Министерство образования Республики Беларусь

ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра технологий программирования

**Методические указания  
 к лабораторной работе № 2\_8  
 по курсу «Основы алгоритмизации   
и программирования»**

«Динамические структуры данных:

очереди, стеки, деки»

Преподаватель: Войтехович   
Агния Витольдовна

Составитель: Войтехович   
Агния Витольдовна

Полоцк, 2017

# **ЦЕЛЬ РАБОТЫ**

Изучить методы организации списочных структур в динамической памяти. Реализовать алгоритмы помещения и изъятия элементов из стека, дека или очереди.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

# 1 CТЕК

**Стек** характерен тем, что получить доступ к его элементам можно лишь с одного конца, называемого **вершиной стека**; иначе говоря:

***стек – структура данных типа «список», функционирующая по принципу LIFO (last in — first out, «последним пришёл — первым вышел»).***

Графически его удобно изобразить в виде вертикального списка (смотрите рисунок 1), например, стопки книг, где чтобы воспользоваться одной из них, и не нарушить установленный порядок, нужно поднять все те книги, что лежат выше нее, а положить книгу можно лишь поверх всех остальных.

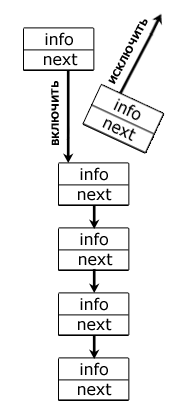


Рисунок 1 - Стек

Впервые стек был предложен в 1946 году Аланом Тьюрингом, как средство возвращения из подпрограмм. В 1955 году немцы Клаус Самельсон и Фридрих Бауэр из Технического университета Мюнхена использовали стек для перевода языков программирования и запатентовали идею в 1957 году. Но международное признание пришло к ним лишь в 1988 году.

На рисунке 1 показан стек, операции над элементами которого, происходят строго с одного конца: для включения нужного элемента в ***n-ую*** ячейку, необходимо сдвинуть ***n-1*** элементов, и исключить тот элемент, который занимает ***n-ую*** позицию.

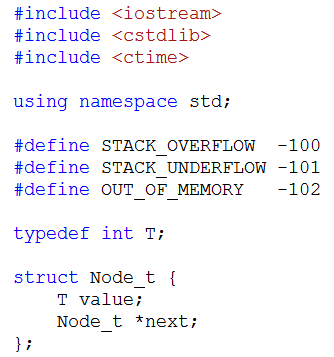
Стек, чаще всего, реализуется на основе обычных массивов, односвязных и двусвязных списков. В зависимости от конкретных условий, выбирается одна из этих структур данных.

Основными операциями над стеками являются:

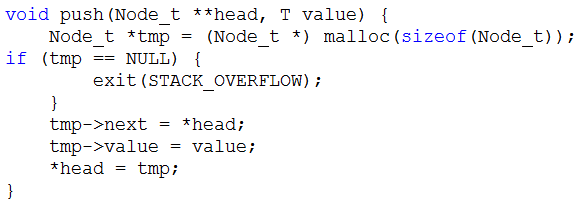
* добавление элемента;
* удаление элемента;
* чтение верхнего элемента.

## 1.1 Динамическая реализация стека на основе списка

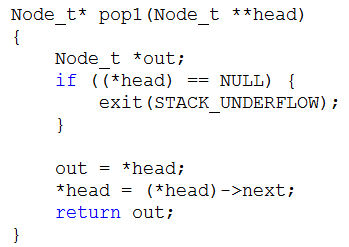
Для начала определим структуру **узел:**

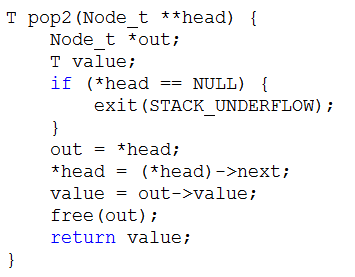


**Функция вставки первого элемента** проста: создаём новый узел. Указатель ***next*** кидаем на старый узел. Далее указатель на вершину стека перекидываем на вновь созданный узел. Теперь вершина стека указывает на новый узел.



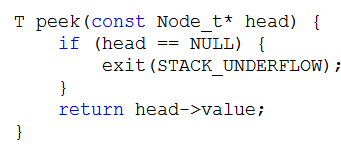
Функция ***pop*** берёт первый элемент (тот, на который указывает вершина), перекидывает указатель на следующий элемент и возвращает первый. Здесь есть два варианта – можно вернуть узел или значение. Если вернём значение, то придётся удалять узел внутри функции



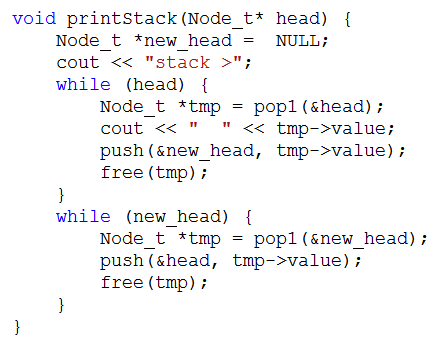


Теперь вместо проверки на длину массива везде используется проверка на равенство NULL вершины стека.

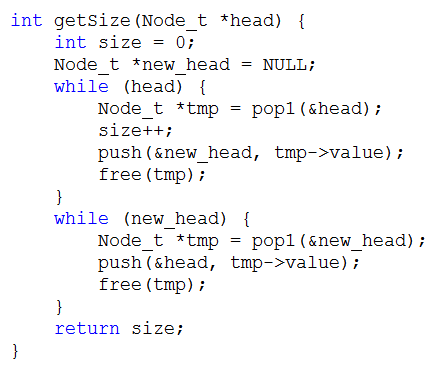
Простая функция peek возвращает значение, хранящееся в голове стека.



Печать стека достаточно интересна: для этого необходимо извлекать элемент из стека, печатать его значение на экран, а затем добавлять его в дополнительный временный (служебный) стек. После того, как основной стек будет опустошен, все элементы по очереди возвращаются из дополнительного стека в основной, затем дополнительный стек уничтожается.

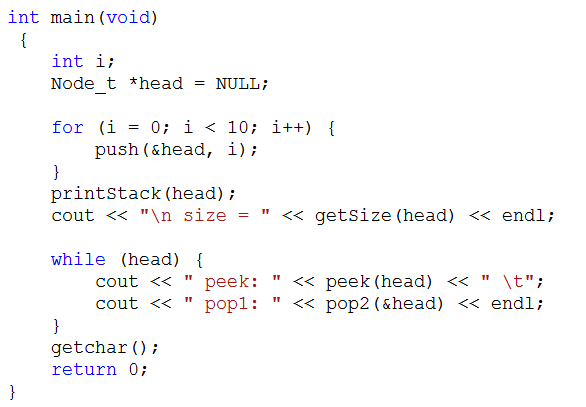


И ещё одна проблема – теперь нельзя просто посмотреть размер стека. Функция, возвращающая количество элементов стека работает аналогично функции печати:

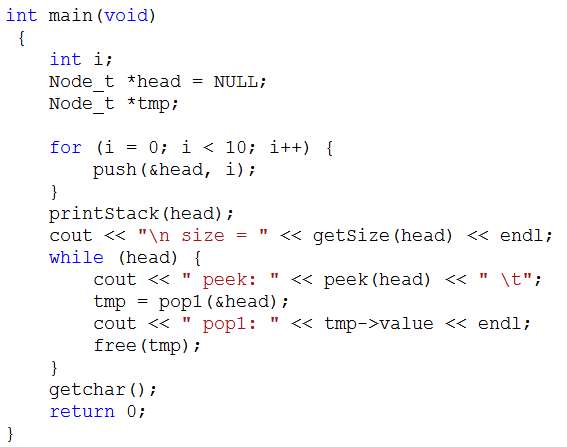


Конечно, можно хранить размер отдельно, можно обернуть стек со всеми данными ещё в одну структуру и т.д.

Тестируем



или так



На рисунках 2 и 3 представлены результаты тестирования работы двух вариантов функции ***main().***

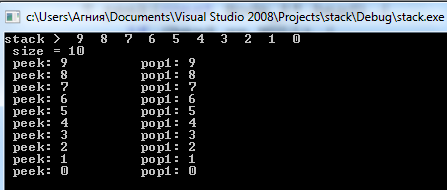


Рисунок 2 – Результат тестирования первого варианта функции main()

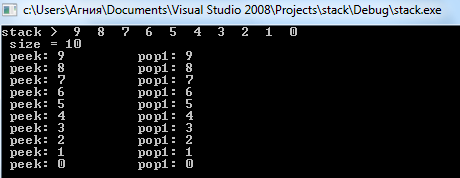


Рисунок 3 – Результат тестирования второго варианта функции main()

# 2 Односвязная ОЧЕРЕДЬ FIFO

**Очередь –** структура данных типа «список», позволяющая добавлять элементы лишь в конец списка, и извлекать их из его начала. Она функционирует по принципу **FIFO** (First In, First Out — «первым пришёл — первым вышел»), для которого характерно, что все элементы a1, a2, …, an-1, an, добавленные раньше элемента an+1, должны быть удалены прежде, чем будет удален элемент an+1.

Также очередь может быть определена как частный случай односвязного списка, который обслуживает элементы в порядке их поступления. Как и в «живой» очереди, здесь первым будет обслужен тот, кто пришел первым.

Графическое отображение однонаправленной очереди представлено на рисунке 4.

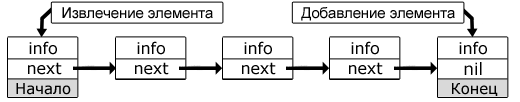


Рисунок 4 – Однонаправленная очередь

Стандартный **набор операций** (часто у разных авторов он не идентичен), выполняемых над очередями, совпадает с тем, что используется при обработке стеков:

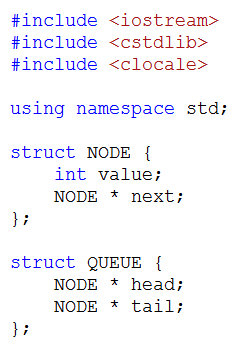
* добавление элемента;
* удаление элемента;
* чтение первого элемента.

Только, если в отношении стека в момент добавления или удаления элемента допустимо задействование лишь его вершины, то касательно очереди эти две операции должны быть применены так, как это регламентировано в определении этой структуры данных, т. е. добавление – в конец, удаление – из начала. Далее, при реализации интерфейса очереди, список стандартных операций будет расширен.

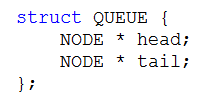
Выделяют два способа программной реализации очереди. Первый из них основан на базе массива, а второй на базе указателей (связного списка). Первый способ – статический, т. к. очередь представляется в виде простого статического массива, второй – динамический.

## 2.1 Динамическая реализация односвязной очереди на основе списка

Данный способ предполагает работу с динамической памятью. Для представления очереди используется односвязный список, в конец которого помещаются новые элементы, а старые извлекаются, соответственно, из начала списка. Здесь каждый узел списка имеет два поля: информационное и связующее:



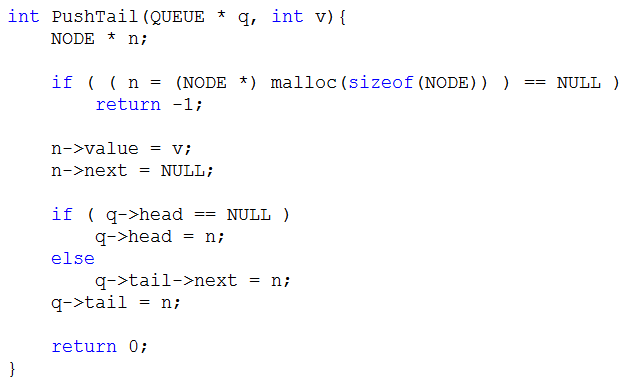
Также понадобиться определить указатели на начало и конец очереди:



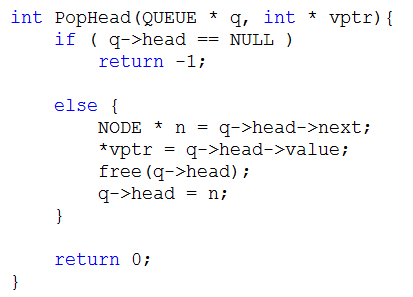
Следующее консольное приложение обслуживает очередь, каждый элемент которой – целое число. Весь процесс обуславливают две основные операции:

* PushTail;
* PopHead.

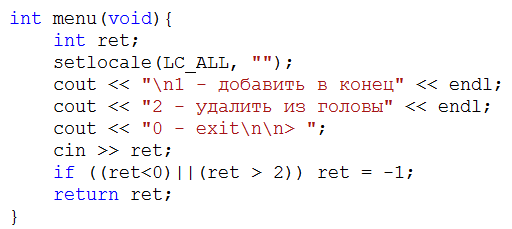
Функция ***PushTail()*** добавляет в конец очереди новый элемент:



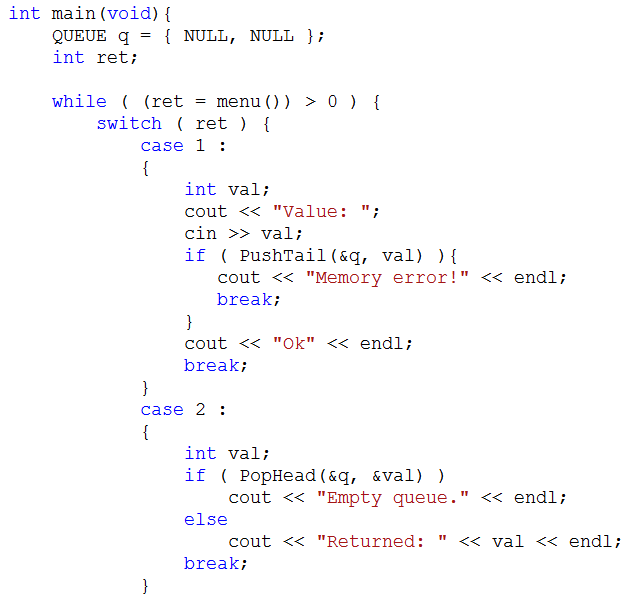
Функция ***PopHead()*** удаляет один элемент из головы очереди.

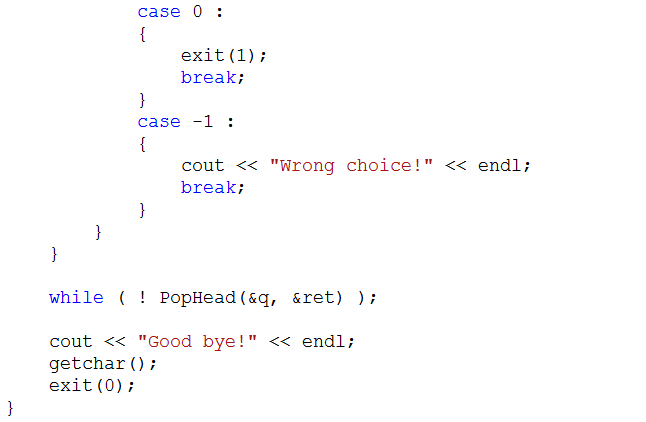


Для упрощения читаемости исходного кода вынесем операции работы с меню в отдельную функцию ***menu().***



Протестируем функции, написав следующее содержание функции ***main()***. Результаты тестирования представлены на рисунке 5.





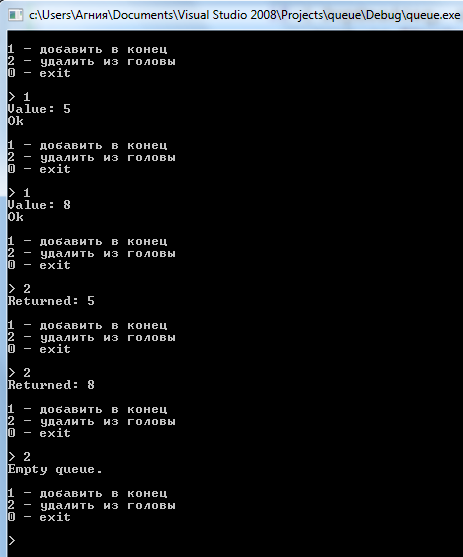


Рисунок 5 – Результат тестирования функций работы с очередью

# 3 ДЕК

**Дек** (deque — double ended queue, «двусторонняя очередь») – структура данных типа «список», функционирующая одновременно по двум принцам организации данных: **FIFO и LIFO** (смотрите рисунок 6). Определить дек можно как очередь с двумя сторонами, так и стек, имеющий два конца. То есть данный подвид списка характерен двухсторонним доступом: выполнение поэлементной операции, определенной над деком, предполагает возможность выбора одной из его сторон в качестве активной.

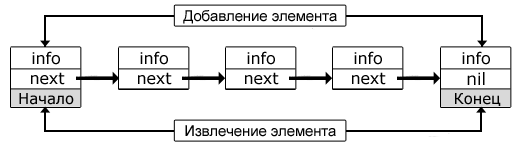


Рисунок 6 – Дек (двунаправленная очередь)

Число основных операций, выполняемых над стеком и очередью, как помнит читатель, равнялось трем: добавление элемента, удаление элемента, чтение элемента. При этом не указывалось место структуры данных, активное в момент их выполнения, поскольку ранее оно однозначно определялось свойствами (определением) самой структуры. Теперь, ввиду дека как обобщенного случая, для приведенных операций следует указать эту область. Разделив каждую из операций на две: одну применительно к «голове» дека, другую – его «хвосту», получим набор из шести операций:

* добавление элемента в начало;
* добавление элемента в конец;
* удаление первого элемента;
* удаление последнего элемента;
* чтение первого элемента;
* чтение последнего элемента.

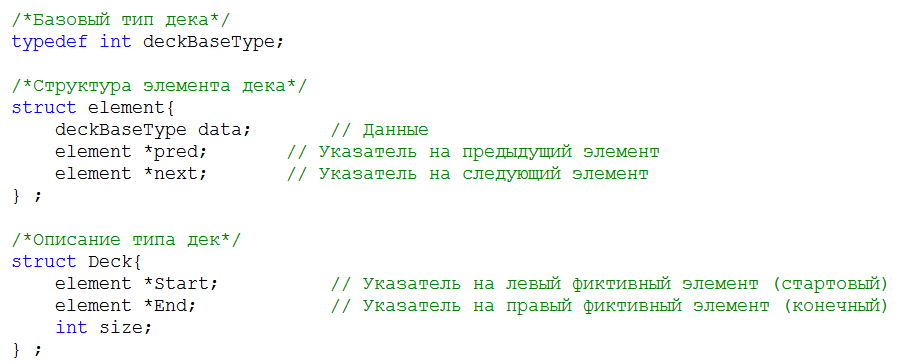
На практике этот список может быть дополнен проверкой дека на пустоту, получением его размера и некоторыми другими операциями.

В плане реализации двусторонняя очередь очень близка к стеку и обычной очереди: в качестве ее базиса приемлемо использовать как массив, так и список.

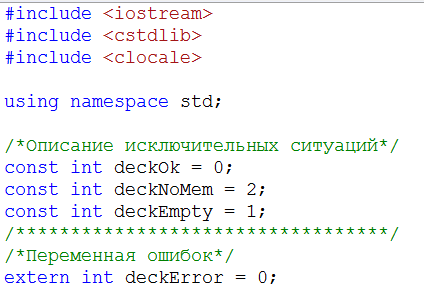
## 3.1 Реализация дека (двунаправленной очереди)

Обычно для реализации дека используют ***двусвязный линейный список***. То есть, на основе готовых функций списка создают новые для **дека**. Иными словами, для реализации двунаправленной очереди необходимо переписать несколько функций, которые будут обеспечивать корректную работу с динамически организованным двусвязным списком, как с деком.

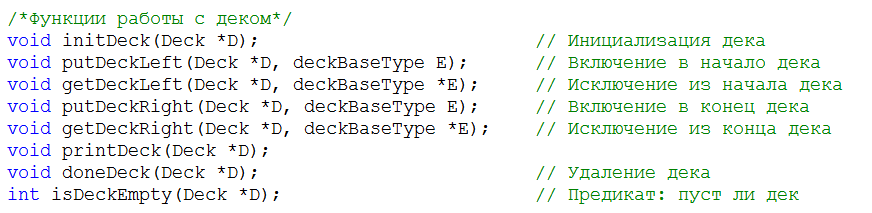
В первую очередь опишем узел и саму структуру данных:



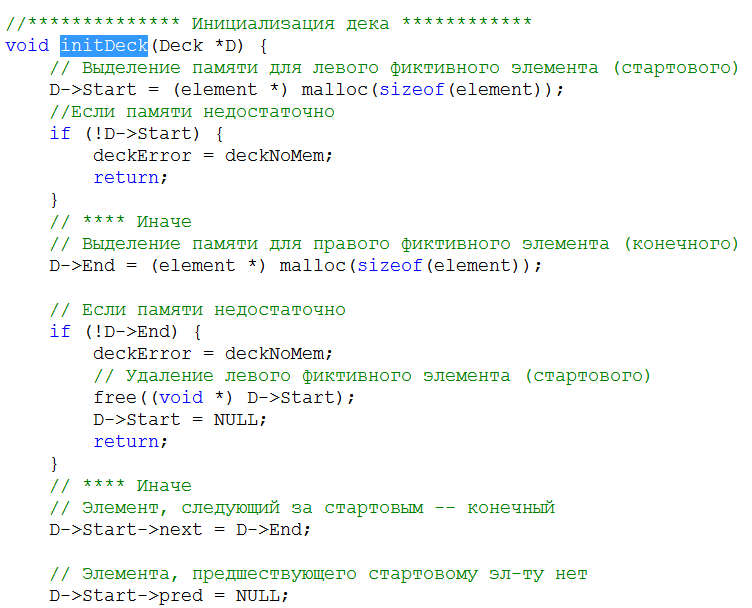
Кроме того, добавим в начале файла переменные, отвечающие за служебную информацию, и подключим нужные библиотеки.

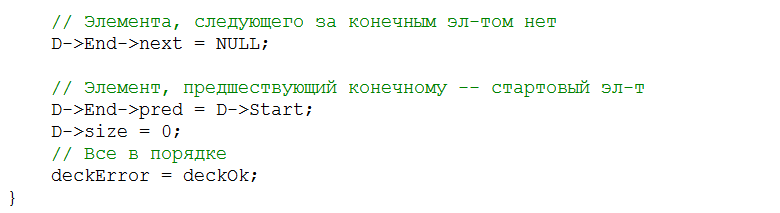


Опишем функции для непосредственной работы с **деком**.

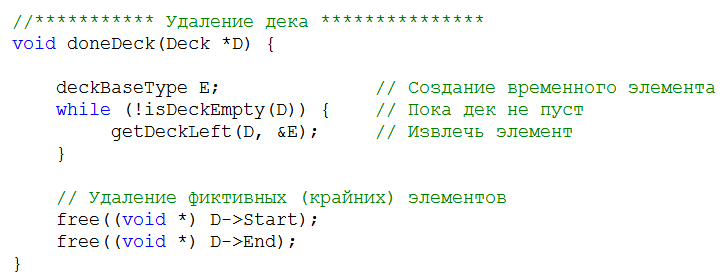


Инициализация нового дека:

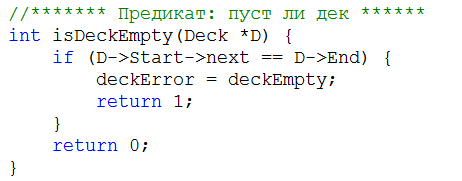




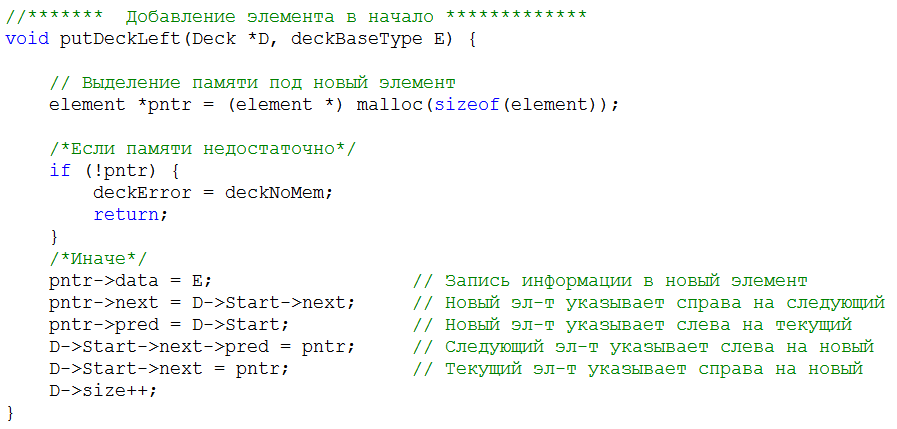
Очистка дека:

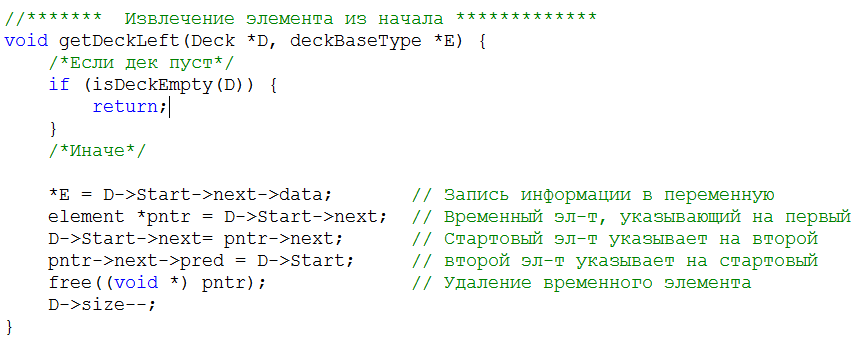


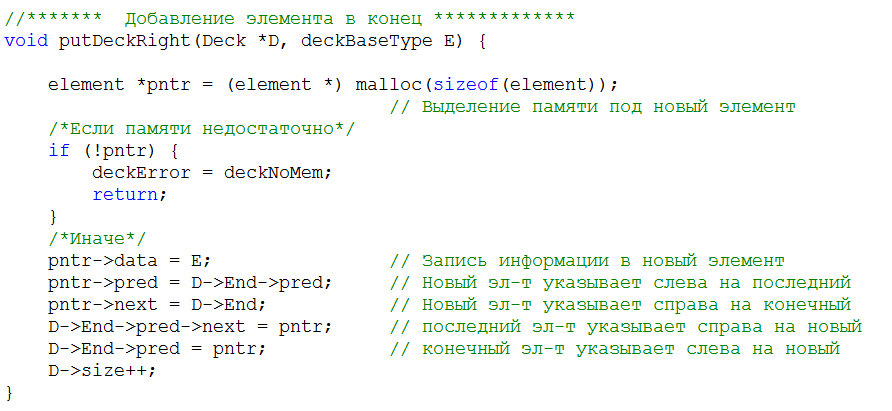
Функция, проверяющая, пустой дек или нет:

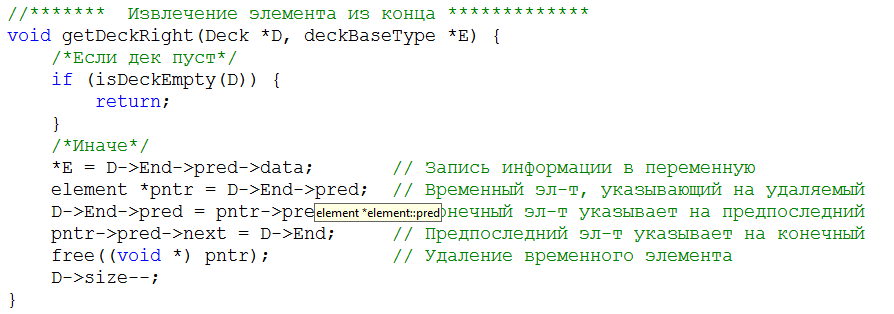


Далее описаны функции, которые обеспечивают добавление и изъятие элементов из обоих концов двунаправленной очереди.

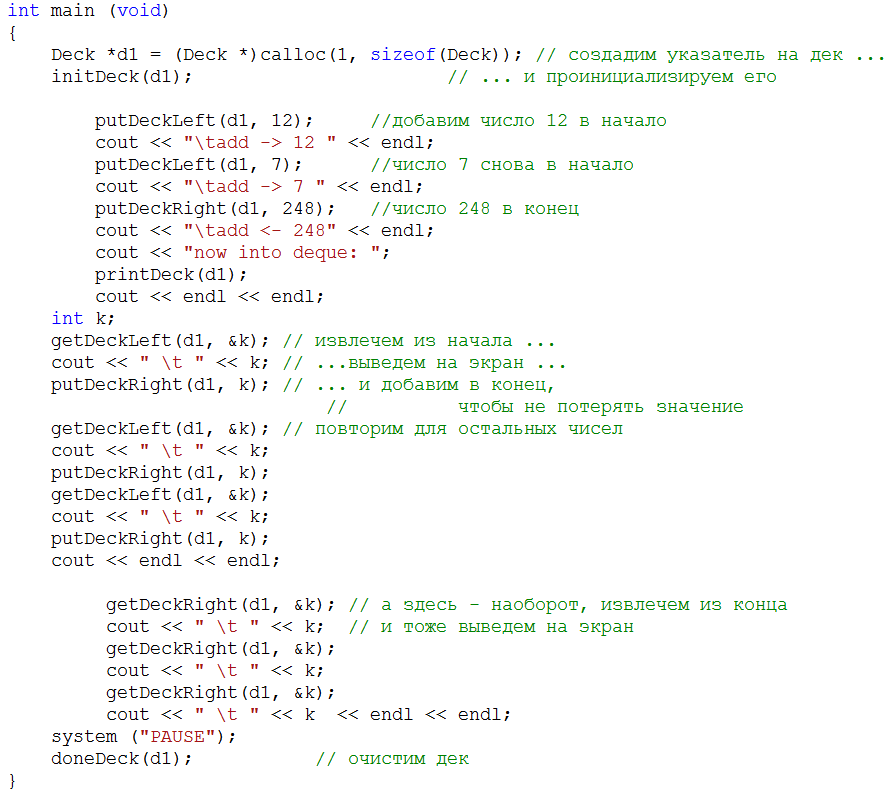








Протестируем работу описанных выше функций, посредствам функции ***main()*** следующего содержания:



Результат тестирования функций представлен на рисунке 7.

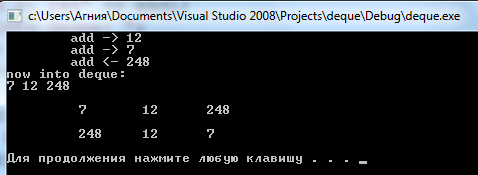


Рисунок 7 – Результат тестирования функций   
работы с деком (двунаправленной очередью)

ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНОЙ РАБОТЫ № 2\_8

**Создайте указанную динамическую структуру данных (ДСД) на основе списка для хранения данных, указанных в варианте задания. Напишите следующие функции:**

**Задание 1.** Функции для создания новой ДСД и ее удаления.

**Задание 2.** Функции добавления элементов (все, что необходимы для полноценной работы с указанной ДСД).

**Задание 3.** Функции удаления элементов (все, что необходимы для полноценной работы с указанной ДСД).

**Задание 4.** Функция печати всех элементов ДСД.

**Задание 5.** Функция очистки ДСД (т.е. удаления всех вложенных в нее элементов).

# ВАРИАНТЫ ЗАДАНИЙ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Вар.** | **Задание** | **Вар.** | **Задание** | **Вар.** | **Задание** |
| **1.1** | стек  char[15] | **2.1** | стек  double | **3.1** | очередь FIFO  char[20] |
| **1.2** | очередь FIFO  float | **2.2** | дек  char | **3.2** | дек  double |
| **1.3** | стек  char[20] | **2.3** | очередь FIFO  double | **3.3** | стек  char |
| **1.4** | дек  double | **2.4** | стек  char[15] | **3.4** | очередь FIFO  char[15] |
| **1.5** | стек  char | **2.5** | очередь FIFO  float | **3.5** | стек  float |
| **1.6** | очередь FIFO  char[15] | **2.6** | дек  char[20] | **3.6** | дек  char[20] |
| **1.7** | дек  float | **2.7** | стек  double | **3.7** | очередь FIFO  double |
| **1.8** | очередь FIFO  char[20] | **2.8** | очередь FIFO  char | **3.8** | дек  char |
| **1.9** | дек  double | **2.9** | стек  char[15] | **3.9** | стек  char[15] |
| **1.10** | стек  char | **2.10** | дек  float | **3.10** | очередь FIFO  float |
| **1.11** | дек  char[15] | **2.11** | очередь FIFO  float | **3.11** | дек  char[20] |
| **1.12** | очередь FIFO  float | **2.12** | стек  char[20] | **3.12** | стек  char[20 |
| **1.13** | стек  char[15] | **2.13** | дек  double | **3.13** | очередь FIFO  char[20] |