Stack

Collection of items in which elements can be inserted or deleted in a specific order (not from any random position).

* Last in First Out(LIFO) Mechanism
* Ex- Heap of Chair, Pile of Books

Operations-

* push() - Insertion of element
* pop() - Deletion of element
* top() - Find top of the element
* size() – Check size of stack
* isempty() - Check whethere stack is empty or not

Types to Implement stack-

* Static Stack- Size of stack is fixed by user.

CODE-

#include<iostream>

using namespace std;

class stackuse

{

    int \*data;

    int nextindex;

    int capacity;

public:

    stackuse(int totalsize)

    {

        data = new int[totalsize];

        nextindex=0;

        capacity= totalsize;

    }

int size(){

    return nextindex;

}

bool isempty(){

    return nextindex==0;

}

void push(int element)

{

    if(nextindex==capacity){

        cout<<"Stack is Full"<<endl;

        return;

    }

    data[nextindex]= element;

    nextindex++;

}

int pop()

{

    if(isempty()){

        cout<<"Stack is Empty"<<endl;

    }

    nextindex--;

    return data[nextindex];

}

int top()

{

    return data[nextindex-1];

}

};

int main()

{

    stackuse s(4);

    s.push(10);

    s.push(20);

    s.push(30);

    s.push(40);

    s.push(50);

    cout<<s.top()<<endl;

    cout<<s.pop()<<endl;

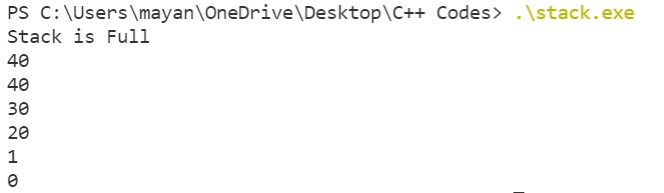
    cout<<s.pop()<<endl;

    cout<<s.pop()<<endl;

    cout<<s.size()<<endl;

    cout<<s.isempty()<<endl;

}



* Dymanic Stack – Size is not Fixed.

CODE-

#include<iostream>

using namespace std;

class stackuse

{

    int \*data;

    int nextindex;

    int capacity;

public:

    stackuse()

    {

        data = new int[4];

        nextindex=0;

        capacity= 4;

    }

int size(){

    return nextindex;

}

bool isempty(){

    return nextindex==0;

}

void push(int element)

{

    if(nextindex==capacity){

        int \*newdata = new int[2\*capacity];

        for(int i=0;i<capacity;i++){

            newdata[i] = data[i];

        }

        capacity \*= 2;

        delete []data;

        data = newdata;

    }

    data[nextindex]= element;

    nextindex++;

}

int pop()

{

    if(isempty()){

        cout<<"Stack is Empty"<<endl;

    }

    nextindex--;

    return data[nextindex];

}

int top()

{

    return data[nextindex-1];

}

};

int main()

{

    stackuse s;

    s.push(10);

    s.push(20);

    s.push(30);

    s.push(40);

    s.push(50);

    s.push(60);

    s.push(70);

    s.push(80);

    cout<<s.top()<<endl;

    cout<<s.pop()<<endl;

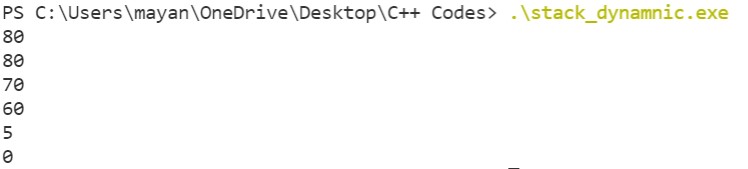
    cout<<s.pop()<<endl;

    cout<<s.pop()<<endl;

    cout<<s.size()<<endl;

    cout<<s.isempty()<<endl;

}



* Using Template

CODE-

#include<iostream>

using namespace std;

template <typename T>

class stackuse

{

    T \*data;

    int nextindex;

    int capacity;

public:

    stackuse()

    {

        data = new T[4];

        nextindex=0;

        capacity= 4;

    }

int size(){

    return nextindex;

}

bool isempty(){

    return nextindex==0;

}

void push(T element)

{

    if(nextindex==capacity){

        T \*newdata = new T[2\*capacity];

        for(int i=0;i<capacity;i++){

            newdata[i] = data[i];

        }

        capacity \*= 2;

        delete []data;

        data = newdata;

    }

    data[nextindex]= element;

    nextindex++;

}

T pop()

{

    if(isempty()){

        cout<<"Stack is Empty"<<endl;

        return 0;

    }

    nextindex--;

    return data[nextindex];

}

T top()

{

    if(isempty()){

        cout<<"Stack is Empty"<<endl;

        return 0;

    }

    return data[nextindex-1];

}

};

int main()

{

    stackuse<int> s;

    s.push(100);

    s.push(101);

    s.push(102);

    s.push(103);

    s.push(104);

    cout<<s.top()<<endl;

    cout<<s.pop()<<endl;

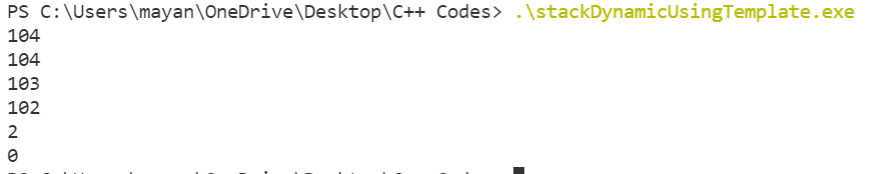
    cout<<s.pop()<<endl;

    cout<<s.pop()<<endl;

    cout<<s.size()<<endl;

    cout<<s.isempty()<<endl;

}



* Using Linked List

CODE-

#include<iostream>

using namespace std;

template <typename T>

class Node

{

public:

    T data;

    Node<T> \*next;

    Node(T data){

        this->data = data;

        next= NULL;

    }

};

template <typename T>

class stack

{

    Node<T> \*head;

    int size;

public:

    stack(){

        head = NULL;

        size=0;

    }

    int getsize(){

        return size;

    }

    bool isempty(){

        return head==NULL;

    }

    void push(T element) {

        Node<T> \*newnode = new Node<T>(element);

        newnode->next = head;

        head = newnode;

        size++;

    }

    T pop() {

        if(isempty())

        {

            return 0;

        }

        T ans = head->data;

        Node<T> \*temp= head;

        head= head->next;

        delete temp;

        size--;

        return ans;

    }

    T top(){

        if(isempty())

        {

            return 0;

        }

        return head->data;

    }

};

int main()

{

    stack<int> s;

    s.push(100);

    s.push(101);

    s.push(102);

    s.push(103);

    s.push(104);

    cout<<s.top()<<endl;

    cout<<s.pop()<<endl;

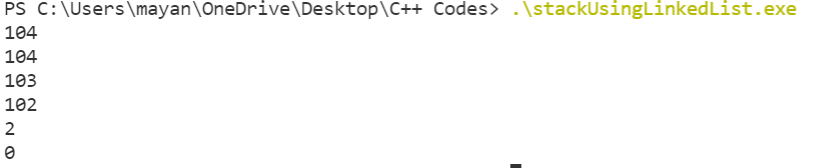
    cout<<s.pop()<<endl;

    cout<<s.pop()<<endl;

    cout<<s.getsize()<<endl;

    cout<<s.isempty()<<endl;

}



* Inbuilt Stack

CODE-

#include<iostream>

using namespace std;

#include <stack>

int main()

{

    stack<int> s;

    s.push(10);

    s.push(20);

    s.push(30);

    s.push(40);

    s.push(50);

    s.push(60);

    cout<<s.top()<<endl;

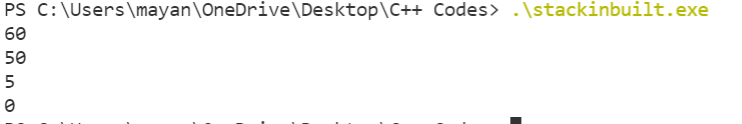
    s.pop();

    cout<<s.top()<<endl;

    cout<<s.size()<<endl;

    cout<<s.empty()<<endl;

}



Queue

Similar as Stack, element inserted & deleted in specific order but it based on FIFO (First in First Out) Mechanism.

Operations-

* enqueue() - Insertion of element
* dequeue () - Deletion of element
* front() - Find top of the element
* size() – Check size of queue
* isempty() - Check whethere stack is empty or not

Types to Implement Queue-

* using Array-

#include<iostream>

using namespace std;

template <typename T>

class myqueue

{

    T \*data;

    int ni; //nextindex

    int fi; //firstindex

    int size;

    int capacity;

public:

    myqueue(int s)

    {

        data = new T[s];

        ni=0;

        fi=-1;

        size=0;

        capacity =s;

    }

    int getsize(){

        return size;

    }

    bool isempty(){

        return size==0;

    }

    void enqueue(T element)

    {

        if(size==capacity){

            cout<<"Queue is Full"<<endl;

            return;

        }

        data[ni]= element;

        ni = (ni+1)%capacity;

        if(fi==-1){

            fi= 0;

        }

        size++;

    }

    T front(){

        if(isempty()){

            cout<<"Queue is Empty";

            return 0;

        }

        return data[fi];

    }

    T dequeue(){

        if(isempty()){

            cout<<"Queue is Empty";

            return 0;

        }

        T ans = data[fi];

        fi = (fi+1)%capacity;

        size--;

        if(size==0)

        {

            fi=-1;

            ni=0;

        }

        return ans;

    }

};

int main()

{

    myqueue<int> q(5);

    q.enqueue(10);

    q.enqueue(20);

    q.enqueue(30);

    q.enqueue(40);

    q.enqueue(50);

    q.enqueue(60);

    cout<<q.front()<<endl;

    cout<<q.dequeue()<<endl;

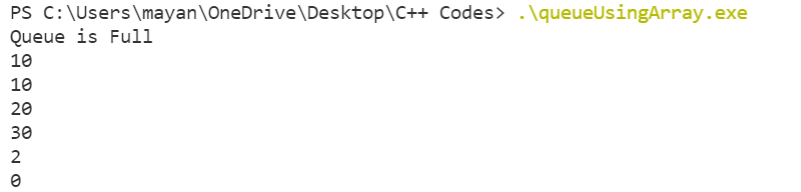
    cout<<q.dequeue()<<endl;

    cout<<q.dequeue()<<endl;

    cout<<q.getsize()<<endl;

    cout<<q.isempty()<<endl;

}



* using Dyanamic Array

#include<iostream>

using namespace std;

template <typename T>

class myqueue

{

    T \*data;

    int ni; //nextindex

    int fi; //firstindex

    int size;

    int capacity;

public:

    myqueue(int s)

    {

        data = new T[s];

        ni=0;

        fi=-1;

        size=0;

        capacity =s;

    }

    int getsize(){

        return size;

    }

    bool isempty(){

        return size==0;

    }

    void enqueue(T element)

    {

        if(size==capacity){

            T \*newdata = new T[2\*capacity];

            int j=0;

            for(int i=fi;i<capacity;i++)

            {

                newdata[j]=data[i];

                j++;

            }

            for(int i=0;i<fi;i++)

            {

                newdata[j]=data[i];

                j++;

            }

            delete []data;

            data= newdata;

            fi=0;

            ni=capacity;

            capacity \*=2;

            //cout<<"Queue is Full"<<endl;

            //return;

        }

        data[ni]= element;

        ni = (ni+1)%capacity;

        if(fi==-1){

            fi= 0;

        }

        size++;

    }

    T front(){

        if(isempty()){

            cout<<"Queue is Empty";

            return 0;

        }

        return data[fi];

    }

    T dequeue(){

        if(isempty()){

            cout<<"Queue is Empty";

            return 0;

        }

        T ans = data[fi];

        fi = (fi+1)%capacity;

        size--;

        if(size==0)

        {

            fi=-1;

            ni=0;

        }

        return ans;

    }

};

int main()

{

    myqueue<int> q(5);

    q.enqueue(10);

    q.enqueue(20);

    q.enqueue(30);

    q.enqueue(40);

    q.enqueue(50);

    q.enqueue(60);

    cout<<q.front()<<endl;

    cout<<q.dequeue()<<endl;

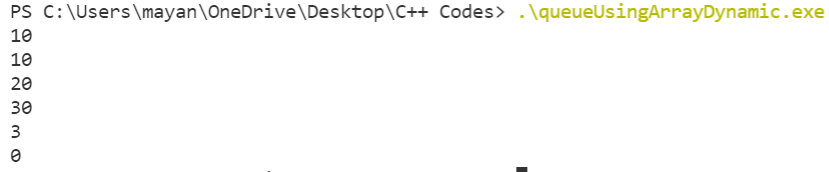
    cout<<q.dequeue()<<endl;

    cout<<q.dequeue()<<endl;

    cout<<q.getsize()<<endl;

    cout<<q.isempty()<<endl;

}



* using Linked List-

#include<iostream>

using namespace std;

template <typename T>

class Node

{

public:

    T data;

    Node<T> \*next;

    Node(T data){

        this->data= data;

        next= NULL;

    }

};

template <typename T>

class queuell

{

    Node<T> \*head;

    Node<T> \*tail;

    int size;

public:

    queuell(){

        head= NULL;

        tail= NULL;

        size=0;

    }

    int getsize(){

        return size;

    }

    bool isempty(){

        return size==0;

    }

    void enqueue(T element)

    {

        Node<T> \*newnode= new Node<T>(element);

        if(head==NULL)

        {

            head=newnode;

            tail=newnode;

        }else

        {

            tail->next= newnode;

            tail= newnode;

        }

        size++;

    }

    T front()

    {

        if(isempty()){

            return 0;

        }

        return head->data;

    }

    T dequeue()

    {

        if(isempty()){

            return 0;

        }

        T ans= head->data;

        Node<T> \*temp=head;

        head= head->next;

        delete temp;

        size--;

        return ans;

    }

};

int main()

{

    queuell<int> q;

    q.enqueue(10);

    q.enqueue(20);

    q.enqueue(30);

    q.enqueue(40);

    q.enqueue(50);

    q.enqueue(60);

    cout<<q.front()<<endl;

    cout<<q.dequeue()<<endl;

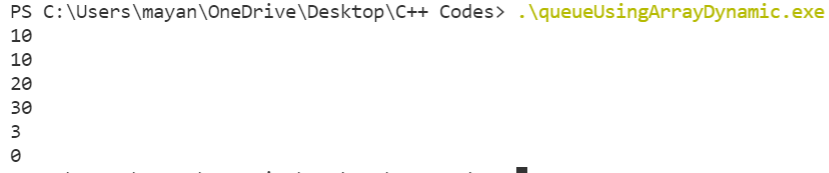
    cout<<q.dequeue()<<endl;

    cout<<q.dequeue()<<endl;

    cout<<q.getsize()<<endl;

    cout<<q.isempty()<<endl;

}



* **Inbuilt Queue**

#include<iostream>

using namespace std;

#include <queue>

int main()

{

    queue<int> q;

    q.push(10);

    q.push(20);

    q.push(30);

    q.push(40);

    q.push(50);

    q.push(60);

    cout<<q.front()<<endl;

    q.pop();

    cout<<q.front()<<endl;

    cout<<q.size()<<endl;

    cout<<q.empty()<<endl;

    while(!q.empty())

    {

        cout<<q.front()<<endl;

        q.pop();

    }

}

