Internal Assessment Question Paper – 1

M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of CSE

Programme: B.E | **Term: Oct 2024 – Jan 2025**
Course: Principles of Programming in C | **Course Code: PPC18**
Sem: I | **CIE: I** | **Section: All**
Max Marks: 30 | **Time:** 1Hr | **Portions for Test:** L1-L13

Scheme

Question 1 is compulsory. Answer any one from question 2 and 3.

Sl #		Question	Marks	CO Mapping

1	a)	i. Value of z:
		z = x / y;
		Given x = 2, y = 3 (both integers), the division will result in integer division. Hence, z = 0.
		To ensure floating-point division, type casting is necessary: z = (float)x / y.
		
		ii. Value of y:
		y = --a * (3 + b) / 2 - c++ * b;
		Steps:
		- a is pre-decremented: a = 2.
		- b = 4, so 3 + b = 7.
		2*7/2-5*4
		14/2-20
		7-20
		=-13
		
		iii. Output of switch statement:
		number = 5;
		The case will fall into 4, 5, or 6 and print: 'Four, Five, or Six.'.
		
		iv. Output of while loop:
		Infinite loop. count is initialized as 1 and decremented (count = count - 1), causing it to never reach the termination condition.
		
		v. Error in for loop:
		The loop decrements (i--) instead of incrementing, resulting in an infinite loop. The semicolon after the loop header causes the loop to do nothing but iterate endlessly.
			One mark each (5)	**CO1,CO2**

	b)	**i. Uninitialized elements of the array will automatically be set to 0.** In C, when you partially initialize an array (e.g., providing values for only some of its elements), the remaining uninitialized elements are automatically set to zero. This behavior is part of the standard C language specification. For example: c Copy code	1+1+2+1	CO3

		int arr[5] = {1, 2};
		Here:
		• The first two elements (arr[0] and arr[1]) are explicitly initialized to 1 and 2, respectively.
		• The remaining three elements (arr[2], arr[3], and arr[4]) are automatically set to 0.
		This is particularly useful when you want to initialize only certain elements and ensure the rest of the array starts with default zero values.
		
		ii. Yes, the declaration is valid. If fewer elements are initialized, the remaining ones are set to 0.
		This means that arrays in C do not necessarily need all their elements explicitly initialized. For example:
		int arr[5] = {1, 2};
		This is a valid declaration:
		• It defines an array of size 5.
		• Initializes the first two elements (arr[0] = 1, arr[1] = 2).
		• Leaves the remaining three elements uninitialized. As explained above, the C language automatically sets them to 0.
		Such behavior simplifies initialization when you are only concerned about setting specific elements, making the code cleaner and more concise.
		
		iii. Modified code to initialize all elements to 0:
		int arr[5] = {0};
		This declaration is an efficient way to initialize **all elements of the array** to 0:
		• By providing just a single 0 within the curly braces, the C compiler sets the first element to 0 and automatically sets all other elements to 0 as well.
		For example:
		int arr[5] = {0};
		This results in the array:
		arr[0] = 0
		arr[1] = 0
		arr[2] = 0
		arr[3] = 0
		arr[4] = 0
		This approach is particularly useful for arrays with many elements (e.g., int arr[1000]), as you don't need to explicitly write out 0 for each element or use a loop to initialize them.
		Key Point:
		If you wanted to initialize all elements to a non-zero value, you would need to use a loop. For example:
		int arr[5];
		for (int i = 0; i < 5; i++) {
		arr[i] = 1; // Set each element to 1
		}
		iv. Output of the code: 1 2 3 0 0.		

	c)	**1. Declaration of an Array** The **syntax** for declaring an array in C is: c Copy code	**2.5+2.5**	**CO3**

		data_type array_name[array_size];
		• **data_type**: Type of data the array will hold (e.g., int, float, char, etc.).
		• **array_name**: Name of the array.
		• **array_size**: The number of elements in the array (a positive integer).
		Example:
		c
		Copy code
		int numbers[5]; // Declares an array of 5 integers.
		float grades[10]; // Declares an array of 10 floats.
		char letters[26]; // Declares an array of 26 characters.
		
		An array is a data structure in programming that stores a fixed-size sequential collection of elements of the same data type. Each element in the array can be accessed using an index, which starts from 0 in most programming languages, including C.
		a) Initialization During Declaration
		You can initialize an array when you declare it by providing values inside curly braces {}.
		Syntax:
		data_type array_name[array_size] = {value1, value2, ..., valueN};
		• The number of values in {} must not exceed array_size.
		Example:
		int numbers[5] = {10, 20, 30, 40, 50}; // Initialize array with specific values.
		float grades[3] = {9.5, 8.0, 7.2}; // Initialize array with float values.
		char vowels[5] = {'a', 'e', 'i', 'o', 'u'}; // Initialize array of characters.
		If you initialize fewer elements than the declared size, the remaining elements are automatically set to 0:
		int numbers[5] = {10, 20}; // numbers[2], numbers[3], and numbers[4] are set to 0.
		If the array size is omitted, it is automatically determined by the number of elements provided:
		c
		Copy code
		int numbers[] = {10, 20, 30}; // Array size is automatically set to 3.
		
		b) Initialization After Declaration
		You can also initialize an array after it has been declared by assigning values to individual elements using their **index**.
		Syntax:
		array_name[index] = value;
		Example:
		int numbers[5];
		numbers[0] = 10; // Initialize the first element.
		numbers[1] = 20; // Initialize the second element.
		numbers[2] = 30; // Initialize the third element.
		numbers[3] = 40; // Initialize the fourth element.
		numbers[4] = 50; // Initialize the fifth element.		

2	**a)**	**structure of a C Program**	**5**	**CO1**

A C program follows a standard structure consisting of several sections. Below is a detailed breakdown of the structure, along with examples.

1. Preprocessor Directives
Preprocessor directives are lines that begin with # and are used to include files or define constants before the program is compiled.
Example:
```
#include <stdio.h>  // Includes the standard input-output library
#define PI 3.14     // Defines a constant named PI
```

2. Global Declarations
Global variables and function prototypes are declared outside any function. These variables or functions can be accessed by all functions in the program.
Example:
```
int global_variable = 10;  // Global variable declaration

void myFunction();  // Function prototype
```

3. The main() Function
The main() function is the **entry point** of every C program. Execution of the program begins from this function. It must return an int value (commonly 0 to indicate successful execution).
Syntax:
```
int main() {
    // Code here
    return 0;
}
```

4. Variable Declarations
Variables are declared inside the main() function or any other user-defined functions. They are used to store data during program execution.
Example:
```
int x = 5;    // Declares and initializes an integer variable
float y = 3.2; // Declares and initializes a float variable
```

5. Input/Output Statements
C programs use functions like printf() for output and scanf() for input.
Example:
```
int num;
printf("Enter a number: ");  // Prompt for input
scanf("%d", &num);           // Reads input from the user
printf("You entered: %d", num);  // Prints the input
```

6. User-Defined Functions
User-defined functions are additional functions created by the programmer to modularize and simplify the program. They are called from the main() function or other functions.
Syntax:
```
return_type function_name(parameters) {
    // Function code
    return value;
```

}

Example:

c

Copy code

```
void greet() {  // A function with no return type
   printf("Hello, World!\n");
}
```

7. Comments

Comments are used to improve code readability. They are ignored by the compiler.

- **Single-line comment:** // This is a comment
- **Multi-line comment:**

b)

In C, **formatted input and output statements** allow us to display information to the user and accept input in a structured format. These statements are provided by the <stdio.h> library and are among the most commonly used functions.

1. Formatted Output: printf()

The printf() function is used to display (output) information in a formatted manner.

Syntax:

c

Copy code

```
printf("format string", variable1, variable2, ...);
```

Components:

- **Format string**: Specifies how the data should be displayed. It may include:
  - Plain text.
  - Format specifiers (placeholders for variables, starting with %).
- **Variables**: The values to be displayed in the positions specified by the format specifiers.

2.5+2.5

CO1

Common Format Specifiers:

Format Specifier	Description
%d	Integer (decimal)
%f	Float
%c	Character
%s	String
%x	Hexadecimal (lowercase)
%o	Octal

```
int age = 25;
float height = 5.9;
char grade = 'A';
char name[] = "John";

// Formatted output
printf("Name: %s\n", name);
```

```
    printf("Age: %d\n", age);
    printf("Height: %.1f\n", height);  // Displays 1 digit after the decimal
point
    printf("Grade: %c\n", grade);
```

2. Formatted Input: scanf()

The scanf() function is used to take (input) data from the user in a formatted manner.

Syntax:

scanf("format string", &variable1, &variable2, ...);

Components:

- **Format string**: Specifies the type of data to be read using format specifiers (same as in printf()).
- **&variable**: The **address-of operator** (&) is used to store the input value into the variable.

Example:

```
int age;
    scanf("%d", &age);
```

	c)	**Different Data Types in C**
		C provides several built-in data types that can be classified into the following categories:
		1. Basic Data Types
		• **int**: Used to store integer values (whole numbers).
		  ○ Size: Typically 4 bytes (varies by system).
		  ○ Example: int a = 10;
		• **float**: Used to store floating-point numbers (numbers with decimal points).
		  ○ Size: Typically 4 bytes.
		  ○ Example: float b = 3.14;
		• **double**: Used to store double-precision floating-point numbers. It offers more precision than float.
		  ○ Size: Typically 8 bytes.
		  ○ Example: double c = 3.14159;
		• **char**: Used to store single characters.
		  ○ Size: Typically 1 byte.
		  ○ Example: char d = 'A';
		2. C Program to Calculate the Area and Circumference of a Circle Using PI as a Defined Constant
		C Program:
			2+3	**CO1**

```
#include <stdio.h>
#define PI 3.14159  // Define PI as a constant
int main() {
    float radius, area, circumference;
    printf("Enter the radius of the circle: ");
    scanf("%f", &radius);
    area = PI * radius * radius;  // Area formula
    circumference = 2 * PI * radius;  // Circumference formula
    printf("Area of the circle: %.2f\n", area);
    printf("Circumference of the circle: %.2f\n", circumference);
    return 0;
```

		}		

3	a)	**1. Implicit Type Conversion (Automatic Type Conversion)**
		Definition: Implicit type conversion is automatically performed by the compiler when it detects that a variable of one type is assigned to a variable of another type. This conversion is done **without the need for explicit instructions** from the programmer.
		Also known as: **Type coercion**.
		When it occurs: It happens when a lower data type (e.g., int) is assigned to a higher data type (e.g., float), or when the types are compatible.
		Example: Converting int to float automatically.
		Example of Implicit Conversion:
		#include <stdio.h>
		int main() {
		int a = 5;
		float b; // Implicit conversion from int to float
		b = a;
		printf("The value of b: %f\n", b); // Output will be 5.000000
		return 0;
		}
		Explanation: The compiler automatically converts the integer value a to a floating-point value when it is assigned to b.
		2. Explicit Type Conversion (Type Casting)
		Definition: Explicit type conversion, or type casting, requires the programmer to manually specify the type conversion using casting syntax.
		When it occurs: It happens when you want to convert a value to a different data type that is not automatically handled by C.
		Syntax: (new_data_type) variable
		Example of Explicit Conversion:
		#include <stdio.h>
		int main() {
		float a = 9.99;
		int b; // Explicit conversion from float to int using type casting
		b = (int)a; // Type casting explicitly converts the float value 'a' to an integer
		printf("The value of b: %d\n", b); // Output will be 9
		return 0;
		}
		Program
		1. #include<stdio.h>
		2. int main()
		3. {
		4. int n,sum=0,m;
		5. printf("Enter a number:");
		6. scanf("%d",&n);
		7. while(n>0)
		8. {
		9. m=n%10;
		10. sum=sum+m;
		11. n=n/10;
		12. }
		13. printf("Sum is=%d",sum);
		14. return 0;
		15. }
			2.5+2.5	**CO1**

b) Loops in C can be categorized as **pretest** or **post-test** loops based on when the loop condition is tested.

1. Pretest Loop (Condition Tested Before Loop Execution)

Definition: A pretest loop tests the condition **before** executing the loop body. If the condition is false initially, the loop body may not be executed at all.

Examples: for loop and while loop.

General Syntax:

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

or

```
for (initialization; condition; increment/decrement) {
   // Code to be executed
}
```

When to Use: Use pretest loops when you want to **test the condition before executing** the loop body.

Example of Pretest Loop (while loop):

```
#include <stdio.h>
int main() {
   int i = 1;
   while (i <= 10) {  // Pretest: Condition checked before loop execution
      printf("%d ", i);
      i++;
   }
   return 0;
}
```

Explanation:

- The loop will print the numbers from 1 to 10.
- The condition i <= 10 is checked **before** each iteration. If it's false, the loop will stop.

Output:

Copy code

1 2 3 4 5 6 7 8 9 10

2. Post-Test Loop (Condition Tested After Loop Execution)

Definition: A post-test loop tests the condition **after** executing the loop body. This means the loop body will always execute **at least once** before the condition is tested.

Example: do-while loop.

General Syntax:

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

- **When to Use**: Use post-test loops when you want the loop body to execute **at least once** regardless of the condition.

Example of Post-test Loop (do-while loop):

```
#include <stdio.h>
int main() {
   int i = 1;
   do {
      printf("%d ", i);
      i++;
   } while (i <= 10);  // Posttest: Condition checked after loop execution
   return 0;
}
```

Explanation:

- The loop will print the numbers from 1 to 10.

1.5+1.5+2 | **CO2**

		• The condition i <= 10 is checked **after** each iteration. The body of the loop will be executed **at least once**.
		Output:
		1 2 3 4 5 6 7 8 9 10
		Program to Calculate the Sum of the First 10 Natural Numbers Using a Pretest Loop
		This program will use a **for loop** (pretest loop) to calculate the sum of the first 10 natural numbers.
		Program:
		#include <stdio.h>
		int main() {
		int sum = 0;
		// Pretest loop (for loop) to calculate the sum of first 10 natural numbers
		for (int i = 1; i <= 10; i++) {
		sum += i; // Add the current number to sum
		}
		printf("The sum of the first 10 natural numbers is: %d\n", sum);
		return 0;
		}		

	c)	In C, **conditional statements** allow the program to execute different blocks of code based on certain conditions. The main conditional statements in C are:
		1. **if statement**
		2. **if-else statement**
		3. **else-if ladder**
		4. **switch statement**
		
		1. if Statement
		The if statement executes a block of code only if a specified condition is true.
		Syntax:
		if (condition) {
		// Code to be executed if condition is true
		}
		Example:
		#include <stdio.h>
		
		int main() {
		int num = 5;
		if (num > 0) {
		printf("The number is positive.\n");
		}
		return 0;
		}
		Output:
		The number is positive.
		2. if-else Statement
		The if-else statement executes one block of code if the condition is true, and a different block of code if the condition is false.
		Syntax:
		if (condition) {
		// Code if condition is true
		} else {
			1.25 EACH	**CO2**

```
    // Code if condition is false
}
```

Example:

```
#include <stdio.h>
int main() {
    int num = -5;
    if (num >= 0) {
        printf("The number is non-negative.\n");
    } else {
        printf("The number is negative.\n");
    }
    return 0;
}
```

Output:

The number is negative.

3. else-if Ladder

The else-if ladder allows multiple conditions to be checked, and different blocks of code will be executed based on which condition is true.

Syntax:

```
if (condition1) {
    // Code if condition1 is true
} else if (condition2) {
    // Code if condition2 is true
} else {
    // Code if none of the conditions are true
}
```

Example:

```
#include <stdio.h>

int main() {
    int num = 0;
    if (num > 0) {
        printf("The number is positive.\n");
    } else if (num < 0) {
        printf("The number is negative.\n");
    } else {
        printf("The number is zero.\n");
    }
    return 0;
}
```

Output:

The number is zero.

4. switch Statement

The switch statement tests a variable against multiple possible values. It executes the block of code corresponding to the matched value.

Syntax:

```
switch (expression) {
    case value1:        // Code to execute if expression == value1
        break;
    case value2:        // Code to execute if expression == value2
        break;
    default:        // Code to execute if no case matches
```

}

Example:

```
#include <stdio.h>
int main() {
   int day = 3;
   switch (day) {
      case 1:
         printf("Monday\n");
         break;
      case 2:
         printf("Tuesday\n");
         break;
      case 3:
         printf("Wednesday\n");
         break;
      default:
         printf("Invalid day\n");
   }
   return 0;
}
```

Output:

Wednesday

Course Outcomes meant to be assessed by the IA Test-I:

- CO1: Identify the basic elements of Computing Systems and C Programming Constructs.
- CO2: Demonstrate the use of Operators & Expressions, Decision Making and Looping Statements.
- CO3: Explore Arrays and User-Defined Functions in Implementing Solutions to Real worldProblems

---

29/11/24

29/11/24