2076 CHAITRA REGULAR QUESTION SOLUTION

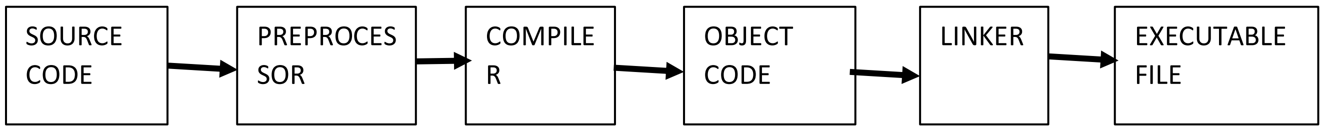
1. What are the different types of computer software? What do you mean by high level and low-level programming language? Along with the block diagram explain the steps involved during the compilation of source code.

→ Software is the set of coded commands or instructions which are provided to the computer to perform meaningful tasks. There are basically two types of software. They are mentioned below.

a. System Software: System software is a type of software which is responsible for controlling, integrating and managing the individual hardware components of a computer system. It compromises software written in low level language, which interacts with hardware at a very basic level.

b. Application Software: Application software is specific purpose software which is used by user for performing specific tasks. It includes a set of programs that do real work for users. There are two types of application software. They are; General Purpose Software and Specific Purpose Software.

A high-level language is a programming language that enables a programmer to write programs independent of a particular type of computer. Every high-level language has its own set of rules to represent instructions. This rule is called syntax. The main advantage of high-level programming over low level programming language is that they are easier to read, write and maintain. On the other hand, A low-level language is a programming language much closer to the hardware. It requires a thorough knowledge of the hardware for which the program is being created. Program written for one type of machine may not run on other machines of different manufacture. It can be divided into two types of mainly machine language and assembly language.



Source Code: The programmer writes the code in high level language which is called source code.

Preprocessor: The preprocessor takes the source code as input and removes all the comments and other irrelevant things in the source code.

Compiler: The code which is expanded by preprocessor is passed to the compiler. The preprocessed data is converted to object code with the help of compiler or assembler.

Object Code: Code produced by the compiler is object code.

Linker: Mainly, all the programs written in C are library functions. These library functions are precompiled, and the object code of these library files is stored in .lib extension. The main working function of the linker is to combine the object code of library files to object code of our main programs.

Executable file: Finally, the linker produces an executable file that can be run directly on the target system. The executable file contains the machine code, data and other necessary information.

2. Explain different types of errors that appear during programming. Define processing directive and explain its type with examples. Write an algorithm and draw a flowchart to find the reverse of the given number.

→ a. Syntax error: Any violation of rules of the language results in syntax error. The compiler can detect and isolate such errors. When syntax errors are present, the compilation fails and is terminated after listing the errors and the line numbers in the source program, where the errors have occurred. Such an error occurs when a computer language compiler cannot understand a command entered by the user. It is easy to correct the syntax errors.

b. Runtime error: Errors such as mismatch of data types or referencing an out-of-range array element go undetected by the compiler. A program with these mistakes will run but produces the erroneous result. Isolating a run time error is usually a difficult task. These errors are encountered after error free compilation, at the time of execution of the program. Some typical examples are:

• Data incorrect or in wrong order

• Incorrect file name

• Infinite loops

• Divide check errors – these errors appear when a quantity is divided by zero.

• Correct outputs only for selected data.

c. Logical errors These errors are related to the logic of the program execution. Such actions as taking the wrong path, failure to consider a particular condition and incorrect order of evaluation of statements belong to this category. Logical errors do not show up as compiler-generated error messages. Rather, they cause incorrect results. These errors are primarily due to a poor understanding of the problem, incorrect translation of the algorithm and a lack of clarity of the hierarchy (precedence) of operators. For Example: for calculating net salary the formula is: Net Salary = Basic Salary + Allowances – Deductions But through the oversight, while coding, the formula is written as: Net Salary = Basic Salary - Allowances + Deductions Such errors can be detected by dry run.

d. Latent errors It is a ‘hidden’ error that shows up only when a particular set of data is used. Example: ratio = (x + y)/(p - q) The above expression generates error when p = q. A Reference B

Step1: start

Step2: declare n, rev_num

Step 3: read n

Step 4: assign rev_num=0

Step 5: if n>0 then goto step 5.1

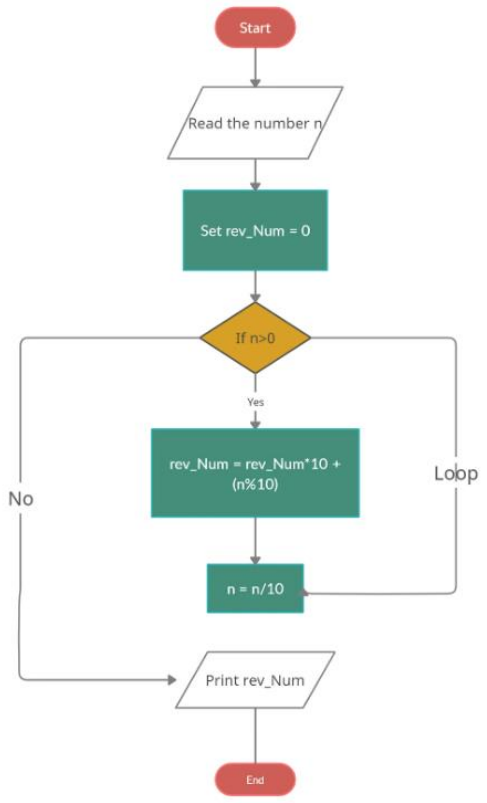
Step 5.1: rev_num=rev_num*10+(n%10)

Step 5.2: n=n/10

Step 5.3: goto step 5

Step 6: display rev_num

Step 7: stop



The C preprocessor is a collection of special statements, called directives that are executed at the

beginning of compilation process. Preprocessors directives usually appear at the beginning of a

program. A preprocessor directive may appear anywhere within a program. Preprocessor

Directives follow special syntax rules that are different from the normal C syntax. They all begin

with the symbol # in column one and do not require a semicolon at the end. We have already

used the directives #define and #include to a limited extent. A set of commonly used

preprocessor directives and their functions are listed below:

#define-----defines a macro substitution

#undef------undefines a macro

#include----specifies the files to be inckuded
#ifdef--------test for a macro definition

3. Why are formatted output important in C language? Write a program to print all the roots (even imaginary roots) of quadratic equation.

$\longrightarrow$ Formatted output refers to the organized and structured presentation of data in a specific manner, often with precise control over elements such as alignment, spacing, decimal places, and formatting styles. Some of its importance is mentioned below:

a. Readability Output that has been formatted is easier to understand and use. It enables you to arrange and structure data presentation, which makes it simpler for consumers to comprehend and analyze the information being shown.

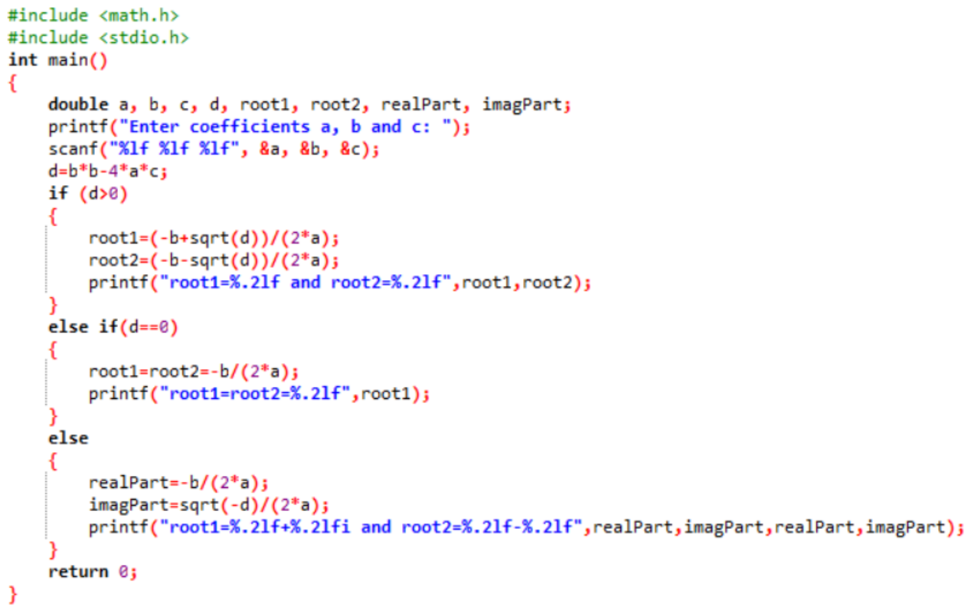
b. Alignment and Space: You may manage the alignment, spacing, and indentation of the output by Precision: When displaying decimal numbers, scientific notation, or other specific numerical forms, formatted output allows you to adjust the precision of numeric data.

c. Unit conversions: Formatted output can be used in scientific or engineering applications to convert units and present measurements in a consistent manner, enhancing clarity and reducing mistakes.

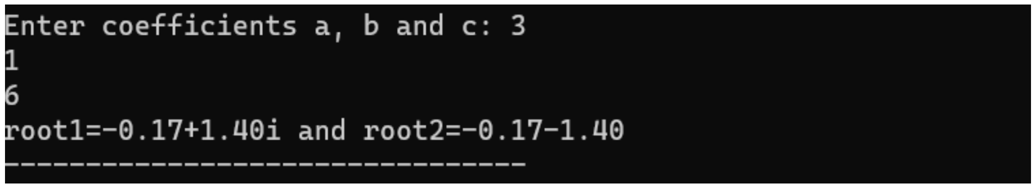
d. Localization and internationalization: Formatted output can be modified to accommodate various linguistic systems, regional cultures, and geographic regions. This is crucial for developing software that is simple to localize for users in many locales.

e. Consistency: You may ensure that your program's output follows a defined and predictable format by consistently employing formatted output throughout your code. This can be helpful for debugging and troubleshooting

using formatted output. This is very helpful for tabular data presentation or for organizing information cleanly into columns.



```c
#include <math.h>
#include <stdio.h>
int main()
{
    double a, b, c, d, root1, root2, realPart, imagPart;
    printf("Enter coefficients a, b and c: ");
    scanf("%lf %lf %lf", &a, &b, &c);
    d=b*b-4*a*c;
    if (d>0)
    {
        root1=(-b+sqrt(d))/(2*a);
        root2=(-b-sqrt(d))/(2*a);
        printf("root1=%.2lf and root2=%.2lf",root1,root2);
    }
    else if(d==0)
    {
        root1=root2=-b/(2*a);
        printf("root1=root2=%.2lf",root1);
    }
    else
    {
        realPart=-b/(2*a);
        imagPart=sqrt(-d)/(2*a);
        printf("root1=%.2lf+%.2lfi and root2=%.2lf-%.2lf",realPart,imagPart,realPart,imagPart);
    }
    return 0;
}
```



```
Enter coefficients a, b and c: 3
1
6
root1=-0.17+1.40i and root2=-0.17-1.40
--------------------------------
```

4. Explain the importance of a switch case statement. Compare it with if-else ladder. Write a program to find sum of numbers 1 to 100 which are exactly, divisible by 5 and not by 3.

→The switch statement causes a group of statements to be executed and rest skipped as done by a series of if else if statements. Its importance is mentioned below:

Readability and Clarity**:** The switch statement provides a concise and structured way to handle multiple possible values of a single variable. This results in more readable code, making it easier for other programmers (including yourself) to understand the logic and intention behind the code.

Efficiency: In some cases, a switch statement can be more efficient than using a series of if-else statements. Many compilers and interpreters can optimize switch statements, leading to faster execution times compared to equivalent if-else structures.

Default Case: The switch statement includes a default case that can be used to handle situations where none of the specified cases match. This is a powerful tool for handling unexpected or exceptional cases.

A switch-case ladder and an if-else ladder are two different control structures used in programming to make decisions based on different conditions. A switch-case ladder is particularly useful when you have a single variable to compare against multiple constant values, leading to cleaner and more efficient code. On the other hand, an if-else ladder is more versatile and can handle more complex conditions and comparisons, making it suitable for scenarios where you need greater flexibility.

if (expression1) { statement1;

}else if(expression2){ statement2;

}else if(expression3){ statement3

... ...

}else{ statement N; }

switch (expression){

case value1: statement1;

break;

case value2: statement2;

break;

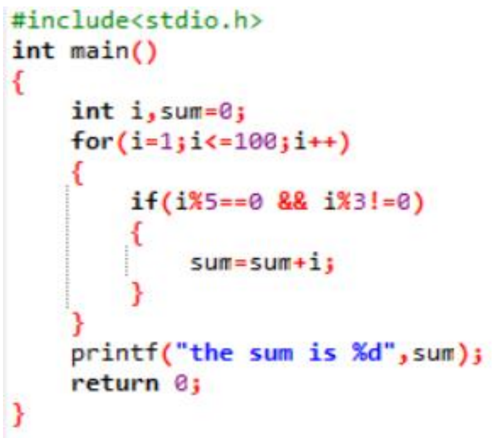
...

case value N: statement N;

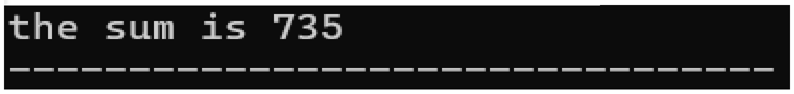
break;

default: default actions;

}



```c
#include<stdio.h>
int main()
{
    int i,sum=0;
    for(i=1;i<=100;i++)
    {
        if(i%5==0 && i%3!=0)
        {
            sum=sum+i;
        }
    }
    printf("the sum is %d",sum);
    return 0;
}
```



```
the sum is 735
--------------------------------
```

5. How is function declared? Why is function prototype necessary? Write recursive function segment that returns the sum of numbers from 1 to n given by the user.

→ A function is a group of statements that together perform a task. Every computer program has at least one function, which is main (), and all the most trivial programs can define additional functions.

Syntax: return_type function_name (parameters_list);

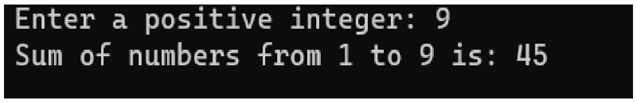
The general form of function definition in C programming language is:

Syntax: return_type function_name (type1 arg1, type2 arg2, type3 arg3, ..., type n arg n)

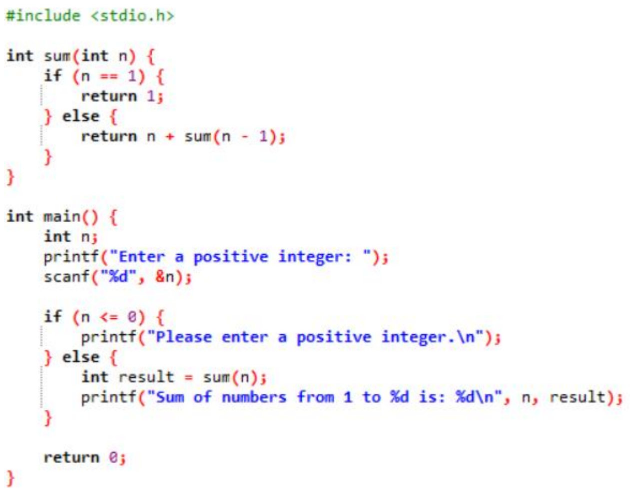
{   /* body of function */

}

Function Prototypes help in organizing your code by separating the interface from the implementation. This makes your code more modular and easier to understand, as other programmers can quickly grasp how to use the function without diving into the implementation details.



```
Enter a positive integer: 9
Sum of numbers from 1 to 9 is: 45
```



```c
#include <stdio.h>

int sum(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n + sum(n - 1);
    }
}

int main() {
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Please enter a positive integer.\n");
    } else {
        int result = sum(n);
        printf("Sum of numbers from 1 to %d is: %d\n", n, result);
    }

    return 0;
}
```

6. How can you pass one dimensional array to function and what does name of an array in function call represents? Write a program to find the largest and smallest element of an array using a single function and display the result in calling function.

→ Array is a derived data type in 'C' that store same/similar type of elements. That is, it is a fixed size sequenced collection of elements of same data type.

Passing one dimensional array to function

We can pass a one dimensional array to a function by passing the base address(address of first element of an array) of the array. We can either pas the name of the array(which is equivalent to base address) or pass the address of first element of array like &array[0]. Similarly, we can pass multi dimensional array also as formal parameters.

Different ways of declaring function which takes an array as input.

- Function argument as a pointer to the data type of array.

```
int testFunction(int *array){
/* Function body */
}
```

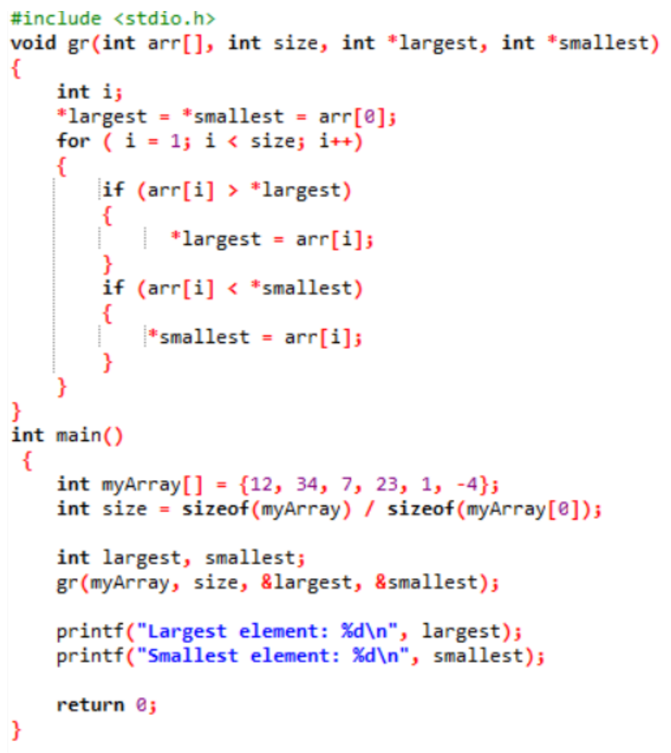
- By specifying size of an array in function parameters.

```
int testFunction(int array[10]){
/* Function body */
}
```

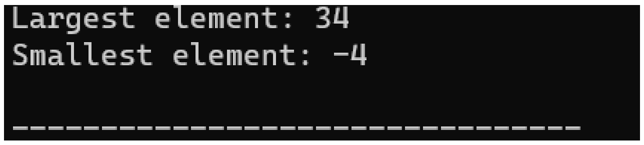
- By passing unsized array in function parameters.

```
int testFunction(int array[]){
/* Function body */
}
```

The name of the array in the function call represents a way to access the elements of the array and manipulate them directly, as if you were working with a pointer.



```
#include <stdio.h>
void gr(int arr[], int size, int *largest, int *smallest)
{
    int i;
    *largest = *smallest = arr[0];
    for ( i = 1; i < size; i++)
    {
        if (arr[i] > *largest)
        {
            *largest = arr[i];
        }
        if (arr[i] < *smallest)
        {
            *smallest = arr[i];
        }
    }
}
int main()
 {
    int myArray[] = {12, 34, 7, 23, 1, -4};
    int size = sizeof(myArray) / sizeof(myArray[0]);

    int largest, smallest;
    gr(myArray, size, &largest, &smallest);

    printf("Largest element: %d\n", largest);
    printf("Smallest element: %d\n", smallest);

    return 0;
}
```



```
Largest element: 34
Smallest element: -4

--------------------------------
```

7. Explain how a structure can be defined and structure variables can be declared in C. Write a program that reads name, roll numbers, program and marks obtained in five subjects by students until the user enters 'e' and display the student detail and total marks obtained by each student.

→ Structure is a collection of variables (may be of different types) under a single name. A structure provides a means of grouping variables under a single name for easier handling and identification. It can be defined by declaring it using below syntax:

Syntax:

```
struct structure_name
{
data_type member 1;
data_type member 2;
.................................
.................................
data_type member n };
```

We can use two different types for structure variable declaration.

a) Using tag name, we can declare variables anywhere in the program.

Syntax: struct structure_name var_name1,var_name2,. ,var_namen;

example: from above

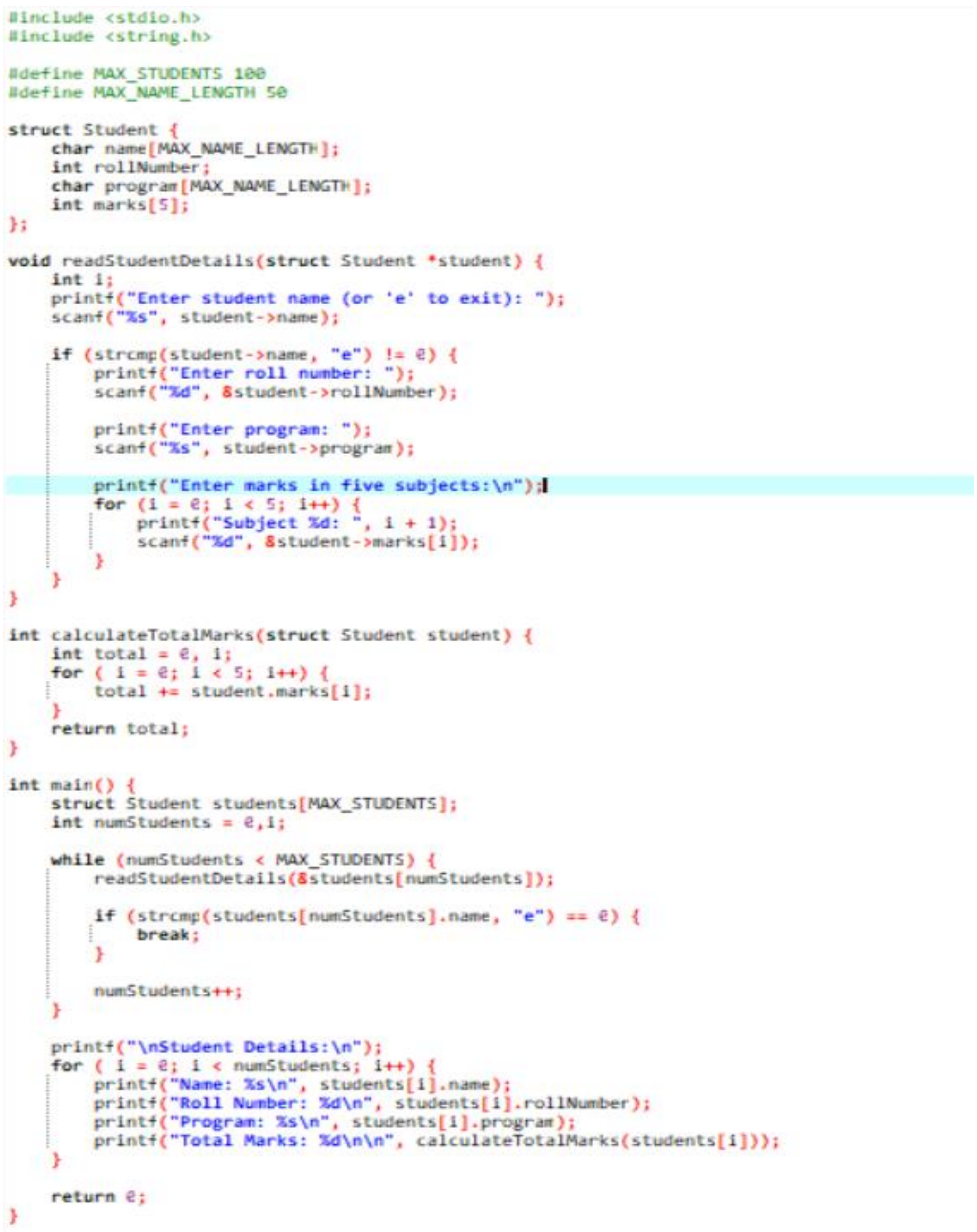
```
struct student
{
char name[20];
int rollno;
float marks;
} ;
int main()
{
```

```
struct student s1,s2,s3;

}
```

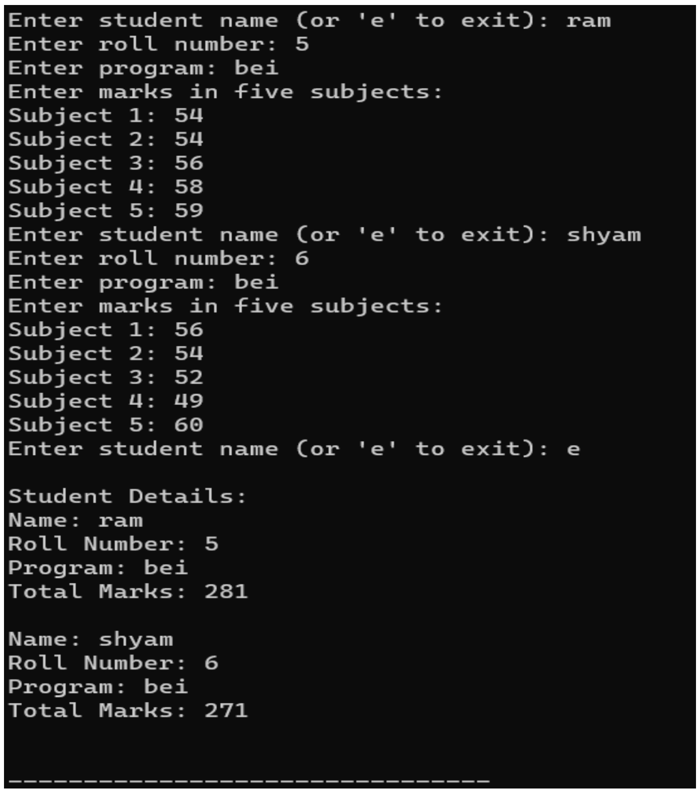
b) For global declaration

```
Syntax: struct structure_name

{

data_type member1;

.................................

data_type membern;

}var_name1,. ,var_namen;

example: struct student

{

int rollno;

char gender;

float marks;

}s1;
```

Here s1,s2,s3 are variables of type struct student. Each of these variables has 3 members as specified by template.



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

#define MAX_STUDENTS 100
#define MAX_NAME_LENGTH 50

struct Student {
    char name[MAX_NAME_LENGTH];
    int rollNumber;
    char program[MAX_NAME_LENGTH];
    int marks[5];
};

void readStudentDetails(struct Student *student) {
    int i;
    printf("Enter student name (or 'e' to exit): ");
    scanf("%s", student->name);

    if (strcmp(student->name, "e") != 0) {
        printf("Enter roll number: ");
        scanf("%d", &student->rollNumber);

        printf("Enter program: ");
        scanf("%s", student->program);

        printf("Enter marks in five subjects:\n");
        for (i = 0; i < 5; i++) {
            printf("Subject %d: ", i + 1);
            scanf("%d", &student->marks[i]);
        }
    }
}

int calculateTotalMarks(struct Student student) {
    int total = 0, i;
    for ( i = 0; i < 5; i++) {
        total += student.marks[i];
    }
    return total;
}

int main() {
    struct Student students[MAX_STUDENTS];
    int numStudents = 0,i;

    while (numStudents < MAX_STUDENTS) {
        readStudentDetails(&students[numStudents]);

        if (strcmp(students[numStudents].name, "e") == 0) {
            break;
        }

        numStudents++;
    }

    printf("\nStudent Details:\n");
    for ( i = 0; i < numStudents; i++) {
        printf("Name: %s\n", students[i].name);
        printf("Roll Number: %d\n", students[i].rollNumber);
        printf("Program: %s\n", students[i].program);
        printf("Total Marks: %d\n\n", calculateTotalMarks(students[i]));
    }

    return 0;
}
```



```
Enter student name (or 'e' to exit): ram
Enter roll number: 5
Enter program: bei
Enter marks in five subjects:
Subject 1: 54
Subject 2: 54
Subject 3: 56
Subject 4: 58
Subject 5: 59
Enter student name (or 'e' to exit): shyam
Enter roll number: 6
Enter program: bei
Enter marks in five subjects:
Subject 1: 56
Subject 2: 54
Subject 3: 52
Subject 4: 49
Subject 5: 60
Enter student name (or 'e' to exit): e

Student Details:
Name: ram
Roll Number: 5
Program: bei
Total Marks: 281

Name: shyam
Roll Number: 6
Program: bei
Total Marks: 271


--------------------------------
```

8. What is pointer? Discuss its relationship with an array. Write a function program that behaves strcpy() function using pointer as argument.

$\longrightarrow$ A Pointer in C language is a variable which holds the address of another variable of same data type. Pointers are used to access memory and manipulate the address. Pointers are one of the most distinct and exciting features of the C language. It provides power and flexibility to the language.

When an array is declared, the compiler allocates enough memory to contain all the elements of the array. Base address i.e. address of the first element of the array is also allocated by the compiler. Suppose we declare an array arr, int arr[5] = { 1, 2, 3, 4, 5 };

Here variable arr will give the base address, which is a constant pointer pointing to the first

element of the array, arr[0]. Hence arr contains the address of arr[0]. In short, arr has two purposes - it is the name of the array and it acts as a pointer pointing towards the first element in the array. arr is equal to &arr[0] by default

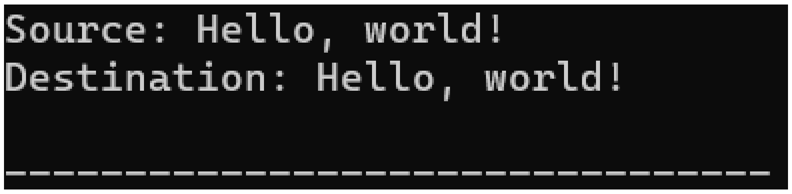
We can also declare a pointer of type int to point to the array arr.

int *p;

p = arr;

Now we can access every element of the array arr using p++ to move from one element to another.

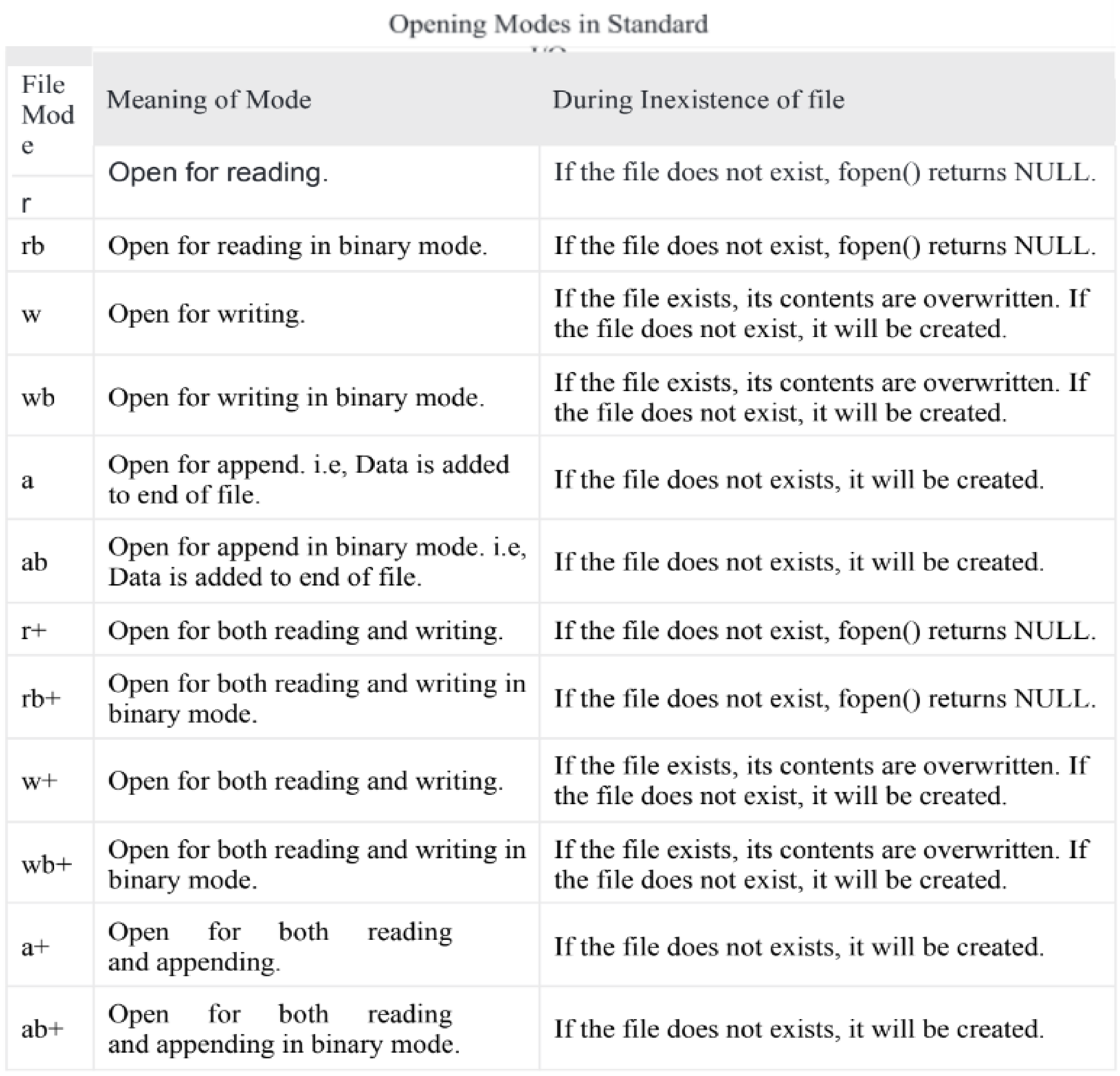
```c
#include <stdio.h>

void customStrcpy(char *d, const char *src) {
    while (*src != '\0') {
        *d = *src;
        d++;
        src++;
    }
    *d = '\0';
}

int main() {
    char source[] = "Hello, world!";
    char destination[20];

    customStrcpy(destination, source);

    printf("Source: %s\n", source);
    printf("Destination: %s\n", destination);

    return 0;
}
```



```
Source: Hello, world!
Destination: Hello, world!


--------------------------------
```

9. Explain different modes in opening file. Write a program to read a string, write it into a file and display the content of a file into a screen.

→ A collection of data which is stored on a secondary device like a hard disk is called a file. A file is generally used as real-life applications that contain a large amount of data.



Opening Modes in Standard

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

10. Explain different data types available in FORTRAN. Write a program in FORTRAN to check whether a number given by user is palindrome or NOT.

→FORTRAN is an acronym for Formula Translation and is one of the oldest computer languages which was originally used for mathematical and engineering computations. The most important feature of FORTRAN is its rich library for in functions.

It has different data types which are mentioned below:

a. Integer: Repsents both positive and negative integral number but doesn't contain fractional number. It occupies 4 bytes of memory in RAM.

b. Real: Represents both positive and negative number including fractional number and occupies 4 bytes of memory in RAM.
c. Double Precision: Same as real but using double the storage and more precision.
d. Complex: Represents an ordered pair of Real data; real and imaginary components.

2076 ASHWIN

A) Define language processor. Explain the main types of language processor in detail.

A language processor is a software or hardware component that interprets, analyzes, and manipulates human language in various ways. It plays a crucial role in enabling computers to understand and generate natural language text, which is essential for a wide range of applications, including communication, data analysis, artificial intelligence, and more.

There are three main types of language processors, each serving distinct purposes:

- Compiler: A compiler is a language processor that translates high-level programming code written in a human-readable language (such as C++, Java, or Python) into machine code or an intermediate code that can be executed by a computer. The compilation process includes several stages, such as lexical analysis, parsing, semantic analysis, optimization, and code generation. The output of a compiler is typically an executable file or an intermediate representation that can be executed efficiently.

Advantages:

- High performance: The generated code is optimized for execution, resulting in faster program execution.
- Error detection: The compiler performs extensive checks on the code, catching many errors before runtime.
- Platform independence: Compiled code can often run on different platforms without modification.

Disadvantages:

- Longer development cycle: Compilation can take some time, especially for large programs.
- Lack of runtime flexibility: Changes to the code usually require recompilation.

- Interpreter: An interpreter is a language processor that directly executes source code written in a high-level language, line by line. It translates and executes each line in real-time, without generating an intermediate code or executable file. Interpreters are commonly used in scripting languages like JavaScript, Python, or Ruby, where rapid development and ease of use are important.

Advantages:

- Quick development: Changes to the code can be tested immediately, making development faster.
- Platform independence: As long as the interpreter is available for a platform, the code can be executed without modification.

Disadvantages:

- Lower performance: Interpreted code is generally slower than compiled code due to the lack of optimization.
- Reduced error checking: Some errors may only be discovered at runtime.
- Assembler: An assembler is a language processor that converts low-level assembly language code into machine code, which is specific to a particular computer architecture. Assembly language is a human-readable representation of machine code, using mnemonic instructions that correspond to specific machine operations.

Advantages:

- Fine control: Assembly language allows programmers to have precise control over the hardware.
- Efficiency: Assembly code can be highly optimized for a specific architecture.

Disadvantages:

- Complexity: Writing assembly language code is more complex and error-prone than high-level languages.
- Platform dependence: Assembly code is specific to a particular architecture, limiting portability.

These three types of language processors serve different purposes in the software development and execution process, catering to the specific needs and priorities of programmers and the applications they're building.

B) List the basic step of problem-solving using computer. Write an algorithm and draw a flowchart to find the sum of N natural numbers.

The basic steps of problem-solving using a computer typically include:

- Understanding the Problem: Clearly define the problem, its input, and the desired output.
- Algorithm Design: Create a step-by-step plan (algorithm) to solve the problem.
- Implementation: Write code based on the algorithm.
- Testing: Verify the code with different inputs, checking if it produces the correct output.
- Optimization: Refine the code for better performance or readability if needed.
- Documentation: Comment the code to explain its logic and usage.

Algorithm to find the sum of N natural numbers:

```
1. Start
2. Initialize a variable 'sum' to 0.
3. Read the value of N.
4. Loop from 1 to N:
   a. Add the current value of the loop variable to 'sum'.
5. Print the value of 'sum'.
6. End
```

Flowchart to find the sum of N natural numbers:

A) Define formatted and Unformatted I/O functions. Write the operations of following functions.

- getch()
- getche()
- getchar().

Formatted and unformatted I/O functions are used in programming to perform input and output operations on the console or other input/output devices. These functions help in reading and writing data to and from the user or the program.

Formatted I/O functions: Formatted I/O functions are used to read and write data in a specific format, such as integers, floating-point numbers, strings, etc. They allow you to control how the data is presented and processed during input and output operations. Some examples of formatted I/O functions in C and C++ include printf, scanf, cout, and cin.

Unformatted I/O functions: Unformatted I/O functions are used to read and write data without any specific formatting considerations. They often deal with characters and are used when precise control over the data's appearance is not necessary. Some examples of unformatted I/O functions in C and C++ include getch(), getche(), and getchar().

Now, let's explore the operations of the specified functions:

I) getch():

- Operation: getch() is an unformatted input function in C that reads a single character from the console without echoing it (not displaying it on the screen). It is commonly used for input without requiring the user to press the Enter key.

Example.

```
#include <conio.h> // Include the necessary header

#include <stdio.h>


int main() {

   char ch;

   printf("Press any key: ");

   ch = getch(); // Read a character without echoing
```

```
    printf("\nYou pressed: %c\n", ch);
    return 0;
}
```

II) getche():

- Operation: getche() is an unformatted input function in C that reads a single character from the console and echoes it (displays it on the screen). It is commonly used for interactive input.
- Example usage:

```
#include <conio.h> // Include the necessary header
#include <stdio.h>

int main() {
    char ch;
    printf("Press any key: ");
    ch = getche(); // Read a character with echoing
    printf("\nYou pressed: %c\n", ch);
    return 0;
}
```

III) getchar():

- Operation: getchar() is an unformatted input function in C that reads a single character from the console and returns it. It reads a character and waits for the Enter key to be pressed, which indicates the end of input.
- Example usage:

```
#include <stdio.h>


int main() {
    char ch;
    printf("Press Enter after typing a character: ");
    ch = getchar(); // Read a character and wait for Enter
    printf("\nYou typed: %c\n", ch);
    return 0;
}
```

2.B) What is an expression? Write a program to display the smallest number between three integers entered from user using conditional operator.

An expression in programming is a combination of values, variables, operators, and function calls that can be evaluated to produce a result. Expressions can represent calculations, comparisons, or other operations that yield a value. They are fundamental building blocks in programming, used in various contexts, such as assignments, conditions, loops, and function arguments.

Now, let's write a C program to display the smallest number among three integers entered by the user using the conditional (ternary) operator:

```
#include <stdio.h>


int main() {
    int num1, num2, num3, smallest;
    // Input three integers from the user

    printf("Enter three integers: ");

    scanf("%d %d %d", &num1, &num2, &num3);


    // Use the conditional operator to find the smallest number

    smallest = (num1 < num2) ? ((num1 < num3) ? num1 : num3) : ((num2 < num3) ? num2 :
num3);


    // Display the smallest number

    printf("The smallest number among %d, %d, and %d is %d.\n", num1, num2, num3, smallest);


    return 0;

}
```

• A) Differentiate between while and do-while looping statement with example.

Feature	while Loop	do-while Loop
Loop Condition	The loop condition is checked before executing the loop body.	The loop body is executed at least once before checking the loop condition.
Entry Control	If the loop condition is false initially, the loop body is never executed.	The loop body is executed at least once, regardless of the initial condition.
Example	c while (condition) { statement(s); } c	c do { statement(s); } while (condition); c

Here are examples illustrating the difference between the while loop and the do-while loop:

Example using while loop:

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

```
int main() {
    int count = 0;

    // Using while loop
    printf("Using while loop:\n");
    while (count < 5) {
        printf("Count: %d\n", count);
        count++;
    }

    return 0;
}
```

Example using do-while loop:

```
#include <stdio.h>

int main() {
    int count = 0;

    // Using do-while loop
    printf("Using do-while loop:\n");
    do {
        printf("Count: %d\n", count);
        count++;
    } while (count < 5);

    return 0;
}
```

- B) Write a program to evaluate the following series up to n terms. Prompt the user to input the value of n and x. f(x)= 1-x^2/2! + x^4/4! – x^6/6! + x^8/8-------- upto n terms.

```
#include <stdio.h>

#include <math.h>


// Function to calculate the factorial of a number

int factorial(int num) {

    if (num <= 1) {

        return 1;

    } else {

        return num * factorial(num - 1);

    }

}


int main() {

    int n;

    double x, result = 1.0;


    // Input the value of n and x from the user

    printf("Enter the value of n: ");

    scanf("%d", &n);

    printf("Enter the value of x: ");

    scanf("%lf", &x);
    // Calculate the series
    for (int i = 1, j = 2; i <= n; i++, j += 2) {
        if (i % 2 == 1) {
            result -= pow(x, j) / (double)factorial(j);
        } else {
            result += pow(x, j) / (double)factorial(j);
        }
    }

    // Display the result
    printf("The value of the series is: %lf\n", result);

    return 0;
}
```

• A) What are the different types of function available In C? What do you mean by pass by reference and pass by value.

In C, there are two main types of functions based on the way arguments are passed: pass by value and pass by reference. Additionally, functions can be classified based on their return type and whether they perform specific tasks or computations. Here's an overview:

• Pass by Value: In pass by value, a copy of the actual arguments (values) is passed to the function. Any changes made to the formal parameters within the function do not affect the original values of the actual arguments.

• Pass by Reference: In pass by reference, a reference (address) to the actual arguments is passed to the function. Changes made to the formal parameters within the function affect the original values of the actual arguments.

• Based on Return Type: Functions can have different return types:

- Functions returning a value: These functions compute a value and return it to the caller using the return statement.
- Functions returning void: These functions do not return a value. They perform a specific task or operation and do not produce a result.
- Based on Task or Computation: Functions can be classified based on what they do:
- Procedural functions: These functions perform a specific task, such as printing a message, calculating a sum, etc.
- Mathematical functions: These functions perform mathematical operations, such as computing the square root, absolute value, etc.
- User-defined functions: These functions are created by the programmer to modularize code and improve maintainability.

Pass by Value vs Pass by Reference:

- Pass by Value: In pass by value, the function receives copies of the arguments, and modifications to the copies do not affect the original values.
- Pass by Reference: In pass by reference, the function receives references (addresses) to the original values, allowing modifications to directly affect the originals.

4. B) What are the similarities and differences between iteration and recursive function?

Iteration and recursion are both techniques used in programming to solve problems that involve repetitive tasks. They have similarities and differences that are worth understanding:

Similarities:

- Repetition: Both iteration and recursion involve repeating a certain operation or function call multiple times until a specific condition is met or a base case is reached.
- Control Flow: Both techniques control the flow of execution within a program, enabling the handling of repetitive tasks efficiently.
- Function Calls: Both techniques involve the use of function calls. In iterative solutions, functions are called explicitly in a loop. In recursive solutions, a function calls itself (recursion) to solve smaller instances of the same problem.

Differences:

- Approach:
- Iteration: Iterative solutions use loops (e.g., for, while) to repeatedly execute a block of code. The control structure explicitly manages the loop condition, initialization, and update.

- Recursion: Recursive solutions use the concept of a function calling itself with modified parameters. The control flow relies on the base case(s) that determines when the recursion stops.
- Readability:
- Iteration: Iterative solutions are generally more straightforward and easier to understand for simple repetitive tasks.
- Recursion: Recursive solutions can be more elegant for problems that have a natural recursive structure, but they can be harder to understand for some people, especially for deeply nested or complex recursive calls.
- Space and Time Complexity:
- Iteration: Iterative solutions often use less memory (stack space) compared to deep recursion, making them more suitable for problems with a large number of iterations.
- Recursion: Recursive solutions can consume more memory due to the call stack. However, some problems can be efficiently solved using recursion, and modern compilers may optimize tail recursion to reduce stack usage.
- Termination:
- Iteration: The termination condition in an iterative solution is typically explicitly specified in the loop control structure.
- Recursion: The termination condition in a recursive solution is defined by the base case, which defines when the recursion stops.

- How do you initialize a 2D array? Explain with example. Write a program to input two matrices of size m*n and p*q respectively. Pass the matrices to a function to calculate the product matrix. Display the product matrix in main() function.

Initializing a 2D array in C involves specifying the dimensions (number of rows and columns) and assigning values to individual elements. Here's an example of how to initialize a 2D array:

```
#include <stdio.h>


int main() {

   int m = 3; // Number of rows

   int n = 4; // Number of columns


   // Initialize a 2D array with predefined values

   int matrix[m][n] = {

      {1, 2, 3, 4},

      {5, 6, 7, 8},

      {9, 10, 11, 12}

   };


   // Access and print the elements of the 2D array

   for (int i = 0; i < m; i++) {

      for (int j = 0; j < n; j++) {

         printf("%d ", matrix[i][j]);

      }

      printf("\n");

   }
```

```
    return 0;
}
```

In this example, a 2D array named matrix is declared and initialized with predefined values. The array has 3 rows and 4 columns. The nested for loops are used to access and print the elements of the 2D array.

Now, let's write a program that takes input for two matrices of sizes m*n and p*q and calculates their product matrix using a separate function. The product matrix is displayed in the main() function:

```
#include <stdio.h>

// Function to calculate the product of two matrices
void multiplyMatrices(int mat1[][10], int mat2[][10], int res[][10], int m, int n, int q) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < q; j++) {
            res[i][j] = 0;
            for (int k = 0; k < n; k++) {
                res[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
}

int main() {
    int m, n, p, q;

    printf("Enter the number of rows and columns for matrix 1 (m n): ");
    scanf("%d %d", &m, &n);
    printf("Enter the number of rows and columns for matrix 2 (p q): ");
    scanf("%d %d", &p, &q);

    if (n != p) {
        printf("Matrices cannot be multiplied. The number of columns in matrix 1 must be equal to
the number of rows in matrix 2.\n");
        return 1;
    }

    int mat1[10][10], mat2[10][10], result[10][10];

    // Input elements for matrix 1
    printf("Enter elements for matrix 1:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &mat1[i][j]);
        }
    }

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

    // Call the function to calculate the product matrix
    multiplyMatrices(mat1, mat2, result, m, n, q);


    // Display the product matrix
    printf("Product matrix:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < q; j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }


    return 0;
}
```

- A). What is structure? When do we use structure?

In programming, a structure is a composite data type that groups together variables of different data types under a single name. A structure allows you to create a custom data type that represents a collection of related information, making it easier to organize and manipulate data in a more meaningful way. Structures are a fundamental feature in many programming languages, including C, C++, and other languages derived from C.

Here's a brief overview of structures:

- Definition: A structure is defined using the struct keyword, followed by a name for the structure, and a set of member variables enclosed in curly braces. Each member variable has a data type, and they collectively define the structure's layout.
- Usage: Structures are used to represent objects or entities in your program. When you need to group different types of data that belong together, you can use a structure. This is particularly useful when dealing with complex data that has multiple attributes.
- Accessing Members: You can access the members of a structure using the dot notation (.). For example, if you have a structure named Person with members name and age, you can access them as person.name and person.age.
- Memory Allocation: Each variable of a structure type occupies memory for all its members, and the layout of the structure in memory is determined by the order in which the members are defined.

When to use structures:

- Organizing Related Data: Use structures when you want to group different variables that are related and collectively represent a concept, object, or entity. For example, you might use a Point structure to represent coordinates (x, y).
- Creating Custom Data Types: Structures allow you to create your own custom data types that are more meaningful than simple built-in types. This can improve the clarity and maintainability of your code.
- Complex Data Representation: When dealing with complex data that has multiple attributes, such as representing a student with name, age, and grades, structures provide a convenient way to organize this data.
- Passing Data as a Unit: Structures can be useful when you need to pass multiple pieces of data as a single unit to functions, making your code more organized and reducing the number of function arguments.

In summary, structures are used to group related data, create custom data types, and improve the organization of your code, making it more modular and readable.

6. B). Write a program using pointer to swap the value of two variable where the swapping operation is performed in separate function.

```
#include <stdio.h>


// Function to swap the values of two integers using pointers
void swapValues(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}


int main() {
    int num1, num2;


    // Input values
    printf("Enter the value of num1: ");
    scanf("%d", &num1);
    printf("Enter the value of num2: ");
    scanf("%d", &num2);


    // Display values before swapping
    printf("Before swapping: num1 = %d, num2 = %d\n", num1, num2);


    // Call the function to swap the values
    swapValues(&num1, &num2);


    // Display values after swapping
    printf("After swapping: num1 = %d, num2 = %d\n", num1, num2);
    return 0;
}
```

- A) Why are fgets(), fputs(), fgetc() and fputc() used?

fgets(), fputs(), fgetc(), and fputc() are functions in C used for file input and output operations. They provide a way to read and write data from/to files in a character-based manner. Here's an overview of each function:

- fgets():
- Purpose: fgets() (file get string) is used to read a line (a sequence of characters terminated by a newline character) from a file.
- Syntax: char *fgets(char *str, int n, FILE *stream);
- Parameters:
- str: A pointer to the character array where the read data will be stored.
- n: The maximum number of characters to read (including the newline character and null-terminator).
- stream: A pointer to the FILE structure representing the input stream (file).
- Returns: The function returns the pointer to the input buffer (str) on success, and NULL on failure (end of file or error).
- fputs():
- Purpose: fputs() (file put string) is used to write a string to a file.
- Syntax: int fputs(const char *str, FILE *stream);
- Parameters:
- str: A pointer to the null-terminated string to be written to the file.
- stream: A pointer to the FILE structure representing the output stream (file).
- Returns: It returns a non-negative value on success and EOF on failure.
- fgetc():
- Purpose: fgetc() (file get character) is used to read a single character from a file.
- Syntax: int fgetc(FILE *stream);

- Parameters:
- stream: A pointer to the FILE structure representing the input stream (file).
- Returns: The function returns the character read as an int, or EOF (end of file) if there's an error or the end of the file is reached.
- fputc():
- Purpose: fputc() (file put character) is used to write a single character to a file.
- Syntax: int fputc(int c, FILE *stream);
- Parameters:
- c: The character to be written, given as an integer.
- stream: A pointer to the FILE structure representing the output stream (file).
- Returns: It returns the character written as an unsigned char, or EOF on failure.

These functions provide a convenient way to perform basic file input and output operations in C. They are particularly useful when dealing with text-based files where you need to read or write character-based data.

- B) Write a program to display the record in sorted order, sorting is performed in ascending order with respect to name using data files concept.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


// Define a structure to represent a record

struct Record {

   char name[100];

   int age;

   // Add more fields as needed

};
```

```
// Function to compare records based on the name field (used by qsort)
int compareRecords(const void *a, const void *b) {
  const struct Record *recordA = (const struct Record *)a;
  const struct Record *recordB = (const struct Record *)b;
  return strcmp(recordA->name, recordB->name);
}

int main() {
  FILE *inputFile, *outputFile;
  struct Record records[100]; // Assuming a maximum of 100 records

  // Open the input file (modify the file path accordingly)
  inputFile = fopen("input.txt", "r");
  if (inputFile == NULL) {
    perror("Error opening input file");
    return 1;
  }

  // Read records from the input file
  int recordCount = 0;
  while (fscanf(inputFile, "%s %d", records[recordCount].name, &records[recordCount].age) !=
EOF) {
    recordCount++;
  }
  fclose(inputFile);

  // Sort the records based on the name field in ascending order
  qsort(records, recordCount, sizeof(struct Record), compareRecords);
```

```
    // Open the output file for writing the sorted records
    outputFile = fopen("sorted_output.txt", "w");
    if (outputFile == NULL) {
        perror("Error opening output file");
        return 1;
    }

    // Write the sorted records to the output file
    for (int i = 0; i < recordCount; i++) {
        fprintf(outputFile, "%s %d\n", records[i].name, records[i].age);
    }
    fclose(outputFile);

    printf("Records sorted and written to the output file.\n");

    return 0;
}
```

• A) Compare logical IF and arithmetic IF statement in fortran with example.

Feature	Logical IF	Arithmetic IF
Syntax	fortran IF (logical_expression) THEN	fortran IF (arithmetic_expression) n, label1, label2
Condition type	Logical expression (Boolean condition)	Arithmetic expression (numeric comparison)
Condition result	True or False	Determined by the numeric comparison (True or False)
Jump to labels	Jump based on whether the condition is True or False	Jump based on the outcome of a specified comparison
Example	fortran IF (num > 0) THEN	fortran IF (num .GT. 0) 1, 2

! Code for num > 0 ! Code for num > 0

ELSE ELSE

! Code for num <= 0 ! Code for num <= 0

END IF 1 CONTINUE

2 CONTINUE

Here's an example illustrating both the logical IF and arithmetic IF in Fortran:

Logical IF example:

```
PROGRAM LogicalIFExample
  INTEGER :: num

  ! Input a number
  WRITE(*,*) "Enter a number: "
  READ(*,*) num

  ! Check if the number is positive
  IF (num > 0) THEN
    WRITE(*,*) "The number is positive."
  ELSE
    WRITE(*,*) "The number is non-positive."
  END IF

END PROGRAM LogicalIFExample
```

Arithmetic IF example:

```
PROGRAM ArithmeticIFExample
  INTEGER :: num

  ! Input a number
  WRITE(*,*) "Enter a number: "
  READ(*,*) num
  ! Check if the number is positive using arithmetic IF
  IF (num .GT. 0) 1, 2
1 WRITE(*,*) "The number is positive."
  GO TO 3
2 WRITE(*,*) "The number is non-positive."
3 CONTINUE
END PROGRAM ArithmeticIFExample
```

• B) Write a fortran program to sort 10 integers from user and display the second largest integer.

```
F PROGRAM SecondLargest
  INTEGER :: i, num(10)
  INTEGER :: largest, second_largest
! Input 10 integers
  WRITE(*,*) "Enter 10 integers:"
  DO i = 1, 10
    READ(*,*) num(i)
  END DO
  ! Initialize the largest and second largest values
  largest = num(1)
  second_largest = num(2)
! Find the largest and second largest
  DO i = 2, 10
    IF (num(i) > largest) THEN
      second_largest = largest
      largest = num(i)
    ELSE IF (num(i) > second_largest .AND. num(i) < largest) THEN
      second_largest = num(i)
    END IF
  END DO
  ! Display the second largest integer
  WRITE(*,*) "The second largest integer is:", second_largest
END PROGRAM SecondLargest
```

DONE BY:

PRABHAT CHAULAGAIN

NIBHRIT SAPKOTA

ASHUTOSH KUMAR SAH

AAYUSH PANDEY