КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 19.3 — Стандартная библиотека C++. Контейнеры.

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2021

Оглавление

Библиотека контейнеров

Последовательные контейнеры

В последовательных контейнерах данные упорядочены. Однако, данные в таких контейнерах сами по себе не сортируются — для этого используются соответсвующие алгоритмы.

К последовательным контейнерам относятся:

- массив (array);
- дек (deque);
- однонаправленный список (forward list);
- список (list);
- вектор (vector),
- стек (stack)
- очередь (queue).

Ассоциативные контейнеры

Ассоциативные контейнеры — это такие коллекции, в которых позиция элемента зависит от его значения, то есть после занесения элементов в коллекцию порядок их следования будет задаваться их значениями. К ассоциативным контейнерам относятся:

-набор (set);
- словарь (map).

Дек (deque — двусторонняя очередь)

Предоставляет:
Шаблоны классов:
deque — двусторонняя очередь

Функции:
begin() — возвращает итератор на начало
end() — возвращает итератор на конец

```
#include <deque>

template < class T, class Alloc = allocator<T> > class deque;
```

deque (обычно произносится как «дек») — неправильное сокращение от двусторонней очереди.

Двусторонние очереди — это последовательные контейнеры с динамическими размерами, которые можно изменять (расширять или сжимать) с обоих концов.

Конкретные библиотеки могут реализовывать двухсторонние очереди по-разному, как правило, в виде некоторой формы динамического массива. Но в любом случае они позволяют получать доступ к отдельным элементам напрямую через итераторы произвольного доступа, при этом хранение обрабатывается автоматически путем расширения и сжатия контейнера по мере необходимости.

Следовательно, деки обеспечивают функциональность, аналогичную векторам, но с эффективной вставкой и удалением элементов не только в конце, но и в начале последовательности.

Однако, в отличие от векторов, двухсторонние очереди не гарантируют хранения всех своих элементов в смежных местах — доступ к элементам двухсторонней очереди путем смещения указателя на другой элемент вызывает неопределенное поведение.

И векторы, и двухсторонние очереди предоставляют очень похожий интерфейс и могут использоваться для аналогичных целей, но внутри они работают совершенно по-разному — векторы используют один массив, который необходимо время от времени перераспределять в случае роста контейнера, элементы же двухсторонней очереди могут быть разбросаны по различным фрагментам памяти, при этом контейнер хранит внутри необходимую информацию, чтобы обеспечить прямой доступ к любому из его элементов за постоянное время и с помощью единообразного последовательного интерфейса (через итераторы).

Следовательно, деки немного сложнее векторов, но это позволяет им при определенных обстоятельствах расти более эффективно, особенно в случае очень длинных последовательностей, когда перераспределение становится дорогостоящим.

Для операций, которые включают частую вставку или удаление элементов в позициях, отличных от начала или конца, деки работают хуже и имеют менее согласованные итераторы и ссылки, чем дву- и однонаправленные списки.

Свойства контейнера

Последовательность

Элементы в контейнерах последовательности упорядочены в строгой линейной последовательности.

Доступ к отдельным элементам осуществляется по их положению в этой последовательности.

Динамический массив

Обычно дек, реализованный как динамический массив, обеспечивает прямой доступ к любому элементу в последовательности и обеспечивает относительно быстрое добавление/удаление элементов в начале или в конце последовательности.

Используется аллокатор

Контейнер использует объект-аллокатор для динамической обработки своих потребностей в памяти.

Параметры шаблона

`T` — тип элементов.

Псевдоним типа члена **`deque::value_type`**.

`Alloc` — тип объекта аллокатора, используемого для определения модели распределения памяти. По умолчанию используется шаблон класса аллокатора, который определяет простейшую модель распределения памяти.

Псевдоним типа члена **`deque::allocator_type`**.

Типы членов

тип члена	определение	примечание
value_type	Первый параметр шаблона (T)	
allocator_type	Второй параметр шаблона (Alloc)	по умолчанию: allocator<value_type>
reference	value_type&	
const_reference	const value_type&	
pointer	allocator_traits<allocator_type>::pointer	для аллокатора по умолчанию: value_type*
const_pointer	allocator_traits<allocator_type>::const_pointer	для аллокатора по умолчанию: const value_type*
iterator	итератор произвольного доступа к value_type	convertible to const_iterator
const_iterator	итератор произвольного доступа к const value_type	
reverse_iterator	reverse_iterator<iterator>	
const_reverse_iterator	reverse_iterator<const_iterator>	
difference_type	целочисленный знаковый тип, эквивалентный типу iterator_traits<iterator>::difference_type	обычно ptrdiff_t
size_type	целочисленный беззнаковый тип, способный представлять любое неотрицательное значение difference_type	обычно size_t

Функции-члены

Все функции-члены являются общедоступными (public)

(**конструктор**) — создает список
(**деструктор**) — уничтожает список
operator= — присваивает деку содержимое

Итераторы

begin(), end() — Возвращает итератор на начало/конец дека
rbegin(), rend() — Возвращает реверсивный итератор на реверсивное начало/конец дека
cbegin(), cend() — Возвращает константный итератор[1] на начало/конец дека
crbegin(), crend() — Возвращает константный реверсивный итератор[2] на реверсивное начало/конец дека

Емкость

empty()	Проверить, пуст ли контейнер
size()	Возвращает количество элементов в деке
max_size()	Возвращает максимально допустимое количество элементов в деке
resize()	Изменить размер
shrink_to_fit	Сократить память до размеров содержимого

1 const_iterator
2 const_reverse_iterator

Доступ к элементам

`operator[]`	Получить элемент
`at`	Получить элемент
`back`	Доступ к последнему элементу
`front`	Доступ к первому элементу

Модификаторы

`assign()`	Назначает новое содержимое контейнеру
`push_back()`, `pop_back()`	Добавляет/удаляет элемент в конце дека
`push_front()`, `pop_front()`	Вставляет/удаляет элемент в начале дека
`insert()`, `erase()`	Вставляет/удаляет элемент в произвольном месте
`swap()`	Обменивает содержимое двух деков
`clear()`	Удаляет содержимое дека
`emplace()`	Создает элемент и вставляет его
`emplace_front()`, `emplace_back()`	Создает элемент и вставляет его в начале/конце списка

Аллокатор

`get_allocator()` — получить аллокатор

Перегруженные функции-не-члены

Реляционные операторы

(1)
```
template <class T, class Alloc>
bool operator== (const deque<T,Alloc>& lhs, const deque<T,Alloc>& rhs);
```

(2)
```
template <class T, class Alloc>
bool operator!= (const deque<T,Alloc>& lhs, const deque<T,Alloc>& rhs);
```

(3)
```
template <class T, class Alloc>
bool operator<  (const deque<T,Alloc>& lhs, const deque<T,Alloc>& rhs);
```

(4)
```
template <class T, class Alloc>
bool operator<= (const deque<T, Alloc>& lhs, const deque<T, Alloc>& rhs);
```

(5)
```
template <class T, class Alloc>
bool operator>  (const deque<T, Alloc>& lhs, const deque<T, Alloc>& rhs);
```

(6)
```
template <class T, class Alloc>
bool operator>= (const deque<T, Alloc>& lhs, const deque<T, Alloc>& rhs);
```

swap() Обменивает содержимое двух деков

Конструкторы дека

Создают объект контейнера двухсторонней очереди, инициализируя его содержимое в зависимости от используемой версии конструктора.

(1) Конструктор по умолчанию

```
explicit deque(const allocator_type& alloc = allocator_type());
```

Конструктор пустого контейнера. Создает пустой контейнер без элементов.

(2) Заполняющий конструктор

```
explicit deque(size_type n);
         deque(size_type n, const value_type& val,
               const allocator_type& alloc = allocator_type());
```

Создает контейнер из **n** элементов.

Каждый элемент является копией **val** (если предоставляется).

(3) Диапазонный конструктор

```
template <class InputIterator>
deque(InputIterator first, InputIterator last,
      const allocator_type& alloc = allocator_type());
```

Создает контейнер с таким количеством элементов, как у диапазона [`first`, `last`), причем каждый элемент создается из соответствующего элемента в этом диапазоне и в таком же порядке.

(4) Конструктор копирования

```
deque(const deque& x);
deque(const deque& x,
      const allocator_type& alloc);
```

Конструктор копирования и копирования с распределением памяти создает контейнер с копией каждого из элементов из **x** в таком же порядке.

(5) Конструктор перемещения

```
deque(deque&& x);
deque(deque&& x,
      const allocator_type& alloc);
```

Конструкторы перемещения и перемещения с помощью аллокатора создают контейнеры, в который попадают элементы из **x**.

Если указано значение **alloc** и оно отличается от аллокатора объекта **x**, элементы перемещаются. В противном случае никакие элементы не создаются (передается напрямую право владения). Элемент **x** остается в неопределенном, но допустимом состоянии.

(6) Конструктор из списка инициализаторов

```
deque(initializer_list<value_type> il,
      const allocator_type& alloc = allocator_type());
```

Создает контейнер с копией каждого из элементов **il** в таком же порядке.

Контейнер хранит внутреннюю копию аллокатора **alloc**, которая используется для выделения и освобождения памяти под его элементы, а также для их создания и уничтожения (как указано в его `allocator_traits`).

Параметры

alloc — объект аллокатора для объекта дека.

Контейнер хранит и использует внутреннюю копию этого аллокатора.

n — начальный размер контейнера (то есть количество элементов в контейнере при создании). Тип элемента **size_type** — это целочисленный тип без знака.

val — значение для заполнения контейнера. Каждый из **n** элементов в контейнере будет инициализирован копией этого значения.

Тип элемента **value_type** — это тип элементов в контейнере, определенный в **deque** как псевдоним его первого параметра шаблона (**T**).

first, **last** — итераторы ввода начальной и конечной позиции в диапазоне.

Используемый диапазон — **[first, last)**, который включает все элементы между первым и последним, включая элемент, на который указывает **first**, но не на элемент, на который указывает **last**. Аргумент шаблона функции **InputIterator()** должен быть типом итератора ввода, который указывает на элементы такого типа, из которого могут быть построены объекты **value_type**.

x — другой объект «двухсторонняя очередь» того же типа (с теми же аргументами шаблона класса **T** и **Alloc**), содержимое которого либо копируется, либо перемещается.

il — объект типа список инициализации

Эти объекты автоматически создаются из деклараторов списка инициализаторов.

Сложность

Константная для конструктора по умолчанию (1) и для конструкторов перемещения (5), если только **`alloc`** не отличается от аллокатора, который используется для **`x`**.

Во всех остальных случаях линейно зависит от полученного размера контейнера.

Замечание

Конструкторы перемещения (5) аннулируют все итераторы, указатели и ссылки, связанные с x, если элементы перемещаются.

Пример — создание двусторонней очереди

```
#include <iostream>
#include <deque>

int main () {

  unsigned int i;

  // constructors used in the same order as described above:
  std::deque<int> first;                                // пустой дек из int
  std::deque<int> second(4,100);                        // 4 int'a 100
  std::deque<int> third(second.begin(),second.end()); // диапазон из second
  std::deque<int> fourth(third);                        // копия third

  // итераторы для диапазонного конструктора создаются из массива "на лету"
  int myints[] = {16, 2, 77, 29};
  std::deque<int> fifth(myints, myints + sizeof(myints)/sizeof(int));

  std::cout << "The contents of fifth are:";
  for (std::deque<int>::iterator it = fifth.begin(); it!=fifth.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

Вывод

```
The contents of fifth are: 16 2 77 29
```

Деструктор

```
~deque( );
```

Линейная сложность от **`deque::size`**.

Оператор присваивания

Присваивает новое содержимое контейнеру, заменяя текущее и соответствующим образом изменяет его размер.

Копирования (1)	`deque& operator= (const deque& x);`
Перемещения (2)	`deque& operator= (deque&& x);`
Из списка инициализации (3)	`deque& operator= (initializer_list<value_type> il);`

Присваивание при копировании (1) копирует все элементы из **`x`** в контейнер, при этом **`x`** сохраняет свое содержимое.

Присваивание при перемещении (2) перемещает элементы из **`x`** в контейнер, при этом **`x`** остается в неопределенном, но валидном состоянии.

Присваивание из списка инициализации (3) копирует элементы из **`il`** в контейнер.

Контейнер сохраняет свой текущий аллокатор, за исключением случаев, когда характеристики аллокаторов не совпадают и должен использоваться аллокатор их **`x`**.

Возвращаемое значение — **`*this`**

Пример — оператор присваивания и деки

```
#include <iostream>
#include <deque>

int main () {

  std::deque<int> first(3);    // deque with 3 zero-initialized ints
  std::deque<int> second(5);   // deque with 5 zero-initialized ints

  second = first;
  first = std::deque<int>();

  std::cout << "Size of first: " << int (first.size()) << '\n';
  std::cout << "Size of second: " << int (second.size()) << '\n';
  return 0;
}
```

Вывод

```
Size of first: 0
Size of second: 3
```

Сложность линейная по размеру.

Все итераторы, ссылки и указатели, относящиеся к этому контейнеру до вызова, становятся недействительными.

При присвоении перемещения итераторы, указатели и ссылки, относящиеся к элементам из **x**, также становятся недействительными.

Доступ к элементу — operator[] и функция at()

```
#include <deque>

      reference operator[] (size_type n);
const_reference operator[] (size_type n) const;
```

Возвращает элемент в указанной позиции **n** в контейнере двухсторонней очереди.

Аналогичная функция-член **deque::at()** имеет то же поведение, что и эта операторная функция, за исключением того, что **deque::at()** проверяется насчет границ (bound-checking) и сигнализирует, если указанная позиция находится вне диапазона, вызывая исключение **out_of_range.**

Стек (LIFO stack)

```
#include <stack>

template <class T, class Container = deque<T> > class stack;
```

Стеки — это тип контейнера, специально разработанный для работы в контексте LIFO (последним пришел — первым вышел), когда элементы вставляются и извлекаются только с одного конца контейнера.

Стеки реализованы как адаптеры контейнеров — они используют инкапсулированный объект определенного контейнерного класса в качестве своего базового контейнера, предоставляя определенный набор функций-членов для доступа к его элементам.

Элементы помещаются/извлекаются из конца конкретного контейнера, которая называется вершиной стека.

Базовый контейнер может быть любым из стандартных шаблонов контейнерных классов или каким-либо другим специально разработанным контейнерным классом и должен поддерживать следующие операции:

- **empty()**
- **size()**
- **back()**
- **push_back()**
- **pop_back()**

Этим требованиям удовлетворяют стандартные классы контейнеров **`vector`**, **`deque`** и **`list`**.

По умолчанию, если для конкретного экземпляра класса стека класс контейнера не указан, используется стандартная двухсторонняя очередь.

Типы членов

Тип	Определение	Замечания
`value_type`	`Первый параметр шаблона (T)`	`Тип элементов`
`container_type`	`Второй параметр шаблона (Container)`	`Тип нижележащего контейнера`
`reference`	`container_type::reference`	`обычно value_type&`
`const_reference`	`container_type::const_reference`	`обычно const value_type&`
`size_type`	`беззнаковый целочисленный тип`	`обычно the same as size_t`

Функции-члены

`(constructor)` Создает стек

`empty()` Проверить, пуст ли контейнер

`size()` Возвращает количество элементов в стеке

`top()` Доступ к элементу на вершине

`push()` Вставить элемент

`emplace()` Создать и вставить элемент

`pop()` Удалить элемент с вершины стека

`swap()` Обмен содержимым

Перегруженные функции-не члены

Реляционные операторы

`swap (stack)` — обменять содержимое стеков

Конструкторы

(1) Конструктор инициализации

```
explicit stack(const container_type& ctnr);
```

Создает адаптер контейнера, внутренний контейнер которого инициализируется копией **ctnr**.

ctnr — объект контейнера.

container_type — это тип базового типа контейнера (определяется как псевдоним второго параметра шаблона класса, см. типы членов).

(2) Конструктор инициализации при перемещении

```
explicit stack(container_type&& ctnr = container_type());
```

Создает адаптер контейнера, внутренний контейнер которого получает значение **ctnr** путем перемещения.

(3) Конструктор с назначением аллокатора

```
template <class Alloc> explicit stack(const Alloc& alloc);
```

Создает контейнер-адаптер, внутреннему контейнеру которого в качестве аргумента передается параметр **alloc**.

alloc — объект аллокатора.

Alloc должен быть типом, для которого значение **uses_allocator::value** истинно. Т

(4) Конструктор с инициализацией и с назначением аллокатора

```
template <class Alloc> stack(const container_type& ctnr, const Alloc& alloc);
```

Создает адаптер контейнера, внутренний контейнер которого создается с помощью **cntr** и **alloc** в качестве аргументов.

(5) Инициализация перемещением и назначением аллокатора

```
template <class Alloc> stack(container_type&& ctnr, const Alloc& alloc);
```

Создает адаптер контейнера, внутренний контейнер которого создается с помощью **std::move(cntr)** и **alloc** в качестве аргументов.

(6) Конструктор копирования с назначением аллокатора

```
template <class Alloc> stack(const stack& x, const Alloc& alloc);
```

Создает адаптер контейнера, внутренний контейнер которого создается с внутренним контейнером типа **x** в качестве первого аргумента и аллокатором в качестве второго.

x — стек такого же типа, то есть с одинаковыми аргументами шаблона, **T** и **Container**).

(7) Конструктор перемещения с назначением аллокатора

```
template <class Alloc> stack(stack&& x, const Alloc& alloc);
```

Создает адаптер контейнера, внутренний контейнер которого создается путем перемещения внутреннего контейнера **x** в качестве первого аргумента и передачи **alloc** в качестве второго.

Пример — создание стека

```
#include <iostream>       // std::cout
#include <stack>          // std::stack
#include <vector>         // std::vector
#include <deque>          // std::deque

int main () {

  std::deque<int>  mydeque (3,100);      // deque with 3 elements
  std::vector<int> myvector (2,200);     // vector with 2 elements

  std::stack<int> first;                 // empty stack
  std::stack<int> second(mydeque);       // stack initialized to copy of deque

  std::stack< int, std::vector<int> > third;        // empty stack using vector
  std::stack< int, std::vector<int> > fourth(myvector);

  std::cout << "size of first: "  << first.size()  << '\n';
  std::cout << "size of second: " << second.size() << '\n';
  std::cout << "size of third: "  << third.size()  << '\n';
  std::cout << "size of fourth: " << fourth.size() << '\n';

  return 0;
}
```

Вставить/удалить элемент — push()/pop()

```
#include <stack>

void push (const value_type& val);
void push (value_type&& val);
void pop( );
```

push( ) вставляет новый элемент на вершине стека, над его текущим верхним элементом.

Эта функция-член фактически вызывает функцию-член **push_back( )** базового объекта-контейнера.

Содержимое этого нового элемента инициализируется копией **val**.

val — Значение, которым инициализируется вставленный элемент.

Тип элемента **value_type** — это тип элементов в контейнере (определенный как псевдоним первого параметра шаблона класса, T).

pop( ) удаляет верхний элемент стека, уменьшая его размер на единицу.

Удаленный элемент — это последний элемент, вставленный в стек, значение которого можно получить, вызвав фунrцию-член **stack::top( )**.

pop( ) вызывает деструктор удаляемого элемента.

Доступ к элементу на вершине стека — top ()

```
#include <stack>

reference top( );
const_reference top( ) const;
```

Возвращает ссылку на верхний элемент в стеке.

Поскольку стеки являются контейнерами LIFO («последним вошел — первым ушел», элемент на вершине — это последний элемент, вставленный в стек.

Эта функция-член фактически вызывает член **back( )** базового объекта-контейнера.

Пример push()/pop()/top()

```
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main () {

  std::stack<int> mystack;

  for (int i = 0; i < 5; ++i) mystack.push(i);

  std::cout << "Извлеченные элементы...";
  while (!mystack.empty()) {
     std::cout << ' ' << mystack.top();
     mystack.pop();
  }
  std::cout << '\n';

  return 0;
}
```

Вывод

```
Извлеченные элементы... 4 3 2 1 0
```

Пример top()

```
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main () {

  std::stack<int> mystack;

  mystack.push(10);
  mystack.push(20);

  mystack.top() -= 5;

  std::cout << "mystack.top() is now " << mystack.top() << '\n';

  return 0;
}
```

Вывод

```
mystack.top() is now 15
```

Проверить, пуст ли стек — empty()

```
#include <stack>

bool empty( ) const;
```

Возвращает **true** если стек пуст, т.е. его размер равен нулю, и **false** в противном случае
Эта функция-член фактически вызывает член базового объекта-контейнера **empty( )**.

Пример empty()

```
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main () {

  std::stack<int> mystack;
  int sum (0);

  for (int i = 1;i <= 10; i++) mystack.push(i);

  while (!mystack.empty()) {
     sum += mystack.top();
     mystack.pop();
  }

  std::cout << "total: " << sum << '\n';

  return 0;
}
```

Содержимое стека инициализируется последовательностью чисел (от 1 до 10). Затем элементы выталкиваются один за другим, пока стек не станет пустым, и вычисляеся их сумма.

Вывод

```
total: 55
```

Возвратить размер — size()

```
#include <stack>
size_type size() const;
```

Возвращает количество элементов в стеке (в базовом объекте-контейнере).

Тип элемента **size_type** — это целочисленный тип без знака.

Пример

```
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main () {

  std::stack<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';
  for (int i = 0; i < 5; i++) myints.push(i);
  std::cout << "1. size: " << myints.size() << '\n';
  myints.pop();
  std::cout << "2. size: " << myints.size() << '\n';
  return 0;
}
```

Вывод

```
size: 0
size: 5
size: 4
```

Создать и вставить элемент — emplace ()

```
#include <stack>

template <class... Args> void emplace(Args&&... args);
```

Добавляет новый элемент на вершину стека, над его текущим верхним элементом.

Этот новый элемент создается на месте с передачей аргументов в качестве аргументов для его конструктора.

Эта функция-член фактически вызывает функцию-член **`emplace_back()`** базового контейнера, перенаправляя аргументы в его конструктор.

`args` — аргументы, передаваемые в конструктор базового класса для создания нового элемента.

Пример emplace()

```
#include <iostream>       // std::cin, std::cout
#include <stack>          // std::stack
#include <string>         // std::string, std::getline(string)

int main () {

  std::stack<std::string> mystack; // стек из строк

  mystack.emplace("First sentence");
  mystack.emplace("Second sentence");

  std::cout << "mystack contains:\n";
  while (!mystack.empty()) {
    std::cout << mystack.top() << '\n';
    mystack.pop();
  }

  return 0;
}
```

Поменять местами содержимое — swap()

```
void swap (stack& x) noexcept(/*see below*/);
```

Заменяет содержимое адаптера контейнера (*** this**) на содержимое **x**.

Эта функция-член вызывает неквалифицированную функцию **swap( )**, которая не является членом, чтобы поменять содержимое базовых контейнеров.

x — другой стек такого же типа (т.е. созданный с теми же параметрами шаблона, **T** и **Container**). Размеры могут отличаться.

Пример

```
// stack::swap
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main () {

  std::stack<int> foo, bar;
  foo.push(10);
  foo.push(20);
  foo.push(30);
  bar.push(111);
  bar.push(222);

  foo.swap(bar);

  std::cout << "size of foo: " << foo.size() << '\n';
  std::cout << "size of bar: " << bar.size() << '\n';

  return 0;
}
```

Вывод

```
size of foo: 2
size of bar: 3
```

Реляционные операторы — функции-не-члены

Выполняют соответствующую операцию сравнения между lhs и rhs.

```
(1) template <class T, class Container>
    bool operator== (const stack<T,Container>& lhs, const stack<T,Container>& rhs);
(2) template <class T, class Container>
    bool operator!= (const stack<T,Container>& lhs, const stack<T,Container>& rhs);
(3) template <class T, class Container>
    bool operator<  (const stack<T,Container>& lhs, const stack<T,Container>& rhs);
(4) template <class T, class Container>
    bool operator<= (const stack<T,Container>& lhs, const stack<T,Container>& rhs);
(5) template <class T, class Container>
    bool operator>  (const stack<T,Container>& lhs, const stack<T,Container>& rhs);
(6) template <class T, class Container>
    bool operator>= (const stack<T,Container>& lhs, const stack<T,Container>& rhs);
```

Каждый из этих перегруженных операторов вызывает аналогичный оператор для базовых объектов контейнера.

Общедоступная функция-член — swap(stack)

Обмен содержимым стеков.

```
#include <stack>

template <class T, class Container>
void swap (stack<T,Container>& x, stack<T,Container>& y) noexcept(noexcept(x.swap(y)));
```

Очередь (FIFO queue)

Предоставляет:
Шаблоны классов:
`queue` — очередь FIFO
`priority_queue` — очередь FIFO с приоритетами

```
#include <queue>

template <class T, class Container = deque<T> > class queue;
```

Очереди — это тип адаптера контейнера, специально разработанный для работы в контексте FIFO (first-in first-out), где элементы вставляются в один конец контейнера и извлекаются из другого.

Очереди реализованы как адаптеры контейнеров, которые представляют собой классы, использующие инкапсулированный объект определенного класса контейнера в качестве своего базового контейнера.

Они предоставляют определенный набор функций-членов для доступа к его элементам.

Элементы вставляются в «хвост» конкретного контейнера и выталкиваются из его «головы».

Базовый контейнер может быть одним из стандартного шаблона класса контейнера или каким-либо другим специально разработанным классом контейнера. Этот базовый контейнер должен поддерживать как минимум следующие операции:

empty()
size()
front()
back()
push_back()
pop_front()

Этим требованиям удовлетворяют стандартные классы контейнера **deque** и **list**. По умолчанию, если для конкретного экземпляра класса очереди не указан класс контейнера, используется стандартная двухсторонняя очередь (**deque**) .

Типы членов

Тип члена	Определение	Примечание
value_type	первый параметр шаблона (T)	Тип элемента
container_type	Второй параметр шаблона (Container)	Тип базового контейнера
reference	container_type::reference	обычно value_type&
const_reference	container_type::const_reference	обычно const value_type&
size_type	беззнаковый целочисленный тип	обычно size_t

Функции-члены

Все функции-члены являются общедоступными (public)

(**`конструктор`**)	создает очередь
`empty()`	проверить, пуст ли контейнер
`size()`	возвращает количество элементов в очереди

Доступ к элементам

`front(), back()`	возвращает ссылку на первый/последний элемент в очереди
`push()`	вставляет элемент в очередь
`pop()`	удаляет элемент из очереди
`emplace()`	создает элемент и вставляет его в очередь
`swap()`	обменивает содержимое

Конструкторы очередей

Создают объекты контейнера очереди

Адаптер контейнера хранит внутри объект контейнера как данные, которые инициализируются этим конструктором:

(1) Конструктор инициализации

```
explicit queue(const container_type& ctnr);
```

Создает контейнер-адаптер, внутренний контейнер которого инициализируется копией **`ctnr`**.

`ctnr` — объект-контейнер.

`container_type` — это тип базового типа контейнера (определяется как псевдоним второго параметра шаблона класса, **`Container`**).

(2) Конструктор инициализации при перемещении

```
explicit queue(container_type&& ctnr = container_type());
```

Создает контейнер-адаптер, внутренний контейнер которого получает значение **`ctnr`** путем перемещения.

(3) Конструктор с назначением аллокатора

Создает контейнер-адаптер, внутреннему контейнеру которого в качестве аргумента передается параметр **alloc**.

```
template <class Alloc> explicit queue(const Alloc& alloc);
```

alloc — объект аллокатора

Alloc — должен быть типом, для которого значение **uses_allocator::value** истинно. Т.е. данный конструктор должен использоваться только в том случае, если базовый контейнер поддерживает аллокатор.

(4) Конструктор с инициализацией и назначением аллокатора

```
template <class Alloc> queue(const container_type& ctnr, const Alloc& alloc);
```

Создает контейнер-адаптер, внутренний контейнер которого создается с помощью **cntr** и **alloc** в качестве аргументов.

(5) Конструктор с инициализацией перемещением и назначением аллокатора

```
template <class Alloc> queue(container_type&& ctnr, const Alloc& alloc);
```

Создает контейнер-адаптер, внутренний контейнер которого создается с помощью **std::move(cntr)** и **alloc** в качестве аргументов.

(6) Конструктор копирования с назначением аллокатора

```
template <class Alloc> queue(const queue& x, const Alloc& alloc);
```

Создает адаптер контейнера, внутренний контейнер которого создается копированием из контейнера **x** в качестве первого аргумента и аллокатором в качестве второго.

x — очередь того же типа (то есть с одинаковыми аргументами шаблона, **T** и **Container**).

(7) **Конструктор перемещения с назначением аллокатора**

```
template <class Alloc> queue(queue&& x, const Alloc& alloc);
```

Конструкторы перемещения (2) и (7) могут сделать недействительными все итераторы, указатели и ссылки, связанные с их перемещенным аргументом.

Пример

```
// constructing queues
#include <iostream>       // std::cout
#include <deque>          // std::deque
#include <list>           // std::list
#include <queue>          // std::queue

int main () {

  std::deque<int> mydeck(3,100);        // дек с 3-мя элемениами
  std::list<int>  mylist(2,200);        // список с 2-мя элемениами

  std::queue<int> first;                // пустая очередь
  std::queue<int> second(mydeck);       // очередь, инициализ-ная копией дека

  std::queue<int, std::list<int>> third; // пустая очередь со списком в
                                          // качестве базового контейнера

  std::queue<int, std::list<int>> fourth(mylist);

  std::cout << "size of first: "  << first.size()  << '\n';
  std::cout << "size of second: " << second.size() << '\n';
  std::cout << "size of third: "  << third.size()  << '\n';
  std::cout << "size of fourth: " << fourth.size() << '\n';

  return 0;
}
```

Вывод

```
size of first: 0
size of second: 3
size of third: 0
size of fourth: 2
```

Вставить/удалить элемент – push()/pop()

#include <queue>

```
void push(const value_type& val);
void push(value_type&& val);
void pop();
```

push() – вставляет новый элемент в конец очереди после ее текущего последнего элемента. Содержимое этого нового элемента инициализируется значением **val**.

Эта функция-член фактически вызывает функцию-член **push_back()** базового объекта-контейнера.

pop() – удаляет элемент в голове очереди, фактически уменьшая ее размер на единицу.

Удаленный элемент является «самым старым» элементом в очереди, значение которого можно получить, вызвав функцию-член **queue::front()**.

pop() вызывает деструктор удаленного элемента. Эта функция-член фактически вызывает функцию-член **pop_front()** базового объекта-контейнера.

Тип элемента **value_type** – это тип элементов в контейнере (определенный как псевдоним первого параметра шаблона класса, **T**).

Пример push()/pop()

```
#include <iostream>       // std::cin, std::cout
#include <queue>          // std::queue

int main () {

  std::queue<int> myqueue;
  int myint;

  std::cout << "Please enter some integers (enter 0 to end):\n";

  do {
    std::cin >> myint;
    myqueue.push(myint);
  } while (myint);

  std::cout << "myqueue contains: ";
  while (!myqueue.empty()) {
    std::cout << ' ' << myqueue.front();
    myqueue.pop();
  }
  std::cout << '\n';

  return 0;
}
```

Доступ к следующему/последнему элементу — front()/back()

```
#include <queue>

      reference& front();
const_reference& front() const;
      reference& back();
const_reference& back() const;
```

front() возвращает ссылку на следующий элемент в очереди.

Следующий элемент — это «самый старый» элемент в очереди и тот же элемент, который извлекается из очереди при вызове **queue::pop()**.

Эта функция-член фактически вызывает функцию-член **front()** объекта базового контейнера.

back() — доступ к последнему элементу

Возвращает ссылку на последний элемент в очереди. Это «самый новый» элемент в очереди (т.е. последний элемент, помещенный в очередь).

Эта функция-член фактически вызывает член-обратно базового объекта-контейнера.

Возвращаемое значение

front() — ссылка на следующий элемент в очереди.

back() — ссылка на последний элемент в очереди.

Пример

```
// queue::front
#include <iostream>       // std::cout
#include <queue>          // std::queue

int main () {

  std::queue<int> myqueue;

  myqueue.push(77);
  myqueue.push(16);

  myqueue.front() -= myqueue.back();    // 77 – 16 = 61

  std::cout << "myqueue.front() is now " << myqueue.front() << '\n';

  myqueue.push(12);
  myqueue.push(75);   // this is now the back

  myqueue.back() -= myqueue.front(); // 75 - 61 = 14

  std::cout << "myqueue.back() is now " << myqueue.back() << '\n';

  return 0;
}
```

Output:

```
myqueue.front() is now 61
myqueue.back() is now 14
```

Реляционные операторы

swap (queue)
Exchange contents of queues (public member function)

Non-member class specializations
uses_allocator<queue>
Uses allocator for queue (class template)

Очереди с приоритетами

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

Очереди с приоритетами — это тип адаптеров контейнера, специально разработанный таким образом, что его первый элемент всегда является наибольшим из элементов, которые он содержит, в соответствии с некоторым строгим критерием слабого порядка.

Очереди приоритета реализованы как адаптеры контейнеров, которые используют инкапсулированные объекты определенного класса контейнера в качестве своего базового контейнера, предоставляя определенный набор функций-членов для доступа к его элементам.

Элементы извлекаются из «задней части» (back) конкретного контейнера, которая называется вершиной очереди с приоритетом.

Базовый контейнер может быть любым из стандартных шаблонов контейнерных классов или каким-либо другим специально разработанным контейнерным классом. Контейнер должен быть доступен через итераторы произвольного доступа и поддерживать следующие операции:

`empty()`, **`size()`**, **`front()`**, **`push_back()`**, **`pop_back()`**

Этим требованиям удовлетворяют стандартные классы контейнера **`vector`** и **`deque`**. По умолчанию, если для конкретного экземпляра класса **`priority_queue`** не указан класс контейнера, используется **`std::vector`**.

Параметры шаблона

T — тип элементов.

Псевдоним типа члена **priority_queue::value_type**.

Container — тип внутреннего базового объекта-контейнера, в котором хранятся элементы.

Его **value_type** должен быть **T**. Псевдоним типа члена **priority_queue::container_type**.

Compare — двоичный предикат, который принимает в качестве аргументов два элемента (оба типа **T**) и возвращает логическое значение.

Выражение **comp(a, b)**, где **comp** — объект этого типа, а **a** и **b** — элементы в контейнере, должно возвращать истину, если считается, что **a** идет перед **b** в строгом слабом порядке, определяемом функцией.

priority_queue использует эту функцию для поддержания элементов, отсортированных таким образом, чтобы сохранить свойства кучи (то есть, что выталкиваемый элемент является последним в соответствии с этим строгим слабым порядком).

Это может быть указатель на функцию или функциональный объект, значение по умолчанию - **less<T>**, которое возвращает то же самое, что и применение оператора «меньше» (**a < b**).

Типы членов

Тип члена	Определение	Примечание
value_type	первый параметр шаблона (T)	Тип элемента
container_type	Второй параметр шаблона (Container)	Тип базового контейнера
reference	container_type::reference	обычно value_type&
const_reference	container_type::const_reference	обычно const value_type&
size_type	беззнаковый целочисленный тип	обычно size_t