Subject Name & Code:

PROGRAMMING FOR PROBLEM SOLVING- BE01R00121

Assignment – 1

Q-1: What is a user-defined function? Write types of user-defined functions? Explain each by example.

Answer:

A user-defined function is a block of code written by the programmer to perform a specific task. It is defined once and can be called (invoked) multiple times, promoting code reusability, modularity, and readability.

The four main types of user-defined functions in C are:

1. **Function with no arguments and no return value.**
    - **Explanation:** The function does not receive any data from the calling function and does not send any result back.
    - **Example:**

```
#include <stdio.h>
void greet() { // No arguments
   printf("Hello, Welcome to the program!\n");
   // No return value
}
int main() {
   greet(); // Calling the function
   return 0;
}
```

2. **Function with no arguments but a return value.**
    - **Explanation:** The function does not take any input but computes and returns a value to the calling function.

- **Example:**

```
#include <stdio.h>
int getNumber() { // No arguments
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    return num; // Returns an integer value
}
int main() {
    int a = getNumber(); // Value is returned and stored in 'a'
    printf("You entered: %d", a);
    return 0;
}
```

3. **Function with arguments but no return value.**
    - **Explanation:** The function receives data (arguments) from the calling function, operates on it, but does not return any value (uses void).
    - **Example:**

```
#include <stdio.h>
void checkEvenOdd(int num) { // Takes an integer argument
    if (num % 2 == 0)
        printf("%d is Even.\n", num);
    else
        printf("%d is Odd.\n", num);
    // No return statement
}
int main() {
    checkEvenOdd(10); // Passing 10 as an argument
```

```
    return 0;
}
```

4. **Function with arguments and a return value.**
    - **Explanation:** This is the most common type. The function takes input as arguments, processes them, and returns a result.
    - **Example:**

```
#include <stdio.h>
int add(int a, int b) { // Takes two integer arguments
    int sum = a + b;
    return sum; // Returns the computed sum
}
int main() {
    int result = add(5, 3); // Arguments passed, return value stored
    printf("Sum is: %d", result);
    return 0;
}
```

Q-2: What is actual argument and formal argument in a user-defined function?

Answer:

- **Formal Arguments:** These are the variables declared in the function definition header. They act as placeholders to receive the values passed from the calling function. Their scope is local to the function.
- **Actual Arguments:** These are the variables or constants (or expressions) used in the function call inside the main() or another calling function. Their values are passed to the formal arguments.

Example:

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

```
// Function Definition
int multiply(int x, int y) { // 'x' and 'y' are Formal Arguments
    return x * y;
}

int main() {
    int a = 5, b = 4;
    int product = multiply(a, b); // 'a' and 'b' are Actual Arguments
    printf("Product is: %d", product);
    return 0;
}
```

In this example, during the function call multiply(a, b), the values of the actual arguments a (5) and b (4) are copied to the formal arguments x and y respectively.

Q-3: Explain call by value (pass by value) and call by reference (pass by reference) with example.

Answer:

- **Call by Value:** In this method, a copy of the actual argument's value is passed to the formal argument. Any changes made to the formal argument *inside the function* do not affect the original actual argument.
    - **Example:**

```
#include <stdio.h>
void swap_by_value(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
    printf("Inside function (Call by Value): x = %d, y = %d\n", x, y);
```

```
}
int main() {
    int a = 10, b = 20;
    swap_by_value(a, b); // Passes copies of the values of a and b
    printf("In main (Call by Value): a = %d, b = %d\n", a, b); // a and b remain unchanged
    return 0;
}
```

Output:

```
Inside function (Call by Value): x = 20, y = 10
In main (Call by Value): a = 10, b = 20
```

- **Call by Reference:** In this method, the memory address (reference) of the actual argument is passed to the formal argument. The formal argument becomes an alias for the actual argument. Therefore, any changes made to the formal argument *inside the function* directly affect the original actual argument. In C, this is simulated using pointers.
    - **Example:**

```
#include <stdio.h>
void swap_by_reference(int *x, int *y) { // x and y are pointer variables
    int temp;
    temp = *x; // Dereferencing pointer x to get the value at address
    *x = *y;   // Assigning value at y's address to x's address
    *y = temp; // Assigning temp value to y's address
    printf("Inside function (Call by Reference): *x = %d, *y = %d\n", *x, *y);
}
int main() {
    int a = 10, b = 20;
    swap_by_reference(&a, &b); // Passes the addresses of a and b
```

```
    printf("In main (Call by Reference): a = %d, b = %d\n", a, b); // a and b are swapped
    return 0;
}
```

Output:

Inside function (Call by Reference): *x = 20, *y = 10

In main (Call by Reference): a = 20, b = 10

Q-4: What care must be taken while writing a program with a recursive function?

Answer:

When implementing a recursive function, extreme care must be taken to avoid logical errors and program crashes. The key considerations are:

1. **Base Case:** This is the most critical element. There must be at least one condition (the base case) where the function returns a value without making a recursive call. This stops the infinite recursion.
2. **Progress Towards Base Case:** Each recursive call must modify the arguments in such a way that the function moves closer to the base case. Otherwise, the recursion will never terminate.
3. **Stack Overflow:** Each recursive call uses space on the call stack. If recursion is too deep (e.g., a very large number or missing base case), it will exhaust the stack memory, leading to a stack overflow error and program termination.
4. **Efficiency:** Recursion can be computationally expensive and memory-intensive compared to iterative solutions (loops) for some problems due to the overhead of repeated function calls.

Q-5: What is an array? Give example and advantages of array. Explain difference between 1D and 2D array.

Answer:

An array is a collection of elements of the same data type, stored in contiguous memory locations. It allows storing multiple values under a single variable name, accessed by an index.

- **Example:** int marks[5] = {85, 90, 78, 92, 88};
- **Advantages:**
    1. **Code Optimization:** Allows efficient handling of a large number of similar data types.
    2. **Ease of Traversal:** Easy to access elements using a loop and an index.
    3. **Random Access:** Any element can be accessed directly in constant time using its index.
    4. **Memory Efficiency:** Contiguous allocation reduces memory overhead.
- **Difference between 1D and 2D Array:**

Feature	One-Dimensional (1D) Array	Two-Dimensional (2D) Array
Concept	A linear list of elements.	A table or matrix of elements (array of arrays).
Declaration	data_type array_name[size]; e.g., int arr[10];	data_type array_name[rows][cols]; e.g., int matrix[3][3];
Access	Single index: arr[i]	Two indices: matrix[i][j]
Memory	Stored as a single contiguous block.	Stored in row-major order as a contiguous block.
Analogy	A single row of lockers.	A grid of lockers with multiple rows and columns.

Q-6: What is a String? How are they declared and also define the null character.

Answer:

In C, a string is defined as a one-dimensional array of characters terminated by a **null character** ('\0'). The null character has an ASCII value of zero and is used to indicate the end of the string.

- **Declaration:**
  Strings can be declared in several ways:
  1. **By character array (with size):**
     char str[20]; // Can hold a string of up to 19 characters + null character
  2. **By character array (with initialization):**
     char str[] = "Hello"; // Compiler automatically adds '\0' and sets size to 6
  3. **By character pointer:**
     char *str = "Hello";
- **The Null Character ('\0'):**
  It is a special character used to mark the end of a string in C. All standard C library string functions (like strlen, strcpy) rely on its presence to work correctly. In memory, the string "Hello" is stored as: ['H', 'e', 'l', 'l', 'o', '\0'].

Q-7: Describe the following string functions in C-Language: 1) strcpy() 2) strcat() 3) strlen() 4) strcmp()

Answer:

(All these functions are defined in the string.h header file.)

1. strcpy() **- String Copy**
   - **Purpose:** Copies the entire contents of one string (including the null character) to another.
   - **Syntax:** char *strcpy(char *destination, const char *source);
   - **Example:**

```
#include <stdio.h>
#include <string.h>
int main() {
    char source[] = "Source String";
    char destination[20];
```

```
    strcpy(destination, source);
    printf("Destination string: %s", destination); // Output: Source String
    return 0;
}
```

2. strcat() **- String Concatenate**
    - **Purpose:** Appends a copy of the source string to the end of the destination string.
    - **Syntax:** char *strcat(char *destination, const char *source);
    - **Example:**

```
#include <stdio.h>
#include <string.h>
int main() {
    char dest[50] = "Hello ";
    char src[] = "World!";
    strcat(dest, src);
    printf("Concatenated string: %s", dest); // Output: Hello World!
    return 0;
}
```

3. strlen() **- String Length**
    - **Purpose:** Calculates the length of a string, excluding the null character.
    - **Syntax:** size_t strlen(const char *str);
    - **Example:**

```
#include <stdio.h>
#include <string.h>
int main() {
    char myString[] = "Programming";
    int length = strlen(myString);
```

```
    printf("Length of '%s' is %d", myString, length); // Output: Length of 'Programming' is 11
    return 0;
}
```

4. strcmp() **- String Compare**
    - **Purpose:** Compares two strings lexicographically (character by character based on ASCII values).
    - **Syntax:** int strcmp(const char *str1, const char *str2);
    - **Return Value:**
        - **0** if both strings are identical.
        - **< 0** if the first differing character in str1 has a lower ASCII value than in str2.
        - **> 0** if the first differing character in str1 has a higher ASCII value than in str2.
    - **Example:**

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[] = "apple";
    char str2[] = "banana";
    int result = strcmp(str1, str2);
    if (result < 0)
        printf("'%s' comes before '%s'\n", str1, str2);
    else if (result > 0)
        printf("'%s' comes after '%s'\n", str1, str2);
    else
        printf("Strings are equal.\n");
    // Output: 'apple' comes before 'banana'
    return 0;
}
```

Assignment – 2

Q-1: What is pointer ? Give its benefits. How to initialize Pointer? How is it different from Array?

Answer:

A pointer is a variable whose value is the memory address of another variable. It "points to" the location where data is stored, rather than storing the data itself.

- **Declaration Syntax:** data_type *pointer_name; (e.g., int *ptr;)
- **Benefits of Pointers:**
    1. **Dynamic Memory Allocation:** Pointers are essential for using functions like malloc(), calloc(), etc., allowing memory to be allocated at runtime.
    2. **Efficient Array Handling:** Accessing array elements using pointers is often faster than using subscript notation.
    3. **Call by Reference:** They allow functions to modify the actual arguments passed to them, enabling the passing of large structures efficiently without copying the entire structure.
    4. **Data Structures:** They form the foundation for complex data structures like linked lists, trees, graphs, etc.
    5. **System Programming:** Essential for direct memory access in low-level programming.
- **Initialization of a Pointer:**
  A pointer is initialized by assigning it the address of a variable using the address-of operator (&).
  Example:

```
#include <stdio.h>
int main() {
   int num = 10;
   int *ptr;       // Declaration of an integer pointer
   ptr = &num;     // Initialization: ptr now holds the address of num

   printf("Value of num: %d\n", num);     // Output: 10
```

```
    printf("Address of num: %p\n", &num);  // Output: (e.g., 0x7ffd3a5b8b4c)
    printf("Value of ptr: %p\n", ptr);     // Output: same as &num
    printf("Value pointed by ptr: %d\n", *ptr); // Output: 10 (Dereferencing)
    return 0;
}
```

- **Difference between Pointer and Array:**

Feature	Pointer	Array
Basic Definition	A variable that stores a memory address.	A collection of elements of the same data type.
Memory Allocation	Dynamic or static. Can be reassigned to point to different memory locations.	Static. Size is fixed at compile time. The name refers to a fixed block of memory.
Sizeof Operator	sizeof(ptr) returns the size of the pointer variable itself (e.g., 4 or 8 bytes).	sizeof(arr) returns the total memory allocated for the entire array (size_of_type * number_of_elements).
Reassignment	Can be incremented, decremented, or reassigned to point to a different address.	The array name is a constant pointer. Its value (the base address) cannot be changed.
Address Arithmetic	Supports pointer arithmetic (e.g., ptr++).	Array subscripting arr[i] is interpreted as pointer arithmetic *(arr + i).

Q-2: Explain array of pointers with suitable example.

Answer:

An array of pointers is an array whose elements are all pointers. Each element of this array holds the address of another variable, which could be an integer, a character, or even another array (like a string).

- **Declaration Syntax:** data_type *array_name[size];
- **Example:**
  This is commonly used to store an array of strings, where each string is a character array.

```
#include <stdio.h>
int main() {
   // An array of 3 character pointers
   char *fruits[3] = {"Apple", "Banana", "Cherry"};

   // Accessing and printing the strings
   for (int i = 0; i < 3; i++) {
      printf("Fruit[%d] = %s\n", i, fruits[i]);
      // fruits[i] is a pointer to the i-th string
   }
   return 0;
}
```

Output:

```
Fruit[0] = Apple
Fruit[1] = Banana
Fruit[2] = Cherry
```

In memory, fruits[0] points to the first character of "Apple", fruits[1] points to "Banana", and so on. This is more memory-efficient than a 2D character array when the strings are of different lengths.

Q-3: What is structure? How to access the elements of structure? How to calculate size of structure? Explain with example.

Answer:

A structure is a user-defined data type in C that allows grouping of variables of different data types under a single name.

- **Defining a Structure:** The struct keyword is used.

```
struct student {
   int roll_no;
   char name[50];
   float marks;
};
```

- **Accessing Structure Elements:**
  Structure members are accessed using the **dot operator (.)** for normal structure variables and the **arrow operator (->)** for structure pointers.
  Example:

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

// Define the structure
struct student {
   int roll_no;
   char name[50];
   float marks;
};

int main() {
   // Declare a structure variable
   struct student s1;
```

```
    // Accessing members using the dot operator
    s1.roll_no = 101;
    strcpy(s1.name, "Alice");
    s1.marks = 85.5;

    printf("Roll No: %d\n", s1.roll_no);
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);

    // Using a pointer and the arrow operator
    struct student *ptr = &s1;
    printf("\nAccessing via pointer:\n");
    printf("Roll No: %d\n", ptr->roll_no); // Same as (*ptr).roll_no

    return 0;
}
```

- **Calculating the Size of a Structure:**
  The sizeof operator is used to find the total memory occupied by a structure variable. The size is not always the simple sum of its members due to **structure padding**, where the compiler adds unused bytes between members to align data for efficient memory access.
  Example:

```
#include <stdio.h>
struct example {
    int a;    // 4 bytes
    char b;   // 1 byte
    double c; // 8 bytes
};
```

```
int main() {
    printf("Size of struct example: %zu bytes\n", sizeof(struct example));
    // Output might be 16 bytes due to padding, not 13 (4+1+8).
    return 0;
}
```

Q-4: Differentiate between structure and union.

Answer:

Feature	Structure (struct)	Union (union)
Memory Allocation	Each member has its own separate memory location.	All members share the same memory location.
Total Size	Size >= sum of sizes of all members. Due to padding.	Size = size of the largest member.
Member Access	All members can be accessed and hold values independently at any time.	Only one member can contain a meaningful value at a time. Accessing one member overwrites the others.
Purpose	To group different data types together to represent a record.	To provide memory efficiency by storing one of many data types at a time.
Example	struct Student { int id; char name[20]; float marks; };	union Data { int i; float f; char str[20]; };

Q-5: What is dynamic memory allocation? Explain important functions associated with it.

Answer:

Dynamic Memory Allocation (DMA) is the process of allocating memory during program execution (runtime) from the heap. This allows programs to request memory as needed and free it when done, providing flexibility in handling data of variable sizes.

The key functions for DMA, declared in stdlib.h, are:

1. malloc() **(Memory Allocation):**
    - **Purpose:** Allocates a single block of memory of a specified size.
    - **Syntax:** void *malloc(size_t size);
    - **Initialization:** The allocated memory is not initialized; it contains garbage values.
    - **Example:**

```
int *ptr;
ptr = (int *)malloc(5 * sizeof(int)); // Allocates memory for 5 integers
if (ptr == NULL) { // Always check if allocation was successful
   printf("Memory allocation failed!");
   exit(1);
}
```

2. calloc() **(Contiguous Allocation):**
    - **Purpose:** Allocates multiple blocks of memory, each of the same size, and initializes all bytes to zero.
    - **Syntax:** void *calloc(size_t num, size_t size);
    - **Example:**

```
int *ptr;
ptr = (int *)calloc(5, sizeof(int)); // Allocates & initializes memory for 5 integers to 0
```

3. realloc() **(Re-allocation):**
    - **Purpose:** Changes the size of a previously allocated memory block (can increase or decrease).
    - **Syntax:** void *realloc(void *ptr, size_t new_size);

- **Example:**

ptr = (int *)realloc(ptr, 10 * sizeof(int)); *// Resizes the array to hold 10 integers*

4. free()**:**
    - **Purpose:** De-allocates (releases) the memory previously allocated by malloc(), calloc(), or realloc(). This is crucial to prevent memory leaks.
    - **Syntax:** void free(void *ptr);
    - **Example:**

free(ptr); *// Releases the allocated memory*

ptr = NULL; *// Good practice to avoid dangling pointer*

Q-6: Describe file management. And List the various file management functions. Explain fopen () and its mode with example to write a string into file.

Answer:

File management in C involves performing operations on files stored in the secondary storage, such as creating, reading, writing, and updating files. It is done using file pointers of type FILE *.

- **Common File Management Functions:**
    - fopen() - Opens a file.
    - fclose() - Closes a file.
    - fprintf(), fscanf() - Formatted write and read.
    - fputc(), fgetc() - Character write and read.
    - fputs(), fgets() - String write and read.
    - fwrite(), fread() - Binary write and read.
    - fseek(), rewind() - Setting the file position.
- **The** fopen() **Function:**
    - **Purpose:** Opens a file and returns a pointer to the FILE structure associated with it.

- **Syntax:** FILE *fopen(const char *filename, const char *mode);

- **File Modes:**
  - "r" (Read): Opens an existing file for reading.
  - "w" (Write): Creates a new file for writing. If the file exists, its contents are erased.
  - "a" (Append): Opens a file for writing at the end. Creates the file if it doesn't exist.
  - "r+" (Read/Write): Opens an existing file for both reading and writing.
  - "w+" (Read/Write): Creates a new file for both reading and writing.
  - "a+" (Read/Append): Opens a file for reading and appending.
- **Example: Writing a string to a file using** fputs() **with** fopen()

```
#include <stdio.h>
int main() {
   FILE *filePointer;
   // Open a file in "write" mode. If "test.txt" exists, it will be overwritten.
   filePointer = fopen("test.txt", "w");

   // Check if file was opened successfully
   if (filePointer == NULL) {
      printf("File could not be opened.\n");
      return 1;
   }

   // Write a string to the file
   fputs("This is a string written to a file using fputs.\n", filePointer);
   // Alternatively, fprintf(filePointer, "This is a string...\n");

   // Close the file to save changes and free resources
```

```
    fclose(filePointer);
    printf("String successfully written to file.\n");
    return 0;
}
```

Q-7: Explain debugging techniques.

Answer:

Debugging is the process of identifying, isolating, and fixing errors (bugs) in a program. Effective techniques include:

1. **Understanding the Error:**
    - **Compile-Time Errors:** Check the compiler's error messages carefully. They indicate syntax violations and are usually easy to fix (e.g., missing semicolon, undeclared variable).
    - **Run-Time Errors:** These cause the program to crash (e.g., segmentation fault, division by zero). Isolate the line causing the crash.
    - **Logical Errors:** The program runs but produces incorrect output. These are the most challenging to find.
2. **Manual Code Review (Desk Checking):**
    - Read the code line by line logically, tracing the flow of execution and the values of variables. This helps catch obvious logical mistakes.
3. **Printf() Debugging:**
    - This is the most common and straightforward technique.
      Insert printf() statements at strategic points in the code to display the values of variables and track the program's flow.
    - **Example:**

```
printf("DEBUG: Entering function calculate(). Value of x = %d\n", x);
// ... some calculation ...
printf("DEBUG: After calculation. Value of result = %d\n", result);
```

4. **Using a Debugger (GDB - GNU Debugger):**

- A debugger is a powerful tool that allows you to control the execution of your program and inspect its state.
- **Key Commands:**
  - gcc -g program.c -o program (Compile with debug info)
  - gdb ./program (Start GDB)
  - break main (Set a breakpoint at main)
  - run (Start execution)
  - next (Execute next line)
  - step (Step into a function)
  - print variable_name (Print variable value)
  - continue (Continue execution until next breakpoint)
  - quit (Exit GDB)

5. **Static Code Analysis Tools:**
   - Use tools like splint, cppcheck, or features within modern IDEs that can analyze code for potential errors, memory leaks, and coding standard violations without even running the program.