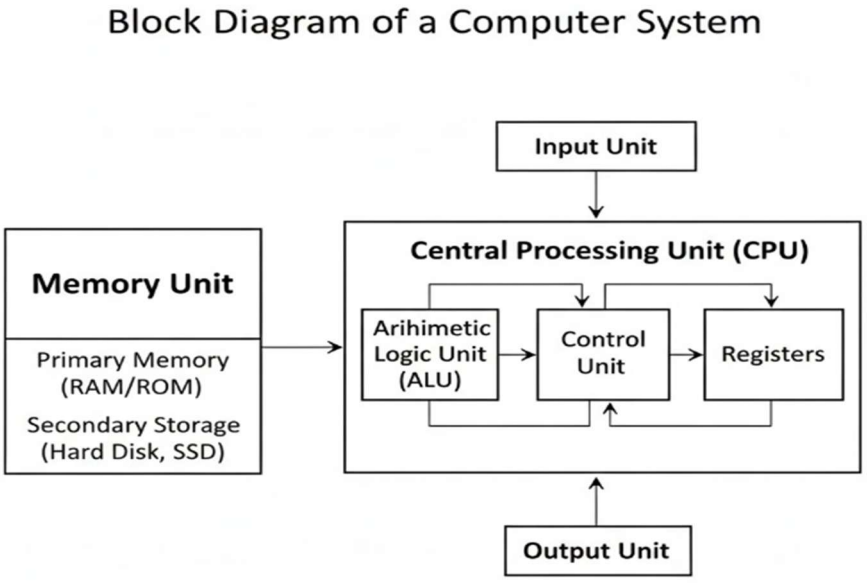
Subject Name & Code:

PROGRAMMING FOR PROBLEM SOLVING- 3110003

Assignment – 1

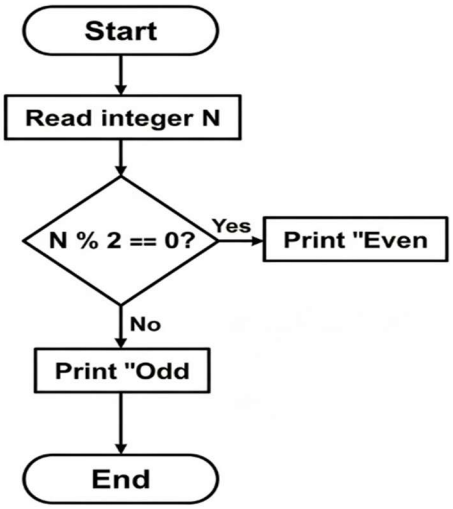
1. **Draw and explain the block diagram of the Computer System.**

   A computer system is an integrated set of hardware and software components designed to process data and produce meaningful output. The block diagram illustrates the major functional units and their interconnections. Below is a textual description that can be used to generate a diagram.



Explanation:
The **Input Unit** (keyboard, mouse, scanner) accepts raw data and instructions, converting them into a binary format understandable by the computer. The **Memory Unit** stores data and instructions before and after processing. Primary memory (RAM) is volatile and fast, used for active tasks, while secondary storage provides permanent, non-volatile storage. The **Central Processing Unit (CPU)** is the brain of the computer. The **Arithmetic Logic Unit (ALU)** performs mathematical and logical operations. The **Control Unit (CU)** manages and coordinates all operations by fetching, decoding, and executing instructions. **Registers** are small, high-speed memory locations within the CPU used to hold temporary data and addresses.
The **Output Unit** (monitor, printer) presents processed information to the user. Data flows between these units via buses (address, data, control), under the supervision of the control unit, ensuring synchronized operation.

2. Draw a flowchart to find whether a given number is odd or even.



Explanation:
The flowchart begins by reading an integer input N. The condition N % 2 == 0 checks divisibility by 2. If true, the number is even; if false, it is odd. The result is printed, and the process ends.

3. Write an algorithm to find the area of a circle.

Algorithm:

1. Start.
2. Define a constant PI with value 3.14159.
3. Read the radius r of the circle.
4. Calculate area using the formula: area = PI * r * r.
5. Display the calculated area.
6. Stop.

4. List out types of software with Examples.

Software is classified into two broad categories:

- **System Software:** Serves as an interface between hardware and user applications. Examples: Operating Systems (Windows, Linux), Device Drivers (printer driver), Utility Programs (disk cleanup tools), Language Translators (compilers, assemblers).
- **Application Software:** Designed to perform specific tasks for end-users. Examples: Word Processors (Microsoft Word), Web Browsers (Chrome), Database Management Systems (MySQL), Engineering Simulation Software (ANSYS).

5. Difference between Assembler, Compiler, and Interpreter.

- **Assembler:** Converts assembly language code (low-level, mnemonic-based) into machine code. Translation is one-to-one; each assembly instruction corresponds to one machine instruction. It is specific to a particular processor architecture.
- **Compiler:** Translates the entire high-level source code (e.g., C, C++) into machine code in one go, generating an executable file. Errors are reported after full scanning. The compiled program runs independently and is generally faster.
- **Interpreter:** Translates and executes high-level code line-by-line. No separate executable is created; the source code is needed each time. Errors are reported immediately upon encounter. Slower execution but easier debugging. Examples: Python, JavaScript.

6. Define the following terms:

1. **Operating System:** A system software that manages computer hardware and software resources, providing common services for computer programs. It handles memory management, process scheduling, file management, and user interaction (e.g., Windows, Linux).
2. **Loader:** A part of the operating system responsible for loading executable program files from secondary storage into main memory for execution. It allocates memory space and resolves relative addresses.
3. **Linker:** A utility program that combines one or more object files generated by a compiler with library functions to produce a single executable file. It resolves external references between modules.
4. **Preprocessor:** A program that processes source code before compilation. In C, it handles directives like #include, #define, and macro expansions, producing an expanded source code for the compiler.

ASSIGNMENT 2

1. Describe print statement provided by the C programming language.

In C, the primary function for producing formatted output to the standard output device (usually the console) is printf(), declared in the stdio.h header. It is a versatile function that prints text, variables, and expressions with precise formatting control. The general syntax is:

printf("format string", argument1, argument2, ...);

The **format string** contains plain text to be printed and **format specifiers** (placeholders beginning with %) that define how subsequent arguments are displayed. Common specifiers include %d for integers, %f for floating-point numbers, %c for characters, %s for strings, and %x for hexadecimal output. printf() returns the number of characters printed and allows escape sequences (like \n for newline, \t for tab) for better text layout. Example:

int age = 20;

printf("I am %d years old.\n", age); *// Output: I am 20 years old.*

It is a fundamental output function essential for debugging and user interaction in C programs.

2. Explain format specifiers used in C and explain any three with examples.

Format specifiers in printf() and scanf() define the data type of the variable being read or written. They begin with a % symbol followed by a character that indicates the type. Below are three key specifiers with examples:

- %d **– Integer specifier:** Used for signed decimal integers.
  Example:

int num = 45;

printf("Number: %d", num); *// Output: Number: 45*

- %f **– Floating-point specifier:** Used for decimal notation of float and double types. By default, it shows six digits after the decimal point. Precision can be controlled (e.g., %.2f for

two decimal places).
Example:

```
float pi = 3.14159;
printf("Pi: %.2f", pi); // Output: Pi: 3.14
```

- %x **– Hexadecimal specifier:** Outputs an unsigned integer in hexadecimal form (lowercase letters a–f). For uppercase letters, %X is used.
  Example:

```
int value = 255;
printf("Hex: %x", value); // Output: Hex: ff
```

Other common specifiers include %c (character), %s (string), %p (pointer address), and %u (unsigned integer).

3. Describe getch() **with examples.**

getch() is a non-standard function typically available in the conio.h header, primarily in older DOS-based compilers like Turbo C. It reads a single character from the keyboard **without echoing** it to the screen and **without waiting for the Enter key** (unlike scanf() or getchar()). It is useful for interactive menus, password input, or pausing the console until a key is pressed.

Example 1 – Reading a character without echo:

```
#include <stdio.h>
#include <conio.h>
int main() {
    char ch;
    printf("Press any key: ");
    ch = getch(); // Key press is not displayed
    printf("\nYou pressed: %c\n", ch);
```

```
    return 0;
}
```

Example 2 – Using getch() to pause the console before exit:

```
#include <stdio.h>
#include <conio.h>
int main() {
    printf("Program finished. Press any key to exit.");
    getch(); // Waits for a key press
    return 0;
}
```

Note: getch() is not part of the standard C library and is not portable. In modern compilers (like GCC), alternatives such as getchar() or platform-specific functions are recommended.

4. Consider the following C statements: enum month {JAN=5, FEB, MAR=8, APR, MAY}; printf("%d %d %d", ++FEB, MAR, APR); **What would be the output?**

In an enum, if an explicit value is assigned to an identifier, subsequent identifiers increment by 1 unless explicitly set. Here:

- JAN = 5 (explicit)
- FEB automatically becomes 6
- MAR = 8 (explicit)
- APR automatically becomes 9
- MAY automatically becomes 10

The expression ++FEB attempts to modify an enumeration constant, which is **invalid** because enum constants are read-only integer constants at compile time. This would lead to a compilation error. However, if we ignore the increment and consider only the values:
FEB is 6, MAR is 8, APR is 9.
Thus, if the increment were allowed, ++FEB would change FEB to 7 and evaluate to 7.
But since FEB **is a constant, the code will not compile.** Assuming the increment is removed, the output would be:

printf("%d %d %d", FEB, MAR, APR); *// Output: 6 8 9*

For the given code with ++FEB, the compiler will generate an error.

5. Consider the following C statements: int y = -5; printf("%x", y); **What would be the output?**

The %x format specifier expects an **unsigned integer** and prints it in hexadecimal. However, y is a signed integer (-5). When a negative integer is passed to %x, it interprets the **bit pattern** of y in two's complement form as an unsigned value.

Assuming a 32-bit system:

- Decimal -5 in two's complement binary:
  5 in binary: 0000 0000 0000 0000 0000 0000 0000 0101
  Invert bits: 1111 1111 1111 1111 1111 1111 1111 1010
  Add 1: 1111 1111 1111 1111 1111 1111 1111 1011
- This hexadecimal representation is 0xFFFFFFFB.

Thus, printf("%x", y); will output:
fffffffb (lowercase hex).
Note: The exact output may vary if int is not 32 bits, but on most modern systems, this is the result.

6. Differentiate between x++ **and** ++x**. What would be the effect on output?**

- x++ **(Post-increment):** Returns the original value of x first, then increments x by 1.
- ++x **(Pre-increment):** Increments x by 1 first, then returns the new value.

Example demonstrating the difference:

int x = 5, y;

y = x++; *// y becomes 5, x becomes 6*

printf("x = %d, y = %d\n", x, y); *// Output: x = 6, y = 5*

x = 5;

y = ++x; *// x becomes 6, y becomes 6*

printf("x = %d, y = %d\n", x, y); *// Output: x = 6, y = 6*

In expressions, this affects the result:

int a = 3;

printf("%d", a++); *// Output: 3 (value used before increment)*

printf("%d", a); *// Output: 4*

int b = 3;

printf("%d", ++b); *// Output: 4 (increment before use)*

Effect on output: Post-increment uses the old value in the current statement; pre-increment uses the new value.

7. For the following C statement, what will be the value of x **in hex?** x = -2 << 2;
The expression -2 << 2 performs a **left shift** on the integer -2.

- -2 in 32-bit two's complement binary:
  2: 0000 0000 0000 0000 0000 0000 0000 0010
  Invert: 1111 1111 1111 1111 1111 1111 1111 1101
  Add 1: 1111 1111 1111 1111 1111 1111 1111 1110 → 0xFFFFFFFE
- Left shifting by 2 bits (<< 2) discards the leftmost bits and fills the right with zeros:
  Binary: 1111 1111 1111 1111 1111 1111 1111 1110 << 2 →
  1111 1111 1111 1111 1111 1111 1111 1000
- This binary corresponds to hexadecimal: 0xFFFFFFF8.
- In decimal, this is -8 (since 0xFFFFFFF8 is the two's complement representation of -8).

Thus, x in hex is 0xFFFFFFF8 (or simply FFFFFFF8 if printed with %x).

8. If integers sum = 25 **and** n = 10**, what would be the results of** avg = sum/n; **and** avg = (float)sum/n; **if** avg **is** float**?**

- **Case 1:** avg = sum/n;
  Both sum and n are integers, so integer division is performed: 25 / 10 = 2 (remainder discarded). The integer result 2 is then implicitly converted to float and stored in avg.
  Result: avg = 2.000000

- **Case 2:** avg = (float)sum/n;
  The explicit cast (float)sum converts sum to floating-point (25.0). Then, floating-point division is performed: 25.0 / 10 = 2.5.
  Result: avg = 2.500000

Example code:

```
#include <stdio.h>
int main() {
    int sum = 25, n = 10;
    float avg;
    avg = sum / n;
    printf("avg = sum/n = %f\n", avg); // Output: 2.000000
    avg = (float)sum / n;
    printf("avg = (float)sum/n = %f\n", avg); // Output: 2.500000
    return 0;
}
```

The key difference is that integer division truncates the fractional part, while casting one operand to float promotes the operation to floating-point division, preserving the decimal result.

Assignment 3

1. Distinguish the data types provided by the C programming language.

C data types are broadly categorized as:

- **Basic/Primitive Data Types:**
    - int: Stores integers (e.g., 5, -10). Size is typically 4 bytes.
    - float: Stores single-precision floating-point numbers (e.g., 3.14). Size is 4 bytes.
    - double: Stores double-precision floating-point numbers (e.g., 3.141592). Size is 8 bytes.
    - char: Stores a single character (e.g., 'A'). Size is 1 byte.
    - void: Represents "no type". Used for functions that return nothing or pointers to generic data.
- **Derived Data Types:** Arrays, pointers, functions, and structures, which are constructed from primitive types.
- **User-Defined Data Types:** enum, struct, union, typedef, allowing custom type definitions.

2. List all operators used in C and explain any three with examples.

C operators include:
Arithmetic (+, -, *, /, %), Relational (==, !=, <, >, <=, >=), Logical (&&, ||, !), Bitwise (&, |, ^, ~, <<, >>), Assignment (=, +=, -=, etc.), Increment/Decrement (++, --), Conditional (? :), Special (sizeof, & (address), * (pointer)).

Explanations:

- **Arithmetic Operator % (Modulus):** Returns the remainder of integer division. Example: 7 % 3 yields 1.
- **Logical Operator && (AND):** Returns true (1) only if both operands are non-zero. Example: (5 > 3) && (2 < 4) returns 1.
- **Increment Operator ++:** Increases the value of a variable by 1. Prefix (++x) increments before use; postfix (x++) increments after use. Example: If x = 5, y = ++x sets y to 6 and x to 6.

3. Describe precedence and associativity of operators with examples.

Precedence determines the order in which operators are evaluated in an expression. Higher precedence operators are evaluated first. **Associativity** defines the direction (left-to-right or right-to-left) when operators have the same precedence.

Example: In 5 + 3 * 2, multiplication (*) has higher precedence than addition (+), so 3 * 2 is evaluated first (6), then 5 + 6 = 11.
For associativity: In a = b = 5, assignment (=) has right-to-left associativity. So b = 5 is evaluated first, then a = b.

4. List any three header files with their usage.

- stdio.h**:** Standard Input/Output header. Provides functions like printf(), scanf(), fopen() for input/output operations.
- math.h**:** Mathematics header. Contains mathematical functions such as sqrt(), pow(), sin().
- string.h**:** String handling header. Includes functions like strlen(), strcpy(), strcmp() for string manipulations.

5. **Explain the ternary (? :) operator in detail. Can a ternary operator be nested (True/False)?**

   The ternary operator is a conditional operator with the syntax:

   condition ? expression1 : expression2

   If condition is true (non-zero), expression1 is evaluated and returned;
   otherwise, expression2 is evaluated and returned. It is often used as a shorthand for an if-else statement.
   Example: int max = (a > b) ? a : b; assigns the larger of a and b to max.

Yes, ternary operators can be nested. This allows multiple conditions in a single line, though it can reduce readability.
Example:
char* result = (marks >= 90) ? "A" : (marks >= 75) ? "B" : "C";

6. **What do you mean by enumerated data types in C language?**

   An enumerated data type (enum) is a user-defined type consisting of a set of named integer constants. It enhances code readability by assigning meaningful names to integral values. The

compiler automatically assigns integer values starting from 0, unless explicitly defined. Example:

```
enum Days {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
enum Days today = Wednesday; // 'today' holds integer value 3
```

7. Write the outputs of the following expressions:

i) 5 % 2 * 2 + 3

- 5 % 2 equals 1 (remainder).
- 1 * 2 equals 2.
- 2 + 3 equals 5.
  Output: 5

ii) 5 / 2.1 (int) 2.5 + 3

- The expression seems syntactically incorrect. Assuming it is 5 / 2.1 + (int)2.5 + 3:
- 5 / 2.1 (float division) ≈ 2.380952.
- (int)2.5 truncates to 2.
- Sum: 2.380952 + 2 + 3 ≈ 7.380952.
  Output: 7.380952 **(float)**
  Note: If the original intended expression was different, please clarify.

iii) (1 > 2) || ((2 < 3) && 5 < 1)

- 1 > 2 is false (0).
- 2 < 3 is true (1).
- 5 < 1 is false (0).
- (2 < 3) && (5 < 1) is 1 && 0 = false (0).
- 0 || 0 = false (0).
  Output: 0

Assignment 4

1. Explain the following: Break, Continue.

- break **statement:** Used to terminate the execution of the nearest enclosing loop (for, while, do-while) or switch statement. Control passes to the statement immediately following the terminated construct. It is commonly used to exit a loop prematurely when a condition is met.
  Example:

```
for(int i=1; i<=10; i++) {
   if(i == 5) break;
   printf("%d ", i); // Prints: 1 2 3 4
}
```

- continue **statement:** Skips the remaining statements in the current iteration of a loop and proceeds to the next iteration. In for loops, the update expression is executed before the next iteration; in while/do-while, control goes to the condition check.
  Example:

```
for(int i=1; i<=5; i++) {
   if(i == 3) continue;
   printf("%d ", i); // Prints: 1 2 4 5
}
```

2. Describe general form of if, if else, Nested if, goto.

- **Simple** if**:**

```
if (condition) {
   // statements executed if condition is true
}
```

- if-else**:**

```
if (condition) {
   // statements if true
} else {
   // statements if false
}
```

- **Nested if:** An if or if-else statement inside another if or else block. Allows multiple level decision-making.

```
if (condition1) {
   if (condition2) {
      // statements
   }
}
```

- goto **statement:** Transfers control unconditionally to a labeled statement within the same function. Generally discouraged as it can lead to spaghetti code, but sometimes used for error handling or breaking out of deeply nested loops.

```
goto label;
// ... code ...
label:
   // statements
```

3. Explain nested case statement with an example to read a number between 1 to 7 and print the relative day (Sunday to Saturday).

C does not have a "nested case" construct explicitly, but switch cases can be nested within an outer switch or other control structures. A more straightforward approach for mapping numbers 1–7 to days is a simple switch:

```
#include <stdio.h>
int main() {
```

```
    int day;
    printf("Enter a number (1-7): ");
    scanf("%d", &day);
    switch(day) {
        case 1: printf("Sunday\n"); break;
        case 2: printf("Monday\n"); break;
        case 3: printf("Tuesday\n"); break;
        case 4: printf("Wednesday\n"); break;
        case 5: printf("Thursday\n"); break;
        case 6: printf("Friday\n"); break;
        case 7: printf("Saturday\n"); break;
        default: printf("Invalid input\n");
    }
    return 0;
}
```

Nested switch *example:* One switch inside a case of another switch can be used for hierarchical menus, but is not needed for this simple mapping.

4. Calculate the output for the following code snippets:

Snippet 1:
x = 1, condition x+1 = 2 (non-zero, true). **Output: "Output 1"**

Snippet 2:
x = -1, -x-1 = -(-1)-1 = 1-1 = 0 (false). **Output: "Output 4"**

Snippet 3:
x = 0, x/x is division by zero, which leads to **undefined behavior** (runtime error, likely program crash). No guaranteed output.

Snippet 4:
x = -2, condition x-1 = -2-1 = -3 (non-zero, true). **Output: "Output 7"**

ASSIGNMENT 5

1. Consider the following C statements:

```
int a[5], sum = 0;
for(i=0; i<5; i++) {
   if((a[i]%2))
      continue;
   sum += a[i];
}
```

What would be output of above code?
The condition (a[i]%2) evaluates to 1 (true) for odd numbers (since a[i]%2 is 1 for odd, 0 for even). When true, continue skips the rest of the loop body, so sum += a[i] is not executed. Therefore, only **even numbers** are added to sum.

Output: The code calculates the **sum of even numbers of a[]**.

2. What the following code performs?

```
int count = 0;
for(i=0; i<10; i++)
   if(!(a[i]%2))
      count++;
```

The condition !(a[i]%2) is true when a[i]%2 is 0, i.e., when a[i] is even. Thus, count increments for every even element in the array.
It counts even values in array.

3. What the following code performs?

```
int count = 0;
for(i=0; i<10; i++)
   if(a[i]%2)
      count++;
```

Here, a[i]%2 is 1 (true) for odd numbers, 0 (false) for even numbers. So count increments for each odd element.
It counts odd values in array.

4. What would be output of above?

```
int a[] = {1,2,3,4,5};
for(i=0; i<4; i++)
   printf("%d ", a[++i]);
```

Loop trace:

- i=0: prints a[++i] → i becomes 1, print a[1] = 2
- i=1: loop increments i to 2 after iteration, then i++ from for makes i=3
  prints a[++i] → i becomes 4, print a[4] = 5
- i=4: fails i<4, loop stops.
  Output: 2 5

5. What is the maximum number of dimensions an array in C may have?

D. Theoratically no limit. The only practical limits are memory size and compilers.
The C standard does not specify a fixed limit; it depends on the compiler and available memory.

6. Starting address of A[49]?

Array A has indices 1..75 (75 elements total). Base address = 1120.
Element size = 3 memory words. Assuming word size = 1 memory unit for simplicity.
Address of A[k] = Base + (k-1)*element_size.
For A[49]: Address = 1120 + (49-1)*3 = 1120 + 48*3 = 1120 + 144 = **1264**.
Answer: A. 1264

7. Array is an example of _____ memory allocation.

A. Compile time
Size of array is fixed at compile time (static allocation). Exception: Variable Length Arrays (VLAs) in C99 are runtime-sized but still allocated on stack.

8. Size of the array need not be specified, when

D. All of the above

- A: Initialization at definition (int arr[] = {1,2,3};)
- B: Formal parameter (void func(int arr[]))
- C: Declaration (extern int arr[];)
  In all cases, size can be omitted.

9. Calculate the output.

First code block:

```
int arr[5] = {1,2,3,4,5};
int p, q, r;
p = ++arr[1];   // arr[1] becomes 3, p = 3
q = arr[1]++;   // q = 3, then arr[1] becomes 4
r = arr[p++];   // p++: use p=3 then p becomes 4, r = arr[3] = 4
printf("%d, %d, %d", p, q, r); // p=4, q=3, r=4
```

Output: 4, 3, 4

Second code block:

```
int a[1] = {100};
printf("%d", 0[a]); // 0[a] is same as a[0] = 100
```

Output: 100

Third code block:

```
int a[5] = {5,1,15,20,25};
int i,j,m;
i = ++a[1];   // a[1] becomes 2, i = 2
j = a[1]++;   // j = 2, then a[1] becomes 3
m = a[i++];   // i++: use i=2 then i becomes 3, m = a[2] = 15
printf("%d, %d, %d", i, j, m); // i=3, j=2, m=15
```

Output: 3, 2, 15

10. Which is true about the given statement?
int arr[10] = {0,1,2,[7]=7,8,9};
This is a **designated initializer** in C (allowed since C99). It initializes arr[0]=0, arr[1]=1, arr[2]=2, arr[7]=7, arr[8]=8, arr[9]=9, others to 0.
C. This is allowed in C.

11. Write a C program to read 10 numbers from user and store them in an array. Display Sum, Minimum and Average.

```
#include <stdio.h>
int main() {
    int arr[10], i, sum = 0, min;
    float avg;

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

    min = arr[0];
    for(i = 1; i < 10; i++) {
        if(arr[i] < min)
            min = arr[i];
    }

    avg = sum / 10.0;
    printf("Sum = %d\n", sum);
    printf("Minimum = %d\n", min);
    printf("Average = %.2f\n", avg);
```

```
    return 0;
}
```

12. Write a program to find a character from the string, string and character to be searched both will be given by user.

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[100], ch;
    int i, found = 0;

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    printf("Enter character to search: ");
    scanf("%c", &ch);

    for(i = 0; str[i] != '\0'; i++) {
        if(str[i] == ch) {
            printf("Character '%c' found at index %d\n", ch, i);
            found = 1;
        }
    }

    if(!found)
        printf("Character '%c' not found.\n", ch);

    return 0;
}
```

13. Show 2D array declaration, initialization and iteration.

```
#include <stdio.h>
int main() {
    // Declaration and initialization
    int matrix[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };

    // Iteration
    int i, j;
    for(i = 0; i < 3; i++) {
        for(j = 0; j < 4; j++) {
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

14. Write a program to display transpose of given 3×3 matrix.

```
#include <stdio.h>
int main() {
    int mat[3][3], transpose[3][3], i, j;

    printf("Enter 3x3 matrix elements:\n");
```

```
    for(i = 0; i < 3; i++)
        for(j = 0; j < 3; j++)
            scanf("%d", &mat[i][j]);

    // Transpose calculation
    for(i = 0; i < 3; i++)
        for(j = 0; j < 3; j++)
            transpose[j][i] = mat[i][j];

    printf("Transpose:\n");
    for(i = 0; i < 3; i++) {
        for(j = 0; j < 3; j++)
            printf("%d ", transpose[i][j]);
        printf("\n");
    }

    return 0;
}
```

15. Discuss various string functions with suitable examples.

Common string functions from string.h:

- strlen(): Returns length of string excluding \0.

```
char str[] = "Hello";
int len = strlen(str); // len = 5
```

- strcpy(dest, src): Copies source string to destination.

```
char src[] = "World", dest[20];
strcpy(dest, src); // dest = "World"
```

- strcat(dest, src): Concatenates source to destination.

```
char str1[20] = "Hello ";
strcat(str1, "World"); // str1 = "Hello World"
```

- strcmp(s1, s2): Compares two strings lexicographically. Returns 0 if equal, <0 if s1<s2, >0 if s1>s2.

```
int res = strcmp("apple", "banana"); // res < 0
```

- strchr(str, ch): Returns pointer to first occurrence of character ch in str, or NULL.

```
char *ptr = strchr("Hello", 'l'); // points to first 'l'
```

16. Write a program to print all Armstrong numbers in a given range.

```
#include <stdio.h>
int main() {
    int start, end, num, digit, sum, temp;

    printf("Enter range (start end): ");
    scanf("%d %d", &start, &end);

    printf("Armstrong numbers in range %d to %d:\n", start, end);
    for(num = start; num <= end; num++) {
        sum = 0;
        temp = num;

        while(temp != 0) {
            digit = temp % 10;
            sum += digit * digit * digit;
            temp /= 10;
        }
```

```
        if(sum == num)
            printf("%d ", num);
    }
    printf("\n");

    return 0;
}
```

17. What is an infinite loop explain with suitable example.

An infinite loop is a loop that never terminates because its terminating condition is never met. This can be intentional (for server programs) or unintentional (logical error).

Example of unintentional infinite loop:

```
int i = 0;
while(i < 5) {
    printf("%d ", i);
    // i is never incremented, so condition always true
}
```

Example of intentional infinite loop (with break condition):

```
while(1) {
    printf("Running...\n");
    // Some condition to break
    if(/* condition */)
        break;
}
```

Infinite loops can cause programs to hang and consume CPU resources indefinitely if not controlled properly.

ASSIGNMENT 6

1. Explain getch()**,** getchar()**,** gets() **and** fputs() **functions.**

- getch(): A non-standard function (from conio.h) that reads a single character from the keyboard **without echoing** it on screen and **without waiting for Enter key**. It is useful for password input or menu-driven programs. Example:

```
char c = getch(); // character is not displayed
```

- getchar(): Standard function (from stdio.h) that reads a single character from stdin and returns it as an int. It **waits for Enter key** and echoes the character. Example:

```
int ch = getchar(); // reads with echo, Enter needed
```

- gets(): Standard function (from stdio.h) that reads a line of text from stdin until newline or EOF, storing it as a string. It **does not check array bounds**, making it unsafe (risk of buffer overflow). Example:

```
char str[100];
gets(str); // deprecated, use fgets() instead
```

- fputs(): Standard function (from stdio.h) that writes a string to a specified output stream (e.g., file or stdout). It **does not add a newline** automatically. Example:

```
fputs("Hello", stdout); // prints "Hello" without newline
FILE *fp = fopen("file.txt", "w");
fputs("Text", fp); // writes to file
```

2. What is UDF? Describe advantages of UDF.

UDF (User Defined Function) is a function written by the programmer to perform a specific task, as opposed to built-in library functions. It consists of a function definition, call, and optionally a prototype.

Advantages:

1. **Modularity**: Breaks a large program into smaller, manageable modules.
2. **Reusability**: Once defined, can be called multiple times without rewriting code.
3. **Readability**: Makes code easier to understand and maintain.
4. **Abstraction**: Hides implementation details, allowing focus on higher-level logic.
5. **Testing & Debugging**: Individual functions can be tested independently.

6. **Collaboration**: Different programmers can work on different functions simultaneously.

3. Explain the function definition, function prototype and function call with relative example.

- **Function Prototype**: Declaration of the function before its use, specifying return type, name, and parameters. It informs the compiler about the function's signature.

```
int add(int a, int b); // prototype
```

- **Function Definition**: Actual implementation of the function containing the code to be executed.

```
int add(int a, int b) { // definition
    return a + b;
}
```

- **Function Call**: Invoking the function from another part of the program.

```
int result = add(5, 3); // call, result = 8
```

Complete Example:

```
#include <stdio.h>
int add(int a, int b); // prototype

int main() {
    int sum = add(10, 20); // call
    printf("Sum = %d\n", sum);
    return 0;
}

int add(int a, int b) { // definition
    return a + b;
}
```

4. Build a function to check number is prime or not. If number is prime then function return value 1 otherwise return 0.

```
#include <stdio.h>
int isPrime(int n) {
    if (n <= 1) return 0;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    if (isPrime(num))
        printf("%d is prime.\n", num);
    else
        printf("%d is not prime.\n", num);
    return 0;
}
```

5. Develop a function to print first N Fibonacci numbers.

```
#include <stdio.h>
void printFibonacci(int n) {
    int a = 0, b = 1, next;
    if (n >= 1) printf("%d ", a);
    if (n >= 2) printf("%d ", b);
    for (int i = 3; i <= n; i++) {
```

```
        next = a + b;
        printf("%d ", next);
        a = b;
        b = next;
    }
    printf("\n");
}

int main() {
    int N;
    printf("Enter N: ");
    scanf("%d", &N);
    printf("First %d Fibonacci numbers: ", N);
    printFibonacci(N);
    return 0;
}
```

6. A. Define recursion.

Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem. Each recursive call works on a smaller instance of the problem until a base condition is reached.

B. What care must be taken while writing a program with recursive function?

1. **Base Case**: Must define a terminating condition to prevent infinite recursion.
2. **Progress**: Each recursive call should move toward the base case.
3. **Stack Overflow**: Deep recursion may exhaust stack memory.
4. **Efficiency**: Recursive solutions can be less efficient than iterative ones due to function call overhead.
5. **Readability vs Performance**: Balance clarity with resource use.

C. Write code to find n Factorial using recursive function

```
#include <stdio.h>
int factorial(int n) {
    if (n == 0 || n == 1) // base case
```

```
    return 1;
  else
    return n * factorial(n - 1);
}

int main() {
  int n;
  printf("Enter n: ");
  scanf("%d", &n);
  printf("%d! = %d\n", n, factorial(n));
  return 0;
}
```

D. List the advantages of recursion

1. **Simplifies code** for problems with recursive nature (e.g., tree traversal).
2. **Elegance**: Natural fit for mathematical definitions (factorial, Fibonacci).
3. **Divide and Conquer**: Facilitates algorithms like quicksort, mergesort.
4. **Easier to write** for problems with hierarchical structures.

7. Write a C program to find $1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots + \frac{1}{n!}$

```
#include <stdio.h>
double seriesSum(int n) {
  double sum = 0.0, fact = 1.0;
  for (int i = 1; i <= n; i++) {
    fact *= i; // fact = i!
    sum += 1.0 / fact;
  }
  return sum;
}
```

```
int main() {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);
    printf("Sum = %lf\n", seriesSum(n));
    return 0;
}
```

8. In user defined function, what is actual argument and formal argument?

- **Formal Arguments**: Variables declared in the function definition to receive values passed during a function call. They act as local variables inside the function.
- **Actual Arguments**: The actual values or variables passed to the function when it is called.

Example:

```
int add(int a, int b) { // a, b are formal arguments
    return a + b;
}
int main() {
    int x = 5, y = 3;
    int sum = add(x, y); // x, y are actual arguments
}
```

9. Write a program to calculate nCr using user defined function.

```
#include <stdio.h>
int factorial(int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++) fact *= i;
    return fact;
}
```

```
int nCr(int n, int r) {
    if (r > n) return 0;
    return factorial(n) / (factorial(r) * factorial(n - r));
}

int main() {
    int n, r;
    printf("Enter n and r: ");
    scanf("%d %d", &n, &r);
    printf("%dC%d = %d\n", n, r, nCr(n, r));
    return 0;
}
```

10. Write a program in C using function to check whether two given strings are anagram or not.

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

int areAnagrams(char str1[], char str2[]) {
    int count[256] = {0}; // assuming ASCII
    int i;

    // Convert to lowercase and count
    for (i = 0; str1[i]; i++)
        count[tolower(str1[i])]++;
    for (i = 0; str2[i]; i++)
        count[tolower(str2[i])]--;

    // Check all counts zero
```

```
    for (i = 0; i < 256; i++)
        if (count[i] != 0) return 0;
    return 1;
}

int main() {
    char s1[100], s2[100];
    printf("Enter first string: ");
    fgets(s1, sizeof(s1), stdin);
    printf("Enter second string: ");
    fgets(s2, sizeof(s2), stdin);

    // Remove newline from fgets
    s1[strcspn(s1, "\n")] = '\0';
    s2[strcspn(s2, "\n")] = '\0';

    if (areAnagrams(s1, s2))
        printf("Strings are anagrams.\n");
    else
        printf("Strings are not anagrams.\n");
    return 0;
}
```

ASSIGNMENT 7

Multiple Choice Questions

1. What is recursion in programming?
    a) Repeating a set of instructions using loops
    b) **A function calling itself directly or indirectly**
    c) Breaking a problem into smaller sub problems
    d) Applying mathematical operations to a variable

2. Which of the following is the base case in a recursive function?
    a) The case where the function calls itself
    b) The first case in the function
    c) **The case that terminates the recursion**
    d) The case where a loop is used

3. What happens if a recursive function does not have a base case?
    a) The program will produce an error
    b) **The recursive function will run indefinitely**
    c) The program will terminate immediately
    d) The recursive function will skip the base case

4. Which data structure is commonly used in recursion?
    a) **Stack**
    b) Queue
    c) Array
    d) Linked List

5. What is the maximum depth of recursion in C?
    a) It depends on the operating system
    b) It depends on the compiler
    c) **It depends on the available memory**
    d) It is fixed and predetermined

6. What is tail recursion?
    a) A recursive function that calls itself at the beginning of the function
    b) **A recursive function that calls itself at the end of the function**
    c) A recursive function that calls another function
    d) A recursive function that has multiple base cases

7. When should recursion be used over iteration?
    a) When the problem can be easily solved using loops
    b) **When the problem requires breaking it into smaller subproblems**
    c) When the problem involves complex mathematical operations
    d) When the problem has a fixed number of iterations

8. Which of the following problems can be efficiently solved using recursion?
    a) Calculating the sum of an array
    b) Finding the maximum element in an array
    c) Sorting an array in ascending order
    d) **Generating all possible permutations of a string**

9. Which of the following is NOT a disadvantage of recursion?
    a. It can consume more memory compared to iteration
    b. It can be difficult to understand and debug
    c. It can lead to stack overflow for large input sizes
    d. **It always results in slower execution time than iteration**

Programming Questions

10. Recursive factorial function:

```
#include <stdio.h>

int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}

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

```
    if (n < 0)
        printf("Factorial is not defined for negative numbers.\n");
    else
        printf("Factorial of %d = %d\n", n, factorial(n));

    return 0;
}
```

11. Recursive Fibonacci function:

```
#include <stdio.h>

int fibonacci(int n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}

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

    if (n < 0)
        printf("Fibonacci is not defined for negative numbers.\n");
    else
        printf("Fibonacci(%d) = %d\n", n, fibonacci(n));
```

```
    return 0;
}
```

12. Recursive sum of digits:

```
#include <stdio.h>

int sumOfDigits(int n) {
    if (n == 0)
        return 0;
    else
        return (n % 10) + sumOfDigits(n / 10);
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("Sum of digits of %d = %d\n", num, sumOfDigits(num));
    return 0;
}
```

13. Recursive power calculation:

```
#include <stdio.h>

double power(double base, int exp) {
    if (exp == 0)
        return 1;
    else if (exp > 0)
```

```
        return base * power(base, exp - 1);
    else
        return (1 / base) * power(base, exp + 1);
}

int main() {
    double base;
    int exp;

    printf("Enter base: ");
    scanf("%lf", &base);
    printf("Enter exponent: ");
    scanf("%d", &exp);

    printf("%.2lf^%d = %.6lf\n", base, exp, power(base, exp));
    return 0;
}
```

ASSIGNMENT 8

Multiple Choice Questions

1. Which operator is used to access the value pointed to by a pointer in C?
   A) *** (Asterisk)**
   B) & (Ampersand)
   C) -> (Arrow)
   D) . (Dot)
2. What is the purpose of the "sizeof" operator in C?
   A) **To determine the size of a variable or data type**
   B) To allocate memory dynamically
   C) To perform bitwise operations
   D) To access the address of a variable
3. Which of the following correctly declares a pointer variable in C?
   A) **int *ptr;**
   B) int ptr;
   C) *int ptr;
   D) pointer int;
4. What is the output of the following code snippet?

```
int  x = 5;
int *ptr = &x;
printf("%d", *ptr);
```

   A) **5**
   B) 0
   C) Garbage value
   D) Compilation error
5. How do you pass a pointer to a function in C?
   A) Pass the pointer using call by value
   B) **Pass the pointer using call by reference**
   C) Pass the value of the pointer
   D) Pointers cannot be passed to functions
6. What is the NULL pointer in C?
   A) A pointer that stores the address of the main function
   B) A pointer that points to the end of a string
   C) **A pointer that does not point to any memory location**
   D) A pointer that contains the value zero
7. What is the purpose of the "const" keyword in a pointer declaration?
   A) **It makes the pointer constant and unmodifiable**
   B) It specifies the pointer type
   C) It ensures that the pointer is always initialized
   D) It allows the pointer to point to any data type

8. What is the result of the following code snippet?

```
int arr[] = {1, 2, 3, 4, 5};
int *ptr = arr;
printf("%d", *(ptr + 2));
```

A) 1
B) 2
C) 3
D) 4

9. What is the purpose of the "->" operator in C?

A) **To access a member of a structure through a pointer**
B) To compare two pointers
C) To perform logical AND operation on two pointers
D) To access the address of a variable

Programming Questions

10. Swap two integers using pointers:

```
#include <stdio.h>

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

int main() {
    int x, y;

    printf("Enter the first integer: ");
    scanf("%d", &x);
    printf("Enter the second integer: ");
    scanf("%d", &y);

    swap(&x, &y);

    printf("\nAfter swapping:\n");
    printf("First integer: %d\n", x);
    printf("Second integer: %d\n", y);

    return 0;
}
```

11. Find max and min using pointers:

```
#include <stdio.h>

int main() {
    int arr[5];
    int *ptr = arr;
    int i, max, min;

    printf("Enter 5 integers: ");
    for(i = 0; i < 5; i++) {
        scanf("%d", ptr + i);
    }

    max = min = *ptr; // Initialize with first element

    for(i = 0; i < 5; i++) {
        if(*(ptr + i) > max)
            max = *(ptr + i);
        if(*(ptr + i) < min)
            min = *(ptr + i);
    }

    printf("Maximum element: %d\n", max);
    printf("Minimum element: %d\n", min);

    return 0;
}
```

12. Sum and average using pointers:

```
#include <stdio.h>

int main() {
    int n, i, sum = 0;
    float avg;

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

    int arr[n];
    int *ptr = arr;

    printf("Enter %d integers: ", n);
    for(i = 0; i < n; i++) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    avg = (float)sum / n;

    printf("Sum: %d\n", sum);
    printf("Average: %.2f\n", avg);

    return 0;
}
```

ASSIGNMENT 9

Multiple Choice Questions

1. What is a structure in C?
    a) A function
    b) **A data type that groups related data elements**
    c) A loop construct
    d) A library in C

2. How do you declare a structure in C?
    a) **struct myStruct;**
    b) int myStruct;
    c) myStruct struct;
    d) struct = myStruct;

3. What is the keyword used to access structure members in C?
    a) **dot (.)** (for direct access)
    b) **arrow (->)** (for pointer access)
    c) colon (:)
    d) exclamation (!)

4. Which of the following statements is true about structures in C?
    a) Structures cannot contain functions.
    b) Structures cannot contain arrays.
    c) Structures can only contain integers.
    d) **Structures can contain members of different data types.**

5. What is the purpose of the sizeof() operator in C?
    a) **It returns the size of a structure in bytes.**
    b) It determines the number of members in a structure.
    c) It calculates the sum of all integers in a structure.
    d) It checks if a structure is empty.

6. How can you initialize a structure variable in C?
    a) **Using the assignment operator (=).**
    b) Using the sizeof() operator.
    c) Using the malloc() function.
    d) Using the new keyword.

7. What is the purpose of typedef in C structures?
    a) **It assigns a new name to an existing data type.**

b) It specifies the size of a structure.
c) It defines a new structure data type.
d) It declares a structure variable.

8. Which operator is used to access structure members through a pointer in C?
   a) asterisk (*)
   b) ampersand (&)
   c) **arrow (->)**
   d) dot (.)

9. What is the maximum number of members a structure can have in C?
   a) 10
   b) 100
   c) **There is no maximum limit.**
   d) It depends on the compiler.

Theoretical Questions

10. Purpose of structure in programming:

Structures allow grouping of related data items of different types under a single name. They enable logical organization of complex data, making programs more readable, maintainable, and modular. Structures facilitate abstraction by representing real-world entities (like student, employee) as single units.

11. Difference between structure and union:

Structure	Union
Allocates memory for all members	Allocates memory equal to largest member
All members store values simultaneously	Only one member stores value at a time
Size = sum of sizes of all members	Size = size of largest member
Members accessed independently	Members share same memory location
Used for grouping related data	Used for saving memory when storing mutually exclusive data

12. Nested structures:

A structure containing another structure as its member.

```
struct Address {
    char city[50];
    char state[50];
    int pincode;
};
```

```
struct Student {
    char name[50];
    int rollNo;
    struct Address addr; // Nested structure
};
```

13. Advantages of using structures:

- Logical grouping of related data
- Code reusability
- Better organization and readability
- Supports complex data modeling
- Can be passed to functions as a single unit
- Supports array of structures

14. Difference between structure and array:

Structure	Array
Can hold different data types	Holds same data type
Members accessed by name	Elements accessed by index
Logical grouping of related data	Collection of similar elements
Size determined by sum of members	Size determined by number of elements

15. Passing structure to a function:

Structures can be passed:

- By value (entire structure copied)
- By reference (passing pointer)

- By address (passing pointer)

16. Real-life scenario for structures:

A library management system uses structures to represent books:

```
struct Book {
    char title[100];
    char author[50];
    char ISBN[20];
    int year;
    float price;
    int copies;
};
```

Programming Questions

17/19. Function to print student info:

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

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

void printStudentInfo(struct Student s) {
    printf("\nStudent Details:\n");
    printf("Name: %s\n", s.name);
    printf("Roll No: %d\n", s.rollNo);
    printf("Marks: %.2f\n", s.marks);
```

```
}

int main() {
    struct Student s1;

    strcpy(s1.name, "John Doe");
    s1.rollNo = 101;
    s1.marks = 85.5;

    printStudentInfo(s1);
    return 0;
}
```

18/20. Array of structures for employees:

```
#include <stdio.h>

struct Employee {
    char name[50];
    int id;
    float salary;
    char department[30];
};

int main() {
    struct Employee emp[5];
    int i;

    printf("Enter details of 5 employees:\n");
    for(i = 0; i < 5; i++) {
```

```
        printf("\nEmployee %d:\n", i + 1);
        printf("Name: ");
        scanf(" %[^\n]", emp[i].name);
        printf("ID: ");
        scanf("%d", &emp[i].id);
        printf("Salary: ");
        scanf("%f", &emp[i].salary);
        printf("Department: ");
        scanf(" %[^\n]", emp[i].department);
    }

    printf("\n\nEmployee Information:\n");
    printf("-----------------------------------------------------\n");
    printf("%-5s %-20s %-10s %-15s %-15s\n", "S.No", "Name", "ID", "Salary", "Department");
    printf("-----------------------------------------------------\n");

    for(i = 0; i < 5; i++) {
        printf("%-5d %-20s %-10d %-15.2f %-15s\n",
               i + 1, emp[i].name, emp[i].id, emp[i].salary, emp[i].department);
    }

    return 0;
}
```

ASSIGNMENT 10

Multiple Choice Questions

1. What is the purpose of dynamic memory allocation in C?
    a) To allocate memory for global variables
    b) To allocate memory for local variables
    c) To allocate memory at compile-time
    d) **To allocate memory at runtime**

2. Which library function is used to dynamically allocate memory in C?
    a) alloc()
    b) **malloc()**
    c) new()
    d) allocate()

3. What is the return type of the malloc function in C?
    a) **void***
    b) int
    c) char*
    d) float*

4. What is the purpose of the calloc function in C?
    a) **To allocate memory for arrays**
    b) To allocate memory for structures
    c) To allocate memory for functions
    d) To allocate memory for pointers

5. What is the difference between malloc and calloc in C?
    a) malloc initializes allocated memory to 0, while calloc does not.
    b) **calloc initializes allocated memory to 0, while malloc does not.**
    c) malloc is used for single variable allocation, while calloc is used for array allocation.
    d) calloc is used for single variable allocation, while malloc is used for array allocation.

6. Which library function is used to reallocate memory in C?
    a) **realloc**
    b) resize
    c) allocate
    d) adjust

7. What happens if the realloc function fails to allocate memory?
    a) **It returns NULL and leaves the original memory block intact.**

b) It returns the original memory block without any changes.
c) It returns a memory block with zero size.
d) It raises a runtime error.

8. What is the purpose of realloc in C?

a) To free dynamically allocated memory.
b) **To resize dynamically allocated memory.**
c) To initialize dynamically allocated memory.
d) To check if dynamically allocated memory is valid.

9. How do you handle an error when malloc or calloc fails to allocate memory?
a) Print an error message and continue execution.
b) **Terminate the program.**
c) Retry the allocation process.
d) Skip the memory allocation step.

10. When using malloc, calloc, or realloc, why is it important to release the allocated memory using free?
a) **To prevent memory leaks.**
b) To improve program performance.
c) To avoid segmentation faults.
d) To reduce the memory footprint of the program.

Theoretical Questions

11. Dynamic memory allocation:

Allocation of memory during program execution (runtime) rather than compile time. Uses heap memory managed through functions like malloc(), calloc(), realloc(), and free().

12. Advantages over static allocation:

- Flexible memory usage (size determined at runtime)
- Efficient use of memory (allocate only what's needed)
- Can resize memory blocks
- Lifetime controlled by programmer
- Suitable for data structures with variable size

13. Drawbacks:

- Manual memory management required
- Risk of memory leaks if not freed
- Fragmentation of heap memory
- Slower than stack allocation
- Pointer dereferencing overhead
- Debugging more difficult

14. Purpose of functions:

- malloc(): Allocates specified bytes of memory (uninitialized)
- calloc(): Allocates memory for array and initializes to zero
- realloc(): Resizes previously allocated memory block
- free(): Releases allocated memory back to heap

15. Difference between malloc and calloc:

- **malloc** takes one argument (bytes), returns uninitialized memory
- **calloc** takes two arguments (count, size), returns zero-initialized memory
- **calloc** is preferred for arrays as it clears memory

16. Memory allocation examples:

```
// Single variable
int *ptr = (int*)malloc(sizeof(int));

// Array of 10 integers
int *arr = (int*)malloc(10 * sizeof(int));

// Array using calloc
int *arr2 = (int*)calloc(10, sizeof(int));
```

Programming Questions

17. Dynamic allocation for Person structure:

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

struct Person {
    char name[50];
    int age;
};

int main() {
    struct Person *p;

    // Allocate memory
```

```
    p = (struct Person*)malloc(sizeof(struct Person));

    if(p == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Input data
    printf("Enter name: ");
    scanf(" %[^\n]", p->name);
    printf("Enter age: ");
    scanf("%d", &p->age);

    // Display data
    printf("\nPerson Information:\n");
    printf("Name: %s\n", p->name);
    printf("Age: %d\n", p->age);

    // Free memory
    free(p);
    printf("\nMemory deallocated successfully.\n");

    return 0;
}
```

18. Dynamic array with sum calculation:

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

int main() {
    int *arr, n, i, sum = 0;
```

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

    // Allocate memory
    arr = (int*)malloc(n * sizeof(int));

    if(arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Input array elements
    printf("Enter %d integers:\n", n);
    for(i = 0; i < n; i++) {
        printf("Element %d: ", i + 1);
        scanf("%d", &arr[i]);
        sum += arr[i];
    }

    // Display sum
    printf("\nSum of all elements: %d\n", sum);
    printf("Average: %.2f\n", (float)sum / n);

    // Free memory
    free(arr);
    printf("Memory deallocated.\n");

    return 0;
}
```