```cpp
#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>
```

```cpp
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;
}
```

```cpp
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;
}
```