Контейнеры

Общее описание

Поскольку рассматриваемые контейнеры имеют много одинаковых функций-членов, в данном разделе эти контейнеры описываются совместно. В каждом пункте все функции-члены приводятся в алфавитном порядке их имен. Если некоторые функции-члены имеются не у всех рассматриваемых типов контейнеров, то это явно указывается.

Класс *string* имеет гораздо больше функций-членов, чем остальные рассмотренные контейнеры, однако в данном разделе приводятся только те из них, которые имеются также и у других последовательных контейнеров.

Если требуется одновременно упомянуть и множество, и мультимножество, то используется слово «(мульти)множество»; если требуется одновременно упомянуть и отображение, и мультиотображение, то используется слово «(мульти)отображение».

В описаниях шаблонов контейнеров и в списках параметров конструкторов и функций-членов не указывается дополнительный тип *Alloc* (который обычно устанавливается по умолчанию).

Все контейнеры определены в пространстве имен *std*.

Последовательные контейнеры

Имя	Описание	Итераторы	Заголовочный файл
vector<T>	*Вектор* с элементами типа *T*	Произвольного доступа	<*vector*>
deque<T>	*Дек* с элементами типа *T*	Произвольного доступа	<*deque*>
list<T>	*Список* с элементами типа *T*	Двунаправленные	<*list*>
string	*Строка* с элементами типа *char*	Произвольного доступа	<*string*>

Ассоциативные контейнеры

Если в шаблоне не указана операция сравнения, то она считается равной *less<Key>*.

Имя	Описание	Итераторы	Заголовочный файл
set<Key[, *Compare*]>	*Множество* с элементами типа *Key* и операцией сравнения *Compare*	Двунаправленные	<*set*>
multiset <Key[, *Compare*]>	*Мультимножество* с элементами типа *Key* и операцией сравнения *Compare*	Двунаправленные	<*set*>
map <Key, T[, *Compare*]>	*Отображение* с ключами типа *Key*, значениями типа *T* и операцией сравнения для ключей *Compare*	Двунаправленные	<*map*>
multimap <Key, T[, *Compare*]>	*Мультиотображение* с ключами типа *Key*, значениями типа *T* и операцией сравнения для ключей *Compare*	Двунаправленные	<*map*>

В последующих описаниях функций-членов некоторые переменные всегда связываются с данными определенного типа (связанного с рассматриваемым контейнером):

- *n* имеет тип *size_type*,
- *k* имеет тип *const key_type*&,
- *x* имеет тип *const value_type*&,
- *pos*, *hintpos*, *first* и *last* (а также *pos_lst*, *first_lst*, *last_lst*) имеют тип итератора соответствующего контейнера (*iterator*).

Переменная *cont* обозначает параметр, являющийся контейнером того же типа, что и текущий контейнер. Переменные *InIterFirst* и *InIterLast* обозначают итераторы чтения (которые могут быть связаны с контейнерами других типов; при этом тип элементов контейнеров должен совпадать с типом элементов данного контейнера).

Функции-члены *begin*, *end*, *rbegin*, *rend*, *front*, *back*, *at*, *equal_range*, *find*, *lower_bound*, *upper_bound*, а также *operator*[] для *последовательных* контейнеров, реализованы в двух вариантах: неконстантном и константном (например, *iterator begin*(…) и *const_iterator begin*(…) *const*); в дальнейшем это особо не оговаривается, и константный вариант не приводится.

Типы, определенные в контейнерах

Для доступа к указанным типам используется нотация ::, например, *vector<int>*::*iterator*.

`iterator, const_iterator, reverse_iterator, const_reverse_iterator`

Типы *итераторов*, связанные с данным контейнером.

`reference, const_reference`

Типы *ссылок* на элементы данного контейнера.

`pointer, const_pointer`

Типы *указателей* на элементы данного контейнера.

`size_type`

Тип, используемый при указании *размера* контейнера.

`value_type`

Тип *элементов* контейнера (*T* для последовательных контейнеров, *Key* для (мульти)множеств, *pair<const Key, T>* для (мульти)отображений).

ассоциативные контейнеры

`key_type`

Тип *Key* (*элементы-ключи* для (мульти)множеств, *ключи* для (мульти)отображений).

(мульти)отображения

`mapped_type`

Тип *значений T* для (мульти)отображений.

ассоциативные контейнеры

`key_compare`

Тип функционального объекта, используемого при *сравнении ключей* типа *key_type*.

ассоциативные контейнеры

`value_compare`

Тип функционального объекта, используемого при *сравнении элементов* типа *value_type* по ключу типа *key_type*.

Параметры конструкторов

`конструктор без параметров`

Создает пустой контейнер. Для ассоциативных контейнеров можно дополнительно указать операцию сравнения *comp* (по умолчанию используется операция сравнения *Compare*(), взятая из шаблона).

`(InIterFirst, InIterLast)`

Создает контейнер, содержащий элементы (типа *value_type*) из диапазона [*InIterFirst*, *InIterLast*). Для ассоциативных контейнеров можно дополнительно указать операцию сравнения *comp* (по умолчанию используется операция сравнения *Compare*(), взятая из шаблона). Для ассоциативных контейне-

ров вставляемые элементы не обязаны быть упорядоченными, однако если они упорядочены, то время их вставки ускоряется.

```
(cont)
```

Создает копию контейнера *cont* (тип контейнера *cont* должен совпадать с типом создаваемого контейнера).

последовательные контейнеры

```
(n, x = T())
```

Создает последовательный контейнер, содержащий *n* копий значения *x*. Для строк *string* обязательными являются оба параметра.

Функции-члены всех контейнеров

```
operator=(cont)
```

Удаляет все элементы контейнера и копирует в него все элементы контейнера *cont* (тип контейнера *cont* должен совпадать с типом преобразуемого контейнера). Возвращает полученный контейнер.

```
iterator begin()
```

Возвращает итератор, указывающий на первый элемент контейнера.

```
void clear()
```

Удаляет все элементы контейнера.

```
bool empty() const
```

Возвращает *true*, если контейнер пуст, и *false* в противном случае.

```
iterator end()
```

Возвращает итератор, указывающий на позицию за последним элементом контейнера.

```
size_type max_size() const
```

Возвращает максимально возможный размер контейнера.

```
reverse_iterator rbegin()
```

Возвращает обратный итератор, связанный с последним элементом контейнера.

```
reverse_iterator rend()
```

Возвращает обратный итератор, связанный с позицией перед первым элементом контейнера.

```
size_type size() const
```

Возвращает текущий размер контейнера.

```
void swap(cont)
```

Меняет местами содержимое данного контейнера и контейнера *cont* того же типа.

Функции-члены последовательных контейнеров

```
void assign(n, x)
void assign(InIterFirst, InIterLast)
```

Удаляет все элементы контейнера и копирует в него новые данные (*n* копий значения *x* или элементы из диапазона [*InIterFirst*, *InIterLast*)).

vector, deque, string

```
reference operator[](n)
```

Возвращает элемент с индексом *n* (0 <= *n* < *size*()). Выход за границы не контролируется. Для типа *string* в случае *n* == *size*() возвращается символ с кодом 0.

vector, deque, string

```
reference at(n)
```

Возвращает элемент с индексом *n* (0 <= *n* < *size*()). Выход за границы приводит к возбуждению исключения *out_of_range*.

vector, deque, list

```
reference back()
```

Возвращает последний элемент контейнера. Для пустого контейнера поведение не определено.

vector, string

```
size_type capacity()
```

Возвращает текущую емкость контейнера.

```
iterator erase(pos)
iterator erase(first, last)
```

Удаляет элемент на позиции *pos* или все элементы в диапазоне [*first*, *last*) и возвращает итератор, указывающий на следующий элемент контейнера (или итератор *end*(), если были удалены конечные элементы контейнера).

vector, deque, list

```
reference front()
```

Возвращает первый элемент контейнера. Для пустого контейнера поведение не определено.

```
iterator insert(pos, x)
void insert(pos, n, x)
void insert(pos, InIterFirst, InIterLast)
```

Вставляет в контейнер новые данные, начиная с позиции *pos* (соответственно одно или *n* значений *x* или элементы из диапазона [*InIterFirst*, *InIterLast*)). Первый вариант функции-члена возвращает итератор, указывающий на вставленный элемент.

vector, deque, list

```
void pop_back()
```

Удаляет последний элемент. Для пустого контейнера поведение не определено.

deque, list

```
void pop_front()
```

Удаляет первый элемент. Для пустого контейнера поведение не определено.

```
void push_back(x)
```

Добавляет *x* в конец контейнера.

deque, list

```
void push_front(x)
```

Добавляет *x* в начало контейнера.

vector, string

```
void reserve(n)
```

Резервирует емкость размером не менее *n*.

```
void resize(n, x = T())
```

Изменяет размер контейнера, делая его равным *n*. Если *n* > *size*(), то в конец контейнера добавляется требуемое число копий *x*. Если *n* < *size*(), то удаляется требуемое количество конечных элементов контейнера.

Дополнительные функции-члены класса list

Все дополнительные функции-члены класса *list*, кроме *splice*, представляют собой специальные реализации соответствующих алгоритмов, которые необходимо использовать вместо стандартных алгоритмов при обработке списков.

```
void merge(lst[, comp])
```

Выполняет операцию слияния текущего списка и списка *lst* того же типа (оба списка должны быть предварительно отсортированы). При слиянии элементы сравниваются с помощью операции < или предиката *comp*, если он явно указан (и эта же операция или предикат должны быть ранее использованы для сортировки списков). Слияние является *устойчивым*, т. е. относительный порядок следования элементов исходных списков не нарушается; если «одинаковые» элементы присутствуют как в текущем списке, так и в списке *lst*, то элемент из *lst* помещается после элемента, уже присутствующего в текущем списке. В результате слияния список *lst* становится пустым.

```
void remove(x)
void remove_if(pred)
```

Удаляет из списка соответственно все вхождения элемента *x* или все элементы, для которых предикат *pred* возвращает значение *true*.

```
void reverse()
```

Изменяет порядок элементов списка на обратный.

```
void sort([comp])
```

Выполняет сортировку списка, используя операцию < или предикат *comp*, если он явно указан. Сортировка является *устойчивой*, т. е. относительный порядок элементов с одинаковыми ключами сортировки не изменяется.

```
void splice(pos, lst)
void splice(pos, lst, pos_lst)
void splice(pos, lst, first_lst, last_lst)
```

Перемещает элементы из списка *lst* в текущий список (элементы размещаются, начиная с позиции *pos*). Перемещаются соответственно все элементы списка *lst*, элемент списка *lst*, расположенный на позиции *pos_lst*, и элементы списка *lst* из диапазона [*first_lst*, *last_lst*) (если текущий список совпадает со списком *lst*, то итератор *pos* не должен входить в диапазон [*first_lst*, *last_lst*)).

```
void unique([pred])
```

Удаляет соседние «одинаковые» элементы списка, оставляя первый из набора «одинаковых» элементов. Для сравнения элементов используется операция == или предикат *pred*, если он явно указан.

Функции-члены ассоциативных контейнеров

map

```
T& operator[](k)
```

Возвращает ссылку на значение, связанное с ключом *k*. Если ключ *k* отсутствует в контейнере, то он добавляется вместе со значением по умолчанию *T*(), и операция [] возвращает ссылку на это значение. Фактически данная операция возвращает следующее выражение: *insert*(*make_pair*(*k*, *T*())).*first->second*.

```
size_type count(k) const
```

Возвращает число ключей со значением *k*. Для множества и отображения это либо 0, либо 1; для мультимножества и мультиотображения возвращаемое значение может быть больше 1.

```
pair<iterator, iterator> equal_range(k)
```

Возвращает результат вызова функций *lower_bound* и *upper_bound* в виде пары итераторов: *make_pair*(*lower_bound*(*k*), *upper_bound*(*k*)).

```
void erase(pos)
void erase(first, last)
size_type erase(k)
```

Удаляет соответственно элемент на позиции *pos*, все элементы в диапазоне [*first*, *last*) или элемент (элементы) с ключом *k*. В последнем случае возвращает количество удаленных элементов (для множества и отображения это либо 0, либо 1).

```
iterator find(k)
```

Ищет ключ *k* и возвращает итератор, указывающий на соответствующий элемент контейнера, или *end*(), если ключ не найден. В случае мультимножества и мультиотображения итератор может указывать на любой из элементов с ключом *k*.

```
pair<iterator, bool> insert(x)
iterator insert(hintpos, x)
void insert(InIterFirst, InIterLast)
```

Вставляет в контейнер новые данные. Если данные с указанным ключом уже имеются в контейнере, то в случае множества и отображения попытка вставки игнорируется. В первых двух вариантах функция возвращает позицию вставленного элемента, а также (в первом варианте) логическое значение, определяющее, была ли произведена вставка (значение *false* возможно только в случае множества и отображения; при этом в качестве первого элемента пары возвращается позиция уже имеющегося в множестве/отображении элемента со значением *x*). Параметр *hintpos* является «подсказкой» для позиции вставки; если элемент *x* вставляется сразу за позицией *hintpos*, то время вставки уменьшается. Третий вариант обеспечивает вставку всех элементов из диапазона [*InIterFirst*, *InIterLast*); эти элементы не обязаны быть упорядоченными по ключу, однако если они упорядочены, то время их вставки уменьшается.

```
key_compare key_comp() const
```

Возвращает функциональный объект, обеспечивающий сравнение ключей.

```
iterator lower_bound(k)
```

Если в множестве или отображении присутствует элемент с ключом *k*, то возвращается его позиция (в случае мультимножества или мультиотображения возвращается позиция *первого* элемента с ключом *k*); если такого элемента нет, то возвращается позиция, куда будет вставлен такой элемент.

```
iterator upper_bound(k)
```

Если в множестве или отображении присутствует элемент с ключом *k*, то возвращается позиция элемента, следующего за ним (в случае мультимножества и мультиотображения возвращается позиция элемента, следующего за *последним* элементом с ключом *k*); если элемента с ключом *k* нет, то возвращается позиция, куда будет вставлен такой элемент.

```
value_compare value_comp() const
```

Возвращает функциональный объект, обеспечивающий сравнение элементов контейнера по их ключам. В случае (мульти)множества совпадает с объектом key_compare, в случае (мульти)отображения выполняет сравнение пар *pair<const Key, T>* по их первому компоненту *Key*.

Вставка и удаление в последовательных контейнерах: дополнительные сведения

Функция-член *insert* реализована во всех последовательных контейнерах в трех вариантах. Первый параметр во всех вариантах — итератор *pos*, определяющий позицию вставки. Новые данные вставляются, начиная с указанной позиции *pos*; все прежние элементы, начиная с позиции *pos* и далее, смещаются вправо (по направлению к концу контейнера). Варианты различаются параметрами, определяющими, что именно вставляется: это либо (1) отдельное значение *x* типа *T* (вставляется единственное значение *x*), либо (2) значения *n* и *x* (вставляются *n* значений *x*), либо (3) два итератора чтения *InIterFirst* и *InIterLast* (вставляются все элементы из диапазона [*InIterFirst*, *InIterLast*)). Во всех контейнерах только вариант (1) функции *insert* возвращает значение, этим значением является итератор, указывающий на вставленный элемент.

Итератор *pos* и возвращаемый итератор — всегда прямые (обычные) итераторы; обратные итераторы в качестве *pos* использовать нельзя.

Имеются дополнительные функции-члены, связанные со вставкой: это *push_back*(*x*) — вставка одного элемента в конец контейнера (реализована для всех последовательных контейнеров) и *push_front*(*x*) — вставка одного элемента в начало контейнера (реализована для дека и списка). Эти функции не возвращают значения.

Функция-член *erase* реализована во всех последовательных контейнерах в двух вариантах: (1) с параметром-итератором *pos*, определяющим позицию удаляемого элемента, и с двумя параметрами-итераторами *first* и *last*, определяющими диапазон [*first*, *last*) удаляемых элементов. В обоих вариантах возвращается итератор на элемент за удаленным элементом (или удаленным диапазоном).

Имеются дополнительные функции-члены, связанные с удалением: это *pop_back*() — удаление последнего элемента из контейнера (реализована для всех последовательных контейнеров), *pop_front*() — удаление первого элемента из контейнера (реализована для дека и списка), *clear*() — удаление всех элементов из контейнера (реализована для всех контейнеров). Эти функции не возвращают значения.

Альтернативой функциям *insert* и *erase* для списков являются три варианта **функции-члена *splice***, позволяющие перемещать отдельные элементы или их диапазоны между различными списками или между различными позициями одного списка. Все варианты функции *splice* начинаются с параметров *pos* (итератора, определяющего место вставки) и *lst* (списка-источника вставляемых данных). Если других параметров нет, то список-источник *lst* целиком вставляется в позицию *pos* списка-приемника, если имеется один дополнительный параметр-

итератор *pos_lst*, то из списка-источника в список-приемник перемещается единственный элемент, связанный с итератором *pos_lst*, если имеются два дополнительных параметра *first_lst* и *last_lst*, то перемещается диапазон элементов [*first_lst*, *last_lst*). Все перемещаемые элементы удаляются из списка-источника.

При выполнении вставки и удаления важно знать, когда в результате выполнения этих действий итераторы и ссылки становятся недействительными (одновременно со ссылками становятся недействительными и указатели).

Вектор

Вставка:
- если в результате вставки выполняется перераспределение памяти (увеличивается емкость), то становятся недействительными все итераторы и ссылки;
- если перераспределения памяти не производится, то итераторы и ссылки до позиции вставки остаются корректными, а прочие — недействительными.

Удаление:
- все итераторы и ссылки до позиции удаления остаются корректными, а прочие — недействительными.

Дек

Вставка:
- все итераторы делаются недействительными;
- ссылки делаются недействительными при вставке в середину дека и остаются корректными при вставке в начало или конец дека.

Удаление:
- все итераторы делаются недействительными;
- ссылки делаются недействительными при удалении элементов из середины дека и остаются корректными при удалении начальных или конечных элементов дека.

Список

Вставка:
- все итераторы и ссылки остаются корректными.

Удаление:
- все итераторы и ссылки на оставшиеся элементы списка остаются корректными.

Обратные итераторы

Получить обратный итератор *r* можно из обычного (прямого) итератора *p* явным приведением типа, например:

```
r = (vector<int>::reverse_iterator)p;
```

Имеется функция-член *rbegin*(), которая возвращает приведенный к типу обратного итератора итератор *end*(), и функция-член *rend*(), возвращающая приведенный к типу обратного итератора итератор *begin*().

Операции инкремента и декремента прямого и обратного оператора взаимно обратны: *r*++ перемещает итератор в том же направлении, что и *p*--, а *r*-- — в том же направлении, что и *p*++.

Для операции разыменования * выполняется следующее базовое соотношение:

если *r* может быть получен из *p*, то **r* равно *(*p* – 1).

Функция-член *base*() обратного итератора возвращает прямой итератор, который можно было бы использовать для получения данного обратного итератора явным приведением типа: если *r* может быть получен из *p*, то *r.base*() == *p*. Или, иначе говоря, ((*reverse_iterator*)*p*).*base* == *p*.

Примеры

В следующем примере рассматривается последовательный контейнер *cont* с исходными элементами 1, 2, 3, 4, 5.

Итераторы *p2, p3, p4, p5* связаны с элементами 2, 3, 4, 5.

Обратные итераторы *r2, r3, r4, r5* определены следующим образом (*rev* — псевдоним типа обратного итератора для *cont*):

```
r2=(rev)p2;
r3=(rev)p3;
r4=(rev)p4;
r5=(rev)p5;
```

Значения разыменованных итераторов для исходного контейнера:

	*p2	*p3	*p4	*p5	*r2	*r3	*r4	*r5
(вектор)	2	3	4	5	1	2	3	4
(дек)	2	3	4	5	1	2	3	4
(список)	2	3	4	5	1	2	3	4

После выполнения оператора

```
x = cont.erase(p3); // или x =
   cont.erase(r3.base());
```

значения разыменованных итераторов будут следующими («*» означает, что попытка разыменования приводит к непредсказуемым результатам):

	*x	*p2	*p3	*p4	*p5	*r2	*r3	*r4	*r5
(вектор)	4	2	*	*	*	1	*	*	*
(дек)	4	*	*	*	*	*	*	*	*
(список)	4	2	*	4	5	1	*	2	4

Затем повторно выполняется инициализация итераторов *p4* и *r4*

```
p4 = x;
r4 = (rev)p4;
```

и выполняются операторы

```
p3 = cont.insert(p4, 3);
     // или p3 = cont.insert(r4.base(), 3);
r3 = (rev)p3;
```

Значения разыменованных итераторов будут следующими:

	*p2	*p3	*p4	*p5	*r2	*r3	*r4	*r5
(вектор)	2	3	*	*	1	2	*	*
(дек)	*	3	*	*	*	2	*	*
(список)	2	3	4	5	1	2	3	4

Анализ полученных результатов полностью соответствует ранее описанным правилам использования функций *insert* и *erase*, а также правилам, связанным с корректностью итераторов. Имеется лишь одно не вполне очевидное обстоятельство, касающееся того, что происходит с обратными итераторами списка, значения которых были связаны с удаляемым элементом (*r4*) и с элементом, предшествующим удаляемому (*r3*).

В случае итератора *r3* он становится недействительным, что является вполне естественным, так как уничтожается тот элемент, на который указывал итератор *r3.base*().

В случае итератора *r4* ситуация интереснее. Несмотря на то что значение, которое он возвращал, пропало, сам этот итератор сохранился, поскольку сохранился связанный с ним прямой итератор *r4.base*() (и, кстати, хотя это не отражено в приведенных данных, после выполнения операции удаления значение *r4.base*() не изменилось). Однако, поскольку после удаления элемента 3 элементом, предшествующим «базовому» элементу, связанному с итератором *r4.base*(), является элемент 2, именно его значение возвращается при разыменовании обратного итератора *r4*. Таким образом, перед удалением элемента 3 значение итератора *r4* было равно 3, а после его удаления значение становится равным *предшествующему* значению (т. е. 2). При вставке элемента 3 перед элементом 4 базовый элемент для обратного итератора *r4* не изменился (он по-прежнему равен *p4*), но, поскольку теперь перед ним находится элемент 3, именно это значение (3) возвращается разыменованным итератором *r4*.