Q.01 A. Explain the structure of C program in detail. Write a sample program to demonstrate the components in the structure of C program

The structure of a C program can be broken down into several components:

1. Preprocessor directives: These are commands that begin with a hash symbol (#) and are used to tell the compiler to perform certain actions before compiling the code. Examples of preprocessor directives include #include, which tells the compiler to include header files in the code, and #define, which is used to define constants or macros.
2. Main function: The main function is the entry point of the program and is the first function that is executed when the program runs. This function contains the code that performs the main tasks of the program.
3. Variables: Variables are used to store data that the program needs to use. In C, variables must be declared before they can be used. They are declared by specifying the data type of the variable and giving it a name.
4. Statements: Statements are the individual instructions that make up the program. Statements can include variable declarations, assignments, and function calls, among others.
5. Functions: Functions are blocks of code that perform specific tasks. They can take arguments and return values, and can be called from other parts of the program.
6. Comments: Comments are used to explain the code and make it easier to understand. They are ignored by the compiler and do not affect the behavior of the program.

B. Demonstrate formatted output of integer in C with suitable example

In C, formatted output is achieved using the `printf()` function, which allows us to specify the format of the output using format specifiers. The format specifier for an integer is `%d`.

Here is an example program that demonstrates formatted output of an integer:

```
#include <stdio.h>



int main() {

   int num = 42;
   printf("The value of num is: %d\n", num);



   return 0;

}
```

In this program, we have declared an integer variable called `num` and initialized it to the value of 42. We then use the `printf()` function to output the value of `num` using the format specifier `%d`. The `%d` format specifier tells `printf()` to output an integer.

When we run this program, we should see the following output:

```
The value of num is: 42
```

Note that the format specifier `%d` can also be used with modifiers such as `0`, `-`, `+`, and `width`, which allow us to control the alignment and formatting of the output. For example, to output an integer with leading zeros, we can use the format specifier `%04d`, which tells `printf()` to pad the output with zeros until it is four digits wide.

Errors in a program can be broadly categorized into three types: syntax errors, logical errors, and runtime errors.

C. Discuss different types of error occur in program

1. Syntax errors: Syntax errors are errors that occur when the code violates the rules of the programming language. Examples of syntax errors include missing semicolons, mismatched parentheses, and misspelled keywords. Syntax errors are detected by the compiler during the compilation process and prevent the program from being compiled until they are corrected.
2. Logical errors: Logical errors are errors that occur when the code is syntactically correct but does not produce the expected output. Logical errors are usually caused by errors in the program's logic or by incorrect assumptions about how the program will behave. These errors can be difficult to detect and may require careful examination of the code and its output to identify and fix.
3. Runtime errors: Runtime errors are errors that occur while the program is running. Examples of runtime errors include division by zero, accessing an invalid memory location, and calling a function with the wrong number or type of arguments. Runtime errors can be caused by a wide range of factors, including incorrect input data, memory leaks, and hardware or software failures.

It is important to note that errors in a program can have different levels of severity. While some errors may be relatively harmless and only result in minor issues or unexpected behavior, other errors can cause the program to crash or produce incorrect or unexpected results. It is important to identify and fix errors in a program as early as possible to ensure that it is functioning correctly and reliably.

Q.02 A. Explain the various rules for forming identifiers names. Give examples for valid and invalid identifiers for the same.

In programming, an identifier is a name that is used to identify a variable, function, or any other user-defined item. Identifiers play a crucial role in programming as they help us to create meaningful and readable code. The rules for forming valid identifiers in most programming languages are as follows:

1. The first character must be a letter or an underscore (_).
2. The rest of the characters can be letters, digits, or underscores.
3. Identifiers are case-sensitive, which means that uppercase and lowercase letters are considered different.
4. Identifiers cannot be a reserved word or keyword in the programming language.

Examples of valid identifiers:

num

count

_temp

hello_world

first_name

Examples of invalid identifiers:

123num (first character is a digit)

my-name (contains a hyphen)

#temp (contains a special character)

while (a reserved keyword in C)

It is important to follow the rules for forming identifiers in order to avoid errors and ensure that the program runs correctly.

B. Mention various output devices and explain hardcopy devices

Output devices are computer hardware devices that display or produce information generated by a computer system. Some common output devices include:

1. Monitors: Also known as displays or screens, monitors are devices that display visual output from a computer. Monitors come in various sizes and types, including CRT, LCD, LED, and OLED.
2. Printers: Printers are devices that produce hardcopy output by printing characters, images, or graphics onto paper. Printers come in various types, including inkjet, laser, and dot matrix.
3. Speakers: Speakers are devices that produce audio output from a computer system. They can be used for playing music, sound effects, or voice recordings.
4. Projectors: Projectors are devices that display visual output onto a large surface, such as a wall or screen. They are often used in presentations and for home theater systems.
5. Plotters: Plotters are specialized printers that are used to produce large-format output, such as technical drawings, maps, or diagrams. They use pens or other writing instruments to produce the output.

Hardcopy devices are output devices that produce a physical, permanent copy of the output. Examples of hardcopy devices include printers and plotters. Hardcopy output can be useful for creating physical records, reports, or documents that can be stored, shared, or distributed. Hardcopy output can also be used for producing high-quality prints of images or graphics that may be difficult or impossible to reproduce accurately on a screen.

C. Discuss the variants of microcomputer that are widely used today

Microcomputers, also known as personal computers or PCs, are small computers that are designed for individual use. They are widely used today for a variety of tasks, including word processing, internet browsing, gaming, and multimedia playback. Here are some of the variants of microcomputers that are widely used today:

1. Desktop computers: Desktop computers are microcomputers that are designed to be used on a desk or table. They consist of a tower or case that contains the computer's components, such as the motherboard, processor, memory, and storage. Desktop computers typically have larger screens, more powerful processors, and more storage capacity than other types of microcomputers.
2. Laptop computers: Laptop computers, also known as notebook computers, are portable microcomputers that are designed to be used on the go. They are lightweight and compact, making them easy to carry around. Laptops typically have smaller screens, less powerful processors, and less storage capacity than desktop computers, but they are more convenient for mobile use.
3. Tablets: Tablets are microcomputers that are designed to be used with a touch screen interface. They are similar in size and shape to a notepad, making them easy to carry around. Tablets typically have less powerful processors and less storage capacity than laptops, but they are more convenient for casual use, such as browsing the web, reading ebooks, or playing games.
4. Smartphones: Smartphones are microcomputers that are designed to be used as mobile phones. They are similar in size and shape to a small tablet or a large phone, and they are designed to be used with a touch screen interface. Smartphones typically have less powerful processors and less storage capacity than tablets, but they are more convenient for mobile use, such as making phone calls, sending text messages, or using mobile apps.

Q. 03 A. Demonstrate the functioning of Bitwise operator in C

Bitwise operators are operators that manipulate the bits of a variable at the binary level. There are six bitwise operators in C, which are:

1. & (AND): Performs bitwise AND operation on two operands and returns the result.
2. | (OR): Performs bitwise OR operation on two operands and returns the result.
3. ^ (XOR): Performs bitwise XOR (exclusive OR) operation on two operands and returns the result.
4. ~ (NOT): Performs bitwise NOT (complement) operation on a single operand and returns the result.
5. << (LEFT SHIFT): Shifts the bits of the left operand to the left by the number of bits specified by the right operand.
6. (RIGHT SHIFT): Shifts the bits of the left operand to the right by the number of bits specified by the right operand.

Here's an example program that demonstrates the functioning of bitwise operators in C:

```
#include <stdio.h>



int main() {

  int a = 10, b = 6, c;



  c = a & b;

  printf("a & b = %d\n", c);



  c = a | b;

  printf("a | b = %d\n", c);



  c = a ^ b;

  printf("a ^ b = %d\n", c);



  c = ~a;

  printf("~a = %d\n", c);



  c = a << 2;

  printf("a << 2 = %d\n", c);



  c = a >> 2;
```

```
    printf("a >> 2 = %d\n", c);



    return 0;

}
```

B. Write a C program to find roots of quadratic equation

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

int main()
{
    float a, b, c, root1, root2, discriminant, realPart, imaginaryPart;

    printf("Enter coefficients a, b and c: ");
    scanf("%f %f %f", &a, &b, &c);

    discriminant = b*b - 4*a*c;

    // Case 1: Real and unequal roots
    if (discriminant > 0)
    {
        root1 = (-b + sqrt(discriminant)) / (2*a);
        root2 = (-b - sqrt(discriminant)) / (2*a);
        printf("Roots are: %.2f and %.2f", root1, root2);
    }

    // Case 2: Real and equal roots
    else if (discriminant == 0)
    {
        root1 = root2 = -b / (2*a);
        printf("Roots are: %.2f and %.2f", root1, root2);
    }

    // Case 3: Imaginary roots
    else
    {
        realPart = -b / (2*a);
        imaginaryPart = sqrt(-discriminant) / (2*a);
        printf("Roots are: %.2f+%.2fi and %.2f-%.2fi", realPart, imaginaryPart,
realPart, imaginaryPart);
    }
    return 0;
}
```

C. Distinguish between the break and continue statement

In C programming language, `break` and `continue` are two control statements used to alter the flow of execution of loops. However, they are used in different ways and for different purposes.

The `break` statement is used to exit from a loop prematurely. When a `break` statement is encountered inside a loop, the loop is immediately terminated, and the control is transferred to the statement immediately following the loop. This is useful when we want to exit a loop based on a certain condition.

Here's an example of using `break` statement in a `for` loop:

```
for(int i = 0; i < 10; i++){

    if(i == 5){

        break; // terminate the loop when i becomes 5

    }

    printf("%d\n", i);

}
```

In this example, when `i` becomes 5, the `break` statement is executed, and the loop is terminated.

On the other hand, the `continue` statement is used to skip the current iteration of a loop and move on to the next iteration. When a `continue` statement is encountered inside a loop, the control is transferred to the beginning of the loop, and the next iteration starts. This is useful when we want to skip certain iterations based on a certain condition.

Here's an example of using `continue` statement in a `for` loop:

```
for(int i = 0; i < 10; i++){
    if(i == 5){

        continue; // skip the iteration when i becomes 5

    }

    printf("%d\n", i);

}
```

In this example, when `i` becomes 5, the `continue` statement is executed, and the loop moves on to the next iteration, skipping the printing of `5`.

In summary, `break` statement is used to exit a loop prematurely, while `continue` statement is used to skip the current iteration of a loop and move on to the next iteration.

Q.04 A. Illustrate Nested loops in C with suitable example

In C programming language, we can use nested loops to perform iterations within iterations. A nested loop is a loop inside another loop. Here's an example of using nested loops to print a pattern of asterisks:

```
#include <stdio.h>



int main() {

    int rows = 5;



    for(int i = 1; i <= rows; i++) {

        for(int j = 1; j <= i; j++) {

            printf("* ");
    }
```

B. Write a C program to print whether a given number is palindrome or not

A palindrome number is a number that remains the same when its digits are reversed. For example, 121 is a palindrome number because it remains the same when its digits are reversed.

Here's a C program to check whether a given number is palindrome or not:

```
#include <stdio.h>



int main() {

  int num, reversed_num = 0, remainder, original_num;



  printf("Enter a number: ");

  scanf("%d", &num);



  original_num = num;



  while(num != 0) {

    remainder = num % 10;

    reversed_num = reversed_num * 10 + remainder;

    num /= 10;

  }
```

```
    if(original_num == reversed_num) {

        printf("%d is a palindrome number.\n", original_num);

    }

    else {

        printf("%d is not a palindrome number.\n", original_num);

    }



    return 0;

}
```

In this program, we first take a number from the user and store it in the variable `num`. We also create a variable `reversed_num` to store the reversed number, and a variable `original_num` to store the original number.

Then, we use a `while` loop to reverse the digits of the given number. Inside the loop, we extract the last digit of the number using the modulus operator `%` and add it to `reversed_num` after

C. Explain switch statement with syntax. Write a C program to simulate calculator

In C programming language, the `switch` statement is a control statement used to select one of several code blocks to be executed based on the value of an expression. The syntax of the `switch` statement is as follows:

In this syntax, `expression` is evaluated and compared with each of the `constant` values in the `case` statements. If a match is found, the corresponding code block is executed. The `break` statement is used to exit the `switch` statement after the matching code block is executed. If none of the `case` statements match the value of `expression`, the code block in the `default` case is executed.

Here's a C program to simulate a calculator using the `switch` statement:

```
switch(expression) {

    case constant1:

        // code block 1

        break;

    case constant2:

        // code block 2

        break;

    case constant3:

        // code block 3

        break;

    ...

    default:

        // code block to be executed if none of the cases are matched

        break;

}
```

CALCULATOR PROGRAM

```
#include <stdio.h>


int main() {

    char operator;

    double num1, num2, result;


    printf("Enter an operator (+, -, *, /): ");
```

```
scanf("%c", &operator);


printf("Enter two numbers: ");

scanf("%lf %lf", &num1, &num2);


switch(operator) {

    case '+':

        result = num1 + num2;

        printf("%.2lf + %.2lf = %.2lf\n", num1, num2, result);

        break;

    case '-':

        result = num1 - num2;

        printf("%.2lf - %.2lf = %.2lf\n", num1, num2, result);

        break;

    case '*':

        result = num1 * num2;

        printf("%.2lf * %.2lf = %.2lf\n", num1, num2, result);

        break;

    case '/':

        if(num2 == 0) {

            printf("Error: Cannot divide by zero.\n");

        }

        else {
            result = num1 / num2;

            printf("%.2lf / %.2lf = %.2lf\n", num1, num2, result);

        }

        break;

      default:

        printf("Error: Invalid operator.\n");

        break;

  }


  return 0;

}
```

Q. 05 A. Write a C program to implement Bubble sort technique(ascending order)

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. Here's a C program to implement the bubble sort technique to sort an array of integers in ascending order:

```
#include <stdio.h>



void bubbleSort(int arr[], int n) {

   int i, j, temp;



   for(i = 0; i < n - 1; i++) {

      for(j = 0; j < n - i - 1; j++) {
           if(arr[j] > arr[j+1]) {
               // swap arr[j] and arr[j+1]
               temp = arr[j];
               arr[j] = arr[j+1];
               arr[j+1] = temp;
           }
      }
   }
}

int main() {
   int arr[100], n, i;

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

   printf("Enter %d integers:\n", n);
   for(i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   bubbleSort(arr, n);

   printf("Sorted array in ascending order:\n");
   for(i = 0; i < n; i++) {

      printf("%d ", arr[i]);

   }



   return 0;

}
```

B. Illustrate the concept of recursive function with example

A recursive function is a function that calls itself within its own definition. This can be a useful technique in programming when a task can be broken down into smaller sub-tasks that can be solved recursively.

Here's an example of a recursive function in C that calculates the factorial of a given number:

```
#include <stdio.h>



int factorial(int n) {

    if(n == 0) {

        return 1;

    }

    else {

        return n * factorial(n-1);

    }

}



int main() {

    int num;
    printf("Enter a non-negative integer: ");

    scanf("%d", &num);



    printf("Factorial of %d = %d\n", num, factorial(num));



    return 0;
}
```

C. Discuss various scope of variables

In C, the scope of a variable refers to the part of the program where the variable is visible and can be accessed. There are three types of variable scope in C:

1. Global scope: A variable declared outside of any function has global scope. Global variables can be accessed and modified from any part of the program. Global variables are generally declared at the top of the file, before any functions are defined
2. Local scope: A variable declared inside a function has local scope. Local variables can only be accessed and modified within the function where they are declared. Once the function returns, the value of the local variable is lost.
3. Block scope: A variable declared inside a block of code (i.e. within curly braces) has block scope. Block scope variables can only be accessed and modified within the block where they are declared

It's important to understand the scope of variables in C, because it affects how you write your programs and how you organize your data. By choosing the right scope for your variables, you can avoid bugs, improve code readability, and make your programs easier to maintain.

Q. 06 A. Differentiate between call by value and call by reference. Using suitable example

In C, there are two ways to pass arguments to a function: call by value and call by reference.

Call by value: In call by value, a copy of the argument is passed to the function. Any changes made to the parameter inside the function do not affect the original value of the argument. Here's an example:

```
#include <stdio.h>



void swap(int a, int b) {

    int temp = a;

    a = b;

    b = temp;

}



int main() {

    int x = 10;

    int y = 20;

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

    swap(x, y);

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

    return 0;

}
```

In this example, the `swap` function takes two arguments `a` and `b`, which are copied from the values of `x` and `y` in `main`. Inside `swap`, the values of `a` and `b` are swapped using a temporary variable. However, because `swap` uses call by value, the values of `x` and `y` in `main` are not affected by the function. The output of this program will be:

```
Before swap: x = 10, y = 20

After swap: x = 10, y = 20
```

As you can see, the values of `x` and `y` are not swapped, because `swap` uses call by value.

Call by reference: In call by reference, a reference to the argument is passed to the function. Any changes made to the parameter inside the function affect the original value of the argument. Here's an example:

```
#include <stdio.h>



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

    int temp = *a;

    *a = *b;

    *b = temp;

}



int main() {

    int x = 10;

    int y = 20;

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

    swap(&x, &y);

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

    return 0;

}
```

B. Write a C program to transpose a MxN matrix

```
#include <stdio.h>

#define M 3 // number of rows
#define N 4 // number of columns

void transpose(int matrix[M][N]) {
    int temp;

    for (int i = 0; i < M; i++) {
        for (int j = i+1; j < N; j++) {
            temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
        }
    }
}

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

```
int main() {
    int matrix[M][N] = {{1, 2, 3, 4},
                        {5, 6, 7, 8},
                        {9, 10, 11, 12}};

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

    transpose(matrix);

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

    return 0;
}
```

C. Discuss the various storage classes

In C programming, a storage class is used to specify the lifetime, visibility, and scope of a variable or function. There are four types of storage classes in C:

1. `auto`: The `auto` storage class is the default storage class for local variables. It is used to declare variables that are local to a block or function. The memory for an `auto` variable is allocated when the function or block is entered and it is deallocated when the function or block is exited.
2. `static`: The `static` storage class is used to declare variables that are local to a block or function but their value persists across multiple function calls. A `static` variable is allocated once and retains its value throughout the program execution.
3. `register`: The `register` storage class is used to declare variables that should be stored in a CPU register for faster access. This storage class is only a request to the compiler, which can choose to ignore it.

4. `extern`: The `extern` storage class is used to declare variables or functions that are defined in a separate file or module. It is used to provide global access to a variable or function across multiple files.

Q. 07 A. Mention various operations that can be performed on string using built-in functions. Explain any two function

In C programming, strings are defined as an array of characters. There are several built-in functions to perform various operations on strings. Some of the operations that can be performed using built-in functions are:

1. Concatenation of strings: This is done using the strcat() function. It takes two arguments, the first being the destination string and the second being the source string. The function appends the source string to the end of the destination string.
2. Copying strings: This is done using the strcpy() function. It takes two arguments, the first being the destination string and the second being the source string. The function copies the contents of the source string to the destination string.
3. Comparison of strings: This is done using the strcmp() function. It takes two arguments, the first being the first string to be compared and the second being the second string to be compared. The function returns an integer value based on the comparison result.
4. Finding the length of a string: This is done using the strlen() function. It takes one argument, which is the string whose length is to be found. The function returns an integer value which is the length of the string.

2.strcat(): This function is used to concatenate two strings. The syntax of the function is:
char *strcat(char *dest, const char *src);
1.strcmp(): This function is used to compare two strings. The syntax of the function is:
int strcmp(const char *str1, const char *str2);

B. Develop a program using pointer to compute the sum, mean and standard deviation of all element stored in array of N real number

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

#define N 5

void calculateStats(float arr[], int n, float *sum, float *mean, float *stddev) {
    // Calculate sum of elements
    *sum = 0;
    for(int i = 0; i < n; i++) {
        *sum += arr[i];
    }

    // Calculate mean
    *mean = *sum / n;

    // Calculate standard deviation
    *stddev = 0;
    for(int i = 0; i < n; i++) {
        *stddev += pow(arr[i] - *mean, 2);
    }
    *stddev = sqrt(*stddev / n);
}

int main() {
    float arr[N] = {1.2, 2.4, 3.6, 4.8, 6.0};
    float sum, mean, stddev;

    // Calculate stats using pointer
    calculateStats(arr, N, &sum, &mean, &stddev);
    // Print results
    printf("Array: ");
    for(int i = 0; i < N; i++) {
        printf("%0.1f ", arr[i]);
    }
    printf("\n");
    printf("Sum: %0.1f\n", sum);
    printf("Mean: %0.1f\n", mean);
    printf("Standard Deviation: %0.2f\n", stddev);

    return 0;
}
```

C. Explain how strings are represented in main memory

In C, a string is an array of characters terminated by a null character ('\0'). When a string is declared and initialized in C, the characters of the string are stored in consecutive memory locations in main memory.

For example, if we declare and initialize a string as follows:

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

The compiler reserves enough consecutive memory locations to store each character of the string, including the terminating null character, and initializes each memory location with the corresponding character value. The resulting memory representation would look something like this:

```
  +------+------+------+------+------+------+------+------+

 | 'H'  | 'e'  | 'l'  | 'l'  | 'o'  | ','  | ' '  | 'w'  |

  +------+------+------+------+------+------+------+------+
 | 'o'  | 'r'  | 'l'  | 'd'  | '!'  | '\0' |      |      |

 +------+------+------+------+------+------+------+------+
```

Here, each character of the string is stored in a single byte of memory, and the terminating null character is represented by the value 0 (or '\0'). Note that the memory locations beyond the last character and the null character are left uninitialized.

It's important to note that C strings are stored as arrays of characters, which means that we can use array indexing to access individual characters of the string. Additionally, since strings are represented as pointers to the first character of the string, we can also use pointer arithmetic to traverse the string.

Q. 08 A. Write a program to compare two strings without using built-in function

```c
#include <stdio.h>


int compareStrings(char str1[], char str2[]) {
  int i = 0;


  while (str1[i] == str2[i]) {
    if (str1[i] == '\0' || str2[i] == '\0')
      break;
    i++;
  }


  if (str1[i] == '\0' && str2[i] == '\0')
    return 0;
  else
    return 1;
}
```

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

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

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

  if (compareStrings(str1, str2) == 0)
    printf("Both strings are equal.");
  else
    printf("Both strings are not equal.");

  return 0;
}
```

B. What is pointer? Discuss pointer arithmetic with suitable C code

In C programming language, a pointer is a variable that stores the memory address of another variable. Pointers are useful in many situations, such as passing arguments to functions, dynamically allocating memory, and accessing array elements.

Pointer arithmetic is the ability to perform arithmetic operations on pointers in C programming. Pointer arithmetic allows us to navigate through arrays and other data structures by incrementing or decrementing pointers. The arithmetic is done in terms of the size of the data type that the pointer points to.

```
#include <stdio.h>



int main() {
    int arr[] = {10, 20, 30, 40, 50};

    int *ptr = arr; // ptr points to the first element of arr



    // Using pointer arithmetic to access array elements

    printf("arr[0] = %d\n", *ptr); // prints 10

    ptr++; // move pointer to the next element

    printf("arr[1] = %d\n", *ptr); // prints 20

    ptr += 2; // move pointer two elements forward

    printf("arr[3] = %d\n", *ptr); // prints 40

    ptr--; // move pointer one element back

    printf("arr[2] = %d\n", *ptr); // prints 30



    return 0;

}
```

C. Explain gets()and puts() function with example

The gets() function and the puts() function are two string input/output functions in C.

gets() function: The gets() function is used to read a line of text from the standard input (stdin) and store it as a string. It reads the input until it encounters a newline character '\n' or an end-of-file character EOF. It also adds a null character '\0' at the end of the string, replacing the newline character. The gets() function is declared in the stdio.h header file.

```
char *gets(char *str);
```

```
#include <stdio.h>

int main() {

   char name[50];

   printf("Enter your name: ");

   gets(name);

   printf("Hello, %s!\n", name);

   return 0;

}
```

In this example, the gets() function reads a string from the user and stores it in the name array. Then it prints a message with the name.

Note: The gets() function is unsafe to use because it does not check the length of the input, and it can lead to buffer overflow problems. It is recommended to use fgets() function instead of gets() function.

puts() function: The puts() function is used to output a string to the standard output (stdout). It writes the string and a newline character '\n' to the output. The puts() function is declared in the stdio.h header file.

Q. 09 A. Explain various modes in which file can be opened for processing

In C programming language, files can be opened in different modes for processing. These modes are specified in the mode parameter of the fopen() function. The various modes in which a file can be opened are:

1. "r" (Read mode): This mode is used to open an existing file for reading. The file pointer is positioned at the beginning of the file. If the file does not exist, fopen() returns NULL.
2. "w" (Write mode): This mode is used to create a new file or overwrite an existing file. If the file exists, its contents are truncated to zero length. The file pointer is positioned at the beginning of the file.

3. "a" (Append mode): This mode is used to open an existing file for writing at the end of the file. If the file does not exist, it is created. The file pointer is positioned at the end of the file.
4. "r+" (Read/Write mode): This mode is used to open an existing file for both reading and writing. The file pointer is positioned at the beginning of the file.
5. "w+" (Write/Read mode): This mode is used to create a new file or overwrite an existing file for both reading and writing. If the file exists, its contents are truncated to zero length. The file pointer is positioned at the beginning of the file.
6. "a+" (Append/Read mode): This mode is used to open an existing file for both reading and writing at the end of the file. If the file does not exist, it is created. The file pointer is positioned at the end of the file.

B. Implement structure to read, write and compute average marks of the students. List the students scoring above and below the average marks for a class of n students

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

#define MAX_STUDENTS 100

// Define a structure for storing student information
struct student {
    char name[50];
    int roll_number;
    float marks;
};

// Function prototypes
float compute_average(struct student[], int);
void list_above_below_average(struct student[], int, float);
```

```
int main() {
    struct student students[MAX_STUDENTS];
    int n;

    // Read the number of students from user
    printf("Enter the number of students: ");
    scanf("%d", &n);

    // Read the student records from user
    printf("Enter the student records:\n");
    for (int i = 0; i < n; i++) {
        printf("Student %d\n", i+1);
        printf("Name: ");
        scanf("%s", students[i].name);
        printf("Roll Number: ");
        scanf("%d", &students[i].roll_number);
        printf("Marks: ");
        scanf("%f", &students[i].marks);
    }

    // Compute the average marks of all students
    float average_marks = compute_average(students, n);
    printf("The average marks of all students is: %.2f\n", average_marks);

    // List the students above and below the average marks
    list_above_below_average(students, n, average_marks);

    return 0;
}
```

```
// Compute the average marks of all students
float compute_average(struct student students[], int n) {
    float total_marks = 0;
    for (int i = 0; i < n; i++) {
        total_marks += students[i].marks;
    }
    return total_marks / n;
}

// List the students above and below the average marks
void list_above_below_average(struct student students[], int n, float average_marks) {
    printf("The students scoring above average marks are:\n");
    for (int i = 0; i < n; i++) {
        if (students[i].marks > average_marks) {
            printf("%s (Roll Number: %d, Marks: %.2f)\n", students[i].name,
students[i].roll_number, students[i].marks);
        }
    }
    printf("The students scoring below average marks are:\n");
    for (int i = 0; i < n; i++) {
        if (students[i].marks < average_marks) {
            printf("%s (Roll Number: %d, Marks: %.2f)\n", students[i].name,
students[i].roll_number, students[i].marks);
        }
    }
}
```

C. What are enumeration variable? How are they declared

Enumeration variables in C are used to create a new data type with a set of named constants. It is a user-defined data type that contains a finite set of constant values.

Enumeration variables are declared using the `enum` keyword, followed by the name of the enumeration type, and a list of possible values enclosed in braces, as shown below:

enum color {RED, GREEN, BLUE};

In the above example, we have declared an enumeration type called `color` which contains three possible values: `RED`, `GREEN`, and `BLUE`. Each of these values is assigned an integer constant by the compiler, starting from 0 for the first value and incrementing by 1 for each subsequent value. So in the above example, `RED` has the value 0, `GREEN` has the value 1, and `BLUE` has the value 2.

Once the enumeration type is defined, we can declare variables of that type and assign them one of the values defined in the enumeration type. For example, we can declare a variable of type `color` and assign it the value `GREEN` as follows:

enum color mycolor;

mycolor = GREEN;

Q. 10 A. Write a short note on functions used to Read data from a, file Write data to a file

In C programming language, there are several functions available to read data from a file and write data to a file. These functions are used to perform input and output operations on files.

The function used to read data from a file is called "fread()". It takes four parameters: a pointer to the array where the data will be stored, the size of the data type to be read, the number of data elements to be read, and a file pointer that points to the file to be read.

size_t fread(void *ptr, size_t size, size_t count, FILE *stream);

The function used to write data to a file is called "fwrite()". It takes four parameters: a pointer to the data to be written, the size of the data type to be written, the number of data elements to be written, and a file pointer that points to the file to be written.

```
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);

#include <stdio.h>



int main() {

    FILE *fp;

    int num;



    fp = fopen("input.txt", "r");

    if (fp == NULL) {

        printf("Error opening file.\n");

        return 1;

    }



    fread(&num, sizeof(int), 1, fp);

    printf("The number read from the file is: %d\n", num);



    fclose(fp);

    return 0;

}
```

B. Differentiate structures and unions with syntax and example

In C programming language, structures and unions are used for storing data of different types. Both structures and unions are user-defined data types that can hold different variables of different data types. However, there are some differences between structures and unions:

1. Syntax: The syntax for declaring a structure is:
```
struct structure_name
{
    data_type variable_1;
    data_type variable_2;
    data_type variable_3;
    ...
};

UNION SYNTAX:

union union_name
{
    data_type variable_1;
    data_type variable_2;
    data_type variable_3;
    ...
};
```
2. Memory Allocation: When we declare a structure, memory is allocated for all its variables separately. The total memory allocated for a structure is equal to the sum of memory required by each variable in the structure.

When we declare a union, memory is allocated for only one variable at a time. The total memory allocated for a union is equal to the memory required by its largest variable.

3. Accessing Variables: In a structure, we can access each variable separately using the dot operator (.). For example:
```
struct student
{
    char name[20];
    int roll_no;
    float marks;
};

struct student s;
```

```
s.roll_no = 10;
s.marks = 85.5;
strcpy(s.name, "John");
```

C. How to detect END-OF-FILE

In C programming language, the end of a file can be detected using the `feof()` function. The `feof()` function checks whether the end-of-file indicator associated with a file has been set or not. The syntax for `feof()` function is:

```
int feof(FILE *stream);
```

The `feof()` function takes a pointer to a file stream as an argument and returns a non-zero value if the end-of-file indicator associated with the stream has been set, and 0 otherwise.

Here is an example of how to use `feof()` function to detect end-of-file in a file:

```
#include <stdio.h>



int main() {

  FILE *fp;

  int ch;



  fp = fopen("myfile.txt", "r");

  if(fp == NULL) {

    printf("Error opening file.\n");

    return 1;

  }



  while(!feof(fp)) {
    ch = fgetc(fp);

    if(ch == EOF) {

      break;

    }

    putchar(ch);

  }



  fclose(fp);

  return 0;

}
```

In the above example, we first open a file `"myfile.txt"` in read mode. Then, we use a `while` loop to read each character from the file until we reach the end-of-file. Inside the loop, we use the `fgetc()` function to read a character from the file, and check if the character is equal to `EOF`. If it is, we break out of the loop. Otherwise, we print the character using `putchar()` function. Finally, we close the file using `fclose()` function