3.24 Наличие/отсутствие и алгоритмическая сложность различных операций в контейнерах

Первые три контейнера - последовательные (sequence containers), вторые - associative containers.

Container	indexating [*pos*]	push_back	insert(it)	erase(it)	find	iter
vector	$O(1)$	$O(1)$ amort	$O(n)$	$O(n)$	-	RA
deque	$O(1)$	$O(1)$	$O(n)$	$O(n)$	-	RA
list (forward_list)	-	$O(1)$	$O(1)$	$O(1)$	-	BI (FI)
set/map	$O(logn)$	-	$O(logn)$	$O(logn)$	$O(logn)$	BI
unordered_set/map	$O(1)$*	-	$O(1)$*	$O(1)$*	$O(1)$*	FI

Пояснение: $O(1)$* - это $O(1)$ среднее, т.е такое, что можно подборать набор входных данных что операции будут работать за линию, но в среднем операции работатют за $O(1)$, так как хеширование реализованно методом цепочек.
Замечание: Почему в deque push_back за $O(1)$? Память двухсторонней очереди автоматически расширяется и сокращается по мере необходимости. Расширение двухсторонней очереди дешевле, чем расширение std::vector, поскольку оно не включает в себя копирование существующих элементов в новое расположение в памяти. С другой стороны, двухсторонние очереди обычно имеют большую минимальную стоимость памяти; двухсторонняя очередь, содержащая только один элемент, должна выделить свой полный внутренний массив (например из 8 ячеек).
!!! Заметим, что у **forward_list, list** и **deque** есть метод `push_front`, который работает за $O(1)$.

3.25 Основные методы контейнера vector и их правильное применение

reference operator[](size_type pos); Возвращает ссылку на элемент по индексу pos. Проверка на выход за границы не выполняется. В методе **at(pos)** эта проверка есть.

void push_back(const T& value); Добавляет данный элемент value до конца контейнера. Если новый size() больше, чем capacity(), Все итераторы и указатели становятся нерабочими. В противном случае, все они остаются в рабочем состоянии. При этом value должен быть CopyInsertable или MoveInsertable

void pop_back(); Удаляет последний элемент контейнера. Итераторы и указатели остаются в рабочем состоянии.

size_type size() const; Возвращает количество элементов в контейнере, т.е.
`std::distance(begin(), end())`

void resize(size_type count, const value_type& value); Изменяет размер контейнера, чтобы содержать count элементы. Если текущий размер меньше, чем count, дополнительные элементы добавляются и инициализируется value (или конструируются по умолчанию). Если текущий размер больше count, контейнер сводится к ее первые элементы count.

size_type capacity() const; Возвращает количество элементов, для которого сейчас выделена память контейнером.

void reserve(size_type size); Задает емкость контейнера по крайней мере, size. Новая память выделяется при необходимости.

3.26 Основные методы контейнера map и их правильное применение

T& operator[](const Key& key); Вставляет новый элемент в контейнере с помощью key в качестве ключа и конструктором по умолчанию отображенное значения и возвращает ссылку на вновь построенное значение. Если элемент с ключом key уже существует, вставка не выполняется, и возвращается ссылка на это значение. Итераторы и указатели остаются в рабочем состоянии.

T& at(const Key& key); Возвращает ссылку на соответствующее значение элемента с ключом, эквивалентным key. Если такого элемента не существует, бросает исключение типа `std::out_of_range`.

iterator find(const Key& key); Находит элемент с ключом key и возвращает итератор на этот элемент. Если такой элемент не найден, то возвращается end()

std::pair<iterator,bool> insert(const value_type& value); Вставляет элемент value в контейнер, если контейнер еще не содержит элемент с эквивалентным ключом. Возвращает пару из итератора на вставленный элемент (или на тот, который помешал вставке), и bool, указывающий, была ли вставка.

void insert(InputIt first, InputIt last); Вставляет элементы из диапазона [first, last). При этом диапозон задан Input Iterator'ами. Ничего не возвращает.

void insert(std::initializer_list<value_type> ilist); Вставляет список инициализации `std::initializer_list<value_type>` и ничего не возвращает.

iterator erase(const_iterator position); Удаляет из контейнера элемент на позиции pos. Указатели и итераторы к удалённым элементам становятся недействительными. Другие итераторы и указатели остаются без изменений. Возвращает итератор, следующего элемента за удаленным элементом.

iterator erase(const_iterator first, const_iterator last); Удаляет элементы из контейнера в диапазоне [first; last). Указатели и итераторы к удалённым элементам становятся недействительными. Другие итераторы и указатели остаются без изменений. Возвращает итератор, следующего элемента за удаленным элементом.

iterator lower_bound(const Key& key); Возвращает итератор, указывающий на первый элемент, который является *не меньше, чем key*. Если такой элемент не найден, то возвращается end()

iterator upper_bound(const Key& key); Возвращает итератор, указывающий на первый элемент, который является *не больше, чем key*. Если такой элемент не найден, то возвращается end()

3.27 Категории итераторов

Input Iterator: позволяет лишь раз пройтись по последовательности.

- Копирование, присваивание.
- Операции сравнения на равенство `it1 == it2` и `it1 != it2`
- Инкремент: `++it` и `it++`.
- Разыменование для чтения: `*it` и `it->m`,
  при этом **запрещена** запись: `*it = value;`

Пример входного итератора - это итератор чтения из потока: `std::istream_iterator`

Важно: Если его скопировать и пройтись 2ой раз, не гарантируется что получим ту же последовательность.

Forward Iterator: однонаправленные итераторы, могут перемещаться только в одну сторону на 1 позицию, перемещение в обратную сторону занимает продолжительное время.

- Все операции **Input Iterator**
- Разыменование для записи: `*it = value;` и `*it++ = value;`
- Требование *многопроходности*: если `it1 == it2`, то `++it1 == ++it2`, т.е. итератор можно копировать, и обходить им последовательность много раз.

Например, итераторы **forward_list, unordered_map, unordered_set**

Bidirectional Iterator двунаправленные итераторы, могут быстро перемещаться на одну позицию как вперед, так и назад.

- Все операции **Forward Iterator**
- Декремент: `--it, it--, *it--`

Например, итераторы **list, map, set**

Random-access итераторы: могут перемещаться быстро на любую позицию в контейнере.

- Все операции **Bidirectional Iterator**
- Операции сравнения: `it1 < it2, it1 > it2, it1 <= it2, it1 >= it2`
- Сложение/вычитание с числом: `it + n, it += n, it - n, it -= n`
- Разность итераторов: `it2 - it1`
- Индексирование: `it[n]`

Например, итераторы **vector, deque**

Алгоритмы из STL, реализованные с помощью итераторов :

- InputIterator - find()
- ForwardIterator - binary_search() т.е. определяет, находится ли элемент в некотором отсортированном диапазоне
- BidirectionalIterator - next_permutation() т.е. генерирует ближайшую следующую перестановку диапазона элементов
- Random-Access Iterator - sort()

3.28 Использование итераторов для прохода по любому из контейнеров от начала до конца. Использование range-based for.

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};

//access by operator []
for(int i = 0; i < v.size(); ++i)
    std::cout << v[i];

//access by iterator
for(std::vector<int>::iterator it = v.begin(); it < v.end(); ++it)
    std::cout << *it;
```

Начиная с C++11 есть синтаксис **range-based for:**

1. Для **просмотра значений** используется такой синтаксис:

```
for (const auto& elem : container)    // capture by const reference
```

   - Если объекты *дешевые для копирования* (например, int, double, etc), то можно использовать немного упрощенную форму:

```
for (auto elem : container)    // capture by value
```

     В цикле используется локальная копия объекта из контейнера.

2. Для **изменения значений** в контейнере используется такой синтаксис:

```
for (auto& elem : container)    // capture by (non-const) reference
```

   - Если контейнер использует так называемые *proxy iterators* (например, std::vector<bool>), то нужно использовать такую форму:

```
for (auto&& elem : container)    // capture by rvalue reference
```

     Объясним почему именно так. Если запустить код, который инвертирует значения булевского массива, то мы получим CE:

```
vector<bool> v = {true, false, false, true};
for (auto& x : v)  // CE
    x = !x;
```

     Проблема в том, что std::vector специальизированный для bool реализован с оптимизацией памяти (то есть каждое логическое значение лежит в 1 бите $\Rightarrow$ 8 логический значений в одном блоке вектора). Так как невозможно вернуть ссылку на каждый бит, вектор использует *proxy iterators*, который возвращает по значению временный объект, а именно proxy class convertible to bool. Поэтому корректная реализация такая:

```
vector<bool> v = {true, false, false, true};
for (auto&& x : v)  // OK
    x = !x;
```

Вывод – универсальный подход такой:

```
// For observing the elements, use:
for (const auto& elem : container)

// For modifying the elements in place, use:
for (auto&& elem : container)
```