



М. Э. Абрамян

ВВЕДЕНИЕ В СТАНДАРТНУЮ БИБЛИОТЕКУ ШАБЛОНОВ C++

**ОПИСАНИЕ
ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ
УЧЕБНЫЕ ЗАДАЧИ**

STL

Михаил Абрамян

**Введение в стандартную
библиотеку шаблонов C
++. Описание, примеры
использования, учебные задачи**

«Южный Федеральный Университет»

2017

Абрамян М. Э.

Введение в стандартную библиотеку шаблонов C++. Описание, примеры использования, учебные задачи / М. Э. Абрамян — «Южный Федеральный Университет», 2017

ISBN 978-5-9275-2374-0

Учебник состоит из трех основных разделов. Первый раздел содержит описание стандартной библиотеки шаблонов C++, во втором приводятся примеры ее применения, а третий представляет собой задачник из 300 учебных заданий, охватывающих все разделы стандартной библиотеки. При описании библиотеки учитываются нововведения стандарта C++11. В четвертом, дополнительном разделе дается обзор средств электронного задачника Programming Taskbook for STL, позволяющих выполнять учебные задания более быстро и эффективно. Для студентов бакалавриата, обучающихся по направлению подготовки 02.03.02 «Фундаментальная информатика и информационные технологии».

ISBN 978-5-9275-2374-0

© Абрамян М. Э., 2017

© Южный Федеральный
Университет, 2017

Содержание

Предисловие	5
Раздел 1. Описание библиотеки STL	7
1.1. Итераторы	7
1.1.1. Общее описание	7
1.1.2. Итераторы потоков ввода-вывода	8
1.2. Контейнеры	10
1.2.1. Общее описание	10
1.2.2. Типы, определенные в контейнерах. Параметры конструкторов	13
1.2.3. Функции-члены всех контейнеров	15
1.2.4. Функции-члены последовательных контейнеров	16
1.2.5. Дополнительные функции-члены класса list	20
1.2.6. Функции-члены ассоциативных контейнеров	21
Конец ознакомительного фрагмента.	23

М. Э. Абрамян

Введение в стандартную библиотеку шаблонов C++. Описание, примеры использования, учебные задачи

Предисловие

Книга, предлагаемая вашему вниманию, представляет собой практико-ориентированный учебник по стандартной библиотеке шаблонов языка C++. Для библиотеки шаблонов часто используется название STL (Standard Template Library), которое является неформальным, однако позволяет отличить ее от остальных частей стандартной библиотеки C++. Начиная с 1998 г. библиотека STL входит в стандарт C++ ISO/IEC 14882 (C++98); она содержит средства для создания и преобразования различных структур данных и использует технологию обобщенного программирования. Архитектура библиотеки STL базируется на трех основных компонентах: контейнерах, итераторах и алгоритмах. *Контейнеры* предназначены для хранения наборов объектов в памяти; STL включает две группы контейнеров: последовательные и ассоциативные, в каждую группу входят контейнеры с различными свойствами, что позволяет выбирать контейнер, наиболее подходящий для решения поставленной задачи. *Итераторы* обеспечивают унифицированные средства доступа к содержимому контейнера. Благодаря концепции итераторов, базирующейся на средствах обобщенного программирования, оказалось возможным реализовать универсальные *алгоритмы* – вычислительные процедуры, предназначенные для анализа и преобразования контейнеров. Один и тот же алгоритм может быть применен к любым контейнерам, обладающим требуемыми для этого алгоритма свойствами (точнее, имеющим итераторы того типа, который необходим для корректной работы алгоритма). Еще одной составной частью библиотеки STL являются *функциональные объекты*, представляющие собой обобщения функций и фигурирующие во многих алгоритмах. В пересмотренном стандарте C++ ISO/IEC 14882:2011 (C++11) библиотека STL была дополнена рядом новых возможностей.

Библиотека STL является одной из наиболее трудных для изучения частей стандартной библиотеки C++. Во-первых, это достаточно большая часть стандартной библиотеки: она включает 5 основных видов итераторов, а также их модификации, 7 основных и ряд дополнительных контейнеров, около 70 (в стандарте C++11 – около 90) алгоритмов, большинство из которых реализовано в нескольких вариантах, и большое число стандартных функциональных объектов. Во-вторых, архитектура библиотеки STL основана на шаблонах – весьма сложном разделе языка C++ [3]. Следует заметить, что особенности механизма шаблонов языка C++ затрудняют поиск и исправление ошибок, допущенных при использовании средств библиотеки STL (в частности, сообщения компилятора об ошибке нередко связываются с фрагментами стандартного программного кода, а не с теми операторами разрабатываемой программы, в которых фактически была допущена ошибка). В то же время библиотека STL относится к тем основным частям стандартной библиотеки, владение которыми является обязательным условием для квалифицированной разработки программ на языке C++.

По библиотеке STL имеется обширная учебная литература, в том числе и на русском языке. Можно отметить книги [2, 4, 6, 7], целиком посвященные STL, а также соответствующие разделы в известных учебниках [5, 8]. Однако очень немногие издания содержат наборы упражнений, позволяющие закрепить полученные знания (в частности, из перечисленных книг

упражнения содержат лишь учебники универсального содержания [5, 8]). При этом предлагаемые упражнения не охватывают все возможности библиотеки и являются достаточно сложными, что затрудняет их использование при проведении лабораторных занятий. Настоящее издание призвано восполнить этот пробел. Помимо компактного, но в то же время достаточно подробного описания всех элементов стандартной библиотеки шаблонов, приведенного в разделе 1, а также примеров их применения (которым посвящен раздел 2), оно содержит набор из 300 задач по всем основным разделам стандартной библиотеки и, таким образом, позволяет не только ознакомиться с ее возможностями, но и освоить эту библиотеку на практике. Задачи разбиты на 7 групп; содержание групп, их особенности и формулировки всех задач приведены в разделе 3.

В описании основных компонентов библиотеки STL учитываются нововведения стандарта C++11. Задания ориентированы в основном на базовый вариант библиотеки STL, соответствующий стандарту C++98, однако при их выполнении вполне допустимо (и более удобно) использовать новые возможности, появившиеся в стандарте C++11.

Все задачи, приведенные в книге, входят в состав *электронного задачника Programming Taskbook for STL (PT for STL)*, являющегося одним из дополнений универсального задачника по программированию Programming Taskbook. Задачник PT for STL может использоваться совместно со средами программирования Microsoft Visual Studio 2008, 2010, 2012, 2013, 2015, 2017 и Code::Blocks, начиная с версии 13. Он позволяет генерировать программы-заготовки для выбранных заданий, предоставляет программам наборы тестовых исходных данных, проверяет правильность полученных результатов, диагностирует различные виды ошибок и отображает на экране все данные, связанные с заданием. Все эти возможности существенно ускоряют выполнение заданий. Особенности применения задачника при выполнении заданий подробно описываются в разделе 2, а дополнительные средства задачника, упрощающие ввод, вывод и отладочную печать данных, – в разделе 4.

Получить дополнительную информацию об электронном задачнике Programming Taskbook и его дополнении Programming Taskbook for STL (а также других его дополнениях) и скачать их дистрибутивы можно на сайте электронного задачника – <http://ptaskbook.com/>.

Автор считает своим приятным долгом выразить искреннюю благодарность Денису Владимировичу Дуброву и Артему Михайловичу Пеленицыну, которые прочитали первый вариант рукописи и высказали много ценных замечаний.

Раздел 1. Описание библиотеки STL

1.1. Итераторы

1.1.1. Общее описание

В библиотеке STL используются пять основных видов итераторов:

- итераторы чтения;
- итераторы записи;
- однонаправленные итераторы;
- двунаправленные итераторы;
- итераторы произвольного доступа.

Для каждого вида итераторов определен набор операций, причем двумя операциями, доступными для всех видов итераторов, являются *операция инкремента* ++, которая передвигает итератор *r* на следующий элемент последовательности (++*r* и *r*++), и *операция разыменования* *, возвращающая значение текущего элемента (* *r* и вариант *r*->*m* для доступа к члену *m* разыменованного объекта).

Операция разыменования имеет следующие особенности:

- в случае итераторов чтения операция * не может использоваться для изменения элемента;
- в случае итераторов записи операция * не может использоваться для получения значения элемента (выражение * *r* можно использовать только в левой части присваивания);
- для прочих итераторов операция * может использоваться как для получения значения элемента, так и для изменения этого значения.

Операции сравнения итераторов на равенство == и != реализованы для всех итераторов, кроме итераторов записи.

Для однонаправленных итераторов не определяются новые операции (по сравнению с итераторами чтения или записи).

Для двунаправленных итераторов в дополнение к операции инкремента ++ вводится *операция декремента* -- (также в двух видах: --*r* и *r*--).

Наконец, для итераторов произвольного доступа добавляются *операция индексирования* [*i*], позволяющая сразу обратиться к элементу последовательности с требуемым индексом (*r*[*i*]), и *операция смещения на указанное количество элементов*, причем в оба направления (*r* + *i* и *r* - *i*). Имеется также *операция разности двух итераторов*, позволяющая определить расстояние между элементами, с которыми они связаны (*r2* - *r1*).

Таким образом, набор операций для итераторов произвольного доступа аналогичен набору операций для обычных указателей.

Для итераторов, не являющихся итераторами произвольного доступа, также можно выполнять действия, связанные со смещением и определением расстояния, используя функции из заголовочного файла <iterator>:

- `advance(p, n)` – передвигает итератор `p` на `n` позиций вперед ($n \geq 0$); для двунаправленного итератора можно использовать $n < 0$ для перемещения назад;
- `distance(p1, p2)` – возвращает расстояние между итераторами `p1` и `p2` (в предположении, что расстояние неотрицательно, т. е. что итератор `p1` предшествует итератору `p2` или совпадает с ним; для двунаправленных итераторов `p2` может предшествовать итератору `p1`, в этом случае расстояние будет отрицательным).

Два итератора обычно используются для задания *диапазона элементов*, при этом предполагается, что первый итератор (*first*) указывает на начальный элемент диапазона, а второй итератор (*last*) указывает на позицию *за* конечным элементом диапазона (причем эта позиция может не быть связана с существующим элементом). Чтобы подчеркнуть отмеченные особенности для диапазонов, определяемых итераторами, они часто записываются в виде *полуинтервала* [*first, last*) (левая граница диапазона включается, правая – нет). Полуинтервал [*first, first*) не содержит ни одного элемента.

В качестве итераторов чтения и итераторов записи можно использовать итераторы всех остальных видов (однонаправленные, двунаправленные, произвольного доступа); следует лишь учитывать, что итераторы записи можно инкрементировать неограниченно, тогда как итераторы других видов всегда связываются с некоторым диапазоном допустимых элементов. В качестве однонаправленных итераторов можно использовать двунаправленные итераторы и итераторы произвольного доступа, а в качестве двунаправленных итераторов – итераторы произвольного доступа.

Для всех видов итераторов определены их модификации – *константные итераторы*, отличающиеся от обычных тем, что их разыменование дает константное значение.

Особыми итераторами являются *итераторы потоков ввода-вывода* (см. п. 1.1.2), *обратные итераторы* (см. п. 1.2.9) и *итераторы вставки* (см. п. 1.3.4).

1.1.2. Итераторы потоков ввода-вывода

Стандартные потоковые итераторы `istream_iterator<T>` и `ostream_iterator<T>` (шаблонные классы) определены в заголовочном файле `<iterator>`.

Имеются два варианта конструктора для *итератора потокового чтения* `istream_iterator`: вариант с параметром-поток `stream` создает итератор для чтения из данного потока, вариант без параметров создает итератор, обозначающий конец потока (все итераторы, обозначающие конец потока, считаются равными друг другу и не равными никаким другим итераторам потокового чтения).

Ниже перечислены свойства потоковых итераторов чтения:

- тип `T` определяет тип элементов данных, которые считываются из потока;
- чтение элемента из потока выполняется в начальный момент работы с итератором, а затем при каждой операции инкремента `++`;
- имеются два варианта операции `++`: префиксный инкремент (`++p`) и постфиксный инкремент (`p++`);
- операция `*` (и ее вариант `->`) возвращает последнее прочитанное значение, причем эту операцию можно использовать неоднократно для получения того же самого значения;
- при достижении конца потока итератор становится равным итератору конца потока; последующие вызовы операции инкремента игнорируются, а

в результате вызова операции `*` всегда возвращается значение последнего прочитанного из потока элемента (если же с итератором был связан пустой поток, то результат операции `*` не определен, хотя и не приводит к аварийному завершению программы).

Для *итератора потоковой записи* `ostream_iterator<T>` также определены два конструктора: первый конструктор содержит единственный параметр `stream`, задающий поток вывода, а второй конструктор дополнительно к параметру `stream` содержит второй параметр `delim`, задающий *разделитель*, который добавляется в поток вывода после каждого выведенного элемента (если параметр `delim` не указан, то между выводимыми элементами никакой разделитель не добавляется).

Ниже перечислены свойства потоковых итераторов записи:

- специальный конструктор для создания итератора конца потока вывода не предусмотрен;
- операции `*` и `++` не выполняют никаких действий и просто возвращают сам итератор;
- операция присваивания `p = выражение` (где `p` – имя итератора записи) записывает значение выражения в поток вывода.

1.2. Контейнеры

1.2.1. Общее описание

Данный раздел посвящен контейнерам, входящим в стандартную библиотеку шаблонов C++. Подробно описываются те основные виды последовательных и ассоциативных контейнеров, с которыми связаны задания, приводимые в книге: это векторы (`vector`), деки (`deque`), списки (`list`), множества (`set`), мультимножества (`multiset`), отображения (`map`) и мультиотображения (`multimap`), а также текстовые строки (`string`), которые относят к *псевдоконтейнерам*. Другие виды контейнеров кратко описываются в п. 1.2.8: это *контейнеры-адаптеры* стек (`stack`), очередь (`queue`) и очередь с приоритетом (`priority_queue`), а также контейнеры, добавленные в библиотеку STL в стандарте C++11 (`array`, `forward_list` и ассоциативные контейнеры на базе хеш-функций). Все контейнеры определены в пространстве имен `std`.

В таблицах 1 и 2 перечислены характеристики основных видов последовательных и ассоциативных контейнеров.

Таблица 1

Последовательные контейнеры

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

Таблица 2

Ассоциативные контейнеры

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

В описаниях шаблонов контейнеров, приводимых в таблицах 1 и 2, и далее при описании конструкторов и функций-членов этих контейнеров (см. п. 1.2.2–1.2.6) не указывается дополнительный тип Alloc, который обычно устанавливается по умолчанию. Необязательные параметры заключаются в квадратные скобки, набранные полужирным шрифтом: []. В частности, если в шаблоне ассоциативного контейнера не указывается операция сравнения Compare, то она считается равной less<Key>.

Контейнеры могут содержать данные только тех типов T, которые удовлетворяют некоторым естественным условиям (например, в стандарте C++98 требуется, чтобы для типа T был определен конструктор копирования и операция присваивания).

Все рассматриваемые **последовательные контейнеры** допускают вставку новых элементов в любую позицию и удаление элементов из любой позиции. *Векторы* оптимизированы для быстрого (за константное время) выполнения операций вставки и удаления, связанных с концом последовательности элементов (функции-члены push_back и pop_back), а *деки* – для операций, связанных как с началом, так и с концом последовательности (функции-члены push_back и pop_back, push_front и pop_front). В то же время, векторы обладают рядом особенностей, отсутствующих у деков; в частности, они имеют такую характеристику, как *емкость*, которая доступна и для чтения (функция-член capacity) и для изменения (функция-член reserve). *Текстовые строки* string обладают возможностями, аналогичными возможностям векторов с символьными элементами. *Списки* позволяют выполнять быструю вставку и удаление элементов для любой позиции, однако доступ к элементу списка по его номеру требует линейного времени (т. е. зависит от текущего размера списка). По этой причине для списков не реализована операция индексирования, а связанные со списками итераторы являются двунаправленными (а не итераторами произвольного доступа, как для всех остальных последовательных контейнеров). Еще одной особенностью списка является то, что операции вставки и удаления не влияют на корректность итераторов и ссылок, связанных с другими его элементами, в то время как для векторов и деков вставка или удаление элементов может приводить к тому, что некоторые (или все) итераторы и/или ссылки окажутся недействительными (подроб-

ности приведены в п. 1.2.7). Кроме того, для списков предусмотрен набор дополнительных функций-членов, отсутствующих у других последовательных контейнеров и представляющих собой оптимизированные реализации соответствующих алгоритмов (см. п. 1.2.5).

Все рассматриваемые **ассоциативные контейнеры** хранят последовательности своих элементов в отсортированном виде. Сортировка выполняется по ключу, причем для множеств и мультимножеств ключами выступают сами элементы (типа T), а в отображениях и мультиотображениях хранятся пары типа `pair<Key, T>`, первый компонент которых считается *ключом* (key), а второй – *значением* (value). По умолчанию порядок определяется операцией < для типа ключа Key, однако его можно явно указать в шаблоне контейнера в виде функционального объекта, реализующего бинарный предикат с параметрами типа Key и со свойствами операции сравнения «меньше». Мультимножества и мультиотображения, в отличие от множеств и отображений, позволяют хранить *набор* элементов с эквивалентными ключами (ключи считаются эквивалентными, если ни один из них не является меньшим, чем другой). Для отображения определена операция индексирования с дополнительными возможностями (см. п. 1.2.6). Вставка новых элементов в любой ассоциативный контейнер сохраняет его упорядоченность. И операция вставки, и операция удаления для ассоциативных контейнеров требует логарифмического времени, если для этих операций указывается параметр-ключ. За это же время выполняется и поиск элементов по ключу, для реализации которого в ассоциативных контейнерах предусмотрен целый набор функций-членов. Указанные свойства ассоциативных контейнеров делают их удобным механизмом для группировки и объединения наборов данных по ключу.

Поскольку контейнеры, перечисленные в таблицах 1 и 2, имеют много одинаковых функций-членов, все они далее рассматриваются совместно: в п. 1.2.2 перечисляются типы, связанные с контейнерами, и описываются варианты конструкторов, в п. 1.2.3 приводятся функции-члены, имеющиеся у всех контейнеров, в п. 1.2.4 – функции-члены последовательных контейнеров, в п. 1.2.5 – дополнительные функции-члены списков, в п. 1.2.6 – функции-члены ассоциативных контейнеров. 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; кроме того, специальным образом помечаются функции-члены, добавленные в стандарте C++11 (например, текст **vector(C++11)**, **string** означает, что соответствующая функция-член доступна только для классов `vector` и `string`, причем для класса `vector` – только начиная со стандарта C++11). Если один из прежних вариантов функции-члена отсутствует в стандарте C++11, то он помечается текстом **C++98**.

Класс `string` имеет гораздо больше функций-членов, чем остальные 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 последующих описаниях функций-членов некоторые переменные всегда связываются с данными фиксированного типа (этот тип определяется в самом контейнере – см. п. 1.2.2):

- n имеет тип `size_type`;
- k имеет тип `key_type`;
- x (а также `x1`, `x2`, ...) имеет тип `value_type`;
- `init_list` имеет тип *списка инициализации* `initializer_list<value_type>` (элементы списка инициализации разделяются запятыми, сам список заключается в фигурные скобки);
 - `pos`, `hintpos`, `first` и `last` (а также `pos_lst`, `first_lst`, `last_lst`) имеют тип итератора соответствующего контейнера (`iterator`).

Переменная `other` обозначает параметр, являющийся контейнером того же типа, что и контейнер, для которого вызывается функция-член. Переменные `InIterFirst` и `InIterLast` обозначают итераторы чтения, которые могут быть связаны с контейнером другого типа (при этом тип элементов контейнера должен совпадать с типом элементов контейнера, для которого вызывается функция-член).

Функции-члены `begin`, `end`, `rbegin`, `rend`, `front`, `back`, `at`, `equal_range`, `find`, `lower_bound`, `upper_bound` (а также `data` для векторов и `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. В стандарте C++11 константные варианты функций-членов `begin`, `end`, `rbegin`, `rend` можно использовать с именами `cbegin`, `cend`, `crbegin`, `crend` соответственно.

1.2.2. Типы, определенные в контейнерах. Параметры конструкторов

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).

```
key_type
```

ассоциативные контейнеры

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

`mapped_type`

(мульти)отображения

Тип значений `T` для (мульти)отображений.

`key_compare`

ассоциативные контейнеры

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

`value_compare`

ассоциативные контейнеры

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

Ниже приводятся варианты параметров для конструкторов контейнеров. Большинство вариантов может использоваться для всех видов рассматриваемых контейнеров; единственный особый вариант для последовательных контейнеров помечен соответствующим образом. Следует обратить внимание на вариант конструктора, появившийся в стандарте C++11 и использующий список инициализации.

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

`(other)`

Создает копию контейнера `other` (тип контейнера `other` должен совпадать с типом создаваемого контейнера). В стандарте C++11 добавлен вариант конструктора с параметром `other`, обеспечивающий *перемещение* элементов контейнера `other`, если контейнер `other` является

ссылкой на *r*-значение (r-value reference; для описания подобной ссылки используется двойной символ &&)

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

последовательные контейнеры

Создает последовательный контейнер, содержащий *n* копий значения *x*. Для строк `string` обязательными являются оба параметра. В стандарте C++11 вариант с одним параметром оптимизирован таким образом, чтобы избежать создания ненужных копий объектов *T*.

```
init_list
```

C++11

Конструктор, использующий *список инициализации* (initializer list). Перед списком может указываться символ `=`. Например, создать вектор с элементами 1, 2, 4 можно с помощью любого из следующих вариантов описания:

```
vector<int> a{1, 2, 4};  
vector<int> a = {1, 2, 4};
```

1.2.3. Функции-члены всех контейнеров

```
operator=(other)
```

Удаляет все элементы контейнера и копирует в него все элементы контейнера `other` (тип контейнера `other` должен совпадать с типом преобразуемого контейнера). Возвращает полученный контейнер. В стандарте C++11 добавлен вариант операции `=`, обеспечивающий *перемещение* элементов контейнера `other`, если контейнер `other` является ссылкой на *r*-значение (r-value reference), а также вариант со *списком инициализации* `init_list` (см. описание последнего варианта конструктора в п. 1.2.2).

```
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(other)
```

Меняет местами содержимое данного контейнера и контейнера other того же типа.

1.2.4. Функции-члены последовательных контейнеров

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

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

C++11

```
void assign(init_list)
```

Удаляет все элементы контейнера и копирует в него новые данные (n копий значения x или элементы из диапазона [InIterFirst, InIterLast)). В стандарте C++11 добавлен вариант с параметром init_list – списком инициализации. Данная функция расширяет возможности, предоставляемые операцией копирования =.

vector, deque, string

reference `operator[](n)`

Возвращает ссылку на элемент с индексом n ($0 \leq n < \text{size}()$). Выход за границы не контролируется. Для типа `string` в случае $n == \text{size}()$ возвращается символ с кодом 0.

vector, deque, string

reference `at(n)`

Возвращает ссылку на элемент с индексом n ($0 \leq n < \text{size}()$). Выход за границы приводит к возбуждению исключения `out_of_range`.

vector, deque, list, string(C++11)

reference `back()`

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

vector, string

size_type `capacity() const`

Возвращает текущую емкость контейнера.

vector(C++11), string

`T*` `data()`

Возвращает указатель на внутренний массив, содержащий элементы вектора или символы строки. Для строк реализован только в константном варианте и возвращает константный указатель.

vector(C++11), deque(C++11), list(C++11)

iterator `emplace(pos, arg1, arg2, ...)`

Вставляет в позицию `pos` контейнера новый элемент, создавая этот элемент «на месте» и используя при его конструировании параметры `arg1, arg2, ...`. Позволяет избежать дополнительных операций копирования или перемещения, выполняемых при использовании функции-члена `insert`. Возвращает итератор, указывающий на вставленный элемент.

```
vector(C++11), deque(C++11), list(C++11)  
void emplace_back(arg1, arg2, ...)
```

Добавляет в конец контейнера новый элемент, создавая этот элемент «на месте» и используя при его конструировании параметры `arg1`, `arg2`, Позволяет избежать дополнительных операций копирования или перемещения, выполняемых при использовании функции-члена `push_back`.

```
deque(C++11), list(C++11)  
void emplace_front(arg1, arg2, ...)
```

Добавляет в начало контейнера новый элемент, создавая этот элемент «на месте» и используя при его конструировании параметры `arg1`, `arg2`, Позволяет избежать дополнительных операций копирования или перемещения, выполняемых при использовании функции-члена `push_front`.

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

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

```
vector, deque, list, string(C++11)  
reference front()
```

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

```
iterator insert(pos, x)  
void insert(pos, n, x)  
C++98  
void insert(pos, InIterFirst, InIterLast)
```

```
C++11  
iterator insert(pos, InIterFirst, InIterLast)  
iterator insert(pos, init_list)
```

Вставляет в контейнер новые данные, начиная с позиции `pos` (соответственно одно или `n` значений `x`, элементы из диапазона `[InIterFirst, InIterLast)` или элементы из списка инициа-

зации `init_list`). Первый вариант функции-члена возвращает итератор, указывающий на вставленный элемент. Два последних варианта, добавленных в стандарт C++11 вместо третьего варианта, возвращают итератор, указывающий на первый вставленный элемент, или исходное значение `pos`, если диапазон или список инициализации являются пустыми.

vector, deque, list, string(C++11)

```
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. В стандарте C++11 вариант с одним параметром оптимизирован таким образом, чтобы избежать создания ненужных копий объектов `T`.

vector(C++11), deque(C++11), string(C++11)

```
void shrink_to_fit()
```

Позволяет уменьшить размер памяти, выделенной для хранения элементов контейнера, однако не гарантирует, что память действительно будет уменьшена.

1.2.5. 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(list[, 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. В стандарте C++11 добавлены варианты с параметром `lst`, являющимся ссылкой на *r-значение* (*r-value reference*).

```
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`). В стандарте C++11 добавлены варианты с параметром `lst`, являющимся ссылкой на r-значение (r-value reference).

```
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.

1.2.6. Функции-члены ассоциативных контейнеров

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

map

Возвращает ссылку на значение, связанное с ключом `k`. Если ключ `k` отсутствует в контейнере, то в контейнер добавляется пара с ключом `k` и значением по умолчанию `T()`, и операция `[]` возвращает ссылку на это значение. Фактически данная операция возвращает следующее выражение: `insert(make_pair(k, T())).first->second`. В стандарте C++11 добавлен вариант с параметром `k`, являющимся ссылкой на r-значение (r-value reference).

```
T& at(k)
```

map(C++11)

Возвращает ссылку на значение, связанное с ключом `k`. Если ключ `k` отсутствует в контейнере, то возбуждается исключение `out_of_range`.

```
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, bool> emplace(arg1, arg2, ...)
```

set(C++11), map(C++11)

multiset(C++11), multimap(C++11)

```
iterator emplace(arg1, arg2, ...)
```

Вставляет в контейнер новый элемент, создавая этот элемент «на месте» и используя при его конструировании параметры `arg1, arg2, ...`. Позволяет избежать дополнительных операций копирования или перемещения, выполняемых при использовании функции-члена `insert`. Если элемент с указанным ключом уже имеется в контейнере, то в случае множества и отображения попытка вставки игнорируется. Возвращает итератор, указывающий на вставленный элемент, а также (в варианте для множества и отображения) логическое значение, определяющее, была

ли произведена вставка. Если вставка не была произведена из-за того, что в контейнере (множестве или отображении) уже существует элемент с таким же ключом, то возвращается позиция уже имеющегося элемента с этим ключом.

```
iterator emplace_hint(hintpos, arg1, arg2, ...) C++11
```

Вставляет в контейнер новый элемент, создавая этот элемент «на месте» и используя при его конструировании параметры `arg1`, `arg2`, Позволяет избежать дополнительных операций копирования или перемещения, выполняемых при использовании функции-члена `insert`. Параметр `hintpos` является «подсказкой» для позиции вставки: элемент `x` вставляется максимально близко *перед* позицией `hintpos`. Возвращает итератор, указывающий на вставленный элемент (если элемент с указанным ключом уже имеется в контейнере, то в случае множества и отображения попытка вставки игнорируется и возвращается позиция уже имеющегося элемента с этим ключом).

```
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))`.

```
size_type erase(k) C++98  
void erase(pos)  
void erase(first, last)  
  
iterator erase(pos) C++11  
iterator erase(first, last)
```

Удаляет элемент (элементы) с ключом `k`, элемент в позиции `pos` или все элементы в диапазоне `[first, last)`. В первом случае возвращает количество удаленных элементов (для множества и отображения это либо 0, либо 1). Два последних варианта, добавленных в стандарт C++11 вместо двух предыдущих вариантов, возвращают итератор, указывающий на элемент, следующий за последним удаленным элементом (или итератор `end()`, если были удалены конечные элементы контейнера).

```
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`.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «ЛитРес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на ЛитРес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.