Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

Высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий математики механики

Очередь

Отчет по лабораторной работе

Выполнил:

студент ИИТММ гр. 381706-2

Антипин А.С.\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:

ассистент каф. МОСТ, ИИТММ

Лебедев И.Г\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Нижний Новгород

2018 г.

**Содержание**

[Введение 3](#_Toc534290481)

[Постановка целей и задач 4](#_Toc534290482)

[Руководство пользователя 5](#_Toc534290483)

[Руководство программиста 6](#_Toc534290484)

[Описание структуры программы 6](#_Toc534290485)

[Описание структур данных 7](#_Toc534290486)

[Описание алгоритмов 8](#_Toc534290487)

[Заключение 9](#_Toc534290488)

[Литература 10](#_Toc534290489)

[Приложение 11](#_Toc534290490)

# Введение

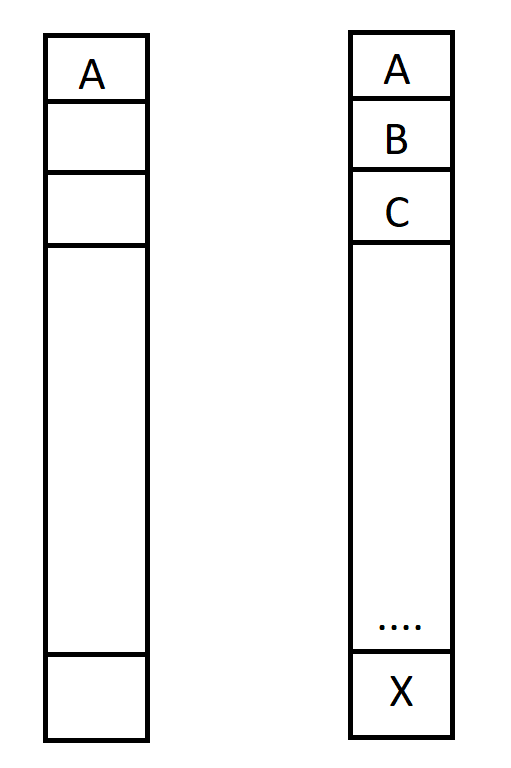
Стек является одной из самых полезных структур хранения данных. Доступ к хранящимся в нем элементам осуществляется посредствам «вынимания» других элементов из конца стека. Но иногда такой способ хранения не всегда удобен, и на ряду со стеком существует другая структура хранения информации – очередь. Очередь – это структура, при которой элементы хранятся по принципу «первым вошел, первым вышел», что является прямой противоположностью стеку. Если у нас есть пустая очередь, несколько элементов, то сначала извлечётся первый элемент, а затем все последующие (рис. 1).

рис. 1 (представление элементов в очереди. 1-вый столбец – положили один элемент в очередь, 2-ой столбец – заполненная очередь)

Практическое применение очередь, как и стек, нашла в алгоритмах задачах.

# Постановка целей и задач

Основной целью лабораторной работы является создание структуры данных «Очередь» и реализация таких методов, как доступ к элементам стека:

* Положить элемент в очередь;
* Вернуть элемент с удалением;
* Вернуть элемент без удаления;

Для реализации алгоритмов будет использоваться шаблонный класс Queue, который будет унаследован от класса Stack с добавлением 2 полей и перегрузкой методов доступа к элементам.

Для проверки правильности работы этих классов будут написаны тесты с использованием фреймворка Google Test, а также тестовый образец программы, которая использует данный класс.

# Руководство пользователя

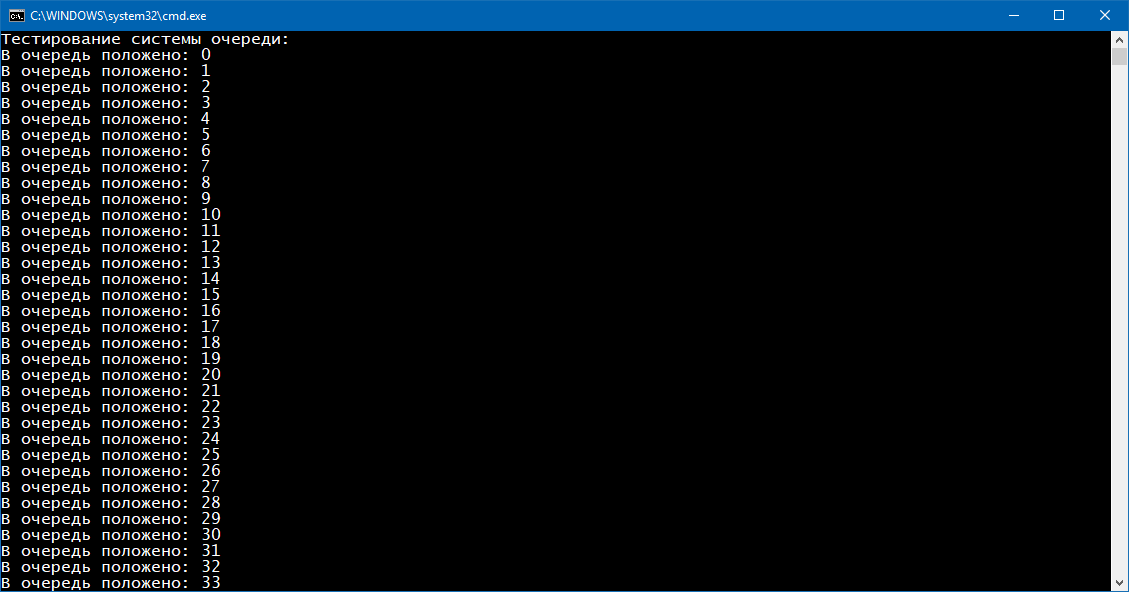
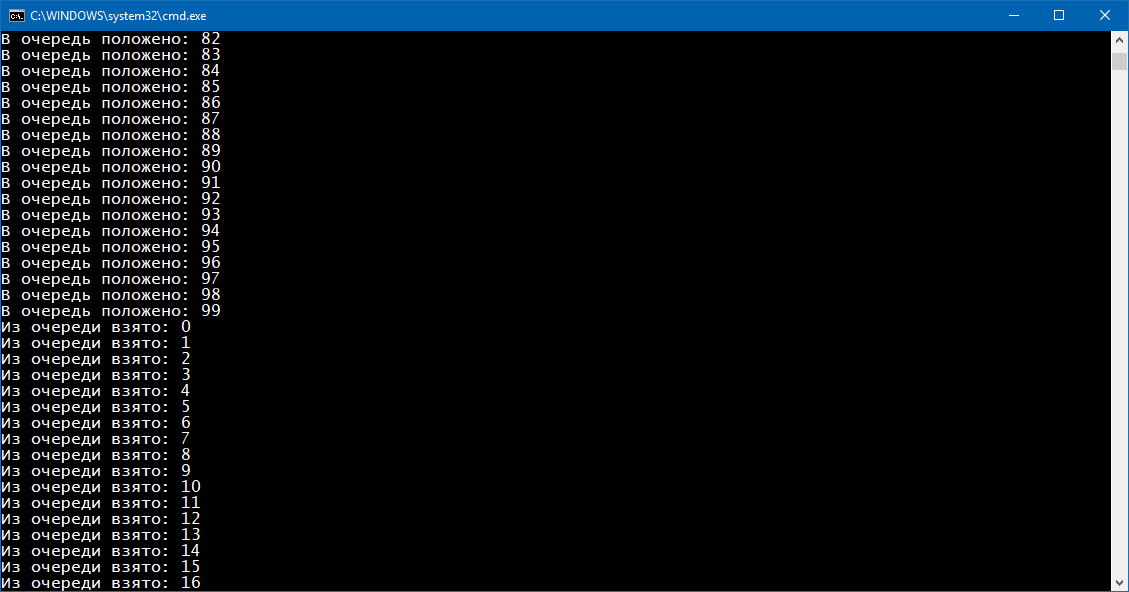
После запуска программы пользователя встречает консольное окно (рис. 2):

рис. 3 (вывод работы очереди для пользователя, из очереди забираются элементы с 0 до 99)

рис. 2 (вывод работы очереди для пользователя, в нее кладутся числа от 0 до 99)

В очередь кладутся числа от 0 до 99, а затем из нее забираются эти элементы, причем сначала забирается 0, 1, и т.д., а уже потом 99 (рис. 3).

# Руководство программиста

## Описание структуры программы

Объект типа очередь будет выделять необходимое количество памяти под объекты определенного типа, которое будет задаваться при его создании. Т.е. максимальное число объектов в очереди будет ограниченно.

Для реализации алгоритмов будет использоваться класс Stack, как предок для класса Queue.

Алгоритм очереди написан таким образом, что она как бы «закольцована», т.е. если у нас имеется свободное место, а индекс первого элемента не является 0, то при окончании места в конце очереди, новые элементы будут записываться в начало. Такое решение позволяет избежать лишних циклов на смещение элементов и сохранить сложность алгоритма доступа к элементам равной O(1).

А также проект использующий фреймворк Google Test, для проверки правильности работы класса стек, а также тесовый проект, который будет показываться пользователю.

**Класс Queue**

**queue.h**

Древо классов

**Класс gtest**

**test\_main.cpp**

**Stack\_test.cpp**

**Проект для пользователя**

**main.cpp**

**stack.h**

**Класс Stack**

**Класс Stack:**

В этом классе объявлены поля, которые необходимы для класса очередь, и реализованы методы работы со стеком, такие как: положить элемент, забрать элемент, вернуть текущее количество элементов в стеке, проверить является ли стек пустым или полным в данный момент.

**Класс Queue:**

В этом классе реализована сама очередь с методами хранения и доступа к элементам.

**Класс gtest:**

Класс gtest реализует тестирование класса Queue, по средствам фреймворка Google Test. Тесты пишутся для каждого метода классов, каждого ветвления этих методов и для всех возможных исключений этих методов.

**Проект Queue:**

В данном проекте реализован примет использования очереди для заполнения элементами и их последующего забора.

## Описание структур данных

**Класс Stack:**

template <class T> - шаблон класса T

int Len; - максимальная длина очереди:

T \*Mem; - указатель на первый элемент очереди типа Т;

int Index; - индекс последнего элемента в очереди;

int counts; - текущее количество элементов в очереди;

int IndexFirst; - индекс первого элемента;

Exceptions\_from\_stack\_queue\_multystack except; - переменная для вызова исключений.

**Описание методов:**

|  |  |
| --- | --- |
| Метод: | Описание: |
| Queue<T>::Queue() : Stack<T>() | Конструктор по умолчанию очереди, который наследуется от конструктора стека. |
| Queue<T>::Queue(const int \_Len) | Конструктор с параметром, который принимает длину очереди. |
| Queue<T>::Queue(const Queue<T>& A) | Конструктор копирования очереди, который принимает константную ссылку на другой стек. |
| void Queue<T>::Put(const T& A) | Метод, который позволяет положить элемент в очередь. |
| T Queue<T>::Get() | Возвращение элемента из очереди с его удалением. |
| T Queue<T>::GetWithoutDelete() | Возвращение элемента из очереди без удаления. |
| bool Queue<T>::IsFull() | Проверка очереди на полноту. |
| bool Queue<T>::IsEmpty() | Проверка очереди на пустоту. |

## Описание алгоритмов

**Подробное описание некоторых методов**

Добавление элемента в очередь:

* Проверка на полноту, если очередь полна, то выдается ошибка о полноте очереди;
* Если в очереди есть свободное место, то элемент кладется в очередь на место Index;
* Если количество элементов в очереди не равно максимальному количеству, и индекс последнего элемента не последний, то этот индекс увеличивается на 1;
* Если количество элементов в очереди не равно максимальному количеству, но индекс последнего элемента последний, то индекс становится равным 0;
* Если количество элементов в очереди равно максимальному количеству, то индекс становится равным 0.

Возврат элемента из очереди:

* Проверка на пустоту, если очередь пуста, то выдается ошибка о пустоте очереди;
* Если в очереди есть элементы, то количество элементов в очереди уменьшается на 1, а нужный элемент запоминается.
* Если количество элементов в очереди не равно 0, и индекс первого элемента не последний, то этот индекс увеличивается на 1;
* Если количество элементов в очереди 0, но индекс первого элемента последний, то индекс первого становится равным 0;
* Если количество элементов в очереди равно 0, то индекс первого и последнего элементов становятся равным 0;
* Возвращается запомнившийся элемент.

# Заключение

Подводя итоги можно сказать, что очередь, как и стек, имеет огромное практическое применение, а ее реализация не составляет труда. Скорость работы очереди также максимальна, т.к. в алгоритмах ее реализации не присутствует циклов, а значит сложность O(1). Все эти неоспоримые преимущества показывает доказывают эффективность применения структуры данных «очередь» в практических задачах.

# Литература

* Учебные материалы к учебному курсу «Методы программирования» - Гергель В.П.
* http://www.cyberforum.ru/cpp-beginners/thread587712.html

# Приложение

**queue.h:**

#pragma once

#include "Stack.h"

template<class T>

class Queue : public Stack<T>

{

protected:

int counts;

int IndexFirst;

public:

Queue();

Queue(const int \_Len);

Queue(const Queue<T>& A);

void Put(const T& A);

T Get();

T GetWithoutDelete();

bool IsFull();

bool IsEmpty();

};

template<class T>

Queue<T>::Queue() : Stack<T>()

{

counts = 0;

IndexFirst = 0;

}

template<class T>

Queue<T>::Queue(const int \_Len)

{

if (\_Len < 0)

except.except\_throw(106);

else if (\_Len == 0)

{

Len = 0;

Mem = NULL;

Index = 0;

counts = 0;

IndexFirst = 0;

}

else

{

Len = \_Len;

Mem = new T[Len];

for (int i = 0; i < Len; i++)

Mem[i] = 0;

Index = 0;

IndexFirst = 0;

counts = 0;

}

}

template<class T>

Queue<T>::Queue(const Queue<T>& A)

{

Len = A->Len;

Mem = new T[Len];

for (int i = 0; i < Len; i++)

Mem[i] = A->Mem[i];

Index = A->Index;

IndexFirst = A.IndexFirst;

counts = A.counts;

}

template<class T>

void Queue<T>::Put(const T& A)

{

if (IsFull() == false)

{

Mem[Index] = A;

counts++;

if (counts != Len && Index != Len)

Index++;

else if (counts != Len && Index == Len)

Index = 0;

else if (counts == Len)

Index = 0;

}

else

except.except\_throw(107);

}

template<class T>

T Queue<T>::Get()

{

if (IsEmpty() == false)

{

counts--;

T A = Mem[IndexFirst];

if (counts != 0 && IndexFirst != Len - 1)

IndexFirst++;

else if (counts != 0 && IndexFirst == Len - 1)

IndexFirst = 0;

else if (counts == 0)

IndexFirst = Index = 0;

return A;

}

else

except.except\_throw(108);

}

template<class T>

T Queue<T>::GetWithoutDelete()

{

if (IsEmpty() == false)

{

T A = Mem[IndexFirst];

return A;

}

else

except.except\_throw(108);

}

template<class T>

bool Queue<T>::IsFull()

{

if (counts == Len)

return true;

else if (counts > Len)

except.except\_throw(109);

else

return false;

}

template<class T>

bool Queue<T>::IsEmpty()

{

if (counts == 0)

return true;

else if (counts > Len)

except.except\_throw(110);

if (counts > 0)

return false;

}

**Queue\_test.cpp:**

#include <gtest.h>

#include "Queue.h"

TEST(Queue, can\_create\_queue\_with\_positive\_len)

{

ASSERT\_NO\_THROW(Queue<int> A(5));

}

TEST(Queue, can\_create\_queue\_with\_negative\_len)

{

ASSERT\_ANY\_THROW(Queue<int> A(-2));

}

TEST(Queue, can\_put\_item)

{

Queue<int> A(3);

A.Put(10);

A.Put(5);

EXPECT\_EQ(10, A.Get());

}

TEST(Queue, can\_throw\_when\_put\_item\_in\_full\_queue)

{

Queue<int> A(3);

A.Put(10);

A.Put(5);

A.Put(1);

ASSERT\_ANY\_THROW(A.Put(3));

}

TEST(Queue, can\_get\_item)

{

Queue<int> A(3);

A.Put(10);

A.Put(5);

A.Put(2);

EXPECT\_EQ(10, A.Get());

}

TEST(Queue, can\_get\_without\_delete\_item)

{

Queue<int> A(3);

A.Put(10);

A.Put(5);

A.Put(2);

EXPECT\_EQ(10, A.GetWithoutDelete());

EXPECT\_EQ(10, A.GetWithoutDelete());

}

TEST(Queue, custom\_put\_and\_get\_item)

{

Queue<int> A(3);

A.Put(10);

A.Put(5);

A.Put(2);

A.Get();

A.Put(15);

A.Get();

EXPECT\_EQ(2, A.Get());

EXPECT\_EQ(15, A.Get());

}

TEST(Queue, can\_throw\_when\_get\_item\_in\_empty\_queue)

{

Queue<int> A(3);

ASSERT\_ANY\_THROW(A.Get());

}

TEST(Queue, can\_report\_if\_queue\_is\_full)

{

Queue<int> A(3);

A.Put(10);

A.Put(5);

A.Put(2);

EXPECT\_EQ(1, A.IsFull());

}

TEST(Queue, can\_report\_if\_queue\_is\_not\_full)

{

Queue<int> A(3);

A.Put(10);

EXPECT\_EQ(0, A.IsFull());

}

TEST(Queue, can\_report\_if\_queue\_is\_empty)

{

Queue<int> A(3);

EXPECT\_EQ(1, A.IsEmpty());

}

TEST(Queue, can\_report\_if\_queue\_is\_not\_empty)

{

Queue<int> A(3);

A.Put(10);

EXPECT\_EQ(0, A.IsEmpty());

}

**main.cpp:**

#include "Queue.h"

#include <iostream>

void main()

{

Queue<int> qu(100);

std::cout << "Тестирование системы очереди:" << std::endl;

for (int i = 0; i < 100; i++)

{

qu.Put(i);

std::cout << "В очередь положено: " << i << std::endl;

}

for (int i = 0; i < 100; i++)

std::cout << "Из очереди взято: " << qu.Get() << std::endl;

}