КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 19.1 — 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

Предоставляет возможности для организации информации в виде коллекций (массивы, вектора, списки, деки, стеки, множества, очереди и отображения).

Контейнер — это специфическая структура данных, которая содержит данные, обычно в неограниченном количестве. У каждого типа контейнера есть ограничения на то, как фактически получать доступ, добавлять или удалять данные.

Контейнерные классы — это классы, которые в состоянии хранить в себе элементы различных типов данных. Почти все контейнерные классы реализованы как шаблонные и, таким образом, могут хранить данные любого типа. Основная идея шаблона состоит в создании родового класса, который определяется при создании объекта этого класса.

Классы контейнеров могут включать целые серии других объектов, которые, в свою очередь, тоже могут являться контейнерами.

Чтобы правильно подобрать контейнер для конкретного случая, очень важно правильно понимать различия разновидностей контейнеров. От этого в значительной степени зависит скорость работы кода и эффективность использования памяти.

Предоставляет две категории разновидностей классов контейнеров:

- последовательные (sequence containers)
- ассоциативные (associative containters).

Последовательные контейнеры — это упорядоченные коллекции, где каждый элемент занимает определенную позицию.

Позиция элемента зависит от места его вставки.

Контейнеры подразделяются на последовательные и ассоциативные.

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).

Массив (array)

Массивы представляют собой последовательные контейнеры фиксированного размера — они содержат определенное количество элементов, упорядоченных в строгой линейной последовательности. Массивы обладают произвольным доступом, что означает, что доступ к любому элементу с целочисленным индексом выполняется за постоянное время.

Вектор (vector)

Векторы — это последовательные контейнеры, представляющие массивы, размер которых может изменяться. Векторы обладают произвольным доступом, что означает, что доступ к любому элементу с целочисленным индексом выполняется за постоянное время (точно так же, как массив).

Однонаправленный список (Forward list)

Однонаправленные списки — это последовательные контейнеры, которые позволяют выполнять операции вставки и удаления в любом месте последовательности за постоянное время.

Список (list)

Списки представляют собой последовательные контейнеры, которые позволяют выполнять операции вставки и удаления с постоянным временем в любом месте последовательности и просмотр в обоих направлениях.

Стек (LIFO stack)

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

Очередь (FIFO queue)

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

Дек (deque — double ended queue)

deque — нестандартное сокращение от «двусторонней очереди». Двусторонние очереди — это последовательные контейнеры с динамическими размерами, которые можно расширять или ужимать с обоих концов (как в начале, так и в конце).

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).

Набор (set)

Наборы — это контейнеры, в которых хранятся уникальные элементы в определенном порядке.

Неупорядоченный набор (unordered set)

Неупорядоченные наборы — это контейнеры, в которых уникальные элементы хранятся в произвольном порядке и которые позволяют быстро извлекать отдельные элементы на основе их значения.

Словарь (map)

Словари — это ассоциативные контейнеры, в которых хранятся элементы, сформированные комбинацией значения ключа и сопоставленного ему значения в определенном порядке.

Неупорядоченный словарь (unordered map)

Неупорядоченные словари — это ассоциативные контейнеры, в которых хранятся элементы, сформированные комбинацией ключевого значения и значения, сопоставленного ему. Они позволяют быстро извлекать отдельные элементы на основе их ключей.

Класс массива <array>

```
#include <array>

template < class T, size_t N > class array;
```

Параметры шаблона

T — тип содержащихся элементов.

N — размер массива, выраженный в количестве элементов.

Массивы представляют собой контейнеры последовательности фиксированного размера: они содержат определенное количество элементов, упорядоченных в строгой линейной последовательности.

Внутренне массив не хранит никаких данных, кроме содержащихся в нем элементов.

Даже размер массива, который является параметром шаблона, фиксируется во время компиляции.

Он так же эффективен с точки зрения размера хранилища, как и обычный массив, объявленный с использованием синтаксиса скобок языка (**[ ]**). Этот класс просто добавляет к нему уровень членов и глобальных функций, в связи с чем массивы могут использоваться в качестве стандартных контейнеров.

В отличие от других стандартных контейнеров, массивы имеют фиксированный размер и распределение его элементов не управляется с помощью аллокатора, поскольку массив представляет собой агрегатный тип, инкапсулирующий массив элементов фиксированного размера.

Следовательно, массивы не могут быть расширены или сжаты динамически.

Допускаются массивы нулевого размера, однако их нельзя разыменовывать (члены front, back и data).

В отличие от других контейнеров в стандартной библиотеке, обмен (swap) двух контейнеров массива является линейной операцией, которая включает в себя замену местами всех элементов в диапазонах по отдельности, что обычно является не слишком эффективной операцией.

Свойства контейнера

Последовательность

Элементы в контейнерах последовательности упорядочены в строгой линейной последовательности.

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

Смежное хранение

Элементы хранятся в смежных ячейках памяти, что обеспечивает возможность произвольного доступа к элементам с постоянным временем. Для доступа к другим элементам к указателям на элементы можно добавлять смещение.

Агрегат фиксированного размера

Контейнер использует неявные конструкторы и деструкторы для статического распределения необходимого пространства. Его размер — константа времени компиляции. Никаких затрат памяти или времени не требуется

Общедоступные функции-члены

Итераторы

`begin` — возвращает итератор, указывающий на первый элемент в последовательности
`end` — возвращает итератор, указывающий на последний элемент в последовательности
`rbegin` — Вернуть обратный итератор в начало обратного направления
`rend` — Вернуть обратный итератор в конец обратного направления
`cbegin` — Вернуть **`const_iterator`** в начало
`cend` — Вернуть **`const_iterator`** в конец
`crbegin` — Вернуть **`const_reverse_iterator`** в начало обратного направления
`crend` — Вернуть **`const_reverse_iterator`** в конец обратного направления

Емкость

`size` — возвращает размер
`max_size` — возвращает максимальный размер
`empty` — проверяет, пуст ли массив

Доступ к элементам

`operator[]` — получить доступ к элементу
`at` — получить доступ к элементу
`front` — доступ к первому элементу
`back` — доступ к последнему элементу

`data` — получить указатель на данные

Модификаторы
`fill` — заполнить массив значением
`swap` — поменять местами содержимое массивов

Перегрузки функций, не являющихся членами
`get(array)` — Получить элемент (интерфейс кортежа)
`relational operators(array)` — Операторы отношений для массива

Специализации классов, не являющихся членами
`tuple_element<array>` — Тип элемента кортежа для массива
`tuple_size<array>` — Характеристики размера кортежа для массива

Пример

```
// array::begin example
#include <iostream>
#include <array>

int main () {

    std::array<int, 5> myarray = { 2, 16, 77, 34, 50 };
    std::cout << "myarray contains:";
    std::array<int, 5>::iterator it; // итератор
    for ( it = myarray.begin(); it != myarray.end(); ++it ) {
        std::cout << ' ' << *it;
    }
    std::cout << '\n';

    return 0;
}
```

Выход

```
myarray contains: 2 16 77 34 50
```

Пример

```
// array::begin example
#include <iostream>
#include <array>

int main () {

    std::array<int, 5> myarray = { 2, 16, 77, 34, 50 };
    std::cout << "myarray contains:";

    for ( auto it = myarray.begin(); it != myarray.end(); ++it )
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

Выход

```
myarray contains: 2 16 77 34 50
```

```
// array::back
#include <iostream>
#include <array>

int main () {

  std::array<int, 3> myarray = {5, 19, 77};

  std::cout << "front is: " << myarray.front() << std::endl;   // 5
  std::cout << " back is: " << myarray.back()  << std::endl;   // 77

  myarray.back() = 50;

  std::cout << "myarray now contains:";
  for ( int& x : myarray ) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Вывод

```
front is: 5
 back is: 77
myarray now contains: 5 19 50
```

Вектор (vector)

Vectors are sequence containers representing arrays that can change in size.

```
#include <vector>

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

Как и массивы, векторы используют смежные места хранения для своих элементов, что означает, что к их элементам также можно получить доступ, используя смещения в обычных указателях на его элементы, причем так же эффективно, как и в массивах. Но в отличие от массивов их размер может изменяться динамически, а память под них автоматически выделяется контейнером.

Внутренне векторы используют динамически выделяемый массив для хранения своих элементов. Возможно, потребуется перераспределять память под этот массив, чтобы он увеличивался в размере при вставке новых элементов. Это может потребовать выделение нового массива и перемещение в него всех элементов. Это относительно дорогостоящая задача с точки зрения временых затрат, и поэтому векторы не перераспределяются каждый раз, когда элемент добавляется в контейнер.

По сравнению с массивами, векторы потребляют больше памяти в обмен на возможность управлять выделением памяти и эффективно динамически расти.

По сравнению с другими динамическими последовательными контейнерами (деки, списки и однонаправленные списка) векторы обеспечивают очень эффективный доступ к своим элементам (как и массивы) и относительно эффективно добавляют или удаляют элементы с его конца.

Для операций, которые включают вставку или удаление элементов в позициях, отличных от конца, они работают хуже, чем другие, и имеют менее согласованные итераторы и ссылки, чем списки и однонаправленные списки.

Список (`std::list`)

Определен класс **`list`** и две функции — итераторы **`begin()`** и **`end()`**

Списки представляют собой последовательные контейнеры, которые позволяют выполнять операции вставки и удаления за постоянное время в любом месте последовательности, а также итерацию в обоих направлениях.

```
#include <list>

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

Контейнеры списков реализованы как двусвязные списки.

Двусвязные списки могут хранить каждый из элементов, которые они содержат, в разных и несвязанных местах хранения. Порядок сохраняется внутри за счет ассоциации с каждым элементом ссылки на предшествующий ему элемент и ссылки на следующий за ним элемент.

Они очень похожи на **`forward_list`**, однако основное отличие состоит в том, что объекты **`forward_list`** являются односвязными списками, и поэтому их можно итерировать только вперед, в обмен на то, что они будут немного меньше и эффективнее.

По сравнению с другими базовыми стандартными последовательными контейнерами (массив, вектор и двухсторонняя очередь), списки обычно лучше работают при вставке, извлечении и перемещении элементов в/из любой позиции в контейнере, для которой уже был получен итератор. Это справедливо и для алгоритмов, которые интенсивно используют списки, такие как алгоритмы сортировки.

Главный недостаток списков **`list`** и **`forward_list`** по сравнению с другими последовательными контейнерами состоит в том, что у них отсутствует прямой доступ к элементам по их положению.

Например, чтобы получить доступ к шестому элементу в списке, нужно выполнить итерирование от известной позиции (например, начала или конца) до этой позиции, что занимает время, линейно зависящее от расстояния между ними. Эти контейнеры также потребляют некоторую дополнительную память для хранения информации о связывании, связанной с каждым элементом, что может быть важным фактором для больших списков элементов небольшого размера.

Свойства контейнера

Последовательность

Элементы в контейнерах последовательности упорядочены в строгой линейной последовательности.

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

Двусвязный список

Каждый элемент хранит информацию о том, как найти следующий и предыдущий элементы, позволяя выполнять операции вставки и удаления за постоянное время до или после определенного элемента (это справедливо даже для целых диапазонов).

Прямой произвольный доступ отсутствует.

Используется аллокатор

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

Функции-члены

Все функции-члены являются общедоступными (public)

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

Итераторы

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

Емкость

`empty()` — Проверить, пуст ли контейнер
`size()` — Возвращает количество элементов в списке
`max_size()` — Возвращает максимально допустимое количество элементов в списке

Доступ к элементам

`front(), back()` — возвращает ссылку на первый/последний элемент в списке

---

1 const_iterator

2 const_reverse_iterator

Модификаторы

`assign()` – Назначает новое содержимое контейнеру
`emplace_front()`, `emplace_back()` – создает элемент и вставляет его в начале/конце списка
`push_front()`, `pop_front()` – вставляет/удаляет элемент в начале списка
`push_back()`, `pop_back()` – добавляет/удаляет элемент в конце списка
`emplace()` – создает элемент и вставляет его
`insert()`, `erase()` – вставляет/удаляет элемент
`swap()` – обменивает содержимое
`resize()` – изменяет размер
`clear()` – удаляет содержимое

Операции

`splice()` – переносит элементы из списка в список
`remove()` – удаляет элементы с определенным значением
`remove_if()` – удаляет элементы, удовлетворяющие условию
`unique()` – удаляет дубликаты элементов
`merge()` – объединяет отсортированные списки
`sort()` – сортирует элементы в контейнере
`reverse()` – переставляет элементы в обратном порядке

Аллокатор

`get_allocator()` – получить аллокатор

Конструкторы списков

Создают объект контейнера **list**, инициализируя его содержимое в зависимости от используемой версии конструктора:

По умолчанию (1)	`list();` `explicit list(const allocator_type& alloc);`

С заполнением (2)	`explicit list(size_type n,`
	`const allocator_type& alloc = allocator_type());`
	`list(size_type n, const value_type& val,`
	`const allocator_type& alloc = allocator_type());`

Из диапазона (3)	`template <class InputIterator>`
	`list(InputIterator first,`
	`InputIterator last,`
	`const allocator_type& alloc = allocator_type());`

Копирования (4)	`list (const list& x);` `list (const list& x, const allocator_type& alloc);`
Перемещения (5)	`list (list&& x);` `list (list&& x, const allocator_type& alloc);`
Из списка иниц. (6)	`list (initializer_list<value_type> il,` `const allocator_type& alloc = allocator_type());`

(1) — пустой контейнер, не содержащий элементов;

(2) — создает контейнер из **n** элементов. Каждый элемент является копией **val** (если предоставляется);

(3) — создает контейнер с таким количеством элементов, которое содержится в диапазоне **`[first, last)`**, причем каждый элемент создается из соответствующего элемента в этом же диапазоне и в том же порядке.

(4) — создает контейнер с копией каждого элемента из **`x`** в том же порядке.

(5) — создает контейнер, в который попадают элементы **`x`** (из r-value).

Если указана значение **`alloc`**, элементы реально перемещаются только в том случае, если указанный в конструкторе аллокатор отличается от того, который использовался в списке **`x`**. В случае одинаковых аллокаторов никакие элементы не создаются (их право владения передается напрямую). Список **`x`** остается в неопределенном, но допустимом состоянии.

(6) — Создает контейнер с копией каждого из элементов списка инициализации **`il`** в том же порядке.

Контейнер хранит внутреннюю копию **`alloc`**, которая используется для выделения и освобождения памяти для его элементов, а также для их создания и уничтожения (как указано в его **`allocator_traits`**). Если конструктору не передается аргумент **`alloc`**, используется аллокатор, созданный по умолчанию, но за исключением следующих случаев:

- конструктор копирования (4.1) создает контейнер, который хранит и использует копию аллокатора, который используется в аллокаторе **`x`**.

- конструктор перемещения (5.1) получает аллокатор от **`x`**.

Все элементы копируются, перемещаются или иным образом создаются путем вызова **`allocator_traits::construct`** с соответствующими аргументами.

```
// конструирование списков
#include <iostream>
#include <list>

int main () {

  // конструкторы используются в том же порядке, как описано выше:
  std::list<int> first;                                // пустой список int'ов
  std::list<int> second(4,100);                        // 4 int'a со значением 100
  std::list<int> third(second.begin(),second.end());
  std::list<int> fourth(third);                        // копия third

  // Для построения списка из массива можно сконструировать итератор
  int myints[] = {16, 2, 77, 29};
  std::list<int> fifth(myints, myints + sizeof(myints)/sizeof(int));
  std::cout << "The contents of fifth are: ";

  for (std::list<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

```
~list( );
```

Вызывает **allocator_traits::destroy( )** для каждого из содержащихся элементов и освобождает всю память, выделенную контейнеру, используя его аллокатор.

Присвоить содержимое (std::list::operator=)

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

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

il — объект типа **initializer_list**. Компилятор автоматически строит такие объекты из деклараторов списка инициализаторов.

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

Пример

```
// assignment operator with lists
#include <iostream>
#include <list>

int main () {

  std::list<int> first(3);      // список из 3-х int, инициализированных нулем
  std::list<int> second(5);     // список из 5-и int, инициализированных нулем

  second = first;
  first  = std::list<int>();    // r-value, перемещение

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

Вывод

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

Оба списка элементов типа **int** инициализируются последовательностями разного размера. Затем second присваивается **first**, поэтому теперь они оба равны и имеют размер 3.

После этого создается пустой объект-контейнер и присваивается контейнеру **first**, поэтому его размер становится равен 0.

Итераторы

`begin()` — указывает на первый элемент, инкремент двигает итератор к концу
`end()` — указывает за последним элементом (в пустое пространство)
`rbegin()` — указывает на последний элемент, инкремент двигает итератор к началу
`rend()` — указывает перед первым элементом (в пустое пространство)

front(), back() — возвращает ссылку на первый/последний элемент в списке

Возвращает ссылку на первый/последний элемент в контейнере списка.

В отличие от функции-члена **`list::begin()`**, которая возвращает итератор на элемент, функция **`list::front()`** возвращает прямую ссылку на первый элемент.

В отличие от функции-члена **`list::end()`**, которая возвращает итератор на пустое место за элементом, функция **`list::back()`** возвращает прямую ссылку на последний элемент.

Вызов этих функций для пустого контейнера вызывает неопределенное поведение.

Вставить элемент в начало/конец (list::push_front/list::push_back)

```
void push_front (const value_type& val); // или push_back
void push_front (value_type&& val);
```

Вставляет новый элемент в начало/конец списка, прямо перед/за его текущим первым/последним элементом. Во вставленный элемент копируется или перемещается содержимое **`val`**.

Вставка фактически увеличивает размер контейнера на единицу.

Удалить первый/последний элемент (list::pop_front/list::pop_back)

```
void pop_front();  // или pop_back
```

Удаляет первый/последний элемент из контейнера списка, фактически уменьшая его размер на единицу. Функции разрушают удаленный элемент.

Пример

```
// list::back
#include <iostream>
#include <list>

int main () {

  std::list<int> mylist;

  mylist.push_back(10);

  while (mylist.back() != 0) {
    mylist.push_back(mylist.back() - 1);
  }

  std::cout << "mylist contains:";
  for (auto it = mylist.begin(); it != mylist.end(); ++it)
    std::cout << ' ' << *it;

  std::cout << '\n';

  return 0;
}
```

Вывод

```
mylist contains: 10 9 8 7 6 5 4 3 2 1 0
```

Вставить элементы (`list::insert`)

Контейнер расширяется путем вставки новых элементов перед элементом в указанной позиции. Вставка фактически увеличивает размер списка на количество вставленных элементов.

В отличие от других стандартных последовательных контейнеров, объекты **`list`** и **`forward_list`** специально разработаны для эффективной вставки и удаления элементов в любой позиции, даже в середине последовательности.

Сколько элементов вставляется и какими значениями они инициализируются, определяется аргументами.

Один элемент (1)	`iterator insert(const_iterator position,` `const value_type& val);`

С заполнением (2)	`iterator insert(const_iterator position,`
	`size_type n,`
	`const value_type& val);`

Диапазон (3)	`template <class InputIterator>`
	`iterator insert(const_iterator position,`
	`InputIterator first,`
	`InputIterator last);`

Перемещение (4)	`iterator insert(const_iterator position,` `value_type&& val);`
Из списка иниц. (5)	`iterator insert(const_iterator position,` `initializer_list<value_type> il);`

Пример — вставка в список

```
#include <iostream>
#include <list>
#include <vector>

int main () {

  std::list<int> mylist;       // создаем список и
  std::list<int>::iterator it; // итератор для него

  // Наполним список чем-нибудь
  for (int i = 1; i <= 5; ++i) mylist.push_back(i); // 1 2 3 4 5

  it = mylist.begin();
  ++it;       // it указывает теперь на 2       // 1 2 3 4 5
                                                    ^
  mylist.insert(it, 10);                        // 1 10 2 3 4 5
  // it все еще указывает на 2                        ^
  mylist.insert(it, 2, 20);                     // 1 10 20 20 2 3 4 5
  --it;       // it указывает теперь на вторую 20          ^

  std::vector<int> myvector (2, 30);
  mylist.insert(it, myvector.begin(), myvector.end());
                                                // 1 10 20 30 30 20 2 3 4 5
                                                //               ^
```

```
  std::cout << "mylist contains:";
  for (it = mylist.begin(); it != mylist.end(); ++it) {
      std::cout << ' ' << *it;
  }
  std::cout << '\n';

  return 0;
}
```

Удалить элементы (`list::erase`)

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
```

Удаляет из контейнера списка либо один элемент, либо диапазон (**`[first, last)`**).

Удаление фактически уменьшает размер контейнера на количество удаленных элементов, которые разрушаются.

В отличие от других стандартных последовательностных контейнеров, объекты **`list`** и **`forward_list`** специально разработаны для эффективной вставки и удаления элементов в любой позиции, включая середину последовательности.

Пример — удаление из списка

```
#include <iostream>
#include <list>

int main () {

  std::list<int> mylist;              // создаем список и
  std::list<int>::iterator it1, it2; // два итератора для него

  // загрузим чем-нибудь список
  for (int i = 1; i < 10; ++i) mylist.push_back(i*10);

                                  // 10 20 30 40 50 60 70 80 90
  it1 = it2 = mylist.begin(); //  ^^
  advance(it2, 6);            //  ^                 ^
  ++it1;                      //     ^              ^

  it1 = mylist.erase(it1);    // 10 30 40 50 60 70 80 90
                              //     ^           ^

  it2 = mylist.erase(it2);    // 10 30 40 50 60 80 90
                              //     ^           ^
  ++it1;                      //        ^        ^
  --it2;                      //        ^     ^

  mylist.erase(it1, it2);     // 10 30 60 80 90          удаляются 40 и 50 [)
                              //        ^^
```

```
  std::cout << "mylist contains:";
  for (it1 = mylist.begin(); it1 != mylist.end(); ++it1) {
    std::cout << ' ' << *it1;
  }
  std::cout << '\n';

  return 0;
}
```

Вывод

```
mylist contains: 10 30 60 80 90
```

Реляционные операторы — функции-не-члены

Выполняют соответствующую операцию сравнения между контейнерами списка lhs и rhs.

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

Сравнение на равенство (**operator==**) выполняется снчала путем сравнения размеров, и если они совпадают, элементы сравниваются последовательно с использованием оператора **==**, останавливаясь при первом несовпадении (как при использовании алгоритма на равенство).

Сравнение меньше чем (**operator<**) ведет себя так, как если бы использовался алгоритм **lexicographic_compare**, который сравнивает элементы с помощью **operator<** последовательно взаимным образом (то есть выполняя проверки типа как **a < b**, так и **b < a**) и останавливаясь при первом появлении несовпадения.

В остальных операциях также используются операторы **==** и **<** для внутреннего сравнения элементов, как если бы имело место следующее

Операция	Эквивалентная операция
`a != b`	`!(a == b)`
`a > b`	`b < a`
`a <= b`	`!(b < a)`
`a >= b`	`!(a < b)`

Пример — сравнение списков

```
#include <iostream>
#include <list>

int main () {

  std::list<int> a = {10, 20, 30};
  std::list<int> b = {10, 20, 30};
  std::list<int> c = {30, 20, 10};

  if (a == b) std::cout << "a and b are equal\n";
  if (b != c) std::cout << "b and c are not equal\n";
  if (b < c)  std::cout << "b is less than c\n";
  if (c > b)  std::cout << "c is greater than b\n";
  if (a <= b) std::cout << "a is less than or equal to b\n";
  if (a >= b) std::cout << "a is greater than or equal to b\n";
  return 0;
}
```

Вывод

```
a and b are equal
b and c are not equal
b is less than c
c is greater than b
a is less than or equal to b
a is greater than or equal to b
```

Однонаправленный список (Forward list)

```
#include <forward_list>

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

Однонаправленные списки — это последовательностные контейнеры, которые позволяют выполнять операции вставки и удаления с постоянным временем в любом месте последовательности.

Однонаправленные списки реализованы как односвязные списки.

В односвязных списках каждый из содержащихся в них элементов может храниться в разных и несвязанных местах хранения. Порядок сохраняется за счет ассоциации с каждым элементом ссылки на *следующий элемент* в последовательности.

Основное различие конструкции между контейнером **`forward_list`** и контейнером **`list`** заключается в том, что **`forward_list`** хранит внутренне только ссылку на следующий элемент, в то время как **`list`** сохраняет две ссылки — одна указывает на следующий элемент, а другая — на предыдущий, что позволяет эффективно итерировать в обоих направлениях, но при этом требуется дополнительное пространство и немного больше времени на вставку и удаление элементов.

Таким образом, объекты **`forward_list`** более эффективны, чем объекты **`list`**, хотя их можно итерировать только вперед.

По сравнению с другими базовыми стандартными последовательными контейнерами (массив, вектор и двухсторонняя очередь), **forward_list** обычно лучше работает при вставке, извлечении и перемещении элементов в/из любой позиции в контейнере и, следовательно, также в алгоритмах, которые их интенсивно используют, например, в алгоритмах сортировки.

Главный недостаток **forward_lists** и **lists** по сравнению с этими другими контейнерами последовательностей состоит в том, что у них отсутствует прямой доступ к элементам по их положению.

Шаблон класса **forward_list** был разработан с учетом эффективности — по дизайну он так же эффективен, как и простой «самописный» односвязный список в стиле C.

forward_list фактически является единственным стандартным контейнером, в котором из соображений эффективности намеренно отсутствует функция-член **size()** — из-за того, что **forward_list** является связанным списком, наличие элемента, содержащего размер, занимающего постоянное время, требует от него внутреннего счетчика для размера (как это имеет место в **list**). Такое решение потребовало бы дополнительного места для хранения и сделало бы операции вставки и удаления немного менее эффективными.

Чтобы получить размер объекта **forward_list**, можно использовать алгоритм расстояния между началом списка и его концом, что является операцией, которая занимает линейное время.

Словарь (map)

Словари — это ассоциативные контейнеры, в которых хранятся элементы, сформированные комбинацией значения ключа и сопоставленного ему значения в определенном порядке.

Библиотека содержит шаблоны двух классов:

`map` — словарь

`multimap` — многоключевой словарь

Функции

`begin` — возвращает итератор на первый элемент в последовательности

`end` — возвращает итератор на последний элемент в последовательности

Словарь (map)

Словари — это ассоциативные контейнеры, в которых хранятся элементы, сформированные комбинацией значения ключа и сопоставленного ему значения в определенном порядке.

В словаре значения ключей обычно используются для сортировки и однозначной идентификации элементов, в то время как сопоставленные ключу значения хранят контент, связанный с этим ключом.

Тип ключа и тип сопоставленного значения могут различаться и группируются вместе в члене типа **value_type**, который является типом **pair**:

```
typedef pair<const Key, T> value_type;
```

Внутренне элементы в словаре всегда сортируются по ключу в соответствии с конкретным строгим критерием слабого упорядочения, который указывается его внутренним объектом сравнения.

При доступе к отдельным элементам по их ключу контейнеры **map** обычно медленнее, чем контейнеры **unordered_map**, но они допускают прямую итерацию в подмножествах, основанных на порядке словаря.

К отображенным значениям в словаре можно получить доступ напрямую по их соответствующему ключу с помощью оператора скобок (**operator[]**).

Словари обычно реализуются как деревья двоичного поиска.

Свойства контейнера

Ассоциативность

На элементы в ассоциативных контейнерах ссылаются по их ключу, а не по их абсолютному положению в контейнере.

Упорядоченность

Элементы в контейнере всегда следуют строгому порядку. Всем вставляемым элементам присваиваются позиции в этом порядке.

Отображение

Каждый элемент связывает ключ с отображаемым значением — ключи предназначены для идентификации элементов, основным содержанием которых является отображаемое значение.

Уникальные ключи

Никакие два элемента в контейнере не могут иметь эквивалентных ключей.

Распределение памяти

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

```
#include <map>

template < class Key,                                     // map::key_type
           class T,                                       // map::mapped_type
           class Compare = less<Key>,                     // map::key_compare
           class Alloc = allocator<pair<const Key,T> >    //
map::allocator_type
           > class map;
```

Параметры шаблона

Key — Тип ключа. Каждый элемент в словаре уникально идентифицируется своим значением ключа. Псевдоним как тип элемента **map::key_type**.

T — тип отображаемого значения. Каждый элемент словаря хранит некоторые данные в качестве отображаемого значения. Псевдоним как тип элемента **map::mapped_type**.

Compare — двоичный предикат, который принимает в качестве аргументов ключи двух элементов и возвращает логическое значение. Выражение **comp(a, b)**, где **comp** — объект этого типа, а **a** и **b** — ключевые значения, должно возвращать истину, если считается, что **a** идет перед **b** в строгом слабом порядке

Объект словаря использует это выражение, чтобы определить как порядок следования элементов в контейнере, так и эквивалентность двух ключей элементов (путем рефлексивного сравнения — они эквивалентны, если **!Comp(a, b) && !Comp(b, a))**.

Никакие два элемента в контейнере словаря не могут иметь эквивалентных ключей.

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

Псевдоним как член типа **`map::key_compare`**.

Alloc

Тип объекта аллокатора, используемого для определения модели управления выделением памяти. По умолчанию используется шаблон класса аллокатора, который определяет простейшую модель распределения памяти и не зависит от значения.

Псевдоним как тип элемента **`map::allocator_type`**.

Пример «Коструирование словаря»

```
#include <iostream>
#include <map>

bool fncomp (char lhs, char rhs) {return lhs < rhs;}

struct classcomp {
  bool operator() (const char& lhs, const char& rhs) const
  {return lhs < rhs;}
};

int main () {

  std::map<char, int> first;
  first['a'] = 10;
  first['b'] = 30;
  first['c'] = 50;
  first['d'] = 70;

  std::map<char, int> second(first.begin(),first.end());
  std::map<char, int> third (second);
  std::map<char, int, classcomp> fourth;     // class as Compare
  bool(*fn_pt)(char, char) = fncomp;
  std::map<char, int, bool(*)(char, char)> fifth (fn_pt); // funcptr as Compare

  return 0;
}
```

Неупорядоченный словарь (unordered map)

Неупорядоченные словари — это ассоциативные контейнеры, в которых хранятся элементы, сформированные комбинацией ключевого значения и значения, сопоставленного ему. Они позволяют быстро извлекать отдельные элементы на основе их ключей.

```
#include <map>

template <class Key,                                    // multimap::key_type
          class T,                                      // multimap::mapped_type
          class Compare = less<Key>,                    // multimap::key_compare
          class Alloc = allocator<pair<const Key, T> > // map::allocator_type
          > class multimap;
```

Свойства контейнера

Ассоциативность
Упорядоченность
Отображение
Несколько эквивалентных ключей
Несколько элементов в контейнере могут иметь эквивалентные ключи.
Распределение памяти

Библиотека итераторов

Предоставляет возможности для упорядоченного доступа к элементам контейнеров.

Определения итератора

Итератор — это любой объект, который, указывая на некоторый элемент в некотором диапазоне элементов, например, массива или контейнера, имеет возможность перебирать элементы этого диапазона, используя набор операторов, таких, как оператор инкремента (**++**) и оператор разыменования (*****).

Наиболее очевидной формой итератора является указатель — указатель может указывать на элементы в массиве и может выполнять итерацию по ним с помощью оператора инкремента (**++**).

Но возможны и другие виды итераторов. Например, каждый тип контейнера (например, список) имеет определенный тип итератора, предназначенный для перебора его элементов.

Следует заметить, что, хотя указатель является формой итератора, не все итераторы обладают той же функциональностью, что и указатели. В зависимости от свойств, поддерживаемых итераторами, они делятся на пять различных категорий.

Категории итераторов

Итераторы подразделяются на пять категорий в зависимости от реализуемой ими функциональности:

1) ввода (Input);
2) вывода (Output);
3) однонаправленные (Forward);
4) двунаправленные (Bidirectional);
5) с произвольным доступом (Random Access).

Итераторы ввода и вывода являются наиболее ограниченными типами итераторов — они могут выполнять только последовательные однопроходные операции ввода или вывода.

Однонаправленные итераторы имеют все функции итераторов ввода и, если они не являются константными итераторами, имеют также функциональные возможности итераторов вывода.

Хотя они ограничены одним направлением для прохода по диапазону (только вперед).

Все стандартные контейнеры поддерживают как минимум итераторы данного типа.

Двунаправленные итераторы подобны однонаправленным итераторам, но также могут выполнять проход и в обратном направлении.

Итераторы с произвольным доступом реализуют все функции двунаправленных итераторов, а также имеют возможность доступа к диапазонам непоследовательно — к дальним (относительно текущего положения) элементам можно получить доступ напрямую, применяя значение смещения к итератору без прохода по всем элементам между ними. Эти итераторы имеют функциональность, аналогичную стандартным указателя (указатели являются итераторами этой категории).

Свойства каждой категории итераторов::

<table>
<tr><th colspan="4">Категория</th><th>Свойства</th><th>Выражение</th></tr>
<tr><td colspan="4" rowspan="2">Все категории</td><td>конструктор копирования
присвоение копированием
деструкторы</td><td>X b(a);
b = a;</td></tr>
<tr><td>может инкрементироваться</td><td>++ a
a++</td></tr>
<tr><td rowspan="6">Произволь-ного доступа</td><td rowspan="6">Дву-направлен-ные</td><td rowspan="5">Одно-направ-ленные</td><td rowspan="2">Ввода</td><td>поддерживает сравнение на равенство/неравенство</td><td>a == b
a != b</td></tr>
<tr><td>может быть разыменован как rvalue</td><td>*a
a->m</td></tr>
<tr><td>Вывода</td><td>может быть разыменован как lvalue (только для изменяе-мых типов итераторов)</td><td>*a = t
*a++ = t</td></tr>
<tr><td rowspan="2"></td><td>конструктор по умолчанию</td><td>X a;
X()</td></tr>
<tr><td>многопроходный — ни разыме-нование, ни инкремент не влияют на разыменование</td><td>{ b=a; *a++;
*b; }</td></tr>
<tr><td colspan="2"></td><td>может декрементироваться</td><td>--a
a--
*a--</td></tr>
</table>

		поддерживают арифметические операторы **+** и **-**	**a + n** **n + a** **a - n** **a - b**
		Поддерживают сравнение на неравенство (**<**, **>**, **<=** и **>=**) между итераторами	**a < b** **a > b** **a <= b** **a >= b**
		Поддерживают составные операции присваивания **+=** и **-=**	**a += n** **a -= n**
		Поддерживает оператор разыменования для смещения (**[ ]**)	**a[n]**

Функциональные шаблоны

Итераторные операции:

begin() — возвращает итератор, указывающий на первый элемент в последовательности
end() — возвращает итератор, указывающий на последний элемент в последовательности

Конструкторы

из контейнера (1)

```
template <class Container>
auto begin(Container& cont)->decltype(cont.begin());

template <class Container>
auto begin(const Container& cont)->decltype(cont.begin());
```

из массива(2)

```
template <class T, size_t N>   constexpr T* begin (T(&arr)
[N]) noexcept;
```

prev() — возвращает итератор, указывающий на предыдущий элемент в последовательности
next() — возвращает итератор, указывающий на следующий элемент в последовательности

advance() — перемещает итератор на **n** элементов.

```
#include <iterator>

template <class InputIterator, class Distance>
void advance(InputIterator& it, Distance n);
```

Если данная операция относится к итератору с произвольным доступом, функция использует **operator+** или **operator-** только один раз. В противном случае функция многократно использует оператор инкрементирования/декрементирования (**operator++** или **operator--**) до тех пор, пока не будет выполнено перемещение на **n** элементов.

it — итератор, подлежащий продвижению. **InputIterator** должен быть как минимум итератором ввода.

n — количество позиций элемента для продвижения.

Количество позиций для продвижения должно быть отрицательным только для итераторов с произвольным доступом и двунаправленных итераторов.

Distance должно быть числовым типом, способным представлять расстояния между итераторами этого типа.

distance() — возвращает расстояние (в элементах) между итераторами.

```
#include <iterator>

template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

Вычисляет количество элементов между **first** и **last**.

Если это итератор с произвольным доступом, функция использует оператор **-** (минус) для его вычисления. В противном случае функция кратно использует оператор инкрементирования (**operator++**).

Генераторы итераторов

back_inserter — создать итератор вставки в конец
front_inserter — создать итератор вставки в начало
inserter — создать итератор вставки
make_move_iterator — построить итератор перемещения

Шаблоны классов

iterator — базовый класс итератора
iterator_traits — свойства итератора

Шаблоны классов предопределенных итераторов

reverse_iterator — обратный итератор
move_iterator — адаптирует итератор так, чтобы тот при разыменовании производил ссылки на rvalue
back_insert_iterator — итератор вставки в конец
front_insert_iterator — итератор вставки в начало
insert_iterator — итератор вставки
istream_iterator — итератор входного потока istream
ostream_iterator — итератор выходного потока ostream
istreambuf_iterator — итератор буфера входного потока
ostreambuf_iterator — итератор буфера выходного потока

Классы тегов категорий (пустые классы для определения катагрии итератора)

`input_iterator_tag` – категория итератора ввода
`output_iterator_tag` – категория итератора вывода
`forward_iterator_tag` – категория однонаправленного итератора
`bidirectional_iterator_tag` – категория двунаправленного итератора
`random_access_iterator_tag` – категория итератора произвольного доступа

Представляют собой следующие структуры:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag {};
struct bidirectional_iterator_tag {};
struct random_access_iterator_tag {};
```

Пары (std::pair)

```
#include <utility>

template <class T1, class T2> struct pair;
```

Этот класс объединяет пару значений, которые могут быть разных типов (**T1** и **T2**).
Доступ к отдельным значениям можно получить через его публичные члены **first** и **second**.
Пары — это частный случай кортежа.
T1 — Тип элемента **first**, псевдоним **first_type**.
T2 — Тип элемента **second**, псевдоним **second_type**.