#include <stdexcept>

template<typename T>

class ABQ {

    public:

        ABQ();

        ABQ(int capacity);

        ABQ(int capacity, float scale\_factor);

        ABQ(const ABQ& other);

        ABQ& operator=(const ABQ& other);

        ~ABQ();

        void enqueue(T item);

        T peek() const;

        T dequeue();

        unsigned int getSize() const;

        unsigned int getMaxCapacity() const;

        T\* getData();

        unsigned int getTotalResizes() const;

    private:

        unsigned int m\_size;

        unsigned int m\_capacity;

        unsigned int head {};

        T\* m\_data;

        unsigned int total\_resizes {};

        static float c\_scale\_factor;

        void copy(const ABQ& other);

        void increase\_capacity();

        void decrease\_capacity();

};

template<typename T>

float ABQ<T>::c\_scale\_factor;

template<typename T>

ABQ<T>::ABQ() {

    m\_size = 0;

    m\_capacity = 1;

    m\_data = new T[m\_capacity];

    c\_scale\_factor = 2.0f;

}

template<typename T>

ABQ<T>::ABQ(int capacity) {

    m\_size = 0;

    m\_capacity = capacity;

    m\_data = new T[m\_capacity];

    c\_scale\_factor = 2.0f;

}

template<typename T>

ABQ<T>::ABQ(int capacity, float scale\_factor) {

    m\_size = 0;

    m\_capacity = capacity;

    m\_data = new T[m\_capacity];

    c\_scale\_factor = scale\_factor;

}

template<typename T>

ABQ<T>::ABQ(const ABQ& other) {

    copy(other);

}

template<typename T>

ABQ<T>& ABQ<T>::operator=(const ABQ& other) {

    copy(other);

    return \*this;

}

template<typename T>

void ABQ<T>::copy(const ABQ& other) {

    delete[] m\_data;

    m\_size = other.m\_size;

    m\_capacity = other.m\_capacity;

    head = other.head;

    total\_resizes = other.total\_resizes;

    m\_data = new T[m\_capacity];

    // Deep copy of array data

    for (unsigned int i{}; i < m\_capacity; ++i) {

        m\_data[i] = other.m\_data[i];

    }

}

template<typename T>

ABQ<T>::~ABQ() {

    delete[] m\_data;

}

template<typename T>

void ABQ<T>::enqueue(T item) {

    // Check if stack is full, and resize if yes

    if (m\_size == m\_capacity) {

        increase\_capacity();

    }

    m\_data[m\_size++] = item;

}

template<typename T>

void ABQ<T>::increase\_capacity() {

    T\* new\_data = new T[static\_cast<int>(m\_capacity \* c\_scale\_factor)];

    // Deep copy of array data

    for (unsigned int i{}; i < m\_capacity; ++i) {

        new\_data[i] = m\_data[head + i];

    }

    // Cleanup

    m\_capacity \*= c\_scale\_factor;

    delete[] m\_data;

    m\_data = new\_data;

    ++total\_resizes;

    head = 0;

}

template<typename T>

T ABQ<T>::peek() const {

    if (m\_size == 0) {

        throw std::runtime\_error("Stack is empty");

    }

    return m\_data[head];

}

template<typename T>

T ABQ<T>::dequeue() {

    if (m\_size == 0) {

        throw std::runtime\_error("Stack is empty");

    }

    T value = m\_data[head];

    ++head;

    // Check if capacity is too small, and resize if yes

    if ((static\_cast<float>(m\_capacity) -1) / --m\_size >= c\_scale\_factor) {

        decrease\_capacity();

    }

    return value;

}

template<typename T>

void ABQ<T>::decrease\_capacity() {

    // If min size

    if (m\_capacity == 1) {

        return;

    }

    m\_capacity /= c\_scale\_factor;

    T\* new\_data = new T[m\_capacity];

    // Deep copy of array data

    for (unsigned int i{}; i < m\_capacity; ++i) {

        new\_data[i] = m\_data[head + i];

    }

    delete[] m\_data;

    m\_data = new\_data;

    ++total\_resizes;

    head = 0;

}

template<typename T>

unsigned int ABQ<T>::getSize() const {

    return m\_size;

}

template<typename T>

unsigned int ABQ<T>::getMaxCapacity() const {

    return m\_capacity;

}

template<typename T>

T\* ABQ<T>::getData() {

    return m\_data;

}

template<typename T>

unsigned int ABQ<T>::getTotalResizes() const {

    return total\_resizes;

}