind of data a variable (a storage location in the program) can hold.
There are so many data types in C programming :



1. **Basic Data Types :**

    1. **Integer (int):**
    This data type is used for whole numbers. For example, if you want to store someone's age, you'd use an integer.
    2. **Floating-point (float):**
    This is for numbers with a decimal point. If you're dealing with someone's height, which can have decimal values, you'd use a float.
    3. **Character (char):**
    This is for single characters, like letters or symbols. If you want to store someone's first initial, you'd use a char.
    4. **Double:**
    Similar to float but can store larger decimal numbers with more precision. If you need more accuracy, you'd use a double.

2. Dived Data Types :

1. **Arrays:**
   Imagine you have a bunch of similar things, like a list of numbers or names. Instead of creating separate variables for each, you can use an array.
2. **Function:**
   Think of a function like a mini-program inside your main program. It's a way to group a set of instructions together that perform a specific task.

3. User Defined Data Types:

1. Structure:
A structure is a way to group different data types under a single name. It allows you to create a composite data type that can hold various pieces of information.

2. Union:

A union is similar to a structure, but it only allows one member to be accessed at a time. All members share the same memory space.

3. Typedef:

The `typedef` keyword is used to create an alias (a new name) for existing data types, including user-defined ones like structures and unions.

Qu2. What is The use of putchar()?

Ans. The `putchar()` function in C is used to print a single character to the standard output, which is usually the console or terminal.

Example:

```
#include <stdio.h>

void main() {
   // Use putchar to print a single character
   putchar('A');
}
```

Que3. Define recursion? Explain with the help of example.
Ans. Recursion in programming is a concept where a function calls itself to solve a smaller instance of the same problem.
Factorial Example:

The factorial of a non-negative integer `n`, denoted as `n!`, is the product of all positive integers less than or equal to `n`. For example, `5!` is calculated as `5 x 4 x 3 x 2 x 1`.

Code to Understand :

```
#include <stdio.h>

// 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);
   }
}

int main() {
   // Calculate and print factorial
   int result = factorial(5);
   printf("Factorial of 5 is: %d\n", result);

   return 0;
}
```

Que4. What is the use of getch() and getchar() function In C language?

Ans. Both `getch()` and `getchar()` are functions in C that are used to read a single character from the standard input (keyboard).

1. `getchar()` Function:

The `getchar()` function is used to read a single character from the standard input (keyboard). It waits for the user to press a key and then returns the ASCII value of that key.

Code:

```
#include <stdio.h>

void main() {
   char ch;

   // Prompt the user to enter a character
   printf("Enter a character: ");

   // Use getchar() to read a character
   ch = getchar();
   printf("You entered: %c\n", ch);
}
```

2. `getch()` Function:

The `getch()` function is part of the `<conio.h>` library (which is not standard and may not be available on all compilers). It reads a single character from the console without echoing it to the screen. This means that the character is input silently without showing on the screen.

Code :

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

void main() {
   char ch;

   // Use getch() to read a character silently
   ch = getch();

   // Display the entered character
   printf("You entered: %c\n", ch);
}
```

Que5. Explain the time and space complexity?
Ans. Time Complexity:

Definition: Time complexity represents the amount of time an algorithm takes to complete based on the input size.

Explanation: Think of time complexity as the total number of basic operations (like additions, comparisons, assignments) an algorithm needs to perform relative to the size of the input. It gives you an idea of how the execution time of an algorithm grows as the input size increases.

Space Complexity:

Definition: Space complexity represents the amount of memory an algorithm uses based on the input size.

Explanation: Space complexity tells you how much additional memory an algorithm needs relative to the size of the input. It includes both the memory required for variables and data structures.

Que6. What is function explain Function definition with suitable example?
Ans. Function Definition:
A function in programming is a block of code that has a name and can be executed whenever it's called. It can take input values (parameters), perform a series of operations, and return a result.

Code :

```
#include <stdio.h>
int addNumbers(int a, int b) {
   int sum = a + b;
   return sum;
}

void main() {
   // Call the addNumbers function with arguments 5 and 7
   int result = addNumbers(5, 7);
   printf("The sum is: %d\n", result);
}
```

Que7. Write short note on:

1. **Loops in C**
2. **If statement**
3. **Higher level language**

4. **Compiler**
5. **pre processor directive**

Ans.

1. Loops in C:

Loops in C are structures that allow you to repeat a certain block of code multiple times. There are different types of loops, but they all serve the purpose of making repetitive tasks more efficient.

Example of for loop :-

```
#include <stdio.h>
void main() {
   // This loop prints numbers from 1 to 5
   for (int i = 1; i <= 5; i++) {
      printf("%d\n", i);
   }
}
```

2. If Statement:

The `if` statement in C is used for decision-making. It allows you to execute a block of code if a specified condition is true.

Example of if Statement :

```
#include <stdio.h>
void main() {
   int age = 18;
   // This if statement checks if the age is greater than or equal to 18
   if (age >= 18) {
      printf("You are eligible to vote!\n");
   }
}
```

3. Higher-Level Language:

A higher-level programming language is designed to be more user-friendly and abstract, providing commands and structures that are closer to human language. This makes it easier for programmers to write code without dealing with low-level details.

4. Compiler:

A compiler is a special program that translates high-level source code written by a programmer into machine code or an intermediate code that can be executed by a computer. It helps in converting human-readable code into something the computer can understand.

5. Preprocessor Directive:

Preprocessor directives in C are special commands that start with a hash symbol (#). They are processed by the preprocessor before actual compilation. One common use is to include header files.

Example of Preprocessor Directive:
```
#include <stdio.h>
void main() {
   // Code using functions from the stdio.h header
   printf("Hello, World!\n");
}
```

Que8. What is operator? Explain different types of operators in C.
Ans. In programming, an operator is a symbol that represents a specific operation to be performed on one or more operands. Operands are the values or variables on which the operation is carried out.
There are Different types of operator :

1. Arithmetic Operators:

Explanation: These operators perform basic mathematical operations like addition, subtraction, multiplication, division, and modulus (remainder).

Example :
```
#include <stdio.h>
void main() {
   int a = 10, b = 3;
   printf("Addition: %d\n", a + b);
   printf("Subtraction: %d\n", a - b);
   printf("Multiplication: %d\n", a * b);
   printf("Division: %d\n", a / b);
   printf("Modulus: %d\n", a % b);
}
```

2. Relational Operators:

Explanation: These operators are used to compare two values and return a boolean result (true or false).

Example :

```
#include <stdio.h>

void main() {
   int x = 5, y = 8;

   printf("Is x equal to y? %d\n", x == y);
   printf("Is x not equal to y? %d\n", x != y);
   printf("Is x greater than y? %d\n", x > y);
   printf("Is x less than or equal to y? %d\n", x <= y);
}
```

3. Logical Operators:

Explanation: These operators are used to perform logical operations (AND, OR, NOT) on boolean values.

Example :

```
#include <stdio.h>

void main() {
   int p = 1, q = 0;

   printf("AND: %d\n", p && q);
   printf("OR: %d\n", p || q);
   printf("NOT: %d\n", !p);
}
```

4. Assignment Operators:

Explanation: These operators are used to assign values to variables.

Example:

```
#include <stdio.h>

void main() {
   int a = 5;
```

```
    // Using assignment operators
    a += 3;  // equivalent to a = a + 3
    printf("New value of a: %d\n", a);
}
```

5. Bitwise Operators:

Explanation: These operators perform operations on individual bits of binary numbers.

Example :

```
#include <stdio.h>

void main() {
    int x = 5, y = 3;

    printf("Bitwise AND: %d\n", x & y);
    printf("Bitwise OR: %d\n", x | y);
    printf("Bitwise XOR: %d\n", x ^ y);
}
```

Programs

Que1. Write a C program to check whether the number is Armstrong or not?

Ans.

```
        #include<stdio.h>
         int main()
        {
        int n,r,sum=0,temp;
        printf("enter the number=");
        scanf("%d",&n);
        temp=n;
        while(n>0)
        {
        r=n%10;
        sum=sum+(r*r*r);
        n=n/10;
```

```
}
if(temp==sum)
printf("armstrong  number ");
else
printf("not armstrong number");
return 0;
}
```