Question Bank Answers

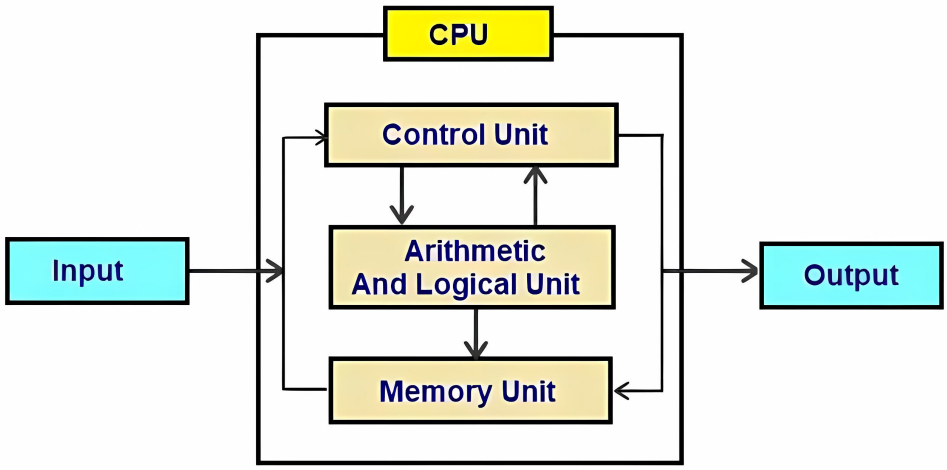
1. Define a computer system. What are its main components? Also, draw the block diagram.

A computer system is an electronic device that accepts data, performs operations, displays results and stores the data or results as per needs.

Main Components:

1. **Input Devices**: The input devices takes input and convert it into binary that computer understands (e.g Keyboard, Mouse, Scanner etc)
2. **Output Devices**: The output devices gives the binary language output to human readable language (e.g Monitor, Printer etc)
3. **Central Processing Unit (CPU)**: The brain of the computer that processes instructions.
4. **Memory**: Stores data and instructions (e.g. RAM, ROM).
5. **Storage Devices**: Used to store data permanently (e.g. Hard Drive, SSD).

Block Diagram:



2. What is the difference between syntax errors and logical errors in C programming?

- **Syntax Errors**:
  - Occur when the code violates the syntax rules of the programming language. (e.g., missing ; or incorrect function names)

    - Detected during compilation.
- **Logical Errors**:
    - Occur when the program runs but produces incorrect results due to mistakes in logic (e.g., wrong calculations or conditions like using length + breadth instead of length * breadth).
    - Not detected by the compiler.

3. Explain what a pseudo code is, and how it is used in programming.

- **Definition**: Pseudocode is a way to plan a program using simple, human-readable language. It's not actual code, but it helps to outline the logic before writing the real code.
- **Usage**:
    i. Helps plan and visualize algorithms before actual coding.
    ii. Used for explaining logic during presentations or discussions.
    iii. Easy to convert into actual programming code.

Example:

```
Start
  Input a, b
  If a > b Then
    Print "A is greater"
  Else
    Print "B is greater"
  EndIf
End
```

4. Explain the functions of a loader and a linker in the process of executing a program.

Loader	Linker
Loads the executable file into memory.	Combines object files and libraries into a single executable.
Assigns memory addresses to program instructions.	Resolves references between functions and variables.
Takes an executable file as input.	Takes object files and library files as input.

Loader	Linker
Prepares the program for execution.	Creates an executable file that can be run.

5. Explain the structure of a C program. What is the significance of the main() function and how it is used in program execution.

1. **Preprocessor Directives**:
   - Instructions starting like `#include<stdio.h>`, used to include libraries or define constants.
2. **Main Function**:
   - Every C program begins execution from the `main()` function.
3. **Functions**:
   - Blocks of reusable code that perform specific tasks (e.g., `void myFunction() {}`).
4. **Variable Declarations**:
   - Define variables to store data (e.g., `int num;`).
5. **Input/Output Statements**:
   - Functions like `printf()` and `scanf()` handle user interaction.

Significance of `main()`: The `main()` function is where program execution begins. The program starts from here and, after execution, returns a value (often `0` for successful execution) to the operating system.

Example:

```c
#include <stdio.h>

int main() {
   printf("Hello, World!\n");
   return 0;
}
```

6. What is the difference between the logical AND (&&) and logical OR (||) operators? Provide an example to illustrate your answer.

- **Logical AND (`&&`)**: Returns `true` if both operands are true. If either operand is false, it returns `false`.

- **Logical OR (`||`)**: Returns `true` if at least one of the operands is true. If both operands are false, it returns `false`.

Example:

```
int a = 5, b = 10;
if (a > 0 && b > 5) {
   printf("Both conditions are true.\n");
}

if (a < 0 || b > 5) {
   printf("At least one condition is true.\n");
}
```

7. What is type conversion in C? Explain the difference between implicit type conversion and explicit type casting with examples.

- **Type Conversion**: The process of converting one data type into another.
  - **Implicit Type Conversion** (Automatic): The compiler automatically converts data types when necessary. It happens when a smaller type is assigned to a larger type.
    - **Example**:

      ```
      int a = 5;
      float b = a;  // Implicit conversion from int to float
      ```

  - **Explicit Type Casting**: The programmer manually converts one data type to another using type casting.
    - **Example**:

      ```
      float a = 5.5;
      int b = (int)a;  // Explicit casting from float to int
      ```

8. Explain the difference between one-dimensional and two-dimensional arrays with examples. How are elements accessed and represented in memory?

One-dimensional Array	Multi-dimensional Array
It stores elements in a single line.	It stores elements in a grid-like structure.
Single index for access items.	Multiple indices to access items.
Requires less memory.	Requires more memory.
Example: int arr[5];	Example: int arr[3][3];

Memory Representation:

- **One-dimensional**: Stored in contiguous memory locations.
- **Two-dimensional**: Stored as a series of rows each row being a one-dimensional array.

9. Describe the differences between while and do-while loops in C. Provide an example where using do-while is more appropriate than while.

For Loop:

- Use when you know how many times the loop will run.
- Combines start, condition and update in one line.
- Cleaner and better for counting or fixed ranges.

Example:

```
for (int i = 0; i < 5; i++) {
    printf("Number: %d\n", i);
}
```

While Loop:

- Use when you don’t know how many times the loop will run.
- Checks a condition and keeps running until it's false.
- You control the start and update separately.

Example:

```c
int i = 0;
while (i < 5) {
    printf("Number: %d\n", i);
    i++;
}
```

When to Prefer Each:

- **For Loop**:
  - Looping through a range of numbers or elements in a collection with a defined start and end.
  - Example: Printing numbers from 1 to 100.
- **While Loop**:
  - Waiting for an external event or condition like user input or checking a file's status.
  - Example: Repeatedly prompting a user until valid input is given.

10. Describe a situation where a `goto` statement might be used in a loop. Why is its use generally discouraged?

The goto statement can be used to jump to a specific label, often used for error handling or exiting deeply nested loops.

Example:

```c
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        if (i == 2 && j == 3) {
            goto exit_loop;  // Exit both loops when i is 2 and j is 3
        }
    }
}
exit_loop:
    printf("Exited loops.\n");
```

Why `goto` is discouraged:
It's discouraged because it can make the code harder to read, maintain and disturbs the programs flow of control.

11. Explain in detail the representation of arrays in memory. Discuss how one-dimensional and multi-dimensional arrays

are stored, accessed, and manipulated with examples.

- **One-Dimensional Arrays**: Arrays are stored as contiguous blocks of memory. Each element is located in a consecutive memory address, and you can access elements using an index.
  Example:

```
int arr[3] = {1, 2, 3};
printf("%d", arr[1]);  // Accesses the second element (2)
```

- **Two-Dimensional Arrays**: A two-dimensional array is a grid of rows and columns, where each row is stored consecutively in memory in row-major order.
  Example:

```
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
printf("%d", arr[1][2]);  // Accesses element at row 1, column 2 (6)
```

- **Memory Representation**:
  - **1D Arrays**: All elements are placed in a single row in memory.
  - **2D Arrays**: Arrays are stored in a linear block, with the first row elements followed by the second row, and so on.

12. Describe the different types of functions in C. Provide examples of each type.

- **Functions with no arguments and no return value**:

```
void printMessage() {
    printf("Hello World\n");
}
```

- **Functions with arguments but no return value**:

```
void printSum(int a, int b) {
    printf("Sum: %d\n", a + b);
}
```

- **Functions with no arguments but a return value**:

```
int getNumber() {
    return 10;
}
```

- **Functions with arguments and a return value**:

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

13. What is an array of pointers? Provide an example where it is used to store and print multiple strings.

An array of pointers is a collection of pointers where each pointer can point to a string or array. This allows you to handle multiple strings efficiently.

Example:

```c
#include <stdio.h>

int main() {
    char *arr[] = {"Hello", "World", "C", "Programming"};
    for (int i = 0; i < 4; i++) {
        printf("%s\n", arr[i]);
    }
    return 0;
}
```

Here, `arr` is an array of pointers, each pointing to a string. The program prints each string in the array.

14. What is the difference between an assembler, compiler, and interpreter? Provide examples of when each might be used.

Here's a tabular comparison between an assembler, compiler, and interpreter:

Assembler	Compiler	Interpreter
Converts assembly language code to machine code.	Converts the entire program into machine code at once.	Translates and executes the program line by line. code.
Produces machine code or object code.	Produces an executable file.	Directly executes the source code.

Assembler	Compiler	Interpreter
One-to-one translation (assembly to machine).	Whole program is translated at once.	Translates and executes one line at a time.
Fast (once written)	Generally slower due to full translation.	Slower execution as it processes line-by-line.
Errors are detected after translation.	Errors are detected after full compilation.	Errors are detected as execution progresses.
Used for low-level programming on specific hardware.	Developing large software applications (e.g., operating systems, games)	Used for scripting, web development etc

15. What are recursive functions? Explain their advantages and limitations.

A recursive function is a function that calls itself. Recursive functions are often used to solve problems that can be divided into similar sub-problems (e.g., factorial, Fibonacci).

Advantages:

- Simplifies complex problems by breaking them into smaller, manageable problems.
- Reduces the need for iterative loops.

Limitations:

- Can lead to stack overflow if the recursion depth is too high.
- May be less efficient due to function calls overhead.

Example:

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

16. What is the difference between call by value and call by reference in C? Write a program to demonstrate swapping two

numbers using both approaches and explain the output.

Call by Value:

- The actual value of the argument is passed to the function.
- Modifications inside the function do not affect the original variable.

Call by Reference:

- The address of the variable is passed to the function.
- Modifications inside the function affects the original variable.

Call by Value Example:

```c
#include <stdio.h>

void modify(int x) {
    x = 10;  // Only changes the local copy of x
}

int main() {
    int num = 5;
    modify(num);
    printf("Value of num: %d", num);  // Outputs 5, not 10
    return 0;
}
```

Call by Refrence Example:

```c
#include <stdio.h>

void modify(int *x) {
    *x = 10;  // Changes the original value of x
}

int main() {
    int num = 5;
    modify(&num);  // Passes the address of num
    printf("Value of num: %d", num);  // Outputs 10
    return 0;
}
```

17. Explain how pointers are declared in C and provide examples of basic pointer operations such as dereferencing and pointer arithmetic.

- **Declaration**: A pointer is declared using the `*` symbol.
  - **Example**:

```
int *ptr;  // Pointer to an integer
```

- **Dereferencing**: Accessing the value pointed to by the pointer using `*`.
  - **Example**:

```
int num = 10;
int *ptr = &num;
printf("%d", *ptr);  // Dereferencing, output: 10
```

- **Pointer Arithmetic**: Pointers can be incremented or decremented to point to the next or previous memory location.
  - **Example**:

```
int arr[3] = {1, 2, 3};
int *ptr = arr;
printf("%d", *(ptr + 1));  // Pointer arithmetic, output: 2
```

18. Explain how arrays are passed to functions in C. Write a program to calculate the sum of all elements in an array using a function.

In C, when an array is passed to a function the function receives the address of the first element (i.e., a pointer to the array). This means that any changes made to the array elements inside the function will affect the original array.

Example program:

```c
#include <stdio.h>

int sum(int arr[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += arr[i];
    }
    return total;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Sum of elements: %d\n", sum(arr, n));  // Output: Sum of elements: 15
    return 0;
}
```

19. What are enumerated data types in C? Explain their use with an example.

Enumerated data types (or enum) are user-defined data types in C that consist of a set of named integer constants. They make the code more readable and meaningful by replacing numeric values with descriptive names.

Example:

```c
#include <stdio.h>

enum day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

int main() {
    enum day today;
    today = Wednesday;
    printf("Day number: %d", today);
    return 0;
}
```

20. Write a program that uses a switch statement to print the season based on user input (e.g., 1 for Spring, 2 for Summer,

etc.).

```c
#include <stdio.h>

int main() {
    int month;
    printf("Enter month number (1 to 12): ");
    scanf("%d", &month);

    switch (month) {
        case 1: case 2: case 3:
            printf("Season: Spring\n");
            break;
        case 4: case 5: case 6:
            printf("Season: Summer\n");
            break;
        case 7: case 8: case 9:
            printf("Season: Autumn\n");
            break;
        case 10: case 11: case 12:
            printf("Season: Winter\n");
            break;
        default:
            printf("Invalid month\n");
    }

    return 0;
}
```

21. Describe the differences between structures and unions in C. How is memory allocated for both? Write a program to demonstrate the usage of a structure and a union for storing employee details.

- **Structures**: A structure is a user-defined data type that can hold multiple different data types. Each member of a structure is allocated memory separately.
- **Unions**: A union is similar to a structure but the members share the same memory location. Only one member can hold a value at any given time, and the memory is allocated for the largest member.

Memory Allocation:

- In a structure, memory is allocated for each member separately.
- In a union, memory is allocated based on the largest member and all members share that memory.

Example program:

```c
#include <stdio.h>

struct Employee {
    int id;
    char name[20];
    float salary;
};

union EmployeeUnion {
    int id;
    char name[20];
    float salary;
};

int main() {
    struct Employee emp1 = {1, "John", 50000};
    union EmployeeUnion empUnion;
    empUnion.id = 1;
    printf("Structure - Employee ID: %d, Name: %s, Salary: %.2f\n", emp1.id, emp1.name, emp1.sa
    printf("Union - Employee ID: %d\n", empUnion.id);

    return 0;
}
```

In this example, a structure holds all the employee details separately, while the union only holds one detail at a time (based on the largest member).

22. Explain in detail how multi-dimensional arrays can be used to solve problems. Write a program to perform matrix addition, subtraction, and multiplication using functions.

A multi-dimensional array is an array of arrays, which is useful for representing tables, matrices and more complex structures like 2D grids.

Example program (Matrix Addition, Subtraction, and Multiplication):

```c
#include <stdio.h>

#define ROWS 2
#define COLS 2

void addMatrices(int a[ROWS][COLS], int b[ROWS][COLS], int result[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            result[i][j] = a[i][j] + b[i][j];
        }
    }
}

void subtractMatrices(int a[ROWS][COLS], int b[ROWS][COLS], int result[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            result[i][j] = a[i][j] - b[i][j];
        }
    }
}

void multiplyMatrices(int a[ROWS][COLS], int b[ROWS][COLS], int result[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            result[i][j] = a[i][j] * b[i][j];
        }
    }
}

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

int main() {
    int matrix1[ROWS][COLS] = {{1, 2}, {3, 4}};
    int matrix2[ROWS][COLS] = {{5, 6}, {7, 8}};
    int result[ROWS][COLS];

    printf("Matrix 1:\n");
    printMatrix(matrix1);
```

```
    printf("Matrix 2:\n");
    printMatrix(matrix2);

    addMatrices(matrix1, matrix2, result);
    printf("Matrix Addition:\n");
    printMatrix(result);

    subtractMatrices(matrix1, matrix2, result);
    printf("Matrix Subtraction:\n");
    printMatrix(result);

    multiplyMatrices(matrix1, matrix2, result);
    printf("Matrix Multiplication:\n");
    printMatrix(result);

    return 0;
}
```

23. Discuss the differences between character arrays and strings in C. Write a program to find the frequency of each vowel in a given string.

- **Character arrays**: A character array is a collection of characters stored sequentially in memory. It can be used to store a string, but it does not automatically handle string termination (null character `\0`).
- **Strings**: A string in C is simply an array of characters terminated by a null character `\0`. Functions like `printf` or `strcpy` are designed to work with strings, treating the null character as the end of the string.

Example program:

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

void vowelFrequency(char str[]) {
    int vowels[5] = {0};  // To store frequency of 'a', 'e', 'i', 'o', 'u'
    int length = strlen(str);

    for (int i = 0; i < length; i++) {
        char ch = tolower(str[i]); // Convert to lowercase to handle both cases
        if (ch == 'a') vowels[0]++;
        else if (ch == 'e') vowels[1]++;
        else if (ch == 'i') vowels[2]++;
        else if (ch == 'o') vowels[3]++;
        else if (ch == 'u') vowels[4]++;
    }

    printf("Vowel frequencies:\n");
    printf("a: %d\n", vowels[0]);
    printf("e: %d\n", vowels[1]);
    printf("i: %d\n", vowels[2]);
    printf("o: %d\n", vowels[3]);
    printf("u: %d\n", vowels[4]);
}

int main() {
    char str[] = "Programming in C is fun!";
    vowelFrequency(str);
    return 0;
}
```

24. What is the purpose of a linker and a loader in program execution? Explain their differences.

- **Linker**: The linker combines object files into a single executable. It resolves references between different modules or libraries, ensuring that function calls and variables are correctly mapped.
- **Loader**: The loader is responsible for loading the executable into memory when the program is executed. It assigns memory locations for variables, initializes the program's runtime environment and prepares it for execution.

Differences:

- **Linker** works before the program is executed, while **loader** works at runtime.

- The linker generates an executable, while the loader loads the executable into memory.

25. Explain the concept of pseudocode. Write pseudocode to find the largest of three numbers.

- **Definition**: Pseudocode is a way to plan a program using simple, human-readable language. It's not actual code, but it helps to outline the logic before writing the real code.

Pseudocode to find the largest of three numbers:

```
BEGIN
    Input three numbers: num1, num2, num3
    IF num1 >= num2 AND num1 >= num3 THEN
        largest = num1
    ELSE IF num2 >= num1 AND num2 >= num3 THEN
        largest = num2
    ELSE
        largest = num3
    END IF
    Output largest
END
```

26. Write the structure of a C program. Explain each section with an example.

1. **Preprocessor Directives**: Used to include libraries or define constants.
2. **Global Declarations**: Variables or functions that are accessible throughout the program.
3. **Main Function**: The entry point of the program where execution begins.
4. **Function Definitions**: Functions that perform specific tasks.

Example program structure:

```c
#include <stdio.h>   // Preprocessor Directive

// Global Variable Declaration
int globalVar = 10;

// Function Declaration
void greet(void);

int main() {   // Main Function
    greet();
    return 0;
}

// Function Definition
void greet() {
    printf("Hello, World!\n");
}
```

- **Preprocessor Directives**: `#include <stdio.h>` includes the standard I/O library for input/output functions.
- **Global Declarations**: `globalVar` is a variable available throughout the program.
- **Main Function**: Execution starts here, calling the `greet()` function.
- **Function Definition**: `greet()` prints "Hello, World!" to the screen.

27. What are storage classes in C? List and explain the four types of storage classes with examples.

Storage classes in C determine where a variable can be used, how long it exists, and whether it can be accessed from other parts of the program. There are four types:

1. **auto**: Default storage class for local variables, with automatic storage duration. The variable is created when the block is entered and destroyed when the block is exited.

   ```c
   void func() {
       auto int x = 5;  // auto is implicit for local variables
   }
   ```

2. **register**: Suggests to the compiler to store the variable in a CPU register for faster access.

   ```c
   void func() {
       register int x = 5;
   }
   ```

3. **static**: Maintains the value of the variable between function calls. It retains its value across function calls and is initialized only once.

```
void func() {
    static int counter = 0;
    counter++;
    printf("%d\n", counter);
}
```

4. **extern**: Used to declare a variable that is defined outside the current file. It allows the use of a global variable across different files.

```
extern int x;  // Declaration in a different file
```

28. What is the significance of return types in functions?

The return type of a function defines the type of value that the function will return to the calling code. It ensures type safety and allows the program to handle different kinds of return values appropriately.

- **void**: Indicates the function does not return any value.
- **Non-void**: Specifies the type of the value returned (e.g., `int`, `float`, etc.).

29. Explain the concept of operator precedence and associativity in C. Provide an example to demonstrate how expressions are evaluated based on precedence.

Operator precedence determines the order in which operators are evaluated in an expression. In arithmetic, multiplication and division have higher precedence than addition and subtraction so they are evaluated first.

Example:

```
#include <stdio.h>

int main() {
    int result = 2 + 3 * 4;
    printf("%d\n", result);  // Output: 14 (Multiplication is performed first due to higher pre
    return 0;
}
```

30. Write the algorithm, pseudocode, and corresponding C code for calculating the area of the following shapes: Circle, Triangle, and Rectangle.

Algorithm:

1. **For Circle:**
   - Input the radius `r`.
   - Calculate the area using the formula: `Area = π * r^2`.
   - Output the area.
2. **For Triangle:**
   - Input the base `b` and height `h`.
   - Calculate the area using the formula: `Area = (b * h) / 2`.
   - Output the area.
3. **For Rectangle:**
   - Input the length `l` and width `w`.
   - Calculate the area using the formula: `Area = l * w`.
   - Output the area.

Pseudocode:

```
BEGIN
    Input shape_type  // Circle, Triangle, or Rectangle
    IF shape_type == "Circle" THEN
        Input radius r
        Area = 3.14 * r * r
    ELSE IF shape_type == "Triangle" THEN
        Input base b, height h
        Area = (b * h) / 2
    ELSE IF shape_type == "Rectangle" THEN
        Input length l, width w
        Area = l * w
    END IF
    Output area
END
```

For Circle:

```c
#include <stdio.h>

#define PI 3.14

void calculateCircleArea() {
    float radius;
    printf("Enter the radius of the circle: ");
    scanf("%f", &radius);
    printf("Area of Circle: %.2f\n", PI * radius * radius);
}

int main() {
    char shape;
    printf("Enter shape type (C for Circle): ");
    scanf(" %c", &shape);
    if (shape == 'C') {
        calculateCircleArea();
    }
    return 0;
}
```

For Triangle:

```c
#include <stdio.h>

void calculateTriangleArea() {
    float base, height;
    printf("Enter the base and height of the triangle: ");
    scanf("%f %f", &base, &height);
    printf("Area of Triangle: %.2f\n", (base * height) / 2);
}

int main() {
    char shape;
    printf("Enter shape type (T for Triangle): ");
    scanf(" %c", &shape);
    if (shape == 'T') {
        calculateTriangleArea();
    }
    return 0;
}
```

For Rectangle:

```c
#include <stdio.h>

void calculateRectangleArea() {
    float length, width;
    printf("Enter the length and width of the rectangle: ");
    scanf("%f %f", &length, &width);
    printf("Area of Rectangle: %.2f\n", length * width);
}

int main() {
    char shape;
    printf("Enter shape type (R for Rectangle): ");
    scanf(" %c", &shape);
    if (shape == 'R') {
        calculateRectangleArea();
    }
    return 0;
}
```

31. What are variables in C? Explain their declaration and initialization with examples. How are they stored in memory?

A **variable** in C is a named storage location in memory that holds a value of a specified type.

- **Declaration**: Specifies the name and type of the variable.
- **Initialization**: Assigns a value to the variable at the time of declaration.

Example:

```c
#include <stdio.h>

int main() {
    int x = 10; // Declaration and initialization
    float y = 20.5; // Declaration and initialization
    char ch = 'A'; // Declaration and initialization

    printf("x = %d, y = %.2f, ch = %c\n", x, y, ch);
    return 0;
}
```

Memory Storage:

- **Integer**: Stored as 4 bytes in most systems.
- **Float**: Stored as 4 bytes.
- **Character**: Stored as 1 byte.

32. Explain the role of memory in a computer system. Differentiate between primary memory and secondary storage with examples.

- **Primary Memory (RAM)**: It is the main memory used to store data and instructions that are currently in use by the CPU. It is fast but volatile, meaning data is lost when power is turned off. Example: RAM.
- **Secondary Storage**: It is used for permanent storage of data. It is slower than primary memory but retains data even when the system is powered off. Example: Hard Disk Drive (HDD), Solid State Drive (SSD).

Primary Memory	Secondary Memory
Temporary storage used while the computer is on.	Permanent storage used for long-term data retention.
Volatile (data is lost when the power is off).	Non-volatile (data is retained even after power off).
Faster compared to secondary memory.	Slower speed compared to primary memory.
Used for running programs, active data processing etc	Used for storing files, documents etc
Examples: RAM (Random Access Memory), Cache	**Examples**: Hard Disk Drive (HDD), Solid State Drive (SSD), CD/DVD

33. What is the purpose of break and continue statements in loops? Write a program that uses continue to skip printing even numbers between 1 and 10.

- **Break**: Terminates the loop immediately and exits.
- **Continue**: Skips the current iteration and moves to the next iteration of the loop.

Example Program (using continue):

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            continue; // Skip even numbers
        }
        printf("%d ", i);  // Print odd numbers
    }
    return 0;
}
```

Output:

```
1 3 5 7 9
```

34. Define object code and executable code. Explain the differences between them with suitable examples.

- **Object Code**: It is the intermediate code generated by the compiler after compiling the source code. It is in machine-readable format but not fully executable. It needs to be linked to create an executable.
- **Executable Code**: It is the final code that can be directly run by the operating system. It is produced after the object code is linked with libraries and other resources.

Differences:

- **Object Code** is not runnable, while **Executable Code** is ready to be executed.
- **Object Code** is generated by the compiler, while **Executable Code** is the result of the linking process.

35. What is recursion in programming? Write a recursive function in C to compute the nth Fibonacci number.

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.

Example Program (Fibonacci Sequence):

```c
#include <stdio.h>

int fibonacci(int n) {
    if (n <= 1) {
        return n;  // Base case
    }
    return fibonacci(n-1) + fibonacci(n-2);  // Recursive case
}

int main() {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    printf("Fibonacci number at position %d is: %d\n", n, fibonacci(n));
    return 0;
}
```

Output (for `n = 5`):

```
Fibonacci number at position 5 is: 5
```

36. Explain how pointers can be used to access elements in a multi-dimensional array. Write a program to demonstrate accessing and modifying elements in a 2D array using pointers.

In C, multi-dimensional arrays are stored in row-major order, which means all elements of the first row are stored first, followed by the elements of the second row, and so on.

Pointers can be used to access these elements by treating the 2D array as a linear array.

Example Program:

```c
#include <stdio.h>

int main() {
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int *ptr = &arr[0][0]; // Pointer to the first element of the array

    // Accessing and modifying elements using pointer arithmetic
    printf("Original values:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", *(ptr + i*3 + j));  // Accessing using pointer arithmetic
        }
        printf("\n");
    }

    // Modify an element (changing arr[1][2] to 100)
    *(ptr + 1*3 + 2) = 100;

    printf("Modified values:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", *(ptr + i*3 + j));
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
Original values:
1 2 3
4 5 6

Modified values:
1 2 3
4 5 100
```

37. How does pointer arithmetic work with multi-dimensional arrays?

Pointer arithmetic works with multi-dimensional arrays in the same way it works with one-dimensional arrays, but with respect to the number of elements in each dimension.

- For a 2D array `arr[M][N]`, the elements are stored as a linear array of size `M * N`.
- Pointer arithmetic allows accessing the element at `arr[i][j]` as `*(arr + i * N + j)`.

38. Explain the different types of user-defined functions in C (e.g., functions with no arguments and no return value, functions with arguments but no return value, etc.). Write an example program for each type.

Types of User-Defined Functions:

1. **Function with No Arguments and No Return Value**:
   - This function performs some task but does not take any input and does not return any value.

   Example:

```c
#include <stdio.h>

void printMessage() {
    printf("Hello, World!\n");
}

int main() {
    printMessage();  // Calling the function
    return 0;
}
```

2. **Function with Arguments but No Return Value**:
   - This function takes input but does not return a value.

   Example:

```c
#include <stdio.h>

void printSum(int a, int b) {
    printf("Sum: %d\n", a + b);
}

int main() {
    int num1 = 10, num2 = 20;
    printSum(num1, num2);  // Calling the function
    return 0;
}
```

3. **Function with Arguments and Return Value**:
   - This function takes input and returns a value.

   Example:

```c
#include <stdio.h>

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

int main() {
    int num1 = 10, num2 = 20;
    printf("Sum: %d\n", add(num1, num2));  // Calling the function
    return 0;
}
```

4. **Function with No Arguments but with Return Value**:
   - This function does not take input but returns a value.

   Example:

```c
#include <stdio.h>

int getNumber() {
    return 42;
}

int main() {
    printf("Returned Value: %d\n", getNumber());  // Calling the function
    return 0;
}
```

39. What is the difference between an array and a structure in C? Provide examples to illustrate when to use each.

- **Array**: A collection of elements of the same type, stored in contiguous memory locations. Arrays are used when you need to store multiple values of the same data type.
- **Structure**: A user-defined data type that can store values of different types. Structures are used when you need to group different types of data together.

Example of Array:

```c
#include <stdio.h>

int main() {
    int arr[3] = {1, 2, 3};
    printf("Array elements: %d, %d, %d\n", arr[0], arr[1], arr[2]);
    return 0;
}
```

Example of Structure:

```c
#include <stdio.h>

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

int main() {
    struct Person p1 = {"John", 30};
    printf("Name: %s, Age: %d\n", p1.name, p1.age);
    return 0;
}
```

When to use:

- Use an **array** when you need to store multiple items of the same type, e.g., a list of integers.
- Use a **structure** when you need to group different types of data together, e.g., storing information about a person (name, age, etc.).

40. Write a program to demonstrate how to declare and initialize an array of structures in C.

Example Program:

```c
#include <stdio.h>

struct Student {
    int roll_no;
    char name[50];
};

int main() {
    struct Student students[2] = {{1, "John"}, {2, "Alice"}};

    printf("Student 1: Roll No: %d, Name: %s\n", students[0].roll_no, students[0].name);
    printf("Student 2: Roll No: %d, Name: %s\n", students[1].roll_no, students[1].name);

    return 0;
}
```

Output:

```
Student 1: Roll No: 1, Name: John
Student 2: Roll No: 2, Name: Alice
```

41. Explain how pointers can be used with an array of structures. Write a program that uses pointers to access and modify the data in an array of structures.

Pointers can be used to access elements in an array of structures by treating the array as a pointer to a structure. Using pointer arithmetic or the `->` operator, we can modify the elements in the array.

Example Program:

```c
#include <stdio.h>

struct Student {
    int roll_no;
    char name[50];
};

int main() {
    struct Student students[2] = {{1, "John"}, {2, "Alice"}};
    struct Student *ptr = students;  // Pointer to the first element of the array

    // Access and modify using pointers
    printf("Original values:\n");
    printf("Student 1: Roll No: %d, Name: %s\n", ptr->roll_no, ptr->name);
    ptr++;  // Move to the next student
    printf("Student 2: Roll No: %d, Name: %s\n", ptr->roll_no, ptr->name);

    // Modify the name of the second student
    ptr--;
    ptr->roll_no = 3;
    snprintf(ptr->name, sizeof(ptr->name), "Bob");

    printf("\nModified values:\n");
    printf("Student 1: Roll No: %d, Name: %s\n", students[0].roll_no, students[0].name);
    printf("Student 2: Roll No: %d, Name: %s\n", students[1].roll_no, students[1].name);

    return 0;
}
```

Output:

```
Original values:
Student 1: Roll No: 1, Name: John
Student 2: Roll No: 2, Name: Alice

Modified values:
Student 1: Roll No: 3, Name: Bob
Student 2: Roll No: 2, Name: Alice
```

42. Discuss in detail the components of a computer system: memory, processor, I/O devices, and storage. How do these components interact to perform tasks?

A computer system consists of four main components: the **processor (CPU)**, **memory**, **input/output devices**, and **storage**. These components interact in the following ways:

1. **Processor (CPU)**:
   - The CPU is the brain of the computer that executes instructions. It performs arithmetic, logic, control, and input/output operations.
   - **Interaction**: The CPU fetches data from memory, processes it, and stores results back in memory or sends it to an I/O device.
2. **Memory (RAM)**:
   - Memory stores data and instructions that are needed by the CPU for immediate processing. It is volatile, meaning it loses its content when power is turned off.
   - **Interaction**: The CPU accesses memory to fetch instructions and store results. The memory is divided into various regions such as registers, cache, and main memory (RAM).
3. **Input/Output Devices (I/O)**:
   - I/O devices allow the computer to interact with the outside world. These include keyboards, mice, printers, monitors, and more.
   - **Interaction**: The CPU processes data and sends results to output devices. It also takes input from input devices to perform tasks.
4. **Storage**:
   - Storage devices (e.g., hard drives, SSDs) store data permanently. Unlike memory, storage is non-volatile, meaning it retains data even when power is off.
   - **Interaction**: The CPU can read from and write to storage devices to retrieve and save data. Programs are loaded from storage into memory for execution.

Interaction: When a task is performed:

- The CPU processes instructions and retrieves or stores data in memory.
- Input data is received from input devices (e.g., keyboard) and processed.
- The output is sent to output devices (e.g., monitor, printer).
- Data is stored on storage devices if needed.