Двусторонние очереди (deque)

Двусторонняя очередь — это последовательный контейнер, который, наряду с вектором, поддерживает произвольный доступ к элементам и обеспечивает вставку и удаление из обоих концов очереди за постоянное время. Те же операции с элементами внутри очереди занимают время, пропорциональное количеству перемещаемых элементов. Распределение памяти выполняется автоматически.

Рассмотрим схему организации очереди (рис. 1). Для того чтобы обеспечить произвольный доступ к элементам за постоянное время, очередь разбита на блоки, доступ к каждому из которых осуществляется через указатель. На рисунке закрашенные области соответствуют занятым элементам очереди. Если при добавлении в начало или в конец блок оказывается заполненным, выделяется память под очередной блок (например, после заполнения блока 4 будет выделена память под блок 5, а после заполнения блока 2 — под блок 1). При заполнении крайнего из блоков происходит перераспределение памяти под массив указателей так, чтобы использовались только средние элементы. Это не занимает много времени. Таким образом, доступ к элементам очереди осуществляется за постоянное время, хотя оно и несколько больше, чем для вектора.

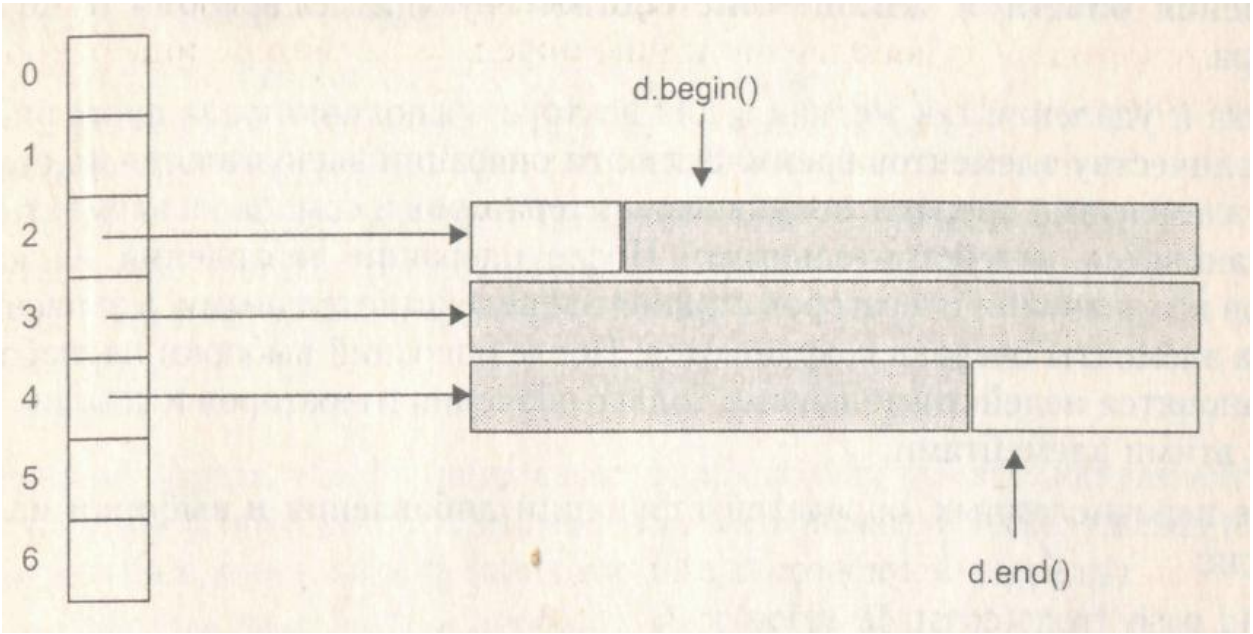


Рис. 1. Организация двусторонней очереди

Для создания двусторонней очереди можно воспользоваться следующими конструкторами (приведена упрощенная запись), аналогичными конструкторам вектора:

```
  explicit deque();                  // 1
  explicit deque(size_type n, const T& value = T());
// 2
  template <class InputIter>         // 3
```

```
deque(InputIter first, InputIter last);
deque(const deque <T>& x);      //4
```

Конструктор 1 является конструктором по умолчанию.

Конструктор 2 создает очередь длиной n и заполняет ее одинаковыми элементами — копиями value.

Конструктор 3 создает очередь путем копирования указанного с помощью итераторов диапазона элементов. Тип итераторов должен быть «для чтения».

Конструктор 4 является конструктором копирования.

Примеры конструкторов:

```
// Создается очередь из 10 равных единице элементов:
deque <int> d2 (10, 1);
// Создается очередь, равная очереди d2:
deque <int> d4 (d2);
// Создается очередь из двух элементов, равных первым двум
// элементам очереди d4 из предыдущего раздела:
deque <int> d3 (d4.begin(), d4.begin() + 2);
// Создается очередь из 10 объектов класса monstr
// (работает конструктор по умолчанию):
deque <monstr> ml (10);
// Создается очередь из 5 объектов класса monstr с заданным именем
// (работает конструктор с параметром char*);
deque <monstr> m2 (5, monstr("Вася в очереди"));
```

В шаблоне deque определены операция **присваивания,** функция **копирования, итераторы, операции сравнения,** операции и функции **доступа к элементам** и **изменения объектов,** аналогичные соответствующим операциям и функциям вектора.

Вставка и удаление так же, как и для вектора, выполняются за пропорциональное количеству элементов время. Если эти операции выполняются над внутренними элементами очереди, все значения итераторов и ссылок на элементы очереди становятся недействительными. После операций добавления в любой из концов все значения итераторов становятся недействительными, а значения ссылок на элементы очереди сохраняются. После операций выборки из любого конца становятся недействительными только значения итераторов и ссылок, связанных с этими элементами.

Кроме перечисленных, определены функции **добавления и выборки из начала** очереди:

```
void push_front(const T& value);
```

`void pop_front();`

При выборке элемент удаляется из очереди.

Для очереди не определены функции **capacity** и **reserve**, но есть функции **resize** и **size**.

К очередям можно применять алгоритмы стандартной библиотеки.

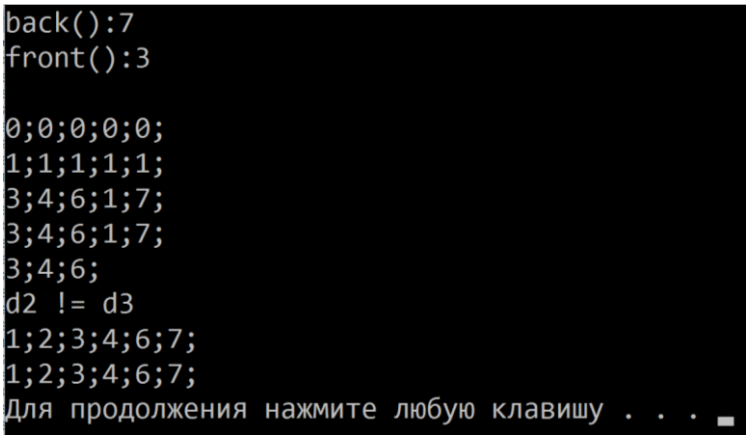
```
// Демонстрация методов push_back(), push_front(), front()
#include "stdafx.h"
#include <iostream>
#include <deque>
#include "windows.h"
using namespace std;

//Вывод значений деки на консоль.
void Print(const deque<int> &x)
{
	for (const int &y : x)
		cout << y << " ";
	cout << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
	int m[] = {3,4,6,1,7};
	deque<int> d1;
	deque<int> d2(5);
	deque<int> d3(5,1);
	deque<int> d4(m,m+5);
	deque<int> d5 = d4;
	cout << "back():" << d5.back() << endl;
	cout << "front():" << d5.front() << endl;

	Print(d1);//пусто
	Print(d2);//0;0;0;0;0;
	Print(d3);//1;1;1;1;1;
	Print(d4);//3;4;6;1;7;
	Print(d5);//3;4;6;1;7;
	deque<int> d6;
	d6.assign(d5.begin(),d5.begin() + 3);
	Print(d6);//3;4;6;
	if (d2 == d3) cout << "d2 == d3" << endl;
	else cout << "d2 != d3" << endl;
	d6.push_front(2);d6.push_front(1);d6.push_back(7);
	Print(d6);//1;3;4;6;7;
	d1.swap(d6);
	Print(d1);//1;3;4;6;7;
    system("PAUSE");
    return 0;
}
```

Результат работы программы:



Списки (list)

Список не предоставляет произвольного доступа к своим элементам, зато вставка и удаление выполняются за постоянное время. Класс list реализован в STL в виде двусвязного списка, каждый узел которого содержит ссылки на последующий и предыдущий элементы. Поэтому операции инкремента и декремента для итераторов списка выполняются за постоянное время, а передвижение на n узлов требует времени, пропорционального n.

После выполнения операций вставки и удаления значения всех итераторов и ссылок остаются действительными.

Список поддерживает **конструкторы, операцию присваивания, функцию копирования, операции сравнения** и **итераторы,** аналогичные векторам и очередям. **Доступ к элементам** для списков ограничивается следующими методами:

```
//чтение и запись в начало
reference   front();
//чтение из начала
const_reference front() const;
//чтение и запись в конец
reference   back();
// чтение из конца
const_reference back() const;
```

Для **занесения в начало и конец** списка определены методы, аналогичные соответствующим методам очереди:

```
//добавить в начало
void push_front(const T& value);
//удалить начало
void pop_front();
//добавить в конец
```

```
void push_back(const T& value);
//удалить конец
void pop_ back ();
```

Кроме того, действуют все остальные **методы для изменения объектов**, аналогичные векторам и очередям:

```
/*вставить в заданную position позицию копию value*/
iterator insert(iterator position, const T& value);
/*вставить в заданную  position позицию n копий value*/
void insert(iterator position, size_type n, const T& value);
template <class Inputlter>
void insert(iterator position, Inputlter first, Inputlter
last);
//удалить значение с позиции position
iterator erase(iterator position);
/*удалить значения начиная с позиции first до last
iterator erase(iterator first, iterator last); */
void swap();
void clear();
```

Для списка не определена функция **capacity**, поскольку память под элементы отводится по мере необходимости. Можно изменить размер списка, удалив или добавив элементы в конец списка (аналогично двусторонней очереди):

```
void resize(size_type sz, T c = T());
```

Кроме перечисленных, для списков определено несколько специфических методов. **Сцепка списков** (splice) служит для перемещения элементов из одного списка в другой без перераспределения памяти, только за счет изменения указателей:

```
void splice(iterator position, list<T>& x);
void splice(iterator position, list<T>& x, iterator i);
void splice(iterator position, list<T>& x, iterator first,
iterator last);
```

Оба списка должны содержать элементы одного типа. Первая форма функции вставляет в вызывающий список перед элементом, позиция которого указана первым параметром, все элементы списка, указанного вторым параметром, например:

```
list <int> L1, L2;
```

... // Формирование списков

```
L1.splice(L1.begin(), L2);
```

Второй список остается пустым. Нельзя вставить список в самого себя.

Вторая форма функции переносит элемент, позицию которого определяет третий параметр, из списка x в вызывающий список. Допускается переносить элемент в пределах одного списка.

```
void splice(iterator position, list<T>& x, iterator i);
```

Третья форма функции аналогичным образом переносит из списка в список несколько элементов. Их диапазон задается третьим и четвертым параметрами функции. Если для одного и того же списка первый параметр находится в диапазоне между третьим и четвертым, результат не определен.

```
void splice(iterator position, list<T>& x, iterator first,
iterator last);
```

Пример:

```
// Листинг 18.cpp: определяет точку входа для консольного приложения.
#include "stdafx.h"
#include <iostream>
#include <list>
#include <string>
#include "windows.h"
using namespace std;

//Вывод значений списка на консоль.
template<class T>
void Print(const T &x, string head)
{
	cout << head << endl;
	for (const auto &y : x)
		cout << y << " ";
	cout << endl;
}


int _tmain(int argc, _TCHAR* argv[])
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	//Создаём список.
	list<int> L1;
	//Создаём три итератора для списка.
	list<int>:: iterator i, j, k;

	//Проталкиваем элементы в список.
	for (int i = 0; i < 5; i++) L1.push_back(i + 1);
	for (int i = 12; i < 14; i++) L1.push_back(i);

	Print(L1,"список перед слиянием  :  ");//1 2 3 4 5 12 13

	i = L1.begin(); i++; //i = 1
```

```
    k = L1.end();               //k = 7
    j = --k;k++;j--;            //j = 5, k = 7
    L1.splice( i, L1, j, k);
    Print(L1,"список после слияния   :  ");        // 1 12 13 2 3 4 5

    L1.sort();
    Print(L1,"список после сортировки:  ");        //1 2 3 4 5 12 13
    cout << endl;
    system("PAUSE");
  return 0;
}
```

Результат работы программы:

```
список перед слиянием  :
1 2 3 4 5 12 13
список после слияния   :
1 12 13 2 3 4 5
список после сортировки:
1 2 3 4 5 12 13

Для продолжения нажмите любую клавишу . . .
```

Перемещенные элементы выделены полужирным шрифтом. Обратите внимание, что для итераторов списков не определены операции сложения и вычитания, то есть нельзя написать j = k - 1, поэтому пришлось воспользоваться допустимыми для итераторов списков операциями инкремента и декремента. В общем случае для поиска элемента в списке используется функция find.

Для **удаления элемента** по его значению применяется функция remove:

```
void remove(const T& value);
```

Если элементов со значением value в списке несколько, все они будут удалены.

Можно удалить из списка элементы, удовлетворяющие некоторому условию. Для этого используется функция remove_if:

```
template <class Predicate>
void remove_if(Predicate pred);
```

Параметром является класс-предикат, задающий условие, накладываемое на элемент списка.

Для **упорядочивания** элементов списка используется метод sort:

```
void sort();
template <class Compare>
void sort(Compare comp);
```

В первом случае список сортируется по возрастанию элементов (в соответствии с определением операции < для элементов), во втором — в соответствии с функ-

циональным объектом Compare. Функциональный объект имеет значение true, если два передаваемых ему значения должны при сортировке остаться в прежнем порядке, и false — в противном случае.

Порядок следования элементов, имеющих одинаковые значения, сохраняется. Время сортировки пропорционально $N*\log_2 N$, где N — количество элементов в списке.

Метод unique оставляет в списке только первый элемент из каждой серии идущих подряд одинаковых элементов. Первая форма метода имеет следующий формат:

```
void unique();
```

Вторая форма метода **unique** использует в качестве параметра бинарный предикат, что позволяет задать собственный критерий удаления элементов списка. Предикат имеет значение true, если критерий соблюден, и false — в противном случае. Аргументы предиката имеют тип элементов списка:

```
template <class BinaryPredicate>
void unique(BinaryPredicate binary_pred);
```

Для **слияния списков** служит метод merge:

```
void merge(list<T>& x);
template <class Compare>
void merge (list<T>& x, Compare comp);
```

Оба списка должны быть упорядочены (в первом случае в соответствии с определением операции < для элементов, во втором — в соответствии с функциональным объектом Compare). Результат — упорядоченный список. Если элементы в вызывающем списке и в списке-параметре совпадают, первыми будут располагаться элементы из вызывающего списка.

Метод reverse служит для **изменения порядка** следования элементов списка на обратный (время работы пропорционально количеству элементов):

```
void reverse();
```

Пример работы со списком:

```
// Листинг 19.cpp: определяет точку входа для консольного приложения.
#include "stdafx.h"
#include <iostream>
#include <string>
#include <list>
#include <fstream>
#include "windows.h"
using namespace std;

template<class T>
void Print(string head, const T &x )
{
	cout << head << endl;
	for (const auto &y : x)
		cout << y << " ";
	cout << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	list<int> L;
	list<int>:: iterator i;
	int x;
	ifstream in("inpnum.txt");
	while ( in >> x, !in.eof()) L.push_back(x);
	Print("список", L);//56 34 54 0 76 23 51 11 51 11 76 88
	L.push_front(1);
	i = L.begin(); L.insert(++i, 2);
	Print("После вставки 1 и 2 в начало", L);
	//1 2 56 34 54 0 76 23 51 11 51 11 76 88
	i = L.end(); L.insert(--i, 100);
	Print("После вставки 100 перед последним", L);
	//1 2 56 34 54 0 76 23 51 11 51 11 76 100 88
	i = L.begin(); x = *i; L.pop_front();
	cout << "Удаление первого " << x << endl;
	//2 56 34 54 0 76 23 51 11 51 11 76 100 88
	i = L.end(); x = *--i; L.pop_back();
	cout << "Удаление последнего " << x << endl;
	Print("Список после удаления", L);
	//2 56 34 54 0 76 23 51 11 51 11 76 100
	L.remove(76);
	Print("Список после удаления 76", L);
	//2 56 34 54 0 23 51 11 51 11 100
	L.sort();
	Print("Список после сортировки ", L);
	//0 2 11 11 23 34 51 51 54 56 100
	L.unique();
	Print("После удаления подряд идущих одинаковых", L);
	//0 2 11 23 34 51 54 56 100
	list<int> L1(L);
	L.reverse();
	Print("После перестановки в обратном порядке", L);//100 56 54 51 34 23 11 2 0
	system("PAUSE");
    return 0;
}
```

Результат работы программы:

```
список:
56 34 54 0 76 23 51 11 51 11 76 88
После вставки 1 и 2 в начало:
1 2 56 34 54 0 76 23 51 11 51 11 76 88
После вставки 100 перед последним:
1 2 56 34 54 0 76 23 51 11 51 11 76 100 88
Удаление первого 1
Удаление последнего 88
Список после удаления:
2 56 34 54 0 76 23 51 11 51 11 76 100
Список после удаления 76:
2 56 34 54 0 23 51 11 51 11 100
Список после сортировки :
0 2 11 11 23 34 51 51 54 56 100
После удаления подряд идущих одинаковых:
0 2 11 23 34 51 54 56 100
После перестановки в обратном порядке:
100 56 54 51 34 23 11 2 0
Для продолжения нажмите любую клавишу . . .
```

К спискам можно применять алгоритмы стандартной библиотеки.

Стеки (stack)

Как известно, в стеке допускаются только две операции, изменяющие его размер — добавление элемента в вершину стека и выборка из вершины. Стек можно реализовать на основе любого из рассмотренных контейнеров: вектора, двусторонней очереди или списка. Таким образом, стек является не новым типом контейнера, а вариантом имеющихся, поэтому он называется ***адаптером*** контейнера. Другие адаптеры (очереди и очереди приоритетов) будут рассмотрены в следующих разделах. В STL стек определен по умолчанию на базе двусторонней очереди:

```
template <class T, class Container = deque<T> >
class stack {
   protected:
        Container c;
   public:
        explicit stack(const Container& = Container());
        bool  empty() const   {return c.empty();}
        size_type size() const     {return c.size();}
        value_type& top()     {return c.back();}
        const value_type& top() const {return c.back();}
        void push(const value_type& x) {c.push_back(x);}
        void pop() {c.pop_back();}
};
```

Из приведенного описания (оно дано с сокращениями) видно, что метод занесения в стек push соответствует методу занесения в конец push_back, метод выборки из стека pop — методу

выборки с конца pop_back, кроме того, добавлен метод top для получения или изменения значения элемента на вершине стека. Конструктору класса stack передается в качестве параметра ссылка на базовый контейнер, который копируется в защищенное поле данных c.

При работе со стеком нельзя пользоваться итераторами и нельзя получить значение элемента из середины стека иначе, чем выбрав из него все элементы, лежащие выше. Для стека, как и для всех рассмотренных выше контейнеров, определены операции сравнения.

Пример использования стека (программа вводит из файла числа и выводит их на экран в обратном порядке):

```
#include "stdafx.h"
#include <fstream>
#include <iostream>
#include <stack>
#include <vector>
using namespace std;


int main()
{
	ifstream in("inpnum.txt");
	//Содержимое файла: 56 34 54 0 76 23 51 11 51 11 76
	stack <int, vector<int> > s;
	int x;
	while (in >> x, !in.eof()) s.push(x);

	s.top() = 1;
	while (!s.empty())
	{
		x = s.top();
		cout << x << " ";
		s.pop();
	}
	cout << endl;

	stack <int, vector<int> > s1;
	s.push(4);
	s1.push(4);
	//Операция == на стеках.
	if (s == s1) cout << "s == s1" << endl;
	else cout << "s != s1" << endl;
	//По умолчанию будет стек на deque<double>
	stack<double> s3;

	return 0;
}
```

Содержимое файла inpnum:

56 34 54 0 76 23 51 11 51 11 76

Результат работы программы:

```
1 11 51 11 51 23 76 0 54 34 56
s == s1
Для продолжения нажмите любую клавишу . . .
```

Очереди (queue)

Для очереди допускаются две операции, изменяющие ее размер — добавление элемента в

конец и выборка из начала. Очередь является адаптером, который можно реализовать на основе двусторонней очереди или списка (вектор не подходит, поскольку в нем нет операции выборки из начала). В STL очередь определена по умолчанию на базе двусторонней очереди:

```
template <class T, class Container = deque<T> >
class queue {
protected:
Container c: public:
explicit queue(const Container& = Container());
bool     empty() const   {return c.empty();}
size_type     size() const    {return c.size();}
value_type& front(){return c.front();}
const value_type& front() const      {return c.front();}
value_type&  back(){return c.back();}
const value_type& back() const {return c.back();}
void push(const value_type& x)        {c.push_back(x);}
void pop()    {c.pop_front();}
};
```

Методы **front** и **back** используются для получения значений элементов, находящихся соответственно в начале и в конце очереди (при этом элементы остаются в очереди).

Пример работы с очередью (программа вводит из файла числа в очередь и выполняет выборку из нее, пока очередь не опустеет):

```
#include "stdafx.h"
#include <fstream>
#include <iostream>
#include <list>
#include <queue>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	ifstream in("inpnum.txt");
	//Содержимое файла: 56 34 54 0 76 23 51 11 51 11 76
	queue <int, list<int> > q;
	int x;
	while (in >> x, !in.eof()) q.push(x);
	q.front() = 10;
	q.back() = 90;
	while (!q.empty()){
		cout << "q.front():  " << q.front() << "\t";
		cout << "q.back():  " << q.back() << endl;
		q.pop();
	}
	cout << "q.size():  " << q.size() << endl;

	queue <int, list<int> > q1;
	queue <int, list<int> > q2;
```

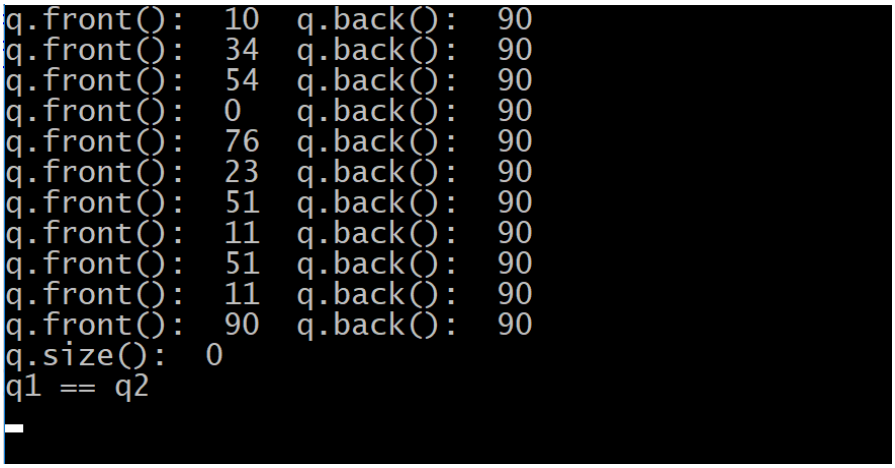
```
        q1.push(6);
        q2.push(6);
        //Операция == на очередях.
        if (q1 == q2) cout << "q1 == q2" << endl;
        else cout << "q1 != q2" << endl;

        cin.get();
        return 0;
}
```

Содержимое файла inpnum:

56 34 54 0 76 23 51 11 51 11 76

Результат работы программы:



К стекам и очередям можно применять алгоритмы стандартной библиотеки.

Очереди приоритетов (priority_queue)

В очереди приоритетов каждому элементу соответствует приоритет, определяющий порядок выборки из очереди. По умолчанию он определяется с помощью операции <; таким образом, из очереди каждый раз выбирается максимальный элемент.

Для реализации очереди приоритетов подходит контейнер, допускающий произвольный доступ к элементам, то есть, например, вектор или двусторонняя очередь. Тип контейнера передается вторым параметром шаблона (первый, как обычно, тип элементов). Третьим параметром указывается функция или функциональный объект, с помощью которого выполняется определение приоритета:

```
template <class T, class Container = vector<T>,
class Compare = less<typename Container::value_type> >
class priority_queue {
protected:
    Container c;
    Compare comp;
```

```
public:
   explicit priority_queue(const Compares x = Compare(),
   const Containers = Container());
   template <class InputIter>
   priority_queue(InputIter first,  InputIter last,
   const Compares x = Compare(), const Containers = Container());
   bool  empty () const{return c.empty();}
   size_type  size() const{return c.size();}
   const value_type&     top() const {return c.front();}
   void  push(const value_type& x);
   void  pop();
}
```

Для элементов с равными приоритетами очередь с приоритетами является простой очередью. Как и для стеков, основными методами являются push, pop и top.

Простой пример:

```
#include "stdafx.h"
#include <queue>
#include <iostream>
#include <functional>
#include <windows.h>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	priority_queue <int, deque<int>, greater<int> > P;
	int x;
	P.push(13);
	P.push(51);
	P.push(200);
	P.push(17);
	while (!P.empty()){
		x = P.top();
		cout << "element: " << x << endl;
		P.pop();
	}
	cout << endl;
	priority_queue <int> P1;
	priority_queue <int> P2;
	P1.push(6);
	P2.push(6);

	cin.get();
	return 0;

}
```

Результат работы программы:

```
element: 13
element: 17
element: 51
element: 200
```

В этом примере третьим параметром шаблона является шаблон, определенный в заголовочном файле <functional>. Он задает операцию сравнения на «меньше». Можно задать стандартные шаблоны greater<тип>, greater_equal<тип>, less_equal<тип>. Если требуется определить другой порядок выборки из очереди, вводится собственный функциональный объект. В приведенном ниже примере выборка выполняется по наименьшей сумме цифр в числе:

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <functional >
#include <queue>
using namespace std;

//По убыванию суммы разрядов чисел.
class CompareSum
public:
	bool operator()(int x, int y){
		int sx = 0, sy = 0;
		while (x){ sx += x % 10; x /= 10; }
		while (y){ sy += y % 10; y /= 10; }
		return sx > sy;
	}
};


int _tmain(int argc, _TCHAR* argv[])
{
	priority_queue <int, vector<int>, CompareSum > P;
	int x;
	P.push(13);
	P.push(51);
	P.push(200);
	P.push(17);
	while (!P.empty())
	{
		x = P.top();
		cout << "element: " << x << endl;
		P.pop();
	}
	cin.get();
	return 0;
}
```

Результат работы программы:

```
element: 200
element: 13
element: 51
element: 17
```