



SNS COLLEGE OF TECHNOLOGY
(An Autonomous Institution)



Coimbatore – 641035.

B.E / B. Tech – Internal Assessment Exam – II

Academic Year 2022 – 2023 (ODD)

FIRST SEMESTER (REGULATION R2019)

19CST101 – PROGRAMMING FOR PROBLEM SOLVING

ANSWER KEY

PART A

1. Define User-defined Data types

The data types that are defined by the user are called the derived data type or user- defined data type.

Eg- array, typedef.

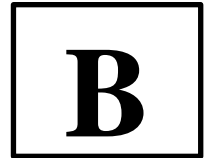
2. Summarize the functions of relational operators

It often compare two quantities and depending on their relation, take certain decisions.

The value of a relational expression is either 1(true) or 0(false).

3. Construct a Function to find whether the given integer is odd or even

```
#include<stdio.h>
#include<conio.h>
void oddeven( int a )
{
if(a%2==0) return “even”;
else
return “odd”;
}
void main( )
{
int a;
```



```
printf("\n Enter any number:");  
scanf("%d", &a);  
oddeven(a);  
getch();  
}
```

4. Define Arrays. And List its Types.

An array is a fixed- size sequenced collection of elements of the same data type

An array can be used to represent a list of numbers, or a list of names.

TYPES OF ARRAY:

1. One-dimensional array
2. Two-dimensional array
3. Multi-dimensional array

5. What is meant by Sorting? And brief the Technique of Merge Sort.

The process of arranging elements either in ascending or descending order is called sorting.

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

PART B

6. (a). i. Define Operators? And Explain its types with Example.

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.

Operators are used in programs to manipulate data and variables.

An expression is a sequence of 'operands' and 'operators' that reduces to a single value.

For example, $10 + 15$ is an expression whose value is 25.

Types of Operators

In C, operators can be classified into various categories based on their utility and action. They are:

1. Arithmetic Operators

2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment and Decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators

1. Arithmetic Operators

Integers, floating point numbers and double precision numbers can be added, subtracted, divided or multiplied. The operators used for these arithmetic operations are called arithmetic operators. C supports all basic arithmetic operators. The list of arithmetic operators and their meanings are given below:

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulus Operators

Integer Arithmetic :

When both the operands in a single arithmetic expression such as $a + b$ are declared as integers, the expression is called an integer expression. An arithmetic operation performed on these integer is called integer arithmetic and it always yields an integer value.

Real Arithmetic :

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result.

Example Program :

```

#include<stdio.h>

void main()
{
int a,b;

printf("Enter any two numbers:");

scanf("%d%d", &a, &b);

printf("a+b=%d \n",a+b);

printf("a-b=%d \n",a-b);

printf("a*b=%d \n",a*b);

printf("a/b=%d \n",a/b);

printf("a%b=%d \n",a%b);

}

```

Output :

Enter any two numbers: 10 5

a+b=15

a-b=5

a*b=50

a/b=2

a%b=0

2. Relational Operators

The relational operators are used to compare the value between two variables or between a variable and a constant. The list of relational operators and their meanings are given below:

Operator	Meaning
----------	---------

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

An expression containing a relational operator is called as relational expression. The relational operators produce an integer result to express the condition of the comparison. If the condition is false, then the integer value is 0. If the condition is true, then the integer value is 1.

Syntax :

exp1 rel_op exp2

Where exp1 and exp2 are arithmetic expressions, which may be simple constants, variables or combination of both and rel_op represents a relational operator.

3. Logical Operators

Logical operators are used to combine two or more relations. The logical operators are called Boolean operators because the tests between values are reduced to either true or false, with zero being false and one being true. The list of logical operators and their meanings are given below:

Operator Meaning

&&	Logical AND
	Logical OR
!	Logical NOT

Example Program :

```
#include<stdio.h>
void main()
{
int a,b;
printf("\nEnter      the two values:");
scanf("%d %d",&a,&b);
printf("\n (a>5)&&(b>2) is %d",(a>5)&&(b>2));
printf("\n      (a>5)||(b>2) is %d",(a>5)||(b>2));
printf("\n      !a is %d",(!a));
}
```

Output :

Enter the two values: 7 4

(a>5)&&(b>2) is 1 (a>5)||(b>2) is 1

!a is 0

4. Assignment Operators

The assignment operator is used to assign the result of an expression to a variable. The most commonly used assignment operator is '='.

Syntax:

var = exp ;

Where var is a variable and exp can be a constant or a variable or a complex expression.

Examples:

num = 25 ; age = 18 ; pi = 3.14 ; area = 3.14 * r * r ;

Example Program :

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

```
void main()
```

```
{
```

```
int a=10, b=20; int c,d,e;
```

```
c=a+b; // expression d=e=b; // nested assignment
```

```
a+=10; // compound assignment printf(“%d %d %d”, c,d,a);
```

```
}
```

Output :

30 20 20

5. Increment and Decrement Operators

C allows two useful unary operators generally not found in other computer languages. These are increment (++) and decrement (--) operators. The operator ++ adds 1 to its operand and -- subtracts 1 to its operand.

The increment and decrement operators and their meanings are given below:

Operator	Meaning
++	Increment operator (Adds 1)
--	Decrement operator (Subtracts 1)

Syntax :

```
(pre) ++variable_name;    (pre) --variable_name;  
                           (or)  
variable_name++(post);    variable_name--(post);
```

The operator is placed either before or after the variable name. If the operator placed before the variable like ++i or --i, it is known as pre-increment and the pre-decrement respectively. If the operator appears after the variable like i++ or i--, it is known as post-increment and post-decrement respectively.

This increment and decrement operator ++m and --m is equivalent to

++m is equivalent to m=m+1 or m+=1 --m is equivalent to m=m-1 or m-=1

This ++m and m++ means same thing when they form statement independently, they behave differently when they are used in expression.

Consider m = 6 y = ++m,

In this case the value of y is 7 and m is also 7. The pre-increment operator first adds to the operand and then the result is assigned to the variable on left.

Consider m=5, y=m++

then the value of y=5, and value of m=6. The post-increment operator first assigns the value to the variable on the left and then increments the operand.

Example Program :

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

```
void main()
```

```
{
```

```
int a,b,c;

printf("\n enter the two values:");

scanf("%d%d",&a,&b);

c=(a>b)?a:b;

printf("\n(a>b)?a:b is %d",c);

}
```

Output:

Enter the two values:

3

4

(a>b)?a:b is 4.

6. Conditional Operators

The conditional operator is also known as a **ternary operator**. The conditional statements are the decision-making statements which depends upon the output of the expression. It is represented by two symbols, i.e., '?' and '!'.

As conditional operator works on three operands, so it is also known as the ternary operator.

The behavior of the conditional operator is similar to the ['if-else'](#) statement as 'if-else' statement is also a decision-making statement.

Syntax of a conditional operator

1. Expression1? expression2: expression3;

In the above syntax, the expression1 is a Boolean condition that can be either true or false value.

If the expression1 results into a true value, then the expression2 will execute.

The expression2 is said to be true only when it returns a non-zero value.

If the expression1 returns false value then the expression3 will execute.

The expression3 is said to be false only when it returns zero value.

Example


```

#include <stdio.h>

int main()
{
    int age; // variable declaration

    printf("Enter your age");

    scanf("%d",&age); // taking user input for age variable

    (age>=18)? (printf("eligible for voting")) : (printf("not eligible for voting")); // conditional
operator

    return 0;
}

```

7. Bitwise Operators

One of C's powerful features is a set of bit manipulation operators. This permits the programmer to access and manipulate individual bits within a piece of data. The bitwise operators can operate upon int and char data types but not on float and double data types. The list of bitwise operators and their meanings are given below:

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
<<	Shift left
>>	Shift right
~	One's complement

All operators except ~ operator are binary operators which requires two operands. While using the bit operators, each operand is treated as a binary number consisting of a series of individual 1s and 0s. The respective bits in each operand are then compared on a bit by bit basis and result is determined based on the selected operation.

Example Program

Write a program to shift the input data two bits right.

```
#include<stdio.h>

#include<conio.h>

void main()

{

int x,y; clrscr();

printf(“Read the integer”);

scanf(“%d”,&x);

x>>2;

y=x;

printf(“Right shifted data is %d”,y);

}
```

Output :

Read the integer 8

Right shifted data is 2

8. Special Operators

C supports some special operators such as comma operators, size of operator, pointer operators (& and *) and member selection operators (. and) The comma and size of operator is discussed below:

Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated left to right and the value of right most expression is the value of the combined expression.

Example :

```
sum = (a = 10, b = 20, a + b) ;
```

Here, the statement first assigns the value 10 to a, then assigns 20 to b, and finally assigns 30 (i.e. 10 + 20) to sum. Since comma operator has the lowest precedence of all operators, the parentheses are necessary.

Sizeof Operator

The sizeof is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier. The syntax is :

```
sizeof(expression) ; (or)sizeof(data_type) ;
```

Example :

```
a= sizeof(sum) ;
```

```
b= sizeof(long int) ;
```

```
c= sizeof(235L) ;
```

The sizeof operator is normally used to determine the length of arrays and structures, when their sizes are not known in advance by the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Example Program :

```
#include<stdio.h>

void main()

{

int a;

printf("\nEnter a value"); scanf("%d",&a);

printf("\nThe sizeof(a) is %d",sizeof(a));

}
```

Output :

```
Enter a value 1
```

```
The sizeof(a) is 2
```

ii. Construct a C-Program to demonstrate a working of arithmetic and relational operators

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

```
#include<conio.h>

void main( )

{

int a;

clrscr( );

printf(“\n Enter the number:”);

scanf(“%d”,&a);

if(a%2==0)

printf(“%d is even”, a);

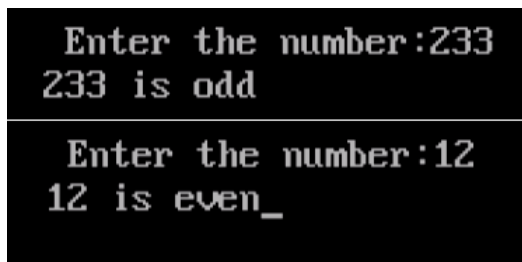
else

printf(“%d is odd”,a);

getch();

}
```

OUTPUT



```
Enter the number:233
233 is odd

Enter the number:12
12 is even_
```

EXPLANATION:

This program is to find whether the given number is odd or even by combining both the arithmetic and relational operators.

Arithmetic operator % used here is to get the remainder of the input divided by 2 and relational operator == is used to check whether the remainder is equal to zero or not.

(b). i. Explain in detail about the Looping Statements with Example.

A segment of the program that is executed repeatedly is called as a loop. Some portion of the program has to be specified several number of times or until a particular condition is satisfied. Such repetitive operation is done through a loop structure.

The Four Methods by which we can repeat a part of a program are:

By using a while loop

By using a do-while loop

By using a for loop

The while loop

The simplest of all the looping structures in C is the while loop. The general form or the syntax of the while loop is:

```
Initialize loop counter; while(condition)
```

```
{
```

```
statement (s) ;
```

```
increment or decrement loop counter ;
```

```
}
```

Here, we can see that firstly, we initialize our iterator. Then we check the condition of `while` loop. If it is false, we exit the loop and if it is true, we enter the loop. After entering the loop, we execute the statements inside the `while` loop, update the iterator and then again check the condition. We do the same thing unless the condition is false.

The do while loop

The do-while loop sometimes referred to as the do loop differs from its counterpart the while loop in checking the condition. The condition of the loop is not tested until the body of the loop has been executed once. If the condition is false, after the first loop iteration the loop terminates. However, if the condition is true the loop continues. The general form or the syntax of the do-while loop is

```
do
```

```
{
```

```
statement (s) ;
```

```
}
```

```
while (condition) ;
```

In do-while the statements would be executed at least once even if the condition fails for the first time itself.

The for loop

The for loop is most commonly and popularly used loop in C. The for loop allows us to specify three things about the loop in single line. Initializing the value for the loop.

Condition in the loop counter to determine whether the loop should continue or not.

Increasing or decreasing the value of loop counter each time the program segment has been executed.

The syntax of the for loop is

```
for(initialization, condition; increment/decrement operation)
```

```
{
```

```
statement (s) ;
```

```
}
```

ii. Write a C-Program to generate a Fibonacci series

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

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
int f1=0,f2=1,f3,i,n;
```

```
clrscr();
```

```
printf("\n FIBONAACI SERIES");
```

```
printf("\n Enter the value:");
```

```
scanf("%d", &n);
```

```
printf("\n %d \n %d",f1,f2);
```

```
for(i=1;i<=n;i++)
```

```

{
f3=f1+f2;

printf("\n %d", f3);

f1=f2;

f2=f3;

}

getch();

}

```

OUTPUT:

```

FIBONAACI SERIES
Enter the value:10

0
1
1
2
3
5
8
13
21
34
55
89

```

7. (a). i. Explain the method to declare and initializing an Array with its types along with an example for each

Arrays

An array is a collection of elements of the same data type, which are stored in contiguous memory locations. The elements in an array can be accessed by their index number. The index number of the first element in an array is 0, and the index number of the last element is the size of the array minus 1. Arrays are useful when we need to store and manipulate a large amount of data of the same type.

Declaring and Initializing Arrays

To declare an array in C, we need to specify its data type and its size:

```
int myArray[5];
```

This declares an integer array named `myArray` with a size of 5.

We can also initialize an array at the time of declaration:

```
int myArray[5] = {10, 20, 30, 40, 50};
```

This initializes the array with the values 10, 20, 30, 40, and 50.

If we don't initialize the array at the time of declaration, its elements will have garbage values.

Multi-Dimensional Arrays

C supports multi-dimensional arrays, which are arrays of arrays.

A two-dimensional array can be declared as follows:

```
int myArray[3][4];
```

This declares a two-dimensional array with 3 rows and 4 columns.

We can initialize a two-dimensional array as follows:

```
int myArray[3][4] = { {10, 20, 30, 40}, {50, 60, 70, 80}, {90, 100, 110, 120} };
```

This initializes a two-dimensional array with the specified values.

We can access a two-dimensional array using its row and column indices:

```
int myArray[3][4] = { {10, 20, 30, 40}, {50, 60, 70, 80}, {90, 100, 110, 120} };  
printf("%d", myArray[1][2]);  
// outputs 70
```

This accesses the element in the second row and third column of the array.

Example

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

```
int main() {  
    int myArray[5] = {10, 20, 30, 40, 50}; // declare and initialize the array  
  
    // print out the elements of the array  
    printf("The elements of the array are: ");  
    for (int i = 0; i < 5; i++) {  
        printf("%d ", myArray[i]);  
    }  
  
    // modify an element of the array
```



```

myArray[2] = 35;

// print out the modified element
printf("\nThe modified element is: %d", myArray[2]);

return 0;
}

```

In this program, we declare a one-dimensional integer array named myArray with a size of 5, and initialize it with the values 10, 20, 30, 40, and 50. We then print out the elements of the array using a for loop and the %d format specifier. We then modify the third element of the array (which has an index number of 2) by setting its value to 35. We print out the modified element using the %d format specifier.

When we compile and run this program, we get the following output:

```

The elements of the array are: 10 20 30 40 50
The modified element is: 35

```

This program demonstrates how to declare, initialize, access, and modify elements of a one-dimensional array in C.

ii. Write a C- Program to print all lower-case alphabets using while loop

C program to print all lower-case alphabets using while loop

Program code:

```

#include<stdio.h>

int main( )
{
char ch = 'a';

printf("\n Lowercase Alphabets");

while(ch <= 'z')
{
printf("%c", ch);

ch++;
}

return 0;
}

```

Output:

Lowercase Alphabets

a b c d e f g h i j k l m n o p q r s t u v w x y z

(b). i. Explain the Searching techniques with Example.

Searching is the process of finding a specific element in a collection of data. There are different searching techniques available to find an element in an array or list of data. Here are some common searching techniques with examples in C:

Linear Search:

Linear search is a simple searching technique that checks every element of the collection sequentially until the target element is found

```
#include <stdio.h>

int linear_search(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            return i;
        }
    }
    return -1;
}

int main() {
    int arr[] = {10, 25, 30, 45, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 30;
    int index = linear_search(arr, n, x);
    if (index == -1) {
        printf("%d not found in the array\n", x);
    }
    else {
```

```

        printf("%d found at index %d\n", x, index);
    }

    return 0;
}

```

In this example, the `linear_search` function takes an integer array `arr` of size `n` and an integer `x` to search for. The function iterates through each element of the array and checks if the current element is equal to the target element `x`. If the element is found, the function returns the index of the element. If the element is not found, the function returns `-1`.

In the main function, we create an integer array `arr` and initialize it with some values. We also specify the size of the array `n` and the element `x` we want to search for. We then call the `linear_search` function and store the result in the variable `index`. If the element is found, we print a message indicating the index where the element was found. If the element is not found, we print a message indicating that the element was not found.

Binary Search:

Binary search is a more efficient searching technique that works on sorted arrays. It repeatedly divides the search interval in half until the target element is found. Here's an example implementation in C:

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

```

int binary_search(int arr[], int l, int r, int x) {
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (arr[m] == x) {
            return m;
        }
        if (arr[m] < x) {
            l = m + 1;
        }
        else {
            r = m - 1;
        }
    }
}

```

```

    }
    return -1;
}

int main() {
    int arr[] = {10, 25, 30, 45, 50};

    int n = sizeof(arr) / sizeof(arr[0]);

    int x = 30;

    int index = binary_search(arr, 0, n - 1, x);

    if (index == -1) {
        printf("%d not found in the array\n", x);
    }

    else {
        printf("%d found at index %d\n", x, index);
    }

    return 0;
}

```

In this example, the `binary_search` function takes an integer array `arr` of size `n`, the left index `l`, the right index `r`, and the integer `x` to search for. The function first checks if the middle element `m` is equal to the target element `x`. If the element is found, the function returns the index of the element. If the element is not found, the function updates the left or right index based on whether the target element is greater or smaller than the middle element. The function continues to divide the search interval in half until the element is found or the interval becomes empty.

In the main function, we create an integer array `arr` and initialize it with some values. We also specify the size of the array `n` and the element `x` we want to search for. We then call the `binary_search` function and store the result in the variable `index`. If the element is found, we print a message indicating the index where the element was found. If the element is not found, we print a message indicating that the element was not found.

Binary search only works on sorted arrays. If the array is not sorted, we must first sort the array before performing binary search.

ii. Write a C-Program to implement any one sorting technique.

Selection sort:

Program code:

```
#include <stdio.h>

int main() {

    int arr[10]={6,12,0,18,11,99,55,45,34,2};

    int n=10;

    int i, j, position, swap;

    for (i = 0; i < (n - 1); i++) {

        position = i;

        for (j = i + 1; j < n; j++) {

            if (arr[position] > arr[j])

                position = j;

        }

        if (position != i) {

            swap = arr[i];

            arr[i] = arr[position];

            arr[position] = swap;

        }

    }

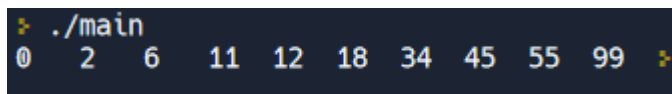
    for (i = 0; i < n; i++)

        printf("%d\t", arr[i]);

    return 0;

}
```

Output:



```
➤ ./main
0 2 6 11 12 18 34 45 55 99 ➤
```

8. (a). Explain in Detail about various String Operations.

In C programming language, a string is a sequence of characters that is terminated by a null character ('\0'). A string is represented as an array of characters, where each character corresponds to an element in the array.

For example, to declare and initialize a string in C

```
char greeting[] = "Hello, World!";
```

In this example, greeting is an array of characters that contains the string "Hello, World!". The size of the array is automatically determined by the length of the string, including the null character at the end.

We can access individual characters in the string using the array subscript notation, for example:

```
char firstLetter = greeting[0]; // gets the first character 'H'
```

C provides a library of functions to manipulate strings, such as strlen, strcpy, strcat, and strcmp. These functions allow us to perform operations such as getting the length of a string, copying one string to another, concatenating two strings, and comparing two strings, respectively.

It is important to note that in C, strings are not a built-in data type, but rather a convention for working with arrays of characters. As such, it is the programmer's responsibility to properly manage the memory allocation and termination of strings to prevent buffer overflows and other errors.

C provides a library of functions to perform operations on strings. Here are some of the commonly used string operations in C:

strlen(): This function returns the length of a string (excluding the null character).

```
char str[] = "Hello";
```

```
int len = strlen(str); // len = 5
```

strcpy(): This function copies one string to another

```
char str1[] = "Hello";
```

```
char str2[10];
```

```
strcpy(str2, str1); // str2 now contains "Hello"
```

strcat(): This function concatenates two strings

```
char str1[] = "Hello";
```

```
char str2[] = "World";
```

```
strcat(str1, str2); // str1 now contains "HelloWorld"
```

strcmp(): This function compares two strings and returns an integer value indicating whether they are equal, greater than, or less than each other

```
char str1[] = "Hello";
```

```
char str2[] = "hello";
```

```
int result = strcmp(str1, str2); // result < 0 because 'H' is less than 'h' in ASCII
```

strncpy(): This function copies a specified number of characters from one string to another.

```
char str1[] = "Hello";
```

```
char str2[10];
```

```
strncpy(str2, str1, 3); // str2 now contains "Hel"
```

strstr(): This function searches for a substring within a string and returns a pointer to the first occurrence of the substring, or NULL if the substring is not found.

```
char str[] = "Hello, World!";
```

```
char* sub = strstr(str, "World"); // sub now points to "World" within str
```

Example:

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

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

```
int main() {
```

```
    char str1[50] = "hello";
```

```
    char str2[] = "world";
```

```
    char str3[100];
```

```
    // strlen()
```

```
    printf("Length of str1: %ld\n", strlen(str1));
```

```
    // strlen()
```

```
    printf("Length of the first 3 characters of str1: %ld\n", strlen(str1, 3));
```

```
    // strcmp()
```

```

printf("Comparing str1 and str2: %d\n", strcmp(str1, str2));

// strncmp()
printf("Comparing the first 3 characters of str1 and str2: %d\n", strncmp(str1, str2, 3));

// strcat()
strcat(str1, str2);

printf("Concatenating str2 to str1: %s\n", str1);

// strcpy()
strcpy(str3, str1);

printf("Copying str1 to str3: %s\n", str3);

// strchr()
printf("Finding the first occurrence of 'l' in str1: %s\n", strchr(str1, 'l'));

// strrchr()
printf("Finding the last occurrence of 'l' in str1: %s\n", strrchr(str1, 'l'));

// strstr()
printf("Finding the first occurrence of 'world' in str1: %s\n", strstr(str1, "world"));

// strcasecmp()
printf("Comparing str1 and str2 case-insensitively: %d\n", strcasecmp(str1, str2));

//strev()
printf("reverse of str1 is : %s\n", strev(str1));

return 0;

}

```

OUTPUT:

```

Length of str1: 5
Length of the first 3 characters of str1: 3
Comparing str1 and str2: -15
Comparing the first 3 characters of str1 and str2: -15
Concatenating str2 to str1: helloworld
Copying str1 to str3: helloworld
Finding the first occurrence of 'l' in str1: lldworld
Finding the last occurrence of 'l' in str1: ld
Finding the first occurrence of 'world' in str1: world
Comparing str1 and str2 case-insensitively: -15

```


(b). Write a C-Program for matrix Multiplication.

Matrix multiplication

Program code:

```
#include<stdio.h>

#include<conio.h>

void main()

{

int i,j,k,m,n,p,q,a[5][5],b[5][5],c[5][5];

clrscr();

printf("\nEnter the size of A matrix:");

scanf("%d%d",&m,&n);

printf("\nEnter the size of B matrix:");

scanf("%d%d",&p,&q);

if(n==p)

{

printf("\nEnter the A Matrix Elements:");

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

{

for(j=0;j<n;j++)

{

scanf("%d",&a[i][j]);

}

}

}
```

```
printf("\nEnter the B Matrix Elements:");

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

{

for(j=0;j<q;j++)

{

scanf("%d",&b[i][j]);

}

}

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

{

for(j=0;j<q;j++)

{

c[i][j]=0;

for(k=0;k<p;k++)

{

c[i][j]=c[i][j]+a[i][k]*b[k][j];

}

}

}

printf("\n\t\tThe Multiplication of the Given Matrix\n"); for(i=0;i<m;i++)

{

for(j=0;j<q;j++)

{
```

```
printf("%d\t\t",c[i][j]);  
  
}  
  
printf("\n");  
  
}  
  
}  
  
else  
  
{  
  
printf("Multiplication Not possible");  
  
}  
  
getch();  
  
}
```

OUTPUT:

```
Enter the size of A matrix:3  
3  
  
Enter the size of B matrix:3  
3  
  
Enter the A Matrix Elements:56  
45  
12  
41  
12  
13  
12  
45  
45  
  
Enter the B Matrix Elements:56  
87  
23  
96  
1  
2  
35  
5  
7  
  
The Multiplication of the Given Matrix  
7876      4977      1462  
3903      3644      1058  
6567      1314      681
```