USN | | | | | | | | | | |

Internal Assessment Test 2 – January 2026

Sub	Programming in C					Sub code	1BEIT105	Branch	ISE, AIDS, AIML, CSML
Date	07/01/26	Duration	90 mins	MaxMarks	50	Sem /Sec	I Sem P Cycle (A, B, C, D, E, F, G, H)		OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1.	a) Explain the importance of strcmp() and strcat() string manipulation fucntions b) Write a C program to implement the string copy operation strcpy(str1, str2) that copies a string str1 to another string str2 without using library function.	[5] [5]	CO1 CO1	L1 L1
2.	a) What is a function? Explain the different types of functions based on parameters and return types. b) Write a program to find the GCD and LCM of two numbers using functions.	[5] [5]	CO1 CO3	L3
3.	a) Explain argc, argv. b) Explain a pointer to a function. Write a C program that declares a function to add two integers and another function to subtract two integers. Use a function pointer to call both functions dynamically and print their results.	[3] [7]	CO1 CO3	L3

CI CCI HOD

4.	a)Differentiate: i) User- defined and built-in function ii) Recursion and iteration b) Write a C function isprime(num) that accepts an integer argument and return 1 if the argument is a prime or 0 otherwise such that to generate prime number between a given range.	[5] [5]	CO2 CO2	L3 L3
5.	What is a pointer? Define Pointer Variable and Pointer Operators with example.	[10]	CO1	L2
6.	a) Give advantages and disadvantages of pointers in C. b) Write a C program to find sum and mean of all elements in an array using pointer.	[5] [5]	CO2 CO3	L3
7.	a) Write note on: i) arrays within structure ii) array of structures b) Create a structure that contains student name (sname) and student marks(smarks). Write a C program that reads the names and marks of two students and compares whether both students have the same marks or not.	[5] [5]	CO4	L2
8.	Explain: a) Unions b)Enumerations c) Typedef	[10]	CO4	L2

CI CCI /Anchor Teacher HOD

CO-PO and CO-PSO Mapping

Course Outcomes		Blooms Level	Modules covered	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	Demonstrate fundamental concepts and language constructs of C programming.	L2	M1	3	2	1	-	1	-	-	-	-	-	-	2	3	-	-	-
CO2	Make use of control structures and arrays to solve basic computational problems.	L3	M2	3	3	2	1	-	-	-	-	1	-	-	2	3	-	-	-
CO3	Develop modular programs using user-defined functions for complex computational problems.	L3	M3	3	3	3	-	2	-	-	-	2	-	-	2	3	-	3	-
CO4	Construct user defined datatypes using structures, unions and enumerations to model simple realworld scenarios.	L3	M4	3	2	3	-	-	-	1	-	-	-	-	2	3	-	3	-
CO5	Choose suitable datatypes and language constructs to solve a given computational or real-world problem	L3	M5	3	3	3	2	-	1	-	-	-	-	-	3	3	-	3	2

COGNITIVE LEVEL	REVISED BLOOMS TAXONOMY KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PO1	Engineering knowledge	PO7	Environment and sustainability	0	No Correlation
PO2	Problem analysis	PO8	Ethics	1	Slight/Low
PO3	Design/development of solutions	PO9	Individual and team work	2	Moderate/ Medium
PO4	Conduct investigations of complex problems	PO10	Communication	3	Substantial/ High
PO5	Modern tool usage	PO11	Project management and finance		
PO6	The Engineer and society	PO12	Life-long learning		
PSO1	Design and develop applications using different stacks of web and programming technologies.				
PSO2	Design and develop secure, parallel, distributed, networked, and digital systems.				
PSO3	Apply software engineering methods to design, develop, test and manage software systems.				
PSO4	Design and develop intelligent applications for business and industry.				

1a. Explain the importance of strcmp() and strcat() string manipulation fucntions

Explanation of strcmp- 2.5 marks, strcat-2.5 marks

In C programming, strcmp() and strcat() are fundamental string manipulation functions provided by the <string.h> library. Their importance lies in enabling effective comparison and construction of strings, which are core operations in most real-world C applications.

- **strcmp()** is used to **compare two strings** character by character, based on ASCII values **Syntax: int strcmp(const char *str1, const char *str2);**
- The comparison results show that if all the characters match, the result is 0, and the output is STRINGS ARE EQUAL.
- Lets take str1 & str2, if strcmp(str1,str2)<0 which means is str1 less than str2.
- Else if(strcmp(str1,str2)>0 str1 is greater than str2.
- In C, strings cannot be compared using ==. strcmp() provides a reliable way to check string equality or ordering.
- The strcmp() functionality is used in Username/password validation, Command matching, and Menu-driven programs.
- Essential for Alphabetical sorting of names, searching records in databases or files.
- **strcat()** is used to **append (concatenate) one string to the end of another**.
- **Syntax: char *strcat(char *dest, const char *src);**
- Strcat() is used for Dynamic String Construction that allows combining multiple strings into a single meaningful output.
- It is also used for building messages, constructing file paths, generating logs and reports.
- Enables string building and output formatting.

1b. Write a small code that supports the string functionality --- [5marks]

```
#include <stdio.h>

int main()

{

   char str1[100], str2[100];

   int i = 0;

   printf("Enter the source string: ");

   scanf("%s", str1);

   //gets(str1);   // reads string into str1

   /* Copy str1 to str2 */

   while (str1[i] != '\0')

   {

      str2[i] = str1[i];

      i++;

   }
```

```
    //str2[i] = '\0';   // add null terminator

    printf("Source String (str1): %s\n", str1);
    printf("Copied String (str2): %s\n", str2);
    return 0;
}
```

Output: **Enter the source string: sowmya**

Source String (str1): sowmya

Copied String (str2): sowmya

2a. What is a function? Explain the different types of functions based on parameters and return types.

Function definition- [1.5marks]

A function in C is a self-contained block of statements that performs a specific task. A function is executed when it is called from another part of the program. Functions help in modular programming, improve code readability, enable code reuse, and make programs easier to test and maintain.

Syntax: return_type function_name(parameter_list)

{

// function body

}

Types Explanation- [3.5marks]:

Types of Functions Based on Parameters and Return Types: Functions are commonly divided into 4 types:

1. Function with No Arguments and No Return Value

- Does not take any input.
- Does not return any output.
- All processing is done inside the function.

Syntax: void function_name(void);

2. Function with Arguments but No Return Value

- Takes input parameters.
- Performs operations using the parameters.
- Does not return a value.

Syntax: void function_name(data_type arg1, data_type arg2);

3. Function with No Arguments but Return Value

- Does not take input parameters.
- Returns a value after computation.

Syntax: data_type function_name(void);

4. Function with Arguments and Return Value

- Takes input parameters.
- Returns a computed value.
- Most commonly used type.

Syntax: data_type function_name(data_type arg1, data_type arg2);

One program that satisfy the above theory consider as an example.

2b. Write a program to find the GCD and LCM of two numbers using functions – [5marks]

```
#include <stdio.h>
/* Function to find GCD using Euclidean algorithm */
int findGCD(int a, int b)
{
   int temp;
   while (b != 0)
   {
      temp = b;
      b = a % b;
      a = temp;
   }
   return a;
}
/* Function to find LCM using GCD */
int findLCM(int a, int b)
{
   return (a * b) / findGCD(a, b);
}
```

```
int main()
{
   int num1, num2, gcd, lcm;

   printf("Enter two numbers: ");
   scanf("%d %d", &num1, &num2);

   gcd = findGCD(num1, num2);
   lcm = findLCM(num1, num2);

   printf("GCD of %d and %d = %d\n", num1, num2, gcd);
   printf("LCM of %d and %d = %d\n", num1, num2, lcm);

   return 0;
}
```

Output: Enter two numbers: 20 30

GCD of 20 and 30 = 10

LCM of 20 and 30 = 60

3a). Explain argc, argv – [3marks]

In C, **argc** and **argv** are parameters of the main() function that allow a program to receive **command-line arguments**. These arguments are supplied when the program is executed from the command prompt or terminal.

1. argc (Argument Count)

- **argc** stands for *argument count*.
- It stores the **total number of command-line arguments**, including the program name.
- The value of argc is always **at least 1**, because the program name itself is counted.

Syntax: int main(int argc, char *argv[])

Example: Sowmya 20 40

Here:

- argc = 3
    - argv[0] → program name ("Sowmya")

- argv[1] → "20"
- argv[2] → "40"

2. argv (Argument Vector)

- **argv** stands for *argument vector*.
- It is an **array of character pointers** (char *argv[]).
- Each element of argv points to a string representing a command-line argument.
- `argv[0]` always contains the **program name**.
- `argv[argc]` is always `NULL` (end marker).
- All command-line arguments are treated as **strings**.

Example:

```
#include <stdio.h>

int main(int argc, char *argv[])

{

   int i;

   printf("Total number of arguments: %d\n", argc);

   for (i = 0; i < argc; i++)

   {

      printf("argv[%d] = %s\n", i, argv[i]);

   }

   return 0;

}
```

Output: program Hello C Language

Total number of arguments: 4

argv[0] = program

argv[1] = Hello

argv[2] = C

argv[3] = Language

3b) Explain a pointer to a function. Write a C program that declares a function to add two integers and another function to subtract two integers. Use a function pointer to call both functions dynamically and print their results.

Pointer Definition: [2 marks]

Ans: A **pointer to a function** is a variable that stores the **address of a function**. Using function pointers, a program can call different functions **dynamically at runtime**, rather than calling them directly by name. Using function pointers, a program can call different functions **dynamically at runtime**, rather than calling them directly by name.

Function pointers are widely used in:

- Callback mechanisms
- Menu-driven programs
- Implementing dynamic behavior
- Passing functions as arguments to other functions

Syntax: return_type (*pointer_name)(parameter_types);

Program ---- [5marks]

```
#include <stdio.h>

/* Function to add two integers */
int add(int a, int b)
{
   return a + b;
}

/* Function to subtract two integers */
int subtract(int a, int b)
{
   return a - b;
}

int main()
{
   int x = 20, y = 10;
   int result;

   /* Declare a function pointer */
```

```
    int (*fp)(int, int);

    /* Point to add function and call it */
    fp = add;
    result = fp(x, y);
    printf("Addition: %d\n", result);

    /* Point to subtract function and call it */
    fp = subtract;
    result = fp(x, y);
    printf("Subtraction: %d\n", result);
    return 0;
}
```

Output: Addition: 30

Subtraction: 10

4a) Differentiate: i) User- defined and built-in functions – [2.5marks]

ii) Recursion and iteration – [2.5marks]

User- defined Functions	built-in function
These are the functions that are created by the programmer or USer to perform specific tasks	Functions already provided by the C standard library
These functions are Highly flexible and can be customized as per program needs	These functions are Limited to the functionality provided.
These functions can be reused, across **user programs**	Reusable across all C programs irrespective of the user.
There is no library required for developing these functions.	Requires header files e.g., <stdio.h>, <string.h>
Can be modified by the programmer	Cannot be modified
Some of user defined functions like add(), factorial()	Some of built in functions like printf(), scanf(), strlen()

Recursion	Iteration
Recursion is A function calling itself to solve a problem	Iteration is Repeated execution of a set of statements using loops
Uses function calls	Uses loops (for, while, do-while).
Requires a base condition	Requires a loop condition

Uses more memory due to function call stack	Uses less memory
Slower due to overhead of function calls	Faster compared to recursion
Example: Problems like factorial, Fibonacci, tree traversal	Example: Problems with simple repetition

4b). Write a C function isprime(num) that accepts an integer argument and return 1 if the argument is a prime or 0 otherwise such that to generate prime number between a given range.

Program- [5marks]

```
#include <stdio.h>

/* Function to check whether a number is prime */

int isprime(int num)

{

   int i;

   if (num <= 1)

      return 0;

   for (i = 2; i <= num / 2; i++)

   {

      if (num % i == 0)

         return 0;

   }

   return 1;

}


int main()

{

   int start, end, i;


   printf("Enter the range: ");

   scanf("%d %d", &start, &end);


   printf("Prime numbers between %d and %d are:\n", start, end);
```

```
for (i = start; i <= end; i++)
{
    if (isprime(i))
        printf("%d ", i);
}   return 0;   }
```

Output: Enter the range: 1

20

Prime numbers between 1 and 20 are:

2 3 5 7 11 13 17 19

5. What is a pointer? Define Pointer Variable and Pointer Operators with an example.

Pointer definition – 2Marks + Pointer Variable – 3 marks + Pointer Operators – 3 marks + Example—2marks.

A pointer is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory.

For example, if one variable contains the address of another variable, the first variable is said to point to the second.

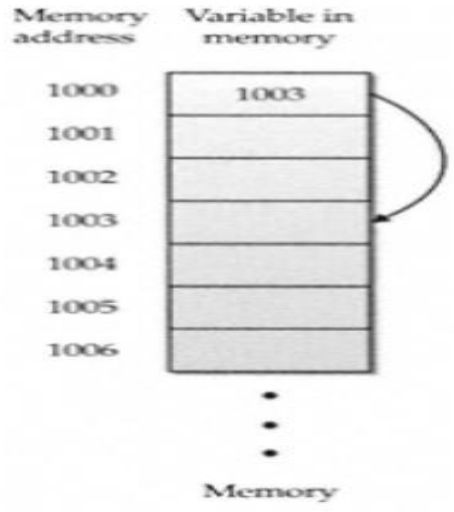

Figure 5-1
One variable points to another

Pointer Variable:

- A pointer declaration consists of a base type, an *, and the variable name.

- The general form for declaring a pointer variable is type *name;

where type is the base type(Data Type) of the pointer and may be any valid type. The name of the pointer variable is specified by name. We can use any variable name for the pointer that is meaningful.

- All pointer operations are done relative to the pointer's base type.
- For example, when you declare a pointer to be of type int *, the compiler assumes that any address that it holds points to an integer, whether it does or not.
- That is, an int * pointer always ''thinks" that it points to an int object, no matter what that piece of memory actually contains.
- Therefore, when you declare a pointer, you must make sure that its type is compatible with the type of object to which you want to point.

If we declare an array like this int *p; The compiler **assumes** that: The memory address stored in p points to an **int** Every time you do p + 1, it should move ahead by **sizeof(int)** bytes (typically 4 bytes). Every time you dereference (*p), it should read **4 bytes** and interpret it as an **integer**

For example, float f = 10.23

Int *a; p=(int *)&f; printf(“%d”, *p);

Here we are forcing int pointer to point to float value

float is stored in 4 bytes, using float format (IEEE 754)

int *p reads those same 4 bytes **as an integer**

Output will be garbage/unexpected
→ because float bytes ≠ int bytes. The pointer “thinks” it's pointing to an int, even though it’s not.

Correct Code is: float f = 3.14;

```
float *p = &f;   // Correct pointer type
printf("%f", *p);
```

Pointer Operators:

There are two pointer operators: * and &. The & is a unary operator that returns the memory address of its operand.

For example, m = &count; places into m the memory address of the variable count.

To understand the above assignment better, assume that the variable count uses memory location 2000 to store its value. Also consider that the count has a value of 100.

Then, after the preceding assignment, m will have the value 2000.

The second pointer operator, *, is the complement of &.

It is a unary operator that returns the value located at the address that follows. For example, if m contains the memory address of the variable count, q = *m; places the value of count into q.

Thus, q will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in m.

```
#include <stdio.h>


int main()
{
   int num = 10;    // normal variable
   int *ptr;       // pointer variable


   ptr = &num;     // & operator: stores address of num in ptr


   printf("Value of num = %d\n", num);
   printf("Address of num = %p\n", &num);
   printf("Value stored in ptr = %p\n", ptr);
   printf("Value pointed by ptr = %d\n", *ptr);  // * operator: access value


   return 0;
}
```

Output: Value of num = 10

Address of num = 0x7ffdfc2a

Value stored in ptr = 0x7ffdfc2a

Value pointed by ptr = 10

6a). Give advantages and disadvantages of pointers in C.

Advantages of Pointers in C --- 2.5marks

1. **Efficient Memory Access**
   Pointers allow direct access to memory locations, making programs faster and more efficient.
2. **Call by Reference**
   Functions can modify actual variables by passing their addresses, avoiding unnecessary copying of data.
3. **Dynamic Memory Allocation**
   Pointers are essential for using dynamic memory functions such as malloc(), calloc(), and free().

4. **Efficient Array and String Handling**
   Arrays and strings are internally handled using pointers, enabling faster traversal and manipulation.
5. **Implementation of Data Structures**
   Pointers are required to create advanced data structures like linked lists, stacks, queues, trees, and graphs.
6. **Memory Sharing**
   Multiple pointers can point to the same memory location, enabling efficient data sharing.
7. **Low-Level System Programming**
   Pointers allow interaction with hardware, memory-mapped I/O, and operating system resources.

Disadvantages of Pointers in C ----2.5marks

1. **Complexity**
   Pointer concepts are difficult to understand for beginners and can make programs hard to read.
2. **Risk of Memory Errors**
   Incorrect use may lead to:
   - Segmentation faults
   - Memory leaks
   - Dangling pointers
3. **Debugging Difficulty**
   Pointer-related errors are often hard to detect and debug.
4. **Security Issues**
   Improper pointer usage can cause buffer overflows and vulnerabilities.
5. **No Automatic Bounds Checking**
   C does not check memory boundaries, increasing the risk of accessing invalid memory.
6. **Maintenance Challenges**
   Code using excessive pointers can be harder to maintain and modify.

6b). b) Write a C program to find sum and mean of all elements in an array using pointer – 5marks

```
#include <stdio.h>

int main()

{

   int arr[100];

   int n, i;

   int sum = 0;
```

```
    float mean;
    int *ptr;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    ptr = arr;   // pointer points to first element of array
    /* Calculate sum using pointer */
    for (i = 0; i < n; i++)
    {
        sum = sum + *(ptr + i);
    }
    mean = (float)sum / n;
    printf("Sum of elements = %d\n", sum);
    printf("Mean of elements = %.2f\n", mean);
    return 0;
}
```

OUTPUT:

Enter number of elements: 5

Enter 5 elements:

10 20 30 40 50

Sum of elements = 150

Mean of elements = 30.00

7a). Write note on: i) arrays within structure ii) array of structures

Arrayswithin structure –2.5marks and array of structures – 2.5marks

- A member of a structure can be either a simple variable, such as an int or double, or an aggregate type.
- In C, aggregate types are arrays and structures. An aggregate type can store multiple values.

- For example, consider this structure:

struct x {

int a[10] [10]; /* 10 x 10 array of ints a 2d array */

float b;

} y;

- To reference integer 3,7 in a of structure y, write : y.a[3][7] → this prints the value at 3rd row 7th column of a
- When a structure is a member of another structure, it is called a nested structure.
- For example, the structure address is nested inside emp in this example:

struct emp {

struct addr address; /* nested structure */

float wage;

} worker;

- Here, the structure emp has been defined as having two members.
- The first is a structure of type addr, which contains an employee's address. (lets assume a structure addr having address, city,zip etc like members)
- The other is wage, which holds the employee's wage.
- As you can see, the members of each structure are referenced from outermost to innermost.
- The C89 standard specifies that structures can be nested to at least 15 levels.
- The C99 standard suggests that at least 63 levels of nesting be allowed.

Example:

```
#include <stdio.h>
#include<string.h>
struct Address {
   char city[20];
   int pincode;
};
struct Student {
   int roll;
   struct Address addr;   // nested structure
};
int main() {
```

```
   struct Student s;
   s.roll = 101;
   strcpy(s.addr.city, "Bangalore");
   s.addr.pincode = 560001;
   printf("Roll Number : %d\n", s.roll);
   printf("City       : %s\n", s.addr.city);
   printf("Pincode    : %d\n", s.addr.pincode);
   return 0;   }
```

Output: Roll Number: 101

City: Bangalore

Pincode: 560001

7b). Create a structure that contains student name (sname) and student marks(smarks). Write a C program that reads the names and marks of two students and compares whether both students have the same marks or not. ---- [5marks program]

```
#include <stdio.h>
struct student
{
   char sname[50];
   int smarks;
};

int main()
{
   struct student s1, s2;

   printf("Enter details of Student 1:\n");
   printf("Name: ");
   scanf("%s", s1.sname);
   printf("Marks: ");
   scanf("%d", &s1.smarks);

   printf("\nEnter details of Student 2:\n");
```

```
    printf("Name: ");
    scanf("%s", s2.sname);
    printf("Marks: ");
    scanf("%d", &s2.smarks);

    if (s1.smarks == s2.smarks)
        printf("\nBoth students have the same marks.\n");
    else
        printf("\nStudents have different marks.\n");

    return 0;
}
```

Output: Enter details of Student 1:

Name: Ravi

Marks: 85

Enter details of Student 2:

Name: Sita

Marks: 85

Both students have the same marks.

8. Explain: a) Unions b)Enumerations c) Typedef

Explanation of Unions+Example --4marks + Enumerations ---3marks + Typedef –3marks

Unions:

- A *union* is a memory location that is shared by two or more different types of variables.
- Declaring a **union** is similar to declaring a structure. Its general form is

union *tag* {

***type member-name*;**

***type member-name*;**

***type member-name*;**

.

.

} *union-variables*;

For example:

union u_type {

int i;

char ch;

};

- **This declaration does not create any variables. You can declare a variable either by placing its name at the end of the declaration or by using a separate declaration statement.**
- To declare a **union variable** called **cnvt** of type **u_type** using the definition just given, write: **union u_type cnvt;**
- In **cnvt**, both integer **i** and character **ch** share the same memory location. Of course, **i** occupy 2bytes (assuming 2-byte integers), and **ch** uses only 1.
- To access a member of a union, use the same syntax that you would use for structures: **the dot and arrow operators.** If you are operating on the **union directly, use the dot operator.**
- If the **union** is accessed through a **pointer**, use the **arrow operator**. For example, to assign the integer 10 to element i of cnvt, **write cnvt.i = 10**;
- using a union, you can easily create a function called putw( ), which writes the binary representation of a short integer to a file one byte at a time.

union pw {

short int i;

char ch[2];

};

Enumerations

- An *enumeration* is a set of named integer constants.
- For example, an enumeration of the coins used in the United States is **penny, nickel, dime, quarter, half-dollar, dollar**
- Enumerations are defined much like structures; the keyword **enum** signals the start of an enumeration type. The general form for enumerations is
- **enum *tag* { *enumeration list* } *variable_list*;**
- Here, both the tag and the variable list are optional. (But at least one must be present.)
- The following code fragment defines an enumeration called **coin**:

- **enum coin { penny, nickel, dime, quarter, half_dollar, dollar};**

Typedef

- You can define new data type names by using the keyword **typedef**.
- You are not actually creating a new data type, but rather defining a new name for an existing type.
- This process can help make machine-dependent programs more portable.
- typedef also can aid in self-documenting your code by allowing descriptive names for the standard data types. The general form of the typedef statement **is typedef type newname;**
- **where type is any valid data type**, **and newname is the new name for this type**.
- For example, you could create a new name for float by using: **typedef float balance;**
- This statement tells the compiler **to recognize balance as another name for float**.
- Next, you could create a float variable using balance: **balance over_due;**
- Here, over_due is a floating-point variable of type balance, which is another word for float.

  An c program example that supports all the 3 definitions.