SCHEME OF EVALUATION
(Scoring Indicators)

Revision: 15 Course Code: 2131

Course Title: PROGRAMMING IN C

Qst. No.	Scoring Indicator	Split up score	Sub Total	Total
I 1.	PART A Exp1? Exp2: Exp3; Where Exp1, Exp2 and Exp3 are expressions. Exp1 is evaluated. If it is true then Exp2 is evaluated and becomes the value of the entire expression. If Exp1 is false then Exp3 is evaluated and its value becomes the value of the expression	Syntax 1 Explanation 1	1+1	2
2.	A function call by itself is called recursion			2
3.	A pointer is used to store address of another variable. For example an integer pointer hold the address of an integer variable. Eg:- int *a,b=5; a=&b;	Definition 1 Example 1	1+1	2
4.	Int a[5]={1,2,3,4,5}; a is an integer array with elements 1,2,3,4,5.	2	2	2
5.	Sequences of characters are called string. It is declared as a character array. Eg: char s[20]="hello";	Definition 1 Example 1	1+1	2

II 1.	**PART B**
	
	A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.
	
	The syntax for a switch statement in C programming language is as follows
	
	`switch(expression){`
	`case constant-expression :`
	`statement(s);`
	`break;/* optional */`
	`case constant-expression :`
	`statement(s);`
	`break;/* optional */`
	`/* you can have any number of case statements */`
		Syntax 2		

```
default:/* Optional */
   statement(s);
}
```

Explanation 2

- The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example 2

```
#include <stdio.h>
void main (){
int choice;
printf("enter the choice");
scanf("%d",&choice);
switch(choice){
case1:printf("one\n");
break;
```

	case2:printf("two\n");
	break;
	case3:printf("three\n");
	break;
	default:printf("Invalid choice\n");
	}
	}
			2+2+2	6

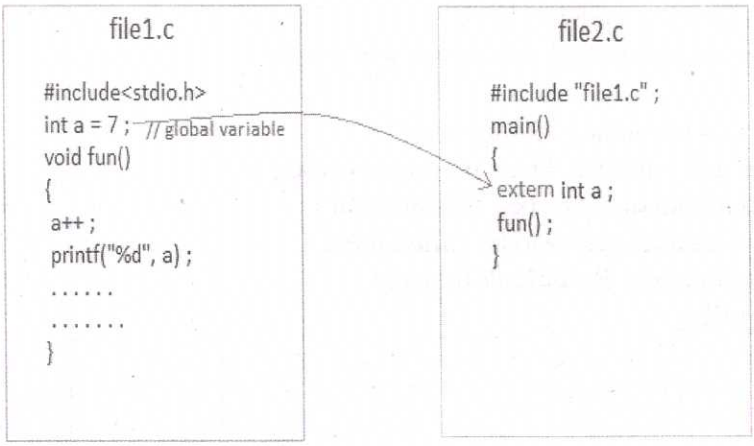
II
2.
	Storage classes in C
	In C language, each variable has a storage class which decides the following things:
	scope :- where the value of the variable would be available inside a program.
	default initial value :- if we do not explicitly initialize that variable, what will be its default initial value.
	lifetime of that variable :- for how long will that variable exist.
	The following storage classes are most often used in C programming,
	• Automatic variables
	• External variables
	• Static variables
	• Register variables
	Automatic variables: auto
	Scope: Variable defined with auto storage class are local to the function block inside which they are defined.
	Default Initial Value: Any random value i.e garbage value.
	Lifetime: Till the end of the function/method block where the variable is defined.
	A variable declared inside a function without any storage class specification, is by default an automatic variable. They are created when a function is called and are destroyed automaticallywhen the function's execution is completed. Automatic variables can also be called local variablesbecause they are local to a function. By default they are assigned garbage value by the compiler.
	#include<stdio.h>
	void main()
	{
	int detail;
	// or
	auto int details; //Both are same
	}
	External or Global variable
	Scope: Global i.e everywhere in the program. These variables are not bound by any function, they are available everywhere.
	Default initial value: 0(zero).
	Lifetime: Till the program doesn't finish its execution, you can access
		Any three 2 marks each		

global variables.
A variable that is declared outside any function is a Global Variable. Global variables remain available throughout the program execution. By default, initial value of the Global variable is 0(zero).

```
#include<stdio.h>
int number;    // global variable
void main()
{
   number = 10;
printf("I am in main function. My value is %d\n", number);
   fun1();    //function calling, discussed in next topic
   fun2();    //function calling, discussed in next topic
}
/* This is function 1 */
fun1()
{
   number = 20;
printf("I am in function fun1. My value is %d", number);
}
/* This is function 1 */
fun2()
{
printf("\nI am in function fun2. My value is %d", number);
}
```

extern keyword

The extern keyword is used with a variable to inform the compiler that this variable is declared somewhere else. The extern declaration does not allocate storage for variables.



global variable from one file can be used in other using extern keyword.

Static variables

Scope: Local to the block in which the variable is defined
Default initial value: 0(Zero).
Lifetime: Till the whole program doesn't finish its execution.
A static variable tells the compiler to persist/save the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, staticvariable is initialized only once

	and remains into existence till the end of the program. They are assigned 0 (zero) as default value by the compiler.
	
	#include<stdio.h>
	void test(); //Function declaration
	int main()
	{
	test();
	test();
	test();
	}
	void test()
	{
	static int a = 0; //a static variable
	a = a + 1;
	printf("%d\t",a);
	}
	Register variable
	Scope: Local to the function in which it is declared.
	Default initial value: Any random value i.e garbage value
	Lifetime: Till the end of function/method block, in which the variable is defined.
	Register variables inform the compiler to store the variable in CPU register instead of memory. Register variables have faster accessibility than a normal variable. Generally, the frequently used variables are kept in registers. But only a few variables can be placed inside registers.
	Syntax :
	register int number;
			2+2+2	6

II 3.	• In a macro call the preprocessor replaces the macro template with its macro expansion. In a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function. • Macros make the program run faster but increase the program size, whereas functions make the program smaller and compact. • If we use a macro hundred times in a program, the macro expansion goes into our source code at hundred different places in the program, it would take the same amount of space in the program. • But passing arguments to a function and getting back the returned value does take time and would therefore slow down the program. This gets avoided with macros since they have already been expanded and placed in the source code before compilation.	Any 3 differences 2 marks for each		6

II 4.	#include<stdio.h>
	void main()
	{
	int a[20], sum=0,i;
	float average;
	for(i=0;i<20;i++)
	{
	printf("enter the number");
	scanf("%d",&a[i]);
	sum=sum+a[i];
	}
	average=sum/20;
	printf('sum= %d",sum);
	printf("Average=%f",average);
	}
		Declaration-2
		
		Looping & Input - 3
		
		Output-1
			2+3+1	6

II 5.	• The one-dimensional array is a list whose elements are of similar datatype. On the other hand, the two-dimensional array is a list whose elements are the array of similar data type. • In C, the one-dimensional array is stored in a contiguous memory location in the indexed order, whereas the two-dimensional array is also stored in the contiguous memory location, but as there are multiple rows in a two-dimensional array, storage of a first row is followed by a second and the third and so on. • Passing of both, one-dimensional array as well as, two-dimensional array to a function is similar i.e. both are passed only by the name of array. • In Both one-dimensional and two-dimensional array, the index plays a very important role because it is the only thing which specifically identifies an element in the array. Both one-dimensional and two-dimensional array can be initialized at the time of their declaration	Any 4 comparisons 1 mark each Example -2	4+2	6
II 6	An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures. /* Usage of an array of structures */ main( ) { struct book { char name ; float price ; int pages ; } ; struct book b[100] ; inti ;	Declaration-2		

	for (i = 0 ; i<= 99 ; i++)
	{
	printf ("\nEnter name, price and pages ") ;
	scanf ("%c %f %d", &b[i].name, &b[i].price, &b[i].pages) ;
	}
	for (i = 0 ; i<= 99 ; i++)
	printf ("\n%c %f %d", b[i].name, b[i].price, b[i].pages) ;
	}
	Here
	the array of structures is declared as
	struct book b[100] ;
	
	This provides space in memory for 100 structures of the type **struct book**.
	
	• The syntax we use to reference each element of the array **b** is similar to the syntax used for arrays of **int**s and **char**s. For example, we refer to zeroth book's price as **b[0].price**. Similarly, we refer first book's pages as **b[1].pages**.
	• In an array of structures all elements of the array are stored in adjacent memory locations. Since each element of this array is a structure, and since all structure elements are always stored in adjacent locations. In our example, **b[0]**'s **name**, **price** and **pages** in memory would be immediately followed by **b[1]**'s **name**, **price** and **pages**, and so on.
		Example -2
		
		Explanation-2
			2+2+2	6

II 7.	#include<stdio.h>
	#include<string.h>
	void main()
	{
	char original[25],reverse[25];
	inti=0,j=0,length;
	printf("Enter the string");
	scanf("%s",original);
	while(original[i]!='\0')
	i++;
	length=i+1;
	i=0;
	for(j=length-1;j>=0;j--)
	{
	reverse[i]=original[j];
	i++;
	}
	reverse[j]='\0';
	printf("reversed string =%s", reverse);
	}
		Declaration -1
		
		Input-1
		
		Reversing -3
		
		Output-1
			1
			+1+3+1
				6

III (a)

PART C

Logical Operators

&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

3

Relational Operators

==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

4

3+4

7

III (b)	#include<stdio.h>
	void main()
	{
	int mark1,mark2,mark3;
	float percentage;
	printf("enter marks in 3 subjects out of 100");
	scanf('%d%d%d",&mark1,&mark2,&mark3);
	percentage=(mark1+mark2+mark3)/3;
	if(percentage>=80)
	printf("Distinction");
	else
	if(percentage>=60 && percentage<=79)
	printf("First class");
	else
	if(percentage>=50 && percentage<=59)
	printf('Second class");
	else
	
	if(percentage>=40 && percentage<=49)
	
	printf("Third class");
	
	else if(percentage<40)
	
	printf("Failed");
	
	}
		Declaration-1
		
		Input-1
		
		Calculation-6
			1+1+6	8

IV (a)	1. For loop
	A for loop is a more efficient loop structure in 'C' programming. The general structure of for loop is as follows:
	for (initial value; condition; increment or decrement)
	{
	statements;
	}
	The initial value of the for loop is performed only once.
	The condition is a Boolean expression that tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.
	The increment/decrement increases (or decreases) the counter by a set value.
	Example for(num=0;num<10;num++)
	{
	printf("%d\n",num);
	}
	2. While loop
	
	The basic format of while loop is as follows:
	while (condition) {
	statements;
	}
		4
		
		3		

	It is an entry-controlled loop. In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed. After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false. Once the condition becomes false, the control goes out of the loop.
	`int num=1;`
	`while(num<=10)`
	`{`
	`printf("%d\n",num);`
	`num++;`
	`}`
			4+3	7

IV (b)	#include<stdio.h>
	void main()
	{
	int num,reverse=0,original,remainder;
	printf("enter the number");
	scanf("%d",&num);
	original=num;
	while(num!=0)
	{
	remainder=num%10;
	num=num/10;
	reverse=(reverse*10)+remainder;
	}
	printf("Reversed no = %d",reverse);
	if(original==reverse)
	printf("original and reversed numbers are equal");
	else
	printf("not equal");
	}
		Declaration-1
		Input-1
		Reverse-4
		Output-2
			1+1+4+2	8

V (b)	#include<stdio.h>
	void swap(int *,int *);
	void main()
	{
	inta,b;
	printf("enter two numbers");
	scanf("%d%d",&a,&b);
	printf("Before swapping a=%d, b=%d",a,b);
	swap(&a,&b);
	printf("After swapping a=%d, b=%d",a,b);
	}
		Declaration-2
		Input-1
		Function call-2		

	void swap(int *x,int *y)
	{
	Int t;
	t=*x;
	*x=*y;
	*y=t;
	}
		Function -3	2+1+2+3	8

V(a)	Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we can use in our program. These datatypes have different storage capacities. C language supports 2 different type of data types: 1. **Primary data types**: These are fundamental data types in C namely integer(`int`), floating point(`float`), character(`char`) and `void`. 2. **Derived data types**: Derived data types are nothing but primary datatypes but a little twisted or grouped together like **array**, **structure**, **union** and **pointer**. These are discussed in details later. Data type determines the type of data a variable will hold. If a variable `x` is declared as `int`. it means x can hold only integer values. Every variable which is used in the program must be declared as what data-type it is. **Integer type:-**Integers are used to store whole numbers. **Size and range of Integer type on 16-bit machine:** **Type** / **Size(bytes)** / **Range** int or signed int — 2 — -32,768 to 32767 unsigned int — 2 — 0 to 65535 short int or signed short int — 1 — -128 to 127 unsigned short int — 1 — 0 to 255 long int or signed long int — 4 — -2,147,483,648 to 2,147,483,647 unsigned long int — 4 — 0 to 4,294,967,295 **Floating point type:** Floating types are used to store real numbers. **Size and range of Integer type on 16-bit machine** **Type** / **Size(bytes)** / **Range** Float — 4 — 3.4E-38 to 3.4E+38 double — 8 — 1.7E-308 to 1.7E+308	2 2		

	long double 10 3.4E-4932 to 1.1E+4932 **Character type:** Character types are used to store characters value. **Size and range of Integer type on 16-bit machine** **Type** **Size(bytes)** **Range** char or signed char 1 -128 to 127 unsigned char 1 0 to 255 **void type:-** `void` type means no value. This is usually used to specify the type of functions which returns nothing.	2 1	2+2+2+1	7

VI (b)	#include<stdio.>
	void main()
	{
	int factorial(int);
	int n;
	printf("enter the no");
	scanf("%d",&n);
	printf("Factorial=%d",factorial(n));
	}
	int factorial(int n)
	{
	if(n<=1)
	return 1;
	else
	return(n*factorial(n-1));
	}
		Declaration-1
		
		Input-1
		
		Factorial using recursion -4
		
		Function call-2
			1+1+4+2	8

VI(a)	There are two ways to pass arguments to a function
	1. Call by value.
	In call by value the value of the actual arguments are copied to the formal arguments.
	For example.
	void sum(int a, int b)
	{
	int s;
	s=a+b;
	printf("sum=%d",s);
	}
	void main()
	{
	int x=5,y=7;
	sum(x,y);
	}
	Here values of x and y are copied to the variable a and b respectively.
	2. Call by reference.
	In call by reference method address of the actual arguments are passed to the formal arguments.
	For example
	void sum(int *a,int *b)
	{
	int s;
	s=a+b;
	printf("sum=%d",s);
	}
	void main()
	{
	int x=5,y=7;
	sum(&x,&y);
	}
	Here the address of x and y are passed to the pointer variables a and b respectively.
	If we want that the value of an actual arguments should not get changed in the function being called, pass the actual arguments by value.
	If we want that the value of an actual arguments should get changed in the function being called, pass the actual arguments by reference.
		3
		
		4
			3+4	7

VII (a) There are three ways in which we can pass a 2-D array to a function.

List methods- 1

```
main( )
{
    int a[3][4] = {
                    1, 2, 3, 4,
                    5, 6, 7, 8,
                    9, 0, 1, 6
                  };

    clrscr( ) ;
    display ( a, 3, 4 ) ;
    show ( a, 3, 4 ) ;
    print ( a, 3, 4 ) ;
}
display ( int *q, int row, int col )
{
    int i, j ;

    for ( i = 0 ; i < row ; i++ )
    {
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", * ( q + i * col + j ) ) ;

        printf ( "\n" ) ;
    }
    printf ("\n" ) ;
}

show ( int ( *q )[4], int row, int col )
{
    int i, j ;
    int *p ;

    for ( i = 0 ; i < row ; i++ )
    {
        p = q + i ;
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", * ( p + j ) ) ;

        printf ( "\n" ) ;
    }
    printf ( "\n" ) ;
}

print ( int q[ ][4], int row, int col )
{
    int i, j ;

    for ( i = 0 ; i < row ; i++ )
    {
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", q[i][j] ) ;
        printf ( "\n" ) ;
    }
    printf ( "\n" ) ;
}
```

2

2

2

And here is the output...

```
1 2 3 4
5 6 7 8
```

	9016 1234 5678 9016 1234 5678 9016 In the display() function we have collected the base address of the 2-D array being passed to it in an ordinary int pointer. The general formula for accessing each array element would be: *(base address +row no*no. of columns + column no.); In the show() function we have defined q to be a pointer to an array of 4 integers through the declaration: Int (*q)[4]; Where q holds the base address of the zeroth 1-D array adding 1 to it would give us the address of the next 1-D array. This address is assigned to p and using it all elements of the 1-D array are accessed. In the third function print(), the declaration of q looks like Int q[][4]; This is same as int (*q)[4]. Where q is pointer to an array of 4 integers.		1+2+2+2	7

(b)	#include<stdio.h>
	void main()
	{
	int a[3][3],b[3][3],s[3][3],i,j,m,n;
	printf("enter the order of the matrix");
	scanf("%d%d",&m,&n);
	printf("Enter the first matrix\n");
	for(i=0;i<m;i++)
	for(j=0;j<n;j++)
	{
	printf("Enter the data");
	scanf("%d",&a[i][j]);
	}
	printf("Enter the second matrix\n");
	for(i=0;i<m;i++)
	For(j=0;j<n;j++);
	{
	printf("Enter the data");
	scanf("%d",&b[i][j]);
	}
	//To find the sum
	for(i=0;i<m;i++)
	for(j=0;j<n;j++)
	{
	s[i][j]=a[i][j]+b[i][j];
		Declaration-1
		
		Input-2
		
		Find sum-4
		
		Output-1		

	}
	printf("matrix sum\n");
	for(i=0;i<m;i++)
	{
	for(j=0;j<n;j++)
	{
	printf("%d",s[i][j]);
	}
	printf("\n");
	}}
			1+2+4+1	8

VIII (a)	#include<stdio.h>
	Void main()
	{
	Int a[1],n,l,pos,i;
	Printf("Enter the value of n");
	Scanf("%d",&n);
	For(i=0;i<n;i++)
	{
	Printf("Enter the number");
	Scanf("%d",&a[i]);
	}
	l=a[0];
	pos=1;
	for(i=0;i<n;i++)
	{
	If(a[i]>l)
	{
	l=a[i];
	pos=i+1;
	}
	}
	Printf("Largest element is %d and position is %d,l,pos);
	}
		Declaration-1
		
		Input-2
		
		Find the largest and position-3
		
		Output-1
			1+2+3+1	7

(b)	Array elements can be passed to a function by calling the function by value or by reference. In the call by value we pass values of array elements to the function, whereas in the call by reference we pass addresses of array elements to the function. These two calls are illustrated below:
	1. Call by value
	/* Demonstration of call by value */
	main( )
	{
	int i ;
	int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;
	for (i = 0 ; i <= 6 ; i++)
	display (marks[i]) ;
	}
	display (int m)
	{
	printf ("%d ", m) ;
	}
	And here's the output...
	55 65 75 56 78 78 90
		Call by value-4		

	Here we are passing an individual array elements at a time to the function display() and getting it printed in the function display().
	In call by reference
	`/* Demonstration of call by reference */`
	`main( )`
	`{`
	`int i ;`
	`int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;`
	`for ( i = 0 ; i <= 6 ; i++ )`
	`disp ( &marks[i] ) ;`
	`}`
	`disp ( int *n )`
	`{`
	`printf ( "%d ", *n ) ;`
	`}`
	`And here's the output...`
	`55 65 75 56 78 78 90`
	Here we are passing addresses of individual array elements to the function display().
		Call by reference -4	4+4	8

IX (a)	C supports a large number of string handling functions in the standard library "string.h"
	Commonly used string handling functions are
	
	1.strlen(): This function is used to calculate the length of a string.
	Eg:-
	char s1[20]="hello";
	int l;
	l=strlen(s1);
	The function strlen(s1) will return the length of the given string s1.
	
	2. strcat(): This function is used for the concatenation of two strings.
	For example
	char s1[20]="Good", s2[20]="Morning";
	strcat(s1,s2);
	printf("%s",s1);
	This printf message will display the concatenated string "GoodMorning".
	
	3.strcmp() :This function is used for comparing two strings.
	Eg: strcmp(s1,s2);
	This function returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2
		2 marks each	2+2+2	6

(b)	`#include<stdio.h>`
	`void main()`
	`{`
	`struct customer{`
	`intacc_no;`
	`char name[20];`
	`int balance;`
	`}c[200];`
	`void main()`
	`{`
	`inti=0;`
		Declaration-1
		
		Read the details-4		

	for(i=0;i<200;i++) // To read the customer details
	{
	printf("enter the account number");
	scanf("%d",&c[i].acc_no);
	printf("enter the name");
	scanf("%s",c[i].name);
	printf("enter the balance");
	scanf("%d",&c[i].balance);
	}
	//To printf the customer details
	for(i=0;i<200;i++)
	{
	printf("%d",c[i].acc_no);
	printf("%s",c[i].name);
	printf("%d",c[i].balance);
	}}
		Formatted output-4	1+4+4	9

X (a)	Structure is a user defined data type in c that allows to combine data items of different kinds. Structures are used to represent record. <u>Defining a Structure</u> struct is the keyword used to define a structure data type. The format of the struct statement is structstructurename { Member variable1; Member variable2; ….. Member variable n; }[one or more structure variables]; Example: A student structure with data members student name, roll number and branch is defined as struct student { char name[20]; introllno; char branch[20]; }s1; <u>Accessing Structure elements</u> To access any member variable use the member access operator (.). The member access operator is coded as a period between the structure variable name and the member variable that we wish to access. For example to read the member variable name Scanf("%s",s1.name);	Definition-2 Declaration-2 Access-2	2+2+2	6

(b)	#include<stdio.h>
	void main()
	{
	char string1[25],string2[25];
	inti=0,j=0;
	while(string1[i]!='\0')
	i++;
	while(string2[j]!='\0')
	{
	string1[i]=string2[j];
	i++;
	j++;
	}
	string1[i]='\0';
	printf("concatenated string is : %s",string1);
	}
		Declaration-2
		
		Concatenation-5
		
		Output-2
			2+5+2	9