Set 1:

Question 1:

**Basic Structure of a C Program:**

A C program typically follows a specific structure, which includes the following components:

1. **Preprocessor Directives**: These are instructions to the compiler to preprocess the information before actual compilation starts. #include <stdio.h>  // Includes the standard input-output library

2. **Global Declarations**: Variables and functions that are declared outside of any function. These are accessible throughout the program.

   ```
   int globalVar = 10;  // Global variable
   ```

3. **Main Function**: The entry point of a C program. Every C program must have a main() function. The execution of the program starts from the main() function.

   ```
   int main() {
     // Code
     return 0;
   }
   ```

4. **Local Declarations**: Variables declared inside a function. These are accessible only within that function. Local variables are not initialized by default and contain garbage values if not explicitly initialized.

   ```
   int localVar = 20;  // Local variable
   ```

5. **Statements and Expressions**: The actual code that performs operations. Statements can include variable assignments, function calls, loops, and conditional statements.

   ```
   printf("Hello, World!\n");  // Statement
   ```

6. **Functions**: Blocks of code that perform specific tasks and can be called from the main() function or other functions.

   ```
   void myFunction() {
    // Function code
   }
   ```

**Example of a Basic C Program:**

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

int globalVar = 10;  // Global Declaration

void display() {  // Function Definition
```

```
    printf("Global Variable: %d\n", globalVar);
}
int main() {  // Main Function
    int localVar = 20;  // Local Declaration
    printf("Local Variable: %d\n", localVar);  // Statement
    display();  // Function Call
    return 0;
}
```

**scanf() Function:**

The scanf() function is used to read formatted input from the standard input (keyboard). It reads data from the user and stores it in the specified variables. The scanf() function is part of the standard input-output library (stdio.h).

**Syntax:**

```
int scanf(const char *format, ...);
```

**Example:**

```
#include <stdio.h>
int main() {
    int age;
    float height;
    char name[50];
    printf("Enter your age: ");
    scanf("%d", &age);  // Reading an integer input
    printf("Enter your name: ");
    scanf("%s", name);  // Reading a string input
    printf("Your age is: %d\n", age);
    printf("Your name is: %s\n", name);
    return 0;
}
```

Question 2:

**Decision Control Statements:**

Decision control statements in C are used to make decisions based on certain conditions. These statements allow the program to execute different blocks of code based on the evaluation of conditions. They are essential for implementing logic and controlling the flow of the program.

**Types of Decision Control Statements:**

1. **if Statement**: Executes a block of code if a specified condition is true.

   ```
   if (condition) {   // Code to execute if condition is true

   }
   ```

2. **if-else Statement**: Executes one block of code if a condition is true, and another block if the condition is false.

   ```
   if (condition) {// Code to execute if condition is true

   } else {// Code to execute if condition is false

   }
   ```

3. **else-if Ladder**: Allows multiple conditions to be checked in sequence.

   ```
   if (condition1) {// Code to execute if condition1 is true

   } else if (condition2) {// Code to execute if condition2 is true

   } else {// Code to execute if all conditions are false}
   ```

4. **switch Statement**: Allows a variable to be tested for equality against a list of values.

   ```
   switch (variable) {

     case value1:

       // Code to execute if variable == value1

        break;

     case value2:

        // Code to execute if variable == value2

        break;

     default:

   }
   ```

**Example:**

```
#include <stdio.h>

int main() {
```

```
    int number = 10;

    if (number > 0) {
        printf("The number is positive.\n");
    } else {
        printf("The number is negative or zero.\n");
    }
    switch (number) {
        case 10:
            printf("The number is 10.\n");
            break;
        case 20:
            printf("The number is 20.\n");
            break;
        default:
            printf("The number is neither 10 nor 20.\n");
    }
    return 0;
}
```

Question 3:

**Purpose of Storage Classes:**

Storage classes in C define the scope (visibility) and lifetime of variables and/or functions within a C program. They determine where the variable is stored, its initial value, and how long it retains its value.

**Types of Storage Classes:**

1. **auto**: The default storage class for local variables. The variable is stored in memory and has a local scope.

   ```
   void function() {
       auto int localVar = 10;
   ```

```
}
```

2. **register**: Suggests that the variable be stored in a CPU register instead of RAM for faster access. The variable has a local scope.

```
void function() {
    register int counter = 0;
}
```

3. **static**: The variable retains its value between function calls. It can be used for both local and global variables.

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

4. **extern**: Declares a global variable that is defined in another file or later in the same file. It extends the visibility of the variable.

```
extern int globalVar;
```

5. **const**: Declares a variable as constant, meaning its value cannot be changed after initialization.

```
const int maxLimit = 100;
```

**Example:**

```
#include <stdio.h>
// Global variable
int globalVar = 10;
void function() {
    // Local auto variable
    auto int localVar = 20;
    // Static variable
    static int staticVar = 30;
    // Register variable
    register int registerVar = 40;
    printf("Global Variable: %d\n", globalVar);
    printf("Local Variable: %d\n", localVar);
    printf("Static Variable: %d\n", staticVar);
    printf("Register Variable: %d\n", registerVar);
}
int main() {
    function();
    return 0;
}
```

Set 2:

Question 4:

**Call by Value:**

In call by value, a copy of the actual parameter's value is passed to the function. Changes made to the parameter inside the function do not affect the original value.

**Example:**

```
#include <stdio.h>
void modifyValue(int x) {
    x = 20;
}
int main() {
    int a = 10;
    modifyValue(a);
    printf("Value of a: %d\n", a);  // Output: 10
    return 0;
}
```

**Call by Reference:**

In call by reference, the address of the actual parameter is passed to the function. Changes made to the parameter inside the function affect the original value.

**Example:**

```
#include <stdio.h>

void modifyValue(int *x) {

    *x = 20;

}

int main() {

    int a = 10;

    modifyValue(&a);

    printf("Value of a: %d\n", a);  // Output: 20

    return 0;

}
```

**Recursion:**

Recursion is a programming technique where a function calls itself to solve a problem. It is useful for problems that can be broken down into smaller, similar sub-problems.

**Example: Factorial Calculation**

```
#include <stdio.h>

int factorial(int n) {

    if (n == 0) {

        return 1;

    } else {

        return n * factorial(n - 1);

    }

}

int main() {

    int num = 5;

    printf("Factorial of %d is %d\n", num, factorial(num));  // Output: 120

    return 0;

}
```

Question 5:

**Pointers in C**:Pointers store the memory address of another variable.

Used for dynamic memory allocation, arrays, functions, and data structures like linked lists, trees, and graphs.

**Declaration:**

Declared using the asterisk (*) symbol.

**Syntax:** data_type *pointer_name;

Pointer Arithmetic:

- Involves operations like addition and subtraction on pointers.
- Used to navigate through arrays and memory blocks.
- Operations are based on the size of the data type the pointer points to.

Types of Pointer Arithmetic:

- Incrementing a Pointer: Moves to the next memory location.
- Decrementing a Pointer: Moves to the previous memory location.
- Addition: Adds an integer value, moving forward by that many elements.
- Subtraction: Subtracts an integer value, moving backward by that many elements.

Example:

```
#include <stdio.h>

int main() {

   int arr[] = {10, 20, 30, 40, 50};  // Array of integers

   int *p = arr;  // Pointer to the first element

   // Accessing array elements using pointer arithmetic

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

      printf("Element %d: %d\n", i, *(p + i));

   }

   // Incrementing the pointer

   p++;

   printf("After incrementing, value pointed by p: %d\n", *p);  // Output: 20

   // Decrementing the pointer

   p--;

   printf("After decrementing, value pointed by p: %d\n", *p);  // Output: 10
```

```
    // Adding an integer to the pointer
    p = p + 2;
    printf("After adding 2, value pointed by p: %d\n", *p);  // Output: 30
    // Subtracting an integer from the pointer
    p = p - 1;
    printf("After subtracting 1, value pointed by p: %d\n", *p);  // Output: 20
    return 0;
}
```

Question 6:

**A. Structures and Unions in C**

**Structure:**

- User-defined data type grouping related variables of different types.
- Each member has its own memory location.

**Example:**

```
#include <stdio.h>
struct Student {
    int id;
    char name[50];
    float marks;
};
int main() {
    struct Student s1 = {1, "Alice", 85.5};
    printf("ID: %d\n", s1.id);
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);
    return 0;
}
```

**Union:**

- User-defined data type grouping related variables of different types.
- All members share the same memory location; only one member can hold a value at a time.

**Example:**

```
#include <stdio.h>

#include <string.h>

union Data {

   int i;

   float f;

   char str[20];

};

int main() {

   union Data data;

   data.i = 10;

   printf("Integer: %d\n", data.i);

   data.f = 220.5;

   printf("Float: %.2f\n", data.f);

   strcpy(data.str, "Hello");

   printf("String: %s\n", data.str);

   return 0;

}
```

**Differences:**

- **Memory Allocation:** Structures allocate separate memory for each member, while unions share the same memory location.
- **Usage:** Structures are used when multiple members need to be accessed simultaneously, while unions are used when only one member needs to be accessed at a time.

**B. Dynamic Memory Allocation in C**

**Functions:**

1. **malloc():** Allocates specified bytes and returns a pointer to the memory. Memory is not initialized.

```
int *ptr = (int *)malloc(sizeof(int) * 5);
```

2. **calloc():** Allocates memory for an array, initializes all bytes to zero, and returns a pointer.

   ```
   int *ptr = (int *)calloc(5, sizeof(int));
   ```

3. **realloc():** Resizes the memory block pointed to by a pointer.

   ```
   ptr = (int *)realloc(ptr, sizeof(int) * 10);
   ```

4. **free():** Deallocates memory previously allocated by malloc(), calloc(), or realloc().

   ```
   free(ptr);
   ```

**Example:**

```
#include <stdio.h>

#include <stdlib.h>

int main() {

   int *ptr;

   int n, i;

   printf("Enter number of elements: ");

   scanf(%d", &n);

   ptr = (int *)malloc(n * sizeof(int));

   if (ptr == NULL) {

      printf("Memory not allocated.\n");

      exit(0);

   }

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

      ptr[i] = i + 1;  }

   printf("Elements of the array: ");

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

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

   }

   free(ptr);

   return 0;}
```