Lab No:1   **Date:2081/09/17**

Title: Write a program to demonstrate Dynamic memory allocation in C.

Dynamic Memory Allocation:

Dynamic Memory Allocation refers to process of allocating memory during run-time of a program. It is opposite of static memory allocation which allocates memory during compile time. DMA allows program to request memory as needed or release memory when no longer required during the run time. The library function is provided in <stdlib.h> header file in C. The four library functions are:

• Malloc

• Calloc

• Realloc

• Free

Malloc**,** calloc**,** realloc, and free are functions in C for dynamic memory management. malloc (memory allocation) allocates a single block of memory of a specified size but does not initialize it, leaving it with garbage values. Calloc (contiguous allocation), on the other hand, allocates memory for an array of elements, initializes all bits to zero, and is often used when initialization to zero is required. Realloc (reallocation) is used to resize an already allocated memory block dynamically, either increasing or decreasing its size, while preserving the existing data up to the smaller of the old and new sizes. Finally**,** free deallocates the memory that was previously allocated by malloc, calloc, or realloc, preventing memory leaks by making the memory available for future use. Proper use of these functions is crucial for efficient memory utilization and avoiding issues like segmentation faults or memory leaks in C programs.

IDE: Visual Studio Code

Language: C

Source Code For Malloc:

```c
#include <stdio.h>
int main()
{ int n, i, *ptr, sum = 0;
    printf("enter the number:\n");
    scanf("%d", &n);
    ptr = (int *) malloc (n * sizeof(int));
    printf("value before initialization is :\n");
    for (i = 1; i <= n; i++)
    {
        printf("%d\n", *(ptr + i));
    }
    printf("after initialization :\n");
    printf("enter the number:\n");
    for (i = 1; i <= n; i++)
    {
        scanf("%d", (ptr + i));
        sum = sum + *(ptr + i);
    }
    printf("the total sum is: %d", sum);
    return 0;
```

Output:

```
enter the number:
2
value before initialization is :
11665600
-593910787
after initialization :
enter the number:
1
2
the total sum is: 3
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```

Source Code for Calloc:

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    printf("enter the number:\n");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    printf("value before initialization is :\n");
    for(i=1;i<=n;i++)
    {
        printf("%d\n",*(ptr+i));
    }
    printf("after initialization :\n");
    printf("enter the number:\n");
    for ( i = 1; i <= n; i++)
    {
        scanf("%d",(ptr+i));
        sum=sum+*(ptr+i);
    }
    printf("the total sum is: %d\n",sum);
    return 0;}
```

Output:

Source code for Realloc:

```c
#include<stdio.h>
int main()
{
    int n,i,*ptr,sum=0,n1;
    printf("enter the number:\n");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));
    printf("value before initialization is :\n");
    for(i=1;i<=n;i++)
    {
        printf("%d\n",*(ptr+i));
    }
    printf("after initialization :\n");
    printf("enter the number:\n");
    for ( i = 1; i <= n; i++)
    {
        scanf("%d",(ptr+i));
        sum=sum+*(ptr+i);
    }
    printf("the total sum is: %d\n",sum);

printf("enter the size of number that you want to add on previous sum:\n");
scanf("%d",&n1);
    ptr=(int*)realloc(ptr,n*sizeof(int));  //reallocating
    printf("value before initialization is :\n");
    for(i=1;i<=n;i++)
    {
        printf("%d\n",*(ptr+i));
    }
    printf("after initialization :\n");
    printf("enter the number:\n");
    for ( i = 1; i <= n; i++)
```

```
    {
        scanf("%d",(ptr+i));
        sum=sum+*(ptr+i);
    }
    printf("the total sum is: %d",sum);
    return 0;
}
```

Output:

```
enter the number:
2
value before initialization is :
13107392
-2143346876
after initialization :
enter the number:
1
2
the total sum is: 3
enter the size of number that you want to add on previous sum:
2
value before initialization is :
1
2
after initialization :
enter the number:
1
2
the total sum is: 6
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain> ▌
```

Lab No:2   **Date:2081/09/**

Title: Write a Menu based program to show the basic operations of Stack.

Stack is a linear data structure that follows **LIFO** (Last In First Out) Principle, the last element inserted is the first to be popped out. It means both insertion and deletion operations happen at one end only.

Types of Stack:

- **Fixed Size Stack** : As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.

- **Dynamic Size Stack** : A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Basic Operations on Stack:

In order to make manipulations in a stack, there are certain operations provided to us.

- **push()** to insert an element into the stack
- **pop()** to remove an element from the stack
- **top()** Returns the top element of the stack.
- **isEmpty()** returns true if stack is empty else false.
- **isFull()** returns true if the stack is full else false.

IDE: Visual Studio Code

Language: C

Source Code:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 2 // array size

struct stack
{
    int arr[MAX];
    int top;
};

void push(struct stack *sp, int item)
{
    if (sp->top == MAX - 1)
    {
        printf("\n The stack is overflow...\n");
    }
    else
    {
        sp->top++;
        sp->arr[sp->top] = item;
        printf("%d is push in stack\n", item);
    }
}

void pop(struct stack *sq)
{
    if (sq->top == -1)
    {
        printf("\n The stack is underflow...\n");
    }
    else
    {
```

```c
        printf("%d is pop from stack\n", sq->arr[sq->top]);
        sq->top--;
    }
}
void display(struct stack *sq)
{
    if (sq->top == -1)
    {
        printf("Stack is empty...\n");
    }
    else
    {
        printf("The data of stack is:\n");
        // for(int i=sq->top;i>=0;i--)
        for (int i = 0; i <= sq->top; i++)
        {
            printf("%d,", sq->arr[i]);
        }
    }
}

int main()
{
    struct stack *sq;
    sq->top = -1;
    int c, choose, item;
    while (1)
    {
        printf("\n\n1.For push\n");
        printf("2. For pop\n");
        printf("3. For display\n");
        printf("4.For exit\n");
        printf("Choose number(1-4):- ");
        scanf("%d", &choose);
```

```c
        switch (choose)
        {
        case 1:
            printf("Enter a number to push:-");
            scanf("%d", &item);
            push(sq, item);
            break;

        case 2:
            pop(sq);
            break;

        case 3:
            display(sq);
            break;

        case 4:
            printf("\n\nProgram END...\n");
            exit(0);
            break;

        default:
            printf("Wrong Entry!!!\n Please choose (1-4) only\n");
            break;
        }
    }
    return 0;
}
```

Output For Push Operation:

```
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
.c -o stackUsingSwitch } ; if ($?) { .\stackUsingSwit


1.For push
2. For pop
3. For display
4.For exit
Choose number(1-4):- 1
Enter a number to push:-23
23 is push in stack


1.For push
2. For pop
3. For display
4.For exit
Choose number(1-4):- 3
The data of stack is:
23,

1.For push
2. For pop
3. For display
4.For exit
Choose number(1-4):- ▋
```

Output For Pop Operation:

```
1.For push
2. For pop
3. For display
4.For exit
Choose number(1-4):- 2
23 is pop from stack


1.For push
2. For pop
3. For display
4.For exit
Choose number(1-4):- 3
The data of stack is:
23,

1.For push
2. For pop
3. For display
4.For exit
Choose number(1-4):- 4


Program END...
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```

Lab No:3   **Date:2081/09/**

Title: Write a Menu based program to show the basic operation of Linear Queue.

Linear Queue:

A Queue Data Structure is a fundamental concept in computer science used for storing and managing data in a specific order. It follows the principle of "First in, First out" (FIFO), where the first element added to the queue is the first one to be removed. Queues are commonly used in various algorithms and applications for their simplicity and efficiency in managing data flow.

Basic Operations on Queue:

Some of the basic operations for Queue in Data Structure are:

• **enqueue()** – Insertion of elements to the queue.

• **dequeue()** – Removal of elements from the queue.

• **peek() or front()-** Acquires the data element available at the front node of the queue without

   deleting it.

• **rear()** – This operation returns the element at the rear end without removing it.

• **isFull()** – Validates if the queue is full.

• **isEmpty()** – Checks if the queue is empty.

• **size()-** This operation returns the size of the queue i.e. the total number of elements it

   contains.

   IDE: Visual Studio Code

   Language: C

Source Code:

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int front = -1, rear = -1, queue[SIZE];
void enqueue();
void dequeue();
void display();
int main()
{
    int choice;
    while (1)
    {
        printf("\nBasic operations on the queue:");
        printf("\n1.Enqueue the element\n2.Dequeue the element\n3.Display\n4.End");
        printf("\n\nEnter the choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            enqueue();
            break;
        case 2:
            dequeue();
            break;
        case 3:
            display();
            break;
        case 4:
            exit(0);
        default:
            printf("\nInvalid choice!!");
        }
    }
}
void enqueue()
{
    int x;
    if (rear == SIZE - 1)

    {
        printf("\nQueue is Full!!");
    }
    else
    {
        if (front == -1)
        {
            front = 0;
```

```c
        }
        printf("the element to be added to the queue: ");
        scanf("%d", &x);
        rear = rear + 1;
        queue[rear] = x;
    }
}
void dequeue()
{
    if (front == -1 || front > rear)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nDequeued element: %d", queue[front]);
        front = front + 1;
    }
}
void display()
{
    if (front == -1 || front > rear)
    {
        printf("\nQUEUE IS FULL!!");
    }
    else
    {
        printf("\nElements present in the queue: \n");
        for (int i = front; i <= rear; i++)
        {
            printf("%d\t", queue[i]);
        }
    }
}
```

Output:

```
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
} ; if ($?) { .\queue }

Basic operations on the queue:
1.Enqueue the element
2.Dequeue the element
3.Display
4.End

Enter the choice: 1
the element to be added to the queue: 23

Basic operations on the queue:
1.Enqueue the element
2.Dequeue the element
3.Display
4.End

Enter the choice: 3

Elements present in the queue:
23
Basic operations on the queue:
1.Enqueue the element
2.Dequeue the element
3.Display
4.End

Enter the choice: █
```

```
Basic operations on the queue:
1.Enqueue the element
2.Dequeue the element
3.Display
4.End

Enter the choice: 2

Dequeued element: 23
Basic operations on the queue:
1.Enqueue the element
2.Dequeue the element
3.Display
4.End

Enter the choice: 3

Elements present in the queue:
24
Basic operations on the queue:
1.Enqueue the element
2.Dequeue the element
3.Display
4.End

Enter the choice: 4
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain> █
```

Lab No:4   **Date:2081/09/**

Title: Write a suitable tail-recursive program of your own idea.

Tail recursion:

The tail recursive functions are considered better than non-tail recursive functions as tail-recursion can be optimized by the compiler.

Compilers usually execute recursive procedures by using a **stack**. This stack consists of all the pertinent information, including the parameter values, for each recursive call. When a procedure is called, its information is **pushed** onto a stack, and when the function terminates the information is **popped** out of the stack. Thus for the non-tail-recursive functions, the **stack depth** (maximum amount of stack space used at any time during compilation) is more.

IDE: Visual Studio Code

Language: C

Source Code:

```c
#include <stdio.h>
int factorial_tail(int n, int a)
{
    if (n == 0 || n == 1)
    {
        return a;
    }
    return factorial_tail(n - 1, n * a);
}
int factorial(int n)
{
    return factorial_tail(n, 1);
}
int main()
{
int n;
    printf("enter the number:\n");
    scanf("%d",&n);
    printf("Factorial of %d is %d\n", n, factorial(n));
    return 0;
}
```

Output:

```
enter the number:
5
Factorial of 5 is 120
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```

Lab No:5   **Date:2081/09/**

Title: Write a program to print all the moves required for Tower of Hanoi problem for user input no of Disk.

Tower of Hanoi:

Puzzle involving three vertical pegs and a set of different sized disks with holes through their centres. The Tower of Hanoi is widely believed to have been invented in 1883 by the French mathematician Édouard Lucas, though his role in its invention has been disputed. Ever popular, made of wood or plastic, the Tower of Hanoi can be found in toy shops around the world.

The typical toy set consists of three pegs fastened to a stand and of eight disks, each having a hole in the centre. The disks, all of different radii, are initially placed on one of the pegs, with the largest disk on the bottom and the smallest on top. The task is to transfer the stack to one of the other pegs subject to two rules: only individual disks may be moved, and no disk may be placed on a smaller disk.

IDE: Visual Studio Code

Language: C

Source Code:

```c
#include<stdio.h>
int TOH(int n,int A, int B,int C)//where n is number of disk and a,b,c is pegs.
{
    static n1=1;
    if (n>0)
    {
        TOH(n-1,B,A,C);
        printf("%d.move disk from %c to %c:\n",n1++,A,C);
        TOH(n-1,A,C,B);// Recursively calculate the number of steps
    }
}
int main()
{
    char a='A';
    char b='B';
    char c='C';
    int n,result;
    printf("enter the value of n:");
    scanf("%d",&n);
    result=TOH(n,a,b,c);
    return 0;
}
```

Output:

```
enter the value of n:3
1.move disk from A to C:
2.move disk from B to C:
3.move disk from B to A:
4.move disk from A to C:
5.move disk from C to B:
6.move disk from A to B:
7.move disk from A to C:
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```

Lab No:6   **Date:2081/09/**

Title: Write a recursive program to find the GCD of user input integers.

Recursion:

It is the process by which function calls itself until some specified condition is satisfied. Each action is started in terms of previous result.

GCD:

The greatest common divisor (GCD) of two or more numbers is the greatest common factor number that divides them, exactly. It is also called the highest common factor (HCF). For example, the greatest common factor of 15 and 10 is 5, since both the numbers can be divided by 5. If a and b are two numbers then the greatest common divisor of both the numbers is denoted by gcd(a, b). To find the gcd of numbers, we need to list all the factors of the numbers and find the largest common factor.

IDE: Visual Studio Code
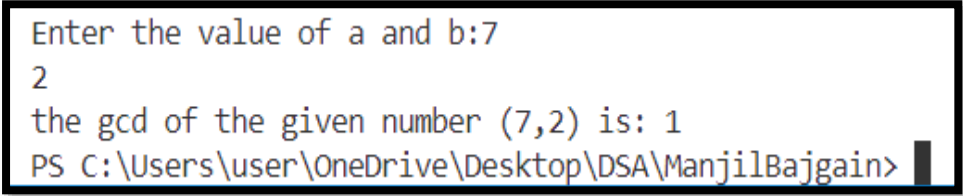
Language: C

Source Code:

```c
#include<stdio.h>
int gcd(int a,int b)
{
    if (b==0)
    {
        return a;
    }
    else
    return gcd(b,a%b);//recursive defination.

}
int main()
{
    int a,b,result;
    printf("Enter the value of a and b:");
    scanf("%d%d",&a,&b);
    result=gcd(a,b);
    printf("the gcd of the given number (%d,%d) is: %d",a,b,result);
    return 0;
}
```

Output:

```
Enter the value of a and b:7
2
the gcd of the given number (7,2) is: 1
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```

Texas

International college

Lab No:7   **Date:2081/09/**

Title: Write a recursive program to calculate the factorial and Fibonacci sequence for user input value.

Recursion:

It is the process by which function calls itself until some specified condition is satisfied. Each action is started in terms of previous result.

Factorial:

In short, a factorial is a function that multiplies a number by every number below it till 1. For example, the factorial of 3 represents the multiplication of numbers 3, 2, 1, i.e. $3! = 3 \times 2 \times 1$ and is equal to 6.

Fibonacci Sequence:

The sequence follows the rule that each number is equal to the sum of the preceding two numbers. The Fibonacci sequence begins with the following 14 integers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233 ... Each number, starting with the third, adheres to the prescribed formula.

IDE: Visual Studio Code

Language: C

Source Code of Factorial:

```
int fact(n)
{
    if (n == 0)

        return 1;

    else
        return (n * fact(n - 1));
}

int factorial(n)
{
    return fact(n, 1);
}
int main()
{
    int n, result;
    printf("Enter the number:");
    scanf("%d", &n);
    result = factorial(n);
    printf("the factorial of the given number is: %d ", result);
}
```

Output:

```
Enter the number:5
the factorial of the given number is: 120
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```

Source Code of Fibonacchi:

```c
#include<stdio.h>

    int fibo(int a)
    {
       if (a==0)
       {
          return 0;
       }
       else if (a==1)
       {
          return 1;
       }
       else
       return fibo(a-1)+fibo(a-2);//recursive defination.

    }
    int main()
{
    int a,b,result;
    printf("Enter the number that you want to find the fibonacci series:");
    scanf("%d",&a);
    result=fibo(a);
    printf("the fibonacci series is:");
    for (int i = 0; i <= a; i++)
    {
       printf("%d,",fibo(i));
    }

    return 0;
}
```

Output:

```
Enter the number that you want to find the fibonacci series:5
the fibonacci series is:0,1,1,2,3,5,
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```