	C Programming - Rev 2015 TED 2131 **ANSWER KEY**		
Q no	**Scoring Indicators**	**split score**	**Total score**
I	**PART A**		
I.1	InteriorAngle = ((n-2)*180)/n;	2	2
I.2	x / y = 2 and x % y = 3	1 + 1	2
I.3	A function that calls itself is known as a *recursive function*.	2	2
I.4	Eg : char ItemName[]; or char ItemName[25];	Type -1 size -1	2
I.5	Eg: float Height[50];	Type -1 size -1	2
II			
II.1	The **break** statement in C programming has the following two usages − • When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. • It can be used to terminate a case in the **switch** statement The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between. For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests.	3+3	6
II.2	The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct. Variable name should not be a keyword eg: Name, item_price, number1	6	6
II.3	The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.	3 + 3	6

II.4	Sample program
	`#include <stdio.h>`
	`void main()`
	`{ int Array[10], sum =0,n;`
	`  printf("\nEnter the size of the array :");`
	`  scanf("%d", &n);`
	`  printf("\nEnter %d numbers ",n);`
	`  for(int k=0; k<n;k++)`
	`  { scanf("%d", &Array[k]);`
	`    sum = sum +Array[k];`
		Declaration – 2
		reading and calculation - 3
		output 1
			6

	}
	printf("\nSum of elements = %d",sum);
	}		

II.5	The **auto** storage class is the default storage class for all local variables.
	`{`
	`   int mount;`
	`   auto int month;`
	`}`
	
	The example above defines two variables with in the same storage class.
	The **register** storage class is used to define local variables that should be stored in a register instead of RAM. .
	`{`
	`   register int  miles;`
	`}`
	
	The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.
	The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.
	
	The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously de-fined.
		Any three 2 x3	6

II.6	**Arrays** a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
	
	Syntax : type arrayName [arraySize];
	eg: double balance[10];
	
	Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds. Structures are used to represent a record.
	`Syntax:`
	`struct [structure tag] {`
	
	`   member definition;`
	`   member definition;`
	`   ...`
	`   member definition;`
	`} [one or more structure variables];`
	
	`eg:struct Books {`
	`   char  title[50];`
	`   char  author[50];`
	`   char  subject[100];`
	`   int   book_id;`
	`} book;`
		3+3	6

II.7

Sample program

```
#include <stdio.h>
struct Student {
   char  name[20];
   int  regno, mark1,mark2,mark3,Total;
   };
void main()
{
  struct Student S;
  printf("\n Enter the name, register number");
scanf("%s %d", S.name, &s.regno);
printf("\nEnter mark1, mark2 and mark3")
scanf("%d %d %d",&s.mark1, &s.mark2,&s.mark3);
s.Total = s.mark1+s.mark2+s.mark3;
printf("Total mark = %d", s.Total);
}
```

Structure definition 2, reading 2 calculation of total - 1 output - 1

6

PART C

III (a)

4+4

15

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

The syntax of a **while** loop is −

```
while(condition) {
   statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

The syntax of a **do...while** loop is −

```
do {
   statement(s);
} while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

III. (b)	<u>Sample program</u>
	
	#include <stdio.h>
	int main()
	{
	    int n, limit;
	
	    printf("Enter multiplier and limit: ");
	    scanf("%d %d",&n,&limit);
	
	    for(int k=1; k<=limit; ++k)
	    {
	        printf("%d * %d = %d \n", n, k, n*k);
	    }
	
	    return 0;
	}
		Logic 4
		program structure 3
			7

IV a	<u>Sample program</u>
	
	#include <stdio.h>
	int main()
	{
	    float BMI, height, weight;
	    printf("\nEnter weight in kilograms and height in meters");
	    scanf("%f %f",&weight,&hight);
	BMI = weight / (height * height);
	if (BMI<18.5)
	     printf("Your BMI is %f \n You are under weight",BMI);
	else if (BMI>=18.5 && BMI<=25)
	         printf("Your BMI is %f \n You are Noraml weight",BMI);
	     else
	         printf("Your BMI is %f \n Caution !! Over weight",BMI);
	
	    return 0;
	}
		Program structure 4
		logic 4
			8

IV. (b)	<u>Sample program</u>
	
	#include <stdio.h>
	int main()
	{
	    int x,n,power=1;
	
	    printf("Enter values of x and n: ");
	    scanf("%d %d",&x,&n);
	
	    for(int k=1; k<=n; ++k)
	    {
	        power = power * x;
	    }
	
	    printf("n th power of x = %d ",power);
	}
		Program structure 3
		logic 4
			7

V (a)	Sample program void swap(int *x, int *y) {	Function declaration 1 calling	8

	```
	   int temp;
	   temp = *x;
	   *x = *y;
	   *y = temp;

	   return;
	}

	#include <stdio.h>

	void swap(int *x, int *y);

	int main ()
	{

	   int a,b;
	    printf("enter the value of a and b ");
	    scanf("%d %d", &a,&b);

	   printf("Before swap, value of a : %d\n", a );
	   printf("Before swap, value of b : %d\n", b );


	   swap(&a, &b);

	   printf("After swap, value of a : %d\n", a );
	   printf("After swap, value of b : %d\n", b );

	   return 0;
	}
	```
		1 function definition 4 other 2	

V. (b)	Integers (long and short) integers signed and unsigned float and double char signed and unsigned basic data types, enumerated data types, void, derived (describe any type of classification with examples)	Any 7 1 x 7	7

VI (a)	
	```
	Sample program

	#include <stdio.h>

	long int factorial(unsigned int n)
	 {

	   if(n <= 1) {
	      return 1;
	   }
	   return n * factorial(n - 1);
	}

	int  main()
	{
	   int n;
	   printf("enter a number ");
	   scanf("%d",&n);
	   printf("Factorial of %d is %d\n", n, factorial(n));
	   return 0;
	}
	```
		8	8

VI. (b)

Eg:

```
#include<stdio.h>
#define NUMBER 10
int main()
{
      printf("%d", NUMBER);
      return 0;
}
```

After expanding the program code is

```
int main()
{
      printf("%d", 10);
      return 0;
}
```

Macro	Function
There is no type checking.	Type checking is occurred.
Macro is Pre-processed	Function is Compiled.
Code Length Increases, when you call macro multiple times.	Code Length remains the same in every calling of the function.
Use of macro can lead Side effect.	No side Effect
Speed of Execution is Faster.	Speed of Execution is Slower as compare to macro.
Generally macro is useful for the small code.	Function is generally useful for the large code.
Because macro is pre- processed, so it is difficult to debug the macro.	Easy to debug the function.

7

VII (a)

Sample program

```
#include <stdio.h>
void main()
{ float temperature[2][12],DayTotal =0, nightTotal=0;
  printf("\n Enter hourly day temperatures 6am-6pm ");
  for (int k=0; k<12; k++)
  { scanf("%f",&temperature[0][k]);
    DayTotal = DayTotal+temperature[0][k];
  }
  printf("\n Enter hourly night temperatures 6pm-6am ");
  for (int k=0; k<12; k++)
  {     scanf("%f",&temperature[1][k]);
        NightTotal = NightTotal + temperature[1][k];
  }
  printf("\nAverage Day temperature = %f", DayTotal/12.0);
  printf("\nAverage Night temperature = %f",NightTotal/12.0);
}
```

Declaration 3 marks reading – 2 marks calculation – 2 marks final display – 1 mark

8

VII. (b)

Passing arrays to a function

The name of the array is itself a pointer to that array. And all el-

Concept 4 eg: 3

7

ements of the array are in contiguous memory location starting from the first element. So we can pass the entire array to a function by just passing the reference of first element of the array.

Eg: here the array named data is declared in the main program. And the reading of the array is done in functions read(). In this function, two arguments are passed, the first one is the reference to the first element of the array (the array name) and the second argument is the number of elements in the array. The first argument is the address and second argument is a value. So in the function definition, the first argument is received by an integer pointer and second argument is received by an integer variable.

```
# include <stdio.h>

int main()
{
         int data[5], i, size =5;
      void read(int*, int);
      void display(int*,int);

      read(data,size);  // function calling for reading the array

}

// function definition for reading array
void read(int *data, int size)
{
      printf("\n enter %d elements ",size);
      for(int i = 0; i < size; ++i)         // reading using for loop
           scanf("%d", &data[i]);
}
```

VIII (a)

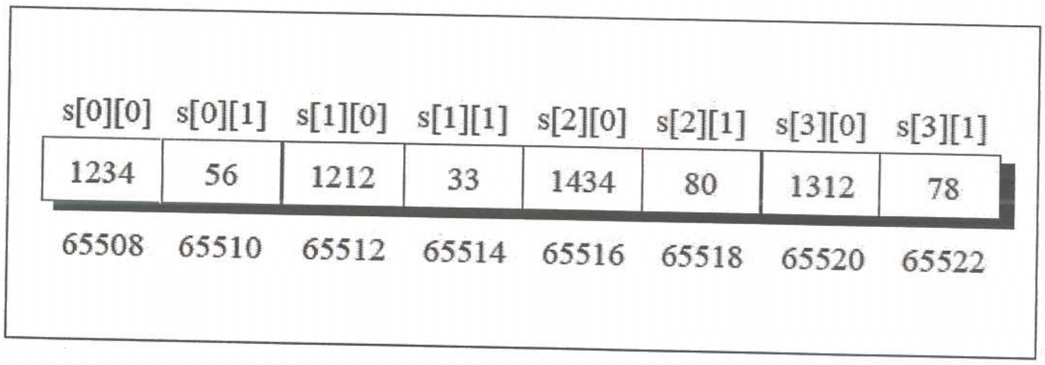
Sample program

```
#include <stdio.h>
int main()
{
   int a[5][5], b[5][5], c[5][5];
   int i, j,m,n;

   printf("Enter the order of matrix\n");
   scanf("%d %d", &m,&n);
   printf("\n enter the elements of first matrix");
   for(i=0; i<m; ++i)
    for(j=0; j<n; ++j)
        scanf("%d", &a[i][j]);


   printf("Enter elements of 2nd matrix\n");

   for(i=0; i<m; ++i)
    for(j=0; j<n; ++j)
           scanf("%d", &b[i][j]);

   // adding corresponding elements of two arrays
   for(i=0; i<m; ++i)
    for(j=0; j<n; ++j)
           c[i][j] = a[i][j] + b[i][j];

   // Displaying the sum
   printf("\nSum Of Matrix:");

   for(i=0; i<m; ++i)
    {
      printf("\n");
      for(j=0; j<n; ++j)
```

Declaration 2 reading 2 adding 2 displaying 2

8

	`printf("%d\t", c[i][j]);`
	`}`
	`return 0;`
	`}`		

VIII (b)	**Example:** consider the array elements in a two-dimensional array of students, which contains roll nos. in one column and the marks in the other int s[4][2] ={ {1234,56}, {1212,33}, {1434,80}, {1312,78} }; In memory whether it is a one-dimensional or a two-dimensional array the array elements are stored in one continuous chain. The arrangement of array elements of a two-dimensional array in memory is shown below:  s[0][0] s[0][1] s[1][0] s[1][1] s[2][0] s[2][1] s[3][0] s[3][1] 1234 56 1212 33 1434 80 1312 78 65508 65510 65512 65514 65516 65518 65520 65522 The expressions s[0] and s[1] would yield the addresses of the zeroth and first one-dimensional array respectively. From Figure these addresses turn out to be 65508 and 65512.	Figure 4 explanation 3	7
IX a)	**strcpy(s1, s2);** Copies string s2 into string s1. **strlen(s1);** Returns the length of string s1. **strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.	2 x 3	6

IX b	Sample program
	
	#include <string.h>
	struct employee
	    {char empname[25];
	     int BP, DA,Total;
	    };
	void main()
	{
	  struct employee Emp[10];
	  for(k=0; k<10; k++)
	  {
		Structure def- 2 declarion 2 reading 2 calculation 2 display 1	9

```
    printf("enter the name and Basic pay of %d employee",k+1);
    scanf("%s %d",Emp[k].empname, &Emp[k].BP;
    Emp[k].DA = Emp[k].BP * 0.25;
    Emp[k].Total = Emp[k].Bp + Emp[k].DA;
   }
 printf("\nName       BP        DA       Total");
 for(k=0; k<10; k++)
    printf("%s %d    %d ",Emp[k].empname, Emp[k].BP, Emp[k].DA,Emp[k].Total);
 }
```

X a.

structure is a user defined data type available in C that allows to combine data items of different kinds. Structures are used to represent a record.

o define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows −

```
struct [structure tag]
{

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The **structure tag (here we give the name of the structure)** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

```
struct Books
 {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Def – 2
syntax 2
eg: 2

6

X b.

```
Sample program

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

void main()
{
    char str1[],str2[], str3[];
    int result;
    printf("\n enter two strings");
    scanf("%s %s", str1,str2);

    // comparing strings str1 and str2
    result = strcmp(str1, str2);
  if (result == 0)
     printf("\n the two strings are same");
  else
     {
     strcat( str3, str1);
     strcat( str3, str2);
     printf("the concatenated string is %s ", str3);
    }
}
```

Declaration 1
rading 2
comparison 2
conactination 2
diaplaying 2

9