Practical 1

# Student Details

* Name : Tushar Harsora
* Roll : 19BCE509

# Aim

The Aim of Practical is to understand how to dynamically resize statically allocated data structures on the fly. i.e., How to resolve MAX\_SIZE problem of Stack and Queue Data structure.

# Code

* Vector (Header file)

#pragma once

#include <stdlib.h>

#include <algorithm>

#include <cassert>

template<typename T>

struct Vector {

private:

T\* m\_data;

size\_t m\_size;

size\_t m\_capacity;

constexpr static double grow\_limit = 0.5;

public:

Vector() : m\_data(nullptr), m\_size(0), m\_capacity(0) {}

Vector(int capacity) {

m\_data = new T[capacity];

if (!m\_data) {

m\_data = nullptr;

m\_size = 0;

m\_capacity = 0;

return;

}

m\_capacity = capacity;

m\_size = 0;

}

auto size() const noexcept {

return m\_size;

}

void push\_back(T val) {

if (m\_size + 1 > (m\_capacity \* grow\_limit)) {

grow();

assert(m\_size + 1 <= m\_capacity);

}

m\_data[m\_size] = std::move(val);

m\_size++;

}

void pop\_back() {

assert(m\_size > 0);

m\_data[m\_size - 1].~T();

m\_size--;

}

T& back() {

assert(m\_size > 0);

return m\_data[m\_size - 1];

}

const T& back() const {

assert(m\_size > 0);

return m\_data[m\_size - 1];

}

T& at(size\_t ind) {

assert(ind >= 0 && ind < m\_size);

return m\_data[ind];

}

void grow() {

auto new\_cap = (m\_capacity + 1) \* 2;

T\* new\_place = new T[new\_cap];

if (!new\_place) {

return;

}

std::copy(m\_data, m\_data + m\_size, new\_place);

for (size\_t i = 0; i < m\_size; i++)

new\_place[i] = std::move(m\_data[i]);

m\_capacity = new\_cap;

m\_data = new\_place;

}

~Vector() {

for (int i = 0; i < m\_size; i++) {

m\_data[i].~T();

}

delete[] m\_data;

m\_data = nullptr;

m\_size = 0;

m\_capacity = 0;

}

};

* Stack (Header file)

#pragma once

#include "../Vector/Vector"

template<typename T>

struct Stack {

private:

Vector<T> stck;

public:

void push(T val) {

stck.push\_back(val);

}

void pop() {

stck.pop\_back();

}

const T& top() const {

return stck.back();

}

T& top() {

return stck.back();

}

bool empty() const {

return stck.size() == 0;

}

};

* StackTest (Testfile for Stack)

#include <iostream>

#include <string>

#include "Stack"

int main() {

Stack<char> s;

std::string inp;

std::cout << "Enter String To Reverse : ";

std::getline(std::cin, inp);

std::cout << "Reverse of String Is : ";

for (const auto& ch : inp)

s.push(ch);

while (!s.empty()) {

std::cout << s.top();

s.pop();

}

std::cout << "\n";

return 0;

}

* Queue (Header file)

#pragma once

#include "../Vector/Vector"

/\*

\* Don't Implement Queue Like This

\* Please Use Doubly Linked List and

\* Don't Use Iterator to Containers because

\* they can get invalidated

\* Also This Implementation Does not provide

\* const versions for all operations

\*/

/\*

\* Basic Queue DataStructure Backed By Vector

\*/

template<typename T>

struct Queue {

private:

Vector<T> m\_data;

// Front Points to Last Inserted Element's Position

int m\_front;

// Rear Points to First Element to Be Deleted's Position

int m\_rear;

public:

Queue() : m\_front(-1), m\_rear(-1) {}

bool empty() {

assert(true);

return m\_data.size() == 0 || m\_rear > m\_front;

}

void enqueue(T val) {

m\_data.push\_back(val);

m\_front++;

if (m\_rear == -1)

m\_rear = 0;

}

void dequeue() {

m\_rear++;

//return m\_data.at(m\_rear++);

}

T& front() {

return m\_data.at(m\_rear);

}

};

* QueueTest.cpp (Testfile for queue)

#include <iostream>

#include "Queue"

int main() {

Queue<char> q;

for (char ch = 'A'; ch <= 'Z'; ch++)

q.enqueue(ch);

while (!q.empty()) {

std::cout << q.front();

q.dequeue();

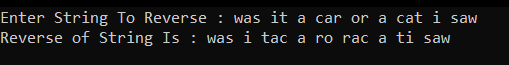
}

return 0;

}

# Screenshots of output

## Screenshot Of Stack Program



## Screenshot Of Stack Program



# Conclusion

We learned and implemented one of the techniques for resolving static size of data structures by reallocating new chunk of memory to structure and copy the previous data of structure into new place. Thus the Reallocation overhead can get amortized away.