BLACK N WHITE

Learn Today Lead Tomorrow

jag....

NAME	
ROLL NUMBER	
SEMESTER	1st
Batch	7th
COURSE CODE	DCA1102
COURSE NAME	**PROGRAMMING IN C**

Black N White (blacksnwhite.com)

SET - I

Q.1) What is the basic structure of C program? Explain printf() and scanf() functions with an example.

Answer .:- I'd be glad to explain the basic structure of a C program and delve into the printf() and scanf() functions with an illustrative example:

Basic Structure of a C Program:

A well-structured C program adheres to a specific format that enhances readability, maintainability, and modularity. Here's a breakdown of the key sections:

1. **Preprocessor Directives (Optional):**
   - Lines starting with # that instruct the preprocessor (a separate program that runs before compilation) to perform certain actions.
   - Common directives include:
     - #include <header_file.h>: Includes standard library functions and definitions from header files.
     - #define MACRO_NAME value: Creates a macro (named constant) for code reusability.
2. **Global Variable Declarations (Optional):**
   - Variables declared outside any function have global scope, meaning they can be accessed from anywhere in the program.
   - While convenient, excessive global variables can lead to naming conflicts and make code harder to manage.
3. **Main Function:**
   - The mandatory entry point of every C program execution.
   - Must have the signature int main(void).
   - The int return type indicates the program's exit status (0 for successful execution).
   - The void argument list specifies that main doesn't take any arguments.
   - All program logic and function calls typically reside within main.
4. **User-Defined Functions (Optional):**
   - Reusable blocks of code that perform specific tasks.
   - Defined using a function prototype (declaration) specifying the return type, name, and argument list (parameters) before main.
   - The actual function definition with the code body comes later.

5. **Function Calls:**
   - Statements that execute the code within a function.
   - Pass arguments (if any) to the function.
   - The function performs its task and optionally returns a value.

Example:

C
```
#include <stdio.h> // Include standard input/output header

int addNumbers(int x, int y) { // Function prototype for addNumbers
  return x + y;
}

int main() {
  int num1, num2, sum;

  printf("Enter two integers: "); // Prompt for input using printf
  scanf("%d %d", &num1, &num2);  // Read input using scanf

  sum = addNumbers(num1, num2); // Call addNumbers function

  printf("The sum of %d and %d is %d\n", num1, num2, sum); // Display result using printf

  return 0; // Indicate successful program termination
}
```

Explanation of printf() and scanf():

- **printf():**
  - Stands for "print formatted."
  - Part of the standard input/output library (stdio.h).
  - Used to print formatted output to the console.
  - Takes a format string as the first argument, followed by optional arguments that correspond to placeholders within the format string.
  - The format string specifies how the output should be displayed, using escape sequences like \n for newline and %d for integers.
- **scanf():**
  - Stands for "scan formatted."
  - Also part of stdio.h.
  - Used to read formatted input from the console.
  - Takes a format string as the first argument, followed by address-of operators (&) before variables where the read values should be stored.

   - The format string specifies the expected input data type, using %d for integers, %f for floats, etc.

Q.2) What is function? Explain different classification of user defined functions based on parameter passing and return type with examples.

Answer .:- Functions are fundamental building blocks that promote code reusability, modularity, and readability. They encapsulate specific tasks, improving program organization and maintainability.

Understanding Functions:

- **Definition:** A function is a named block of code that performs a particular task. It's designed to be self-contained and reusable from different parts of your program.
- **Execution:** Functions are only executed when explicitly called. The main function is the mandatory entry point of a C program, and it often serves as the starting point for calling other user-defined functions.

Classifying Functions by Parameter Passing and Return Type:

C functions can be categorized based on how they handle parameters (data passed into the function) and return values (data sent back from the function). Here's a breakdown of the common classifications:

1. **Functions with No Parameters and No Return Value (Void Functions):**
   - Signature: return_type function_name(void) { ... }
   - Example:

   C
   ```
   void greet() {
     printf("Hello, world!\n");
   }

   int main() {
     greet(); // Call the greet function
     return 0;
   }
   ```
   In this example, the greet function doesn't take any arguments (void) and doesn't return any value (void). It simply prints a message to the console.
2. **Functions with Parameters and No Return Value (Procedures):**
   - Signature: return_type function_name(parameter_list) { ... }
   - Example:

C
```c
void printFullName(char *firstName, char *lastName) {
  printf("%s %s\n", firstName, lastName);
}

int main() {
  char firstName[] = "Alice";
  char lastName[] = "Smith";
  printFullName(firstName, lastName); // Call with arguments
  return 0;
}
```

The printFullName function takes two string pointers (char *) as parameters (firstName and lastName) to print a full name. It doesn't return any value (void).

3. **Functions with No Parameters and a Return Value:**
   - Signature: return_type function_name(void) { ... return value; }
   - Example:

C
```c
int getArea() {
  int length = 5, width = 3;
  return length * width; // Calculate and return area
}

int main() {
  int area = getArea();
  printf("Area: %d\n", area);
  return 0;
}
```

The getArea function doesn't take any parameters (void) but calculates the area (length * width) and returns the result as an integer (int).

4. **Functions with Parameters and a Return Value:**
   - Signature: return_type function_name(parameter_list) { ... return value; }
   - Example:

C
```c
int addNumbers(int num1, int num2) {
  return num1 + num2;
}

int main() {
  int a = 10, b = 20;
  int sum = addNumbers(a, b);
  printf("Sum: %d\n", sum);
  return 0;
}
```

The addNumbers function takes two integer parameters (num1 and num2), adds them, and returns the sum (int).

Q.3) Explain the different kinds of loops available in C with examples.
Answer .:-

1. **for Loop: The Champion of Known Iterations**

   The for loop reigns supreme when you know the exact number of times you need to execute a code block. It excels at iterating a predetermined number of times, making it the go-to choice for tasks like printing sequences or processing fixed-size arrays.

   - **Syntax:**

     C
     ```
     for (initialization; condition; increment/decrement) {
       // Code to be executed repeatedly
     }
     ```
   - **Breakdown:**
     - initialization: This statement executes only once at the loop's beginning, often used for initializing a loop counter variable.
     - condition: This condition is evaluated before each iteration. If it's true, the loop body executes. Once it becomes false, the loop terminates.
     - increment/decrement: This statement updates the loop counter variable after each iteration, controlling the loop's progress.
   - **Example (Printing numbers from 1 to 5):**

     C
     ```
     for (int i = 1; i <= 5; i++) {
       printf("%d ", i);
     }
     ```
     In this example, i is initialized to 1, the condition i <= 5 ensures we iterate 5 times, and i is incremented by 1 after each iteration, printing numbers from 1 to 5.

2. **while Loop: The Persistent Executor**

   The while loop is your partner when you need to execute a code block repeatedly until a specific condition becomes false. It's ideal for scenarios where the number of iterations depends on user input, external factors, or complex logic.

   - **Syntax:**

     C

```
while (condition) {
  // Code to be executed repeatedly
}
```

- **Breakdown:**
  - condition: This condition is the heart of the while loop. Before each iteration, the condition is evaluated. If it's true, the loop body executes. The loop continues to iterate as long as the condition remains true.
- **Example (Reading input until the user enters 'q'):**

  C
  ```
  char input;
  while (input != 'q') {
    printf("Enter a character (q to quit): ");
    scanf(" %c", &input); // Note the space before %c for single character
  }
  ```
  This loop keeps prompting the user for input until they enter 'q'. The loop continues as long as input is not equal to 'q'.

3. **do-while Loop: Guaranteed Execution at Least Once**

   The do-while loop offers a unique twist. It ensures the code block executes at least once, regardless of the initial condition. This is useful when you have essential code that needs to run even if the condition might be false initially.
   - **Syntax:**

     C
     ```
     do {
       // Code to be executed repeatedly
     } while (condition);
     ```
   - **Breakdown:**
     - The code block within the do statement executes at least once before the condition is checked.
     - condition: After the first execution, the condition is evaluated before subsequent iterations. If it's true, the loop continues; otherwise, it terminates.
   - **Example (Ensuring a positive number input):**

     C
     ```
     int number;
     do {
       printf("Enter a positive number: ");
       scanf("%d", &number);
     } while (number <= 0);
     ```
     Here, the loop prompts the user for a number. Even if the user enters a non-positive number initially, the do block ensures the prompt is displayed and the user has a chance to correct their input. The loop continues until a positive number is entered.

SET - II

Q.4) What is the purpose of storage class in C? Explain various types of storage class in C.

Answer .:- In C programming, storage classes play a vital role in defining the **scope, visibility, and lifetime** of variables and functions. They essentially dictate where a variable or function is stored in memory and how long it exists during program execution. Here's a breakdown of the various storage classes in C, keeping the word count within 450:

1. Automatic Storage Class (Default):

- Variables declared within functions (local variables) or as function arguments automatically belong to this class.
- **Scope:** Local to the function where they are declared. They cease to exist once the function execution is complete.
- **Visibility:** Accessible only within the function's code block.
- **Lifetime:** Short-lived, existing only during the function's execution.
- **Example:**

  C
  ```
  void myFunction() {
     int x = 10; // Automatic variable
     // Use x within the function
  }
  ```

2. External Storage Class:

- Used to declare global variables that can be accessed from anywhere in the program.
- **Scope:** Global, accessible from all functions in the source file and potentially other files (with proper linkage).
- **Visibility:** Accessible throughout the program (can lead to naming conflicts if not used judiciously).
- **Lifetime:** Persists throughout the program's execution, even if control exits the block where it's declared.
- **Example:**

  C
  ```
  int globalVariable = 5; // External variable

  void someFunction() {
     printf("Global value: %d\n", globalVariable);
  }
  ```

3. Static Storage Class:

- Can be applied to both local variables (within functions) and global variables.
- **Scope:** Local if declared within a function, global if declared outside any function.
- **Visibility:** Same as the corresponding storage class (local or global).
- **Lifetime:** Unlike automatic variables, static variables retain their value between function calls. They are initialized only once, at the beginning of program execution.
- **Example (Local static variable):**

  C
  ```
  void myFunction() {
     static int counter = 0; // Local static variable
     counter++;
  ```

```
    printf("Counter: %d\n", counter);
}
```
In this example, counter will remember its value across multiple calls to myFunction.

4. Register Storage Class (Hint):

- A suggestion to the compiler to store a variable in a CPU register for faster access.
- Not guaranteed by the compiler, as register allocation depends on hardware architecture and compiler optimizations.
- Use sparingly and only for variables that are frequently accessed and have a small size.
- **Scope and Visibility:** Same as the variable's base storage class (automatic or external).
- **Lifetime:** Same as the variable's base storage class.
- **Example:**

  C
  ```
  register int count = 0; // Suggesting register storage
  ```

Q.5) Describe the process of dynamic memory allocation in C and elaborate on the different dynamic memory allocation functions in the C programming language.

Answer .:- C, unlike some other languages, doesn't offer automatic memory management. This means you have more control over memory allocation but also the responsibility to manage it properly to avoid memory leaks and improve program efficiency. This is where dynamic memory allocation comes in.

Dynamic Memory Allocation in C:

1. **Static vs. Dynamic Memory:**
   - Static memory allocation happens at compile time. The size of statically allocated variables (e.g., local variables, global variables) is fixed and determined beforehand.
   - Dynamic memory allocation, on the other hand, occurs during program execution at runtime. You can allocate memory for variables or data structures whose size might not be known until runtime.
2. **The Heap:**
   - C provides a special memory region called the heap for dynamic allocation.
   - The heap is a pool of memory that can be used and released as needed during program execution.
3. **Dynamic Memory Allocation Functions:**
   - C offers several library functions (stdlib.h) to manage memory allocation on the heap:
     - **malloc(size):** Allocates a block of memory of the specified size (in bytes) on the heap. It returns a void* pointer, which needs to be cast to the appropriate data type to access the allocated memory. If allocation fails (not enough memory available), malloc returns NULL.
     - **calloc(num_elements, element_size):** Allocates memory for an array of num_elements of the specified element_size (in bytes). It initializes all the

elements to zero. Similar to malloc, it returns a void* pointer that needs casting.

- **realloc(ptr, new_size):** Resizes an existing memory block pointed to by ptr to the new size new_size. This can be useful if you need to adjust the size of a previously allocated memory block. It returns a void* pointer to the resized block or NULL on failure.
- **free(ptr):** Releases the memory block pointed to by ptr. It's crucial to free dynamically allocated memory when you're done using it to prevent memory leaks.

Important Considerations:

- **Memory Leaks:** If you allocate memory with malloc, calloc, or realloc but forget to free it with free, you create a memory leak. This means the allocated memory remains occupied even if it's not being used, potentially leading to program crashes or performance issues.
- **Type Casting:** Remember to cast the returned void* pointer to the appropriate data type before accessing the allocated memory.
- **Error Handling:** Always check the return values of allocation functions (malloc, calloc, realloc) to ensure successful allocation. If they return NULL, handle the error condition appropriately (e.g., inform the user or exit gracefully).

Q.6.a) What is the difference between structure and union in C?

Answer .:-

Structure:

- Each member in a structure has its own distinct memory allocation.
- Size of a structure is typically the sum of the sizes of its individual members (not always exact due to padding).
- Allows storing multiple values independently within the same memory block.
- Example:

  C
  ```
  struct Point {
     int x;
     float y;
  };
  ```
  Here, x and y have separate memory locations within a Point structure.

Union:

- All members of a union share the same memory location.
- Size of a union is equal to the size of the largest member.

- Only one member can hold a value at a time. Assigning a value to one member overwrites the previous value in the shared memory location.
- Example:
  C
  ```
  union Data {
     int num;
     float decimal;
  };
  ```
  In this union, num and decimal cannot hold values simultaneously. Assigning a value to one will overwrite the other.

Choosing Between Structures and Unions:

- Use structures when you need to store multiple independent values of different data types.
- Use unions when you need to store different data types that represent alternative states of the same entity, and only one state is relevant at a time (memory efficiency is a concern).

Q.6.b) Discuss the concept of macro substitution in C preprocessor directives. Provide examples of macro definitions and demonstrate how they are expanded during compilation .

Answer .:- In C, macro substitution is a powerful preprocessor feature that allows you to define shortcuts for code snippets or constants. The preprocessor, a separate program that runs before compilation, identifies these macros and replaces them with their corresponding definitions during the preprocessing stage.

Macro Definition:

- Uses the #define directive followed by the macro name, an optional space, and the replacement text.
- Example: #define PI 3.14159 defines a macro named PI with the value 3.14159.

Macro Expansion:

- During preprocessing, the preprocessor encounters the macro name and replaces it with its defined value wherever it appears in the code.
- Example:
  C
  ```
  #define AREA(r) (3.14159 * r * r) // Macro for area of a circle
  ```

```
int radius = 5;
int circleArea = AREA(radius); // Expands to (3.14159 * 5 * 5)
```

Important Considerations:

- Macros are not functions; they simply perform text replacement.
- Be cautious with macro arguments, as they undergo simple text substitution, which can lead to unexpected behavior if not used carefully.
- Use parentheses around macro definitions to avoid unintended side effects.

Benefits:

- Code readability: Macros can improve code readability by replacing complex expressions or repetitive code blocks with short, meaningful names.
- Maintainability: Macros can centralize constants or frequently used expressions, making updates easier.

➔

- Use macros judiciously, as overuse can make code harder to understand and debug.
- Consider using const variables or inline functions for constants or simple expressions when appropriate.