C++ и ООП, Стандартная библиотека шаблонов. Справочник.

мехмат, прикладная математика, IV курс

Содержание

1 Контейнеры

1.1 Общие свойства всех контейнеров

Для любого контейнера `template <class T> class X;` выполняются следующие требования:

- Типы:
  - `X::value_type` — тип хранимых значений (`T`).
  - `X::reference` — l-выражение типа хранимых значений (`T &`).
  - `X::const_reference` — l-выражение на константу типа хранимых значений (`const T &`).
  - `X::iterator` — итератор, указывающий на `T`.
  - `X::const_iterator` — итератор, указывающий на `const T`.
  - `X::difference_type` — целый знаковый тип, хранящий разность в элементах для двух значений типа `iterator` или `const_iterator`.
  - `X::size_type` — целый беззнаковый тип, хранящий количество элементов в контейнере.
- Размещение / освобождение:
  - Конструктор по умолчанию — создаёт пустой контейнер.
  - Семантика значений.
  - Деструктор.
  - `void swap(X <T> &rX);` — обмен содержимым.
  - `void swap(X <T> &rX1, X <T> &rX2);` — глобальная функция.

- Получение размера:
  - `size_type size() const;` — возвращает количество элементов.
  - `size_type max_size() const;` — возвращает максимально возможное количество элементов.
  - `bool empty() const;` — проверка на пустоту.
- Сравнение (операции `==`, `!=`, `>=`, и т. д.) — в лексикографическом порядке (ленивая конкретизация шаблонов).
- Доступ к элементам:
  - (`const_`)`iterator begin()` (`const`); — получение указателя на первый элемент.
  - (`const_`)`iterator end()` (`const`); — получение указателя *за* последним элементом.

1.2 Общие свойства обратимых контейнеров

- Типы:
  - `X::`(`const_`)`reverse_iterator` — итератор, указывающий на (`const`) `T`.
- Доступ к элементам:
  - (`const_`)`reverse_iterator rbegin()` (`const`) — получение указателя на первый элемент.
  - (`const_`)`reverse_iterator rend()` (`const`) — получение указателя *за* последним элементом.

1.3 Общие свойства последовательностей

1.3.1 Общие свойства всех последовательностей

- Размещение / освобождение:
  - `X(size_type n, const T &rcT = T());` — создаёт последовательность из `n` элементов, инициализируя их значением `rcT`.
  - `template <class InpIter> X(InpIter first, InpIter last);` — создаёт последовательность, заполняя её значениями по указателям в диапазоне [`first, last`).
- Вставка / удаление:
  - `iterator insert(iterator pos, const T &rcT = T());` — вставляет копию `rcT` перед `pos`, возвращает его позицию.
  - `void insert(iterator pos, size_type n, const T &rcT);` — вставляет `n` копий `rcT` перед `pos`.

- **template** <**class** InpIter>

  void insert(iterator pos, InpIter first, InpIter last); — вставляет копии значений по указателям в диапазоне [first, last) перед pos (first и last *не указывают* на позиции в самой последовательности).
- **void** erase(iterator pos); — удаляет элемент в позиции pos.
- **void** erase(iterator first, iterator last); — удаляет элементы в диапазоне [first, last).
- **void** clear(); — очищает последовательность.

1.3.2 Дополнительные свойства некоторых последовательностей

- Доступ к элементам:
  - (const_)reference front() (**const**); — возвращает ссылку на первый элемент последовательности (vector, list, deque).
  - (const_)reference back() (**const**); — возвращает ссылку на последний элемент последовательности (vector, list, deque).
  - (const_)reference **operator** [] (size_type n) (**const**); — возвращает ссылку на n-й элемент последовательности (vector, deque).
  - (const_)reference at(size_type n) (**const**); — как **operator** [] (), но с контролем выхода за границы.
- Вставка / удаление:
  - **void** push_front(**const** T &rcT); — вставляет копию rcT перед началом последовательности (list, deque).
  - **void** push_back(**const** T &rcT); — вставляет копию rcT в конец последовательности (vector, list, deque).
  - **void** pop_front(); — удаляет элемент в начале последовательности (list, deque).
  - **void** pop_back(); — удаляет элемент из конца последовательности (vector, list, deque).
  - **template** <**class** InpIter>

    void assign(InpIter first, InpIter last); — заполняет элементы значениями из диапазона [first, last). Размер становится равным количеству элементов в диапазоне (vector, list, deque).
  - **void** assign(size_type n, **const** value_type &rcT); — заполняет элементы n копиями значения rcT. Размер становится равным n (vector, list, deque).

1.3.3 vector

Основные методы:

- Получение размера:
  - `size_type capacity() const;` — возвращает максимально возможное количество элементов на данный момент, не требующее перераспределения памяти.
- Размещение / освобождение:
  - `void reserve(size_type n);` — резервирование заданного размера.

В библиотеке существует явная специализация `vector <bool>`, позволяющая хранить флаги побитово. Метод `flip()` (отрицание) может применяться как ко всему вектору, так и к отдельным элементам.

Тип `const_reference` описан как `bool`.

1.3.4 list

Основные методы:

- Модифицирующие операции:
  - `void splice(iterator pos, list <T> &rList);` — переносит всё содержимое из списка `rList` (он становится пустым) в текущий, перед позицией `pos`.
  - `void splice(iterator pos, list <T> &rList, iterator i);` — переносит элемент из списка `rList` в текущий (может быть тем же самым) из позиции `i` в позицию `pos`.
  - `void splice(`
    `iterator pos, list <T> &rList, iterator first, iterator last);` — переносит все элементы из диапазона [`first`, `last`) перед позицией `pos` (должна быть вне диапазона).
  - `void remove(const T &rcT);` — удаляет все элементы с заданным значением.
  - `template <class Predicate> void remove_if(Predicate pred);` — удаляет все элементы по заданному условию.
  - `void unique();` — удаляет все подряд идущие одинаковые элементы, кроме первого из них.
  - `template <class BinaryPredicate>`
    `void unique(BinaryPredicate pred);` — аналогично, но «одинаковость» определяется бинарным отношением `pred`.
  - `void merge(list <T> &rList);` — сливает текущий упорядоченный список с упорядоченным списком `rList`, который становится в результате пустым.

- `template <class Compare> void merge(list <T> &rList, Compare comp);` — аналогично, но отношение порядка определяется функцией `comp`.
- `void reverse();` — переставляет элементы в обратном порядке.
- `void sort();` — сортирует список.
- `template <class Compare> void sort(Compare comp);` — аналогично, но отношение порядка определяется функцией `comp`.

1.3.5 deque

Сокращение от *"Double-ended queue"*.

1.4 Адаптеры контейнеров

Общие свойства:

- Типы:
  - `value_type` — тип хранимых данных.
  - `size_type` — тип размера.
  - `container` — тип контейнера.
- Получение размера:
  - `size_type size() const;` — возвращает количество элементов.
  - `bool empty() const;` — проверка на пустоту.

1.4.1 stack

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

На основе любой последовательности, поддерживающей операции `back()`, `push_back()` и `pop_back()`. В частности, `vector`, `list` и `deque`.

Свойства:

- Размещение / освобождение:
  - `stack(const Container &rcC = Container());` — конструктор по умолчанию.
- Сравнение (операции `==`, `!=`, `>=`, и т. д.) — в лексикографическом порядке.
- Доступ к элементам:
  - (`const`) `value_type &top()` (`const`); — возвращает ссылку на элемент в вершине стека.

- Вставка / удаление:
  - **void** push(**const** value_type &rcT); — помещает элемент в вершину стека.
  - **void** pop(); — удаляет элемент из вершины стека.

1.4.2 queue

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

На основе любой последовательности, поддерживающей операции front(), back(), push_back() и pop_front(). В частности, list и deque.

Свойства:

- Размещение / освобождение:
  - queue(**const** Container &rcC = Container()); — конструктор по умолчанию.
- Сравнение (операции ==, !=, >=, и т. д.) — в лексикографическом порядке.
- Доступ к элементам:
  - (**const**) value_type &front() (**const**); — возвращает ссылку на первый элемент в очереди.
  - (**const**) value_type &back() (**const**); — возвращает ссылку на последний элемент в очереди.
- Вставка / удаление:
  - **void** push(**const** value_type &rcT); — помещает элемент в конец очереди.
  - **void** pop(); — удаляет элемент из начала очереди.

1.4.3 priority_queue

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

Элементы сортируются автоматически согласно критерию сортировки.

На основе любой последовательности с произвольным доступом, поддерживающей операции front(), push_back() и pop_front(). В частности, vector и deque. Для работы вызывает алгоритмы make_heap() и т. д.

Свойства:

- Размещение / освобождение:
  - `priority_queue(`
    `const Compare rcComp = Compare(),`
    `const Container &rcC = Container());` — конструктор по умолчанию.
  - `template <class InpIter>`
    `priority_queue(`
    `InpIter first, InpIter last,`
    `const Compare rcComp = Compare(),`
    `const Container &rcC = Container());` — заполняет очередь значениями из диапазона [`first, last`).
- Доступ к элементам:
  - (`const`) `value_type &top()` (`const`); — возвращает ссылку на первый элемент в очереди.
- Вставка / удаление:
  - `void push(const value_type &rcT);` — помещает элемент в очередь согласно приоритету.
  - `void pop();` — удаляет элемент из начала очереди.

1.5 Свойства ассоциативных контейнеров

Являются двунаправленными контейнерами.

1.5.1 Свойства всех ассоциативных контейнеров

Пусть есть контейнер «множество» или «отображение»:

```
template < class Key, class Compare = less <Key> > class X;
или
template < class Key, class T, class Compare = less <Key> > class X;
```

Для отображений в качестве хранящихся значений используются пары «ключ — значение»: шаблонные структуры с 2 полями заданных типов — `std::pair`:

Функция-шаблон `std::make_pair(myKey, myVal)` может использоваться для конструирования пары:.

- Типы:
  - `X::key_type` — тип ключа (`Key`) (тип `value_type` для множеств совпадает с `key_type`, для отображений — с `pair <const Key, T>`).
  - `X::key_compare` — функция сравнения ключей (`Compare`).

  - `X::value_compare` — объект сравнения двух значений.

- Размещение / освобождение:
  - `X(const Compare &rcComp = Compare());` — инициализирует пустой контейнер с заданным объектом сравнения.
  - `template <class InpIter>`
    `X(InpIter first, InpIter last, const Compare &rcComp = Compare());` — заполняет контейнер значениями.

- Доступ к объектам сравнения:
  - `key_compare key_comp() const;` — доступ к созданному объекту сравнения ключей.
  - `value_compare value_comp() const;` — доступ к объекту сравнения значений, созданному из объекта сравнения.

- Вставка / удаление:
  - `pair <iterator, bool> insert(const value_type &rcV);` — (только для контейнеров с уникальными ключами) вставляет элемент, только если такого ключа не было. Возвращает позицию элемента с указанным ключом и флаг — произошла ли вставка.
  - `iterator insert(const value_type &rcV);` — (только для контейнеров с повторяющимися ключами) вставляет элемент и возвращает позицию вставки.
  - `iterator insert(iterator pos, const value_type &rcX);` — вставляет элемент (для контейнеров с уникальными ключами — только если нет такого ключа), используя `pos` в качестве подсказки — откуда искать место для вставки.
  - `template <class InpIter>`
    `void insert(InpIter first, InpIter last);` — вставляет элементы из другого контейнера с позициями в заданном диапазоне.
  - `void erase(iterator pos);` — удаляет элемент в заданной позиции.
  - `size_type erase(const key_type &rcKey);` — удаляет все элементы с заданным ключом, возвращает их количество.
  - `void erase(iterator first, iterator last);` — удаляет все элементы в диапазоне.
  - `void clear();` — очищает контейнер.

- Доступ к элементам:
  - `(const_)iterator find(const key_type &rcKey) (const);` — находит позицию элемента по заданному ключу, возвращает `end()`, если он не найден.

- `size_type count(const key_type &rcKey) const;` — возвращает количество элементов с заданным ключом.
- `(const_)iterator lower_bound(const key_type &rcKey) (const);` — находит позицию первого элемента с ключом, не меньшим заданного.
- `(const_)iterator upper_bound(const key_type &rcKey) (const);` — находит позицию первого элемента с ключом, бóльшим заданного.
- `pair <(const_)iterator, (const_)iterator> equal_range(`
  `const key_type &rcKey) (const);` — эквивалентна двум предыдущим вызовам.

1.5.2 map

- Доступ к элементам:
  - `reference operator [] (const key_type &rcT);` — позволяет работать с отображением как с ассоциативным массивом.

1.6 Специализированный контейнер bitset

Битовое множество: `template <size_t NumBits> class bitset;`

- Размещение / освобождение:
  - `bitset();` — конструктор по умолчанию.
  - `bitset(unsigned long ulVal);` — инициализирует контейнер битами из `ulVal` (оставшиеся — нулями).
  - `template <class TChar> bitset(`
    `const basic_string <TChar> &rcStr,`
    `typename basic_string <TChar>::size_type nPos = 0,`
    `typename basic_string <TChar>::size_type nNum =`
    `basic_string <TChar>::npos);` — инициализирует контейнер битами из строки `rcStr` (должна быть в виде `"1101001"`).
- Получение размера:
  - `size_t size() const;` — возвращает размер (значение шаблонного параметра `N`).
- Доступ к элементам:
  - `reference operator [] (size_t nPos);` — доступ к отдельному биту.
  - `unsigned long to_ulong() const;` — преобразование к беззнаковому целому.
  - `template <class TChar>`
    `basic_string <TChar> to_string() const;` — преобразование к строке.

- `size_t count() const;` — возвращает количество установленных бит.
- `bool any() const;` — проверяет, есть ли установленные биты.
- `bool none() const;` — проверяет, все ли биты сброшены.
- `bool test(size_t n) const;` — проверяет, установлен ли `n`-й бит.
- `bitset <N> &(re)set();` — устанавливает (сбрасывает) все биты.
- `bitset <N> &(re)set(size_t n);` — устанавливает (сбрасывает) `n`-й бит.
- `bitset <N> &set(size_t n, int nVal);` — устанавливает `n`-й бит в соответствии с `nVal`.
- `bitset <N> &flip();` — инвертирует все биты.
- `bitset <N> &flip(size_t n);` — инвертирует `n`-й бит.
- Перегруженные побитовые операции: `~`, `&`, `&=`, `<<`, `<<=`, и т. д.

- Ввод / вывод в текстовый поток: перегруженные операции `<<` и `>>`.

Возвращаемая ссылка поддерживает операции `flip()` и `~`.

1.7 Компонент `basic_string`

Предназначен для замены C-строк. Существует две специализации: `string` и `wstring`.

Обладает всеми свойствами векторов.

Свойства для заданного класса `basic_string <TChar>`:

- Типы:
  - `basic_string <TChar>::pointer` — указатель на тип хранимых значений (`TChar *`).
  - `basic_string <TChar>::const_pointer` — указатель на константный тип хранимых значений (`const TChar *`).
- Константы:
  - `static const size_type npos;` — специальное значение, возвращаемое в случае, когда поиск завершился неуспешно, или передаваемое в качестве параметра «количество символов», означающее «все символы до конца строки» (то же самое, что и передача значения, большего количества символов до конца, но значение начальной позиции должно лежать в диапазоне).
- Получение размера:
  - `size_type length() const;` — то же самое, что и `size()`. В отличие от C-строк символы `'\0'` внутри строки не означают её конца (следовательно, строки могут хранить содержимое двоичных файлов).

- Размещение / освобождение:
  - `basic_string(`
    `const basic_string &, size_type pos = 0, size_type n = npos);` — инициализирует строку максимум `n` символами из другой строки, начиная с позиции `pos`.
  - `basic_string(const TChar *);`
    `basic_string(const TChar *, size_type n);` — инициализируют строку символами из массива точной длины `n` (может содержать нулевые символы посередине) или до первого нулевого символа (первый вариант).
  - `basic_string(size_type n, TChar c);` — инициализирует строку, заполняя её `n` копиями символа `c`.
  - Перегруженные операции присваивания для типов `const TChar *` и `TChar`.
  - `void resize(size_type n);`
    `void resize(size_type n, TChar c);` — меняют размер строки, заполняя недостающие элементы справа символом `c` или нулевым символом.
  - `void reserve(size_type n = 0);` — меняет резерв в соответствии с параметром `n` (можно уменьшать, однако это не гарантируется — зависит от реализации). В частности, при `n == 0` делает занимаемое место в точности достаточным для хранения текущей строки. Стандартом гарантируется, что резерв может уменьшаться *только* в результате вызова `reserve()`.
- Сравнение:
  - Перегруженные операции `==`, `!=`, `>=`, и т. д., один из операндов может иметь тип `const TChar *`.
  - `int compare(const basic_string &) const;`
    `int compare(`
    `size_type pos1, size_type n1, const basic_string &) const;`
    `int compare(`
    `size_type pos1, size_type n1, const basic_string &,`
    `size_type pos2, size_type n2) const;`
    `int compare(`
    `size_type pos1, size_type n1,`
    `const TChar *, size_type n2 = npos) const;` — возвращают -1, 0, или 1.
- Ввод / вывод:

- Перегруженная операция >>: пропускаются ведущие пробельные символы, если в потоке установлен флаг `skipws`. Затем считываются символы до первого пробельного (не включая его), до достижения ширины потока `basic_istream <TChar>::width()`, либо до достижения `max_size()`. Ширина потока устанавливается в 0.
- Перегруженная операция <<: в поток записывается максимум `width()` символов.
- ```
  template <class TChar> basic_istream <TChar> &
    getline(basic_istream <TChar> &, basic_string <TChar> &);
  template <class TChar> basic_istream <TChar> &
    getline(
      basic_istream <TChar> &, basic_string <TChar> &);
      TChar delim);
  ```
  — считывают из входного потока строку до первого символа `delim` (по умолчанию — символ перевода строки) или до достижения ширины потока. Ширина потока устанавливается в 0.

- Доступ к элементам:
  - `const TChar *c_str() const;` — возвращает содержимое строки в виде C-строки (с добавлением символа `'\0'` в конец, содержимое строки отражается верно до первого вызова неконстантного метода строки).
  - `const TChar *data() const;` — возвращает содержимое строки в виде C-массива (без добавления символа `'\0'` в конец).
  - `size_type copy(TChar *, size_type n, size_type pos = 0) const;` — копирует содержимое строки в массив, предоставляемый вызывающей стороной. Символ `'\0'` не добавляется.
  - ```
    basic_string &substr(
      size_type pos = 0, size_type n = npos) const;
    ```
    — возвращает подстроку.

- Вставка / удаление:
  - Конкатенация — перегруженная операция +, один из операндов может иметь тип `const TChar *` или `TChar`.
  - Конкатенация — перегруженная операция +=, второй операнд может иметьтип `const basic_string &`,`const TChar *`, или `TChar`.
  - ```
    basic_string &append(const basic_string &);
    basic_string &append(
      const basic_string &, size_type pos, size_type n);
    basic_string &append(const TChar *);
    basic_string &append(const TChar *, size_type n);
    basic_string &append(size_type n, TChar c);
    template <class InpIter> basic_string &append(
      InpIter first, InpIter last);
    ```
    — добавляют указанные символы в конец строки.
    ```

– `basic_string &assign(/* в тех же формах, что и append */);` — заменяют содержимое строки указанными символами.

– `basic_string &insert(size_type pos1, const basic_string &);`
`basic_string &insert(`
`size_type pos1, const basic_string &,`
`size_type pos2, size_type n);`
`basic_string &insert(size_type pos, const TChar *);`
`basic_string &insert(size_type pos, const TChar *, size_type n);`
`basic_string &insert(size_type pos, size_type n, TChar);`
`iterator insert(iterator pos, TChar);`
`void insert(iterator pos, size_type n, TChar);`
`template <class InpIter> void insert(`
`iterator pos, InpIter first, InpIter last);` — вставляют указанные символы в заданную позицию строки.

– `basic_string &erase(size_type pos = 0, size_type n = npos);`
`iterator erase(iterator pos);`
`iterator erase(iterator first, iterator last);` — удаляют символ (символы) из указанных позиций.

– `basic_string &replace(`
`size_type pos1, size_type n1, const basic_string &);`
`basic_string &replace(`
`size_type pos1, size_type n1, const basic_string &,`
`size_type pos2, size_type n2);`
`basic_string &replace(`
`size_type pos, size_type n1, const TChar *, size_type n2);`
`basic_string &replace(`
`size_type pos, size_type n1, size_type n2, TChar);`
`basic_string &replace(`
`iterator pos1, iterator pos2, const basic_string &);`
`basic_string &replace(`
`iterator pos1, iterator pos2, const TChar *);`
`basic_string &replace(`
`iterator pos1, iterator pos2, const TChar *, size_type n);`
`basic_string &replace(`
`iterator pos1, iterator pos2, size_type n, TChar);`
`template <class InpIter> basic_string &replace(`
`iterator pos1, iterator pos2, InpIter first, InpIter last);` — заменяют символы в заданных позициях заданными символами.

- Поиск:

- `size_type find( const basic_string &, size_type pos = 0) const;`
  `size_type find( const TChar *, size_type pos = 0) const;`
  `size_type find( const TChar *, size_type pos, size_type n) const;`
  `size_type find(TChar, size_type pos = 0) const;` — находят первое вхождение значения.
- `size_type rfind( /*` в тех же формах, что и `find`, но с `size_type pos = npos */);` — находят последнее вхождение значения.
- `size_type find_first_of(/*` в тех же формах, что и `find */);` — находят первое вхождение любого символа из значения.
- `size_type find_last_of(/*` в тех же формах, что и `rfind */);` — находят последнее вхождение любого символа из значения.
- `size_type find_first_not_of(/*` в тех же формах, что и `find */);` — находят первое вхождение любого символа не из значения.
- `size_type find_last_not_of(/*` в тех же формах, что и `rfind */);` — находят последнее вхождение любого символа не из значения.

2 Итераторы

2.1 Категории итераторов

2.1.1 Входные итераторы

Свойства:
Для любых итераторов чтения `i`, `i1`, и т. д.:

Операция	Действие
`*i`	Доступ к элементу на чтение.
`i->m`	Доступ к члену элемента на чтение.
`++ i`	Переход к следующему элементу, возвращает новую позицию.
`i ++`	Переход к следующему элементу, возвращает старую позицию.
`i1 == i2`	Проверка на равенство.
`i1 != i2`	Проверка на неравенство.
`ITER_TYPE(i)`	Конструктор копирования.

2.1.2 Выходные итераторы

Свойства:

Операция	Действие
`*i = v`	Запись в элемент.
`++ i`	Переход к следующему элементу, возвращает новую позицию.
`i ++`	Переход к следующему элементу, возвращает старую позицию.
`ITER_TYPE(i)`	Конструктор копирования.

2.1.3 Прямые итераторы

Обладают всеми свойствами входных, и почти всеми свойствами выходных итераторов. В отличие от них могут указывать на один и тот же элемент в той же коллекции и обрабатывать его более одного раза.

Свойства:

Операция	Действие
`*i`	Доступ к элементу на чтение / запись.
`i->m`	Доступ к члену элемента на чтение / запись.
...	Другие свойства входных и выходных итераторов ...
`ITER_TYPE()`	Конструктор по умолчанию.
`i1 = i2`	Операция присваивания.

2.1.4 Двунаправленные итераторы

По сравнению с прямыми вводят дополнительные операции перехода назад.

2.1.5 Итераторы произвольного доступа

По сравнению с двунаправленными вводят дополнительные операции с арифметикой итераторов.

Свойства:

Операция	Действие
...	Все свойства двунаправленных итераторов ...
`i += n`	Переход на `n` элементов вперёд (назад, если `n < 0`).
`i -= n`	Переход на `n` элементов назад (вперёд, если `n < 0`).
`i + n`	Возвращает итератор элемента, отстоящего на `n` элементов
`n + i`	вперёд от текущего.
`i - n`	Возвращает итератор элемента, отстоящего на `n` элементов назад от текущего.
`i[n]`	`*(i + n)` (как для обычных массивов).
`i1 - i2`	Возвращает расстояние между двумя итераторами.
`i1 < i2`	Сравнение итераторов.
`i1 <= i2`	
`i1 > i2`	
`i1 >= i2`	

Перемещение итератора за начало (`coll.begin() - 1`) приводит к неопределённому поведению.

2.2 Вспомогательные функции

Описаны в `<iterator>` (кроме последней).

- **`template <class InpIter, class Dist>`**

  `void advance(InpIter &rPos, Dist n);` — смещает позицию на `n` элементов вперёд (можно назад для двунаправленных итераторов). Выбирает оптимальный способ в зависимости от типа итератора (произвольного доступа или нет). *Не проверяет* на выход за конец.

- **`template <class InpIter>`**

  `typename iterator_traits <InpIter>::difference_type`

  `distance(InpIter first, InpIter last);` — вычисляет разность между итераторами. `last` должен быть достижим из `first`.

- **`template <class FwdIter1, class FwdIter2>`**

  `void iter_swap(FwdIter1 pos1, FwdIter2 pos2);` — меняет местами значения элементов, на которые указывают итераторы. Описана в `<algorithm>`.

2.3 Адаптеры итераторов

2.3.1 Обратные итераторы

Переопределяют операции увеличения / уменьшения итераторов таким образом, что они действуют в обратном направлении.

2.3.2 Итераторы вставки

Переопределяют операции разыменования и инкремента как ничего не делающие, возвращающие текущую позицию

Виды итераторов вставки:

Имя	Класс	Создание	Реализация
Назад	`back_insert_iterator`	`back_inserter(coll)`	`coll.push_back(val)`
Вперёд	`front_insert_iterator`	`front_inserter(coll)`	`coll.push_front(val)`
Общий	`insert_iterator`	`inserter(coll, pos)`	`coll.insert(pos, val)`

Реализация `insert_iterator`:

```
pos = coll.insert(pos, val);
++ pos;
```

2.3.3 Итераторы ввода / вывода

Свойства итераторов ввода:

Операция	Действие
`istream_iterator <T> ()`	Создаёт позицию конца потока.
`istream_iterator <T> (istream)`	Создаёт итератор для потока `istream` (и, возможно, считывает первое значение).
`*i`	Возвращает прочитанное ранее значение (считывает первое, если этого не было сделано в конструкторе).
`i->m`	Доступ к члену прочитанного элемента.
`++ i`	Считывает следующее значение, возвращает новую позицию.
`i ++`	Считывает следующее значение, возвращает старую позицию.
`i1 == i2`	Сравнение итераторов (имеет смысл
`i1 != i2`	сравнивать только с концом потока).

Если при считывании происходит ошибка, итератор ввода становится итератором конца потока.

Поскольку конструктор может считывать первое значение, не следует описывать итератор до того, как он действительно необходим.

Свойства итераторов вывода:

Операция	Действие
`ostream_iterator <T> (ostream)`	Создаёт итератор для потока `ostream`.
`ostream_iterator <T> (ostream, pcszD)`	Создаёт итератор для потока `ostream` со строкой `pcszD` в качестве разделителя между значениями (типа **`const char`** `*`).
`*i`	Ничего не делают (возвращают
`++ i`	значение итератора).
`i ++`	
`i = v`	Записывает значение в выходной поток.

3 Функциональные объекты

3.1 Стандартные функциональные объекты

Определены в заголовке `<functional>`.

Операция	Действие
`negate <T> ()`	`-p`
`plus <T> ()`	`p1 + p2`
`minus <T> ()`	`p1 - p2`
`multiplies <T> ()`	`p1 * p2`
`divides <T> ()`	`p1 / p2`
`modulus <T> ()`	`p1 % p2`
`equal_to <T> ()`	`p1 == p2`
`not_equal_to <T> ()`	`p1 != p2`
`less <T> ()`	`p1 < p2`
`greater <T> ()`	`p1 > p2`
`less_equal <T> ()`	`p1 <= p2`
`greater_equal <T> ()`	`p1 >= p2`
`logical_not <T> ()`	`!p`
`logical_and <T> ()`	`p1 && p2`
`logical_or <T> ()`	`p1 || p2`

3.2 Адаптеры функциональных объектов

3.2.1 Адаптеры для привязывания аргументов и отрицания

Стандартные адаптеры:

Операция	Действие
`bind1st(op, value)`	`op(value, param)`
`bind2nd(op, value)`	`op(param, value)`
`not1(op)`	`!op(param)`
`not2(op)`	`!op(param1, param2)`

3.2.2 Адаптеры для обычных функций

Операция	Действие
`ptr_fun(op)`	`op(param)` или `op(param1, param2)` — в зависимости от того, имеет ли функция `op` один или два параметра.

3.2.3 Адаптеры для функций-членов классов

Операция	Действие
`mem_fun_ref(op)`	`(param.*op)()` или `(param1.*op)(param2)` — в зависимости от того, имеет ли функция-член класса, на которую указывает `op`, ни одного или один параметр.
`mem_fun(op)`	`(param->*op)()` или `(param1->*op)(param2)` — в зависимости от того, имеет ли функция-член класса, на которую указывает `op`, ни одного или один параметр.

4 Алгоритмы

Описаны в заголовке `<algorithm>`. Некоторые численные алгоритмы — в `<numeric>`. В `<algorithm>` также описаны вспомогательные функции `min()`, `max()` и `swap()` — для двух значений произвольного типа.

4.1 Категории алгоритмов

4.1.1 Немодифицирующие алгоритмы

- `for_each(first, last, f);` — применяет функцию `f` ко всем элементам полуинтервала (возвращаемое значение игнорируется), в прямом направлении.
- `count(_if)(first, last, fcond_or_val);` — подсчитывает количество заданных элементов.
- `min_element(first, last(, fcond_bin));` — находит *позицию* минимального элемента.
- `max_element(...);`
- `find(_if)(first, last, fcond_or_val);` — находит *позицию* первого элемента, удовлетворяющего условию (или `last`). Если значения в диапазоне отсортированы, необходимо использовать вместо него `lower_bound()`, `upper_bound()`, `equal_range()`, или `binary_search()`. Ассоциативные контейнеры предоставляют более эффективный метод `find()`.
- `search_n(first, last, num, val(, fcond_bin));` — находят позицию первой последовательности из `num` элементов, равных `val` (или удовлетворяющих условию `fcond_bin(elem, val)` — должен быть *бинарным* предикатом).
- `search(first, last, first_to_find, last_to_find(, fcond_bin));` — находят позицию первой последовательности, совпадающей с $[$`first_to_find`, `last_to_find`) (или равенство может определяться как `fcond_bin(elem, elem_to_find)`).

- `find_end(...);` — то же самое, что и `search`, но *последней* последовательности.
- `find_first_of(...);` — как и `search`, но находит первую позицию *любого элемента* из [`first_to_find`, `last_to_find`).
- `adjacent_find(_if)(first, last(, fcond_bin));` — находит первую позицию элемента, совпадающего со следующим.
- `adjacent_find(_if)(first, last(, fcond_bin));` —
- `equal(first1, last1, first2(, fcond_bin));` — проверяет два диапазона на равенство.
- `mismatch(first1, last1, first2(, fcond_bin));` — возвращает пару (`pair`) позиций соответствующих элементов, которые не совпадают (`!(*i1 == *i2)` или `!fcond_bin(*i1, *i2)`).
- `lexicographical_compare(first1, last1, first2, last2(, fcond_bin));` — сравнивает последовательности в лексикографическом порядке, используя операцию `*i1 < *i2` или `fcond_bin(*i1, *i2)`.

4.1.2 Модифицирующие алгоритмы

- `copy(first, last, first_to);` — копирует данные *в прямом* направлении.
- `copy_backward(first, last, last_to);` — копирует данные *в обратном* направлении, все итераторы должны быть двунаправленными.
- `transform(first1, last1(, first2), first_to, f_un_or_bin);` — записывает в выходной диапазон значения `f_un_or_bin(*i)` или `f_un_or_bin(*i1, i2)`, от значений из двух входных диапазонов. Отличается от `for_each()` тем, что результирующее значение определяется возвращаемым значением функционала (в `for_each()` значение параметра-ссылки должно меняться).
- `swap_ranges(first1, last1, first2);` — меняет значения местами в обоих диапазонах, должны задаваться прямыми итераторами.
- `fill(first, last, val);` — заполняет диапазон заданным значением. Итераторы должны быть прямыми.
- `fill_n(first, n, val);` — заполняет `n` значений, начиная с заданной позиции, заданным значением. Итератор может быть выходным.
- `generate(first, last, f);`

  `generate_n(first, n, f);` — как `fill()` и `fill_n()`, но записываемые значения, генерируются вызовом `f()`.
- `replace(_copy)(_if)(first, last(, first_to), fcond_or_val, val_new);` — заменяет в диапазоне каждое значение, равное `fcond_or_val` (или для которого выполняется `fcond_or_val(*i)`) на `val_new`.

4.1.3 Удаляющие алгоритмы

- `remove(_copy)(_if)(first, last(, first_to), fcond_or_val);` — удаляют заданные элементы в диапазоне, сохраняя относительное расположение остальных.

- `unique(_copy)(first, last(, first_to)(, fcond_bin));` — удаляют все следующие друг за другом одинаковые элементы, кроме первого из них.

4.1.4 Перестановочные алгоритмы

- `reverse(_copy)((first, last(, first_to));` — меняют порядок следования на противоположный, итераторы должны быть двунаправленными (кроме `first_to`).

- `rotate(_copy)((first, first_new, last(, first_to));` — циклически сдвигают все элементы, смещая в начало элемент, на который указывает `first_new`. Все итераторы должны быть прямыми (кроме `first_to`).

- `next_permutation(first, last(, fcond_bin));` — генерирует «следующую» (в лексикографическом порядке) перестановку из последовательности. Если перестановка является последней, генерирует «начальную» перестановку (из упорядоченных по возрастанию элементов) и возвращает **`false`**. Итераторы должны быть двунаправленными.

- `prev_permutation(...);`.

- `random_shuffle(first, last(, rfn_random));` — перемешивает последовательность в случайном порядке с равномерным распределением. Итераторы должны быть произвольного доступа. Дополнительно можно передавать функцию, генерирующую случайную последовательность, по ссылке `rfn_random`, с одним параметром `n` типа `difference_type`, и возвращающую случайное значение от `0` до `n - 1`. Параметр является ссылкой, чтобы можно было иметь возможность передавать функциональный объект, хранящий предыдущее случайное число.

- `(stable_)partition(first, last, fcond_un);` — перемещают в начало все элементы, для которых `fcond_un` возвращает **`true`**. Возвращают позицию первого элемента, для которого условие не выполняется. Итераторы должны быть двунаправленными. Вариант `stable_partition` дополнительно сохраняет исходное взаимное расположение элементов в обеих группах — удовлетворяющих и не удовлетворяющих условию.

4.1.5 Алгоритмы сортировки

- `(stable_)sort(first, last(, fcond_bin));` — выполняют сортировку последовательности. Итераторы должны быть произвольного доступа. Вариант `stable_sort` дополнительно сохраняет исходное взаимное расположение «одинаковых» элементов.

- `partial_sort(first, last_sort, last(, fcond_bin));` — производит сортировку, останавливается, как только окажутся отсортированными все элементы в диапазоне [`first, last_sort`). Итераторы должны быть произвольного доступа.

- `partial_sort_copy(first, last, first_to, last_to(, fcond_bin));` — копирует min{`last - first, last_to - first_to`} элементов из отсортированной исходной последовательности. Возвращают позицию первого неперереписанного элемента в результирующей последовательности. Итераторы источника должны быть входными, итераторы приёмника — произвольного доступа.

- `nth_element(first, nth, last(, fcond_bin));` — упорядочивает последовательность, перенося в диапазон [`first, nth`) все элементы, не бóльшие, чем `*nth`, а в [`nth, last`) — не меньшие (используется при быстрой сортировке).

Алгоритмы пирамиды:

- `make_heap(first, last(, fcond_bin));` — преобразуют последовательность в пирамиду (необходимо только для последовательности более, чем из одного элемента).

- `push_heap(first, last(, fcond_bin));` — помещают последний элемент из диапазона [`first, last`) в пирамиду, находящуюся в диапазоне [`first, last - 1`) (так, что диапазон [`first, last`) становится пирамидой).

- `pop_heap(first, last(, fcond_bin));` — перемещают в конец диапазона первый (наибольший) элемент, и создают новую пирамиду в диапазоне [`first, last - 1`).

- `sort_heap(first, last(, fcond_bin));` — преобразуют пирамиду в сортированную последовательность (перестаёт быть пирамидой), за время $O(n \log n)$.

4.1.6 Алгоритмы обработки упорядоченных последовательностей

Алгоритмы поиска:

- `binary_search(first, last, val(, fcond_bin));` — возвращают **`true`**, если заданный элемент найден. Итераторы должны быть прямыми.

- `includes(first, last, first_to_find, last_to_find(, fcond_bin));` — возвращают **`true`**, если *все* значения из второго диапазона найдены в первом. Итераторы должны быть входными.

- `lower_bound(first, last, val(, fcond_bin));` — возвращают позицию первого элемента, *не большего* заданного, или `last`, если такого нет (первая позиция, куда он может быть вставлен без нарушения упорядоченности всей последовательности). Итераторы должны быть прямыми.

- `upper_bound(...);` — как `lower_bound()`, но находят позицию первого элемента, *большего* заданного (последняя позиция, куда он может быть вставлен без нарушения упорядоченности).

- `equal_range(first, last, val(, fcond_bin));` — эквивалентны вызовам двух предыдущих алгоритмов, возвращают пару (`pair`).

Алгоритмы объединения:

- `merge(first1, last1, first2, last2, first_to(, fcond_bin));` — объединяют два упорядоченных диапазона, записывая результат, начиная с `first_to` (возвращают правую границу результирующего диапазона).

- `set_union(...);` — как `merge()`, однако из одинаковых элементов записывается их *максимальное* количество в каждом из диапазонов (а не их суммарное количество в обоих диапазонах).

- `set_intersection(...);` — как `set_union()`, однако записываются только элементы, общие для двух диапазонов. Из одинаковых элементов записывается их *минимальное* количество в каждом из диапазонов.

- `set_difference(...);` — как `set_union()`, однако записываются только элементы, принадлежащие первому диапазону, но *не входящие* во второй. Из одинаковых элементов записывается их *разность между* их количеством в первом и втором диапазонах (или ни одного, если во втором их больше).

- `set_symmetric_difference(...);` — как `set_union()`, однако записываются только элементы, принадлежащие только *одному из диапазонов*. Из одинаковых элементов записывается их *разность по модулю между* их количеством в первом и втором диапазонах.

- `inplace_merge(first1, last1_first2, last2(, fcond_bin));` — объединяют два смежных сортированных диапазона в один. Итераторы должны быть двунаправленными.

4.1.7 Алгоритмы обработки числовых последовательностей

Определены в заголовке `<numeric>`.

- `accumulate(first, last, val_init(, f_bin));` — возвращают «сумму» всех элементов, выполняя для каждой позиции `val_init = val_init + *i` (или `val_init = f_bin(val_init, *i)`). Значение `val_init` не меняется, если передаётся по значению.

- `inner_product(first1, last1, first2, val_init(, f_bin));` — возвращают «скалярное произведение» двух диапазонов.

- `partial_sum(first, last, first_to(, f_bin));` — записывают во второй диапазон «частичные суммы» для каждой позиции в первом диапазоне: `*i1, *i1 + *i2, *i1 + *i2 + *i3, ...`, возвращают правую границу выходного диапазона. Входной и выходной диапазоны могут совпадать.

- `adjacent_difference(first, last, first_to(, f_bin));` — записывают во второй диапазон для каждой позиции «разность» её и предыдущего элемента: `*i1, *i2 - *i1, *i3 - *i2, ...`, возвращают правую границу выходного диапазона. Входной и выходной диапазоны могут совпадать. Являются «обратными» к `partial_sum()`.