Лекция 01 — STL

Содержание

”Давайте не бежать впереди паровоза, а то он вас может догнать.”
— А. П. Хвастунов

Общие представления

Стандартная библиотека шаблонов (*STL — Standard Template Library*) — условно выделяемый подраздел стандартной библиотеки языка C++, объединяющий в себе ряд широко используемых шаблонизированных классов и алгоритмов. Она разрабатывалась Александром Степановым и Менгом Ли в рамках их работы в компании Hewlett-Packard и

была впервые представлена на встрече Комитета по стандартизации в 1993 году, практически сразу получив восхищённые отзывы.

Стандартная библиотека шаблонов предлагает готовые и эффективные решения повседневных задач и поэтому является неотъемлемой частью языка на сегодняшний день. Если что-то уже реализовано в STL, то у нас почти наверняка не выйдет сделать это лучше, а если и выйдет, то на практике это будет иметь не очень много смысла.

Все сущности, содержащиеся в STL, можно разбить на несколько категорий: *контейнеры*, *алгоритмы*, *итераторы* и *функторы*. Каждая из них будет подробнее рассмотрена далее.

Контейнеры

Контейнеры — объекты, отвечающие за хранение данных и управление памятью, которая используется сохраняемыми сущностями. Они призваны инкапсулировать механизмы решения тех проблем, с которыми мы сталкиваемся при работе с обычными массивами.

Стандарт, впрочем, не закрепляет само понятие контейнера. Вместо этого он выдвигает ряд *именованных требований* (*Named requirements*), которые определяют *свойства*, которыми должен обладать класс, чтобы заслужить право носить гордое имя «контейнер». В случае общего определения контейнера именованные требования включают в себя наличие ряда базовых `typedef`-деклараций, специальных методов, доступа к итератором (*см. Итераторы*), операций сравнения, а также методов `size()`, `empty()` и `swap()`.

Примечание. Начиная с C++20 именованные требования заменяются концептами, что, впрочем, никак не меняет сути дела.

Последовательные контейнеры

О последовательных контейнерах можно думать как об удобной «надстройке» над обычным массивом — иными словами, они последовательно хранят набор объектов одного типа. Помимо выполнения базовых требований контейнера, последовательный контейнер должен поддерживать операцию вставки и ряд специальных конструкторов.

Стандартная библиотека шаблонов предлагает следующие последовательные контейнеры:

> I. `std::array`

Этот контейнер является агрегатным типом с той же семантикой, что и структура, содержащая в качестве единственного поля обычный массив. Его главное применение — адаптировать C-style массивы для работы с алгоритмами STL.

```
std::array<int, 5> a{1, 2, 3, 4, 5};
```

II. `std::vector`

Вектор реализует интерфейс *динамического* массива, автоматически расширяя память при необходимости. Он хранит элементы последовательно друг за другом, что в частности означает, что доступ к ним можно получать при помощи арифметики указателей. Вектор предоставляет быстрый произвольный доступ, а также вставку в конец и удаление с конца за амортизированную константу. Операции произвольной вставки и произвольного удаления линейны относительно количества элементов в векторе.

```
std::vector<int> a{1, 2, 3, 4};
a.push_back(5);
```

Существует отдельная специализация `vector<bool>`, которая минимально перерасходует память для случая массива отдельных битов.

III. `std::list`

Этот контейнер реализует интерфейс *двусвязного списка*. Он поддерживает произвольную вставку и удаление за $O(1)$, однако не поддерживает быстрый произвольный доступ.

```
std::list<int> a{1, 3, 4, 5};
a.insert(std::next(a.begin()), 2);
```

IV. `std::forward_list`

Этот контейнер реализует интерфейс *односвязного списка* и обладает соответствующими ограничениями сравнительно `std::list`, однако он несколько более оптимально расходует память.

```
std::forward_list<int> a{1, 2, 3, 4, 5};
```

V. `std::deque`

Это динамический массив, поддерживающий быстрые манипуляции не только с концом, но и с началом. В отличие от `std::vector`, он не хранит элементы последовательно.

```
std::deque<int> a{2, 3, 4, 5};
a.push_front(1);
```

При выборе контейнера следует чётко понимать, какая задача перед ним стоит, поскольку ни один из них, как видно, не является универсальным. Вектор и список, в частности, можно воспринимать как антагонирующие сущности: преимущества одного почти всегда будут недостатками второго, и наоборот. Впрочем, стандартным решением в ситуации, когда определиться с выбором слишком сложно, всё же является вектор.

Ассоциативные и неупорядоченные ассоциативные контейнеры

Ассоциативные контейнеры в первую очередь решают задачу быстрого поиска элементов. Практически всегда они в своей основе представляют красно-черное дерево, и поэтому сложность операций с ними в среднем составляет $O(\log n)$.

Стандартная библиотека шаблонов предлагает следующие ассоциативные контейнеры:

I. `std::set`

Этот контейнер представляет собой упорядоченную коллекцию уникальных ключей. Упорядочение происходит относительно значений элементов при помощи передаваемого в качестве шаблонного аргумента компаратора. В качестве компаратора по умолчанию используется `std::less<T>` (*см. Функциональные классы*).

```
std::set<int, std::greater<int>> a;
a.insert(1); // 1
a.insert(1); // 1
a.insert(2); // 2 1
```

Если мы хотим позволить элементам множества повторяться, мы должны использовать `std::multiset`:

```
std::multiset<int> a;
a.insert(1); // 1
a.insert(1); // 1 1
```

> II. `std::map`
>
> Этот контейнер представляет собой ассоциативный массив в его канонической форме. Он хранит пары «ключ-значение», где ключ уникален. Эти пары сортируются относительно значений ключей при помощи передаваемого в качестве шаблонного аргумента компаратора. В качестве компаратора по умолчанию используется `std::less<T>` (*см. Функциональные классы*).
>
> ```
> std::map<std::string, int> a;
> a["hvost"] = 238;
> ```
>
> Если мы хотим позволить ключам повторяться, мы должны использовать `std::multimap`. Следует помнить, что в этом случае мы теряем возможность напрямую обращаться к элементам по ключу.

Упорядочение в ассоциативных контейнерах опционально. Если нам не обязательно хранить ключи в отсортированном формате, мы можем сделать выбор в пользу неупорядоченных ассоциативных контейнеров, которые практически всегда реализованы при помощи хеш-таблиц, что *в общем случае* даёт нам константное время поиска, доступа, вставки и удаления. Следует, впрочем, помнить, что время выполнения операций в хеш-таблицах при определенных обстоятельствах может достигать линейного. В частности, при небольшом количестве упорядоченные ассоциативные контейнеры могут оказаться более оптимальны с точки зрения скорости операций.

Стандартная библиотека шаблонов предоставляет неупорядоченные версии каждого из описанных выше контейнеров: `std::unordered_map`, `std::unordered_multimap`, `std::unordered_set`, `std::unordered_multiset`.

Адаптеры контейнеров

При реализации определённых структур данных мы можем прийти к занятному откровению, которое будет заключаться в том, что в их основе лежит некоторый контейнер. Более того, одна и та же структура может работать с различными контейнерами, никак не изменяя свой интерфейс, и каждая из реализаций будет обладать как своими преимуществами, так и своими недостатками. Выбрать что-то одно в таком положении достаточно трудно, поэтому достаточно логичным решением кажется позволить пользователю параметризовать структуру, самостоятельно выбрав, на основе какого контейнера она будет реализована.

Такие структуры, интегрирующие существующие классы в систему с другим интерфейсом, называются *адаптерами*. Конечно, здесь подразумеваются в первую очередь очередь и стек, которые можно реализовать практически на любом

последовательном контейнере. К адаптерам контейнеров также относится очередь с приоритетом — `std::priority_queue`, которая по умолчанию использует вектор. Очередь и стек по умолчанию используют `std::deque`.

Аллокаторы

Практически все динамические контейнеры позволяют явно определить функциональный класс, который будет отвечать за аллокацию памяти, если он удовлетворяет именованным требованиям аллокатора:

```
template<typename T>
class MyAllocator {
  public:

     typedef T value_type;

     T* allocate(int n) {
          std::cout << "Allocate\n";
          return new T[n];
     }

     void deallocate(T* p, int n) {
          std::cout << "Deallocate\n";
          delete[] p;
     }

};

int main() {
     std::vector<int, MyAllocator<int>> vec;
     vec.push_back(5);
}

// Output:
// Allocate
// Deallocate
```

По умолчанию все контейнеры используют `std::allocator`, чья стратегия заключается в простом вызове `new` и `delete` — примерно так же, как показано выше. Тем не менее, таких стратегий сильно больше одной, и каждая из них может оказаться оптимальной при определенных обстоятельствах. Задача аллокатора в этом контексте — инкапсулировать в себя конкретный подход и параметризировать конкретный контейнер, чтобы подобрать подходящий сценарий для отдельно взятой ситуации.

Итераторы

Определение

Как можно видеть, контейнеры бывают достаточно разношёрстными. Нам, тем не менее, хотелось бы, чтобы наши алгоритмы умели одинаково хорошо работать если не со всеми из них, то с как можно бо̀льшим их числом.

Основная задача, которая перед нами встаёт — унификация процедуры доступа к элементам. И задача эта совсем не тривиальная, поскольку какие-то контейнеры хранят данные последовательно, а какие-то — нет; одни предоставляют произвольный доступ за константное время, другие — за линейное. К единому решению в таких условиях прийти откровенно невозможно, поэтому STL предлагает несколько «уровней» обращения к содержимому контейнеров, объединяя их под эгидой единого интерфейса.

Концепция *итераторов* в определенном смысле вдохновлена тем, как мы проходим по обычному массиву, сдвигая единственный указатель. Если бы каждый контейнер предоставлял такой указателеобразный объект, то мы бы могли вообще не придумывать ничего нового. Именно этот объект, обобщающий свойства указателей, и называется итератором.

Любой контейнер поддерживает методы `.begin()` и `.end()`.
`.begin()` возвращает итератор, указывающий на первый элемент контейнера.
`.end()` возвращает итератор, указывающий на позицию сразу за последним элементом контейнера. Таким образом, диапазон итераторов можно выразить в виде полуинтервала `[.begin(), .end())`.
Располагать `.end()` именно таким образом крайне удобно по ряду причин:

- легко проверить пустой контейнер;
- легко проверить конец итерации по контейнеру;
- легко добавить элемент в конец контейнера;
- многим алгоритмам удобно работать с полуинтервалами;
- нет необходимости отводить специальное значение под результат неудачной работы алгоритма и проч.

Следует отметить, что при изменении внутренней структуры контейнера некоторые связанные с ним итераторы могут продолжить указывать на неактуальные данные, и как следствие потерять свой функционал. Это явление называется *инвалидацией* итератора. Каждый контейнер определяет собственный список условий, при которых его итераторы инвалидируются.

Классификация

По уже упомянутым причинам итераторы разделяются на шесть категорий. Категории эти так же, как и контейнеры, определяются именованными требованиями — в частности, любой итератор должен поддерживать оператор инкремента, а также, что не менее важно, гарантировать временную сложность $O(1)$ для всех операций, которые с ними производятся. Остальные требования зависят от непосредственной категории, среди которых выделяются:

> I. **Входной итератор** ($LegacyInputIterator$)
>
> Итератор, использующийся для чтения элемента, на который он указывает. Обязан поддерживать операторы `*` и `->`, то есть, практически, позволять разыменование, а также оператор сравнения. Может быть использован для однократного прохода по элементам.

> II. **Выходной итератор** ($LegacyOutputIterator$)
>
> Итератор, использующийся для изменения значения элемента, на который он указывает. Обязан поддерживать выражение `*r = o`.

> III. **Однонаправленный итератор** ($LegacyForwardIterator$)
>
> Входной итератор, значение которого можно сохранять для многократного чтения. Это требование необходимо, например, в алгоритме бинарного поиска.

> IV. **Двунаправленный итератор** ($LegacyBidirectionalIterator$)
>
> Однонаправленный итератор, поддерживающий декремент, т.е. позволяющий перемещаться в обоих направлениях.

> V. **Итератор с произвольным доступом** ($LegacyRandomAccessIterator$)
>
> Двунаправленный итератор, поддерживающий произвольный доступ к элементам за константное время. С точки зрения именованных требований это практически означает полную поддержку арифметики указателей.

> VI. **Непрерывный итератор** ($LegacyContiguousIterator$)
>
> Итератор произвольного доступа, который гарантирует, что его элементы физически расположены в памяти последовательно. Практически применяется для низкоуровневых оптимизаций.

Отсюда видно, что категории указателей образуют иерархическую структуру: категории III -VI удовлетворяют всем именованным требованиям предыдущих категорий, последовательно образуя надклассы. Отсюда, в частности, следует, что везде, где требуется итератор «младшей» категории, можно предоставить итератор «старшей»

категории. Следует также отметить, что, очевидно, указатель на элемент массива является непрерывным итератором.

Описание контейнеров включает описание тех категорий итераторов, которые они предоставляют, а описание алгоритмов — тех, которые они запрашивают. Таким образом STL гарантирует поддержку эффективных комбинаций и препятствует неэффективным.

Адаптеры итераторов

Так же, как и в случае контейнеров, стандартная библиотека шаблонов предлагает ряд адаптеров для итераторов. Часть из них предназначена для вставки диапазонов значений в произвольные места контейнера — это наиболее полезно в ситуации, когда размер диапазона, получающегося в результате работы алгоритма, может быть произвольным. Другая часть реализует различную неклассифицируемую семантику.

Среди адаптеров итераторов можно выделить:

> ▶ `std::reverse_iterator`
>
> Итератор, предоставляющий возможность итерации по элементам диапазона в обратном порядке. Получить такой итератор можно при помощи функции `std::make_reverse_iterator`.
>
> ```
> int main() {
>     std::vector<int> a{1, 2, 3, 4, 5};
>     std::for_each(std::make_reverse_iterator(a.end()),
> std::make_reverse_iterator(a.begin()), Print); // 5 4 3 2 1
> }
> ```

> ▶ `std::back_insert_iterator`
>
> Итератор, который инкапсулирует механизм вставки элементов в конец контейнера. Он поддерживается только контейнерами, реализующими метод `push_back`. Получить такой итератор можно при помощи функции `std::back_inserter`.
>
> ```
> int main() {
>     std::vector<int> a{1, 2, 3};
>     std::vector<int> b{4, 5, 6};
>     std::copy(b.begin(), b.end(), std::back_inserter(a)); // 1 2 3 4 5 6
> }
> ```

> ▶ `std::front_insert_iterator`

Итератор, который инкапсулирует механизм вставки элементов в начало контейнера. Он поддерживается только контейнерами, реализующими метод `push_front`.Получить такой итератор можно при помощи функции `std::front_inserter`.

```
int main() {
    std::deque<int> a{1, 2, 3};
    std::vector<int> b{4, 5, 6};
    std::copy(b.begin(), b.end(), std::front_inserter(a)); // 4 5 6 1 2
3
}
```

▶ `std::insert_iterator`

Итератор, который инкапсулирует механизм вставки элементов в произвольное место в контейнере. Он поддерживается только контейнерами, реализующими метод `insert`. Получить такой итератор можно при помощи функции `std::inserter`.

```
int main() {
    std::deque<int> a{1, 2, 3};
    std::vector<int> b{4, 5, 6};
    std::copy(b.begin(), b.end(), std::inserter(a, a.begin() + 1)); // 1
4 5 6 2 3
}
```

▶ `std::istream_iterator` / `std::ostream_iterator`

Итераторы, которые считывают данные из входного потока или выводят их в выходной.

```
int main() {
    std::vector<int> a;
    std::copy(std::istream_iterator(std::cin), std::istream_iterator(),
std::back_inserter(a));
    std::copy(a.begin(), a.end(), std::ostream_iterator(std::cout));
}
```

`std::iterator_traits`

Ранее мы упомянули, что категории итераторов практически образуют иерархическую структуру, которая, впрочем, существовала пока что лишь в нашем воображении, хотя иметь все свойства итераторов явно представленными и ранжированными было бы крайне полезно в реализации ряда алгоритмов. Создатели стандартной библиотеки

шаблонов учли это, введя шаблонный класс `std::iterator_traits`, который предоставляет единообразный интерфейс для доступа к свойствам конкретных типов итераторов.

Он хранит в качестве полей все основные `typedef`-декларации, определенные в самом итераторе (исключением служит обыкновенный указатель, для которого эти декларации выводятся отдельно), а также, что главное, *тег*, который также обязан явно определяться для каждого итератора. Эти теги представляют собой пустые структуры, использующие механизм разрешения перегрузок, чтобы выбрать наиболее оптимальную реализацию алгоритма для конкретного набора условий, которые реализуются итератором. Это — яркий и самый что ни на есть прямой пример использования идиомы Tag dispatch, которая будет отдельно обсуждена в разговоре о SFINAE и концептах.

Все теги итераторов из понятных соображений организованы в иерархию наследования:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
struct contiguous_iterator_tag : public random_access_iterator_tag {};
```

Стоит также отметить, что `std::iterator_traits` поддерживает объявление специализаций для пользовательских итераторов.

Операции над итераторами

Как уже выяснилось, далеко не все категории итераторов поддерживают арифметику указателей, позволяющую производить соответствующие операции за константное время. По этой причине вводятся обобщенные функции, которые реализуют ту же семантику, но (ровно благодаря `std::iterator_traits`!) при помощи различных механизмов в зависимости от категории итератора. Сложность всех из них линейная до тех пор, пока итератор не удовлетворяет требованиям итератора с произвольным доступом.

> ▶ `std::advance`
>
> Сдвигает итератор на некоторое количество шагов вправо.

```
int main() {
     std::vector<int> a{1, 2, 3, 4, 5};
     auto it = a.begin();
     std::advance(it, 3);
```

```
    std::cout << *it; // 4
}
```

▶ `std::distance`

Вычисляет расстояние между двумя итераторами.

```
int main() {
    std::vector<int> a{1, 2, 3, 4, 5};
    auto it1 = a.begin();
    auto it2 = a.begin();
    std::advance(it2, 3);
    std::cout << std::distance(it1, it2); // 3
}
```

▶ `std::next` / `std::prev`

Возвращают следующий и предыдущий итераторы соответственно.

```
int main() {
    std::vector<int> a{1, 2, 3, 4, 5};
    auto it = a.begin();
    std::cout << *std::next(it); // 2
}
```

Алгоритмы

Алгоритмы стандартной библиотеки шаблонов — это готовые решения для задач поиска, сравнения или простых вычислений, преимущественно при работе с контейнерами.
По аналогии с тем, как контейнеры представлены шаблонными классами, алгоритмы STL представлены шаблонными функциями и поэтому называются *обобщёнными*. Многие из них поддерживают частичную параметризацию, что бывает особенно полезно в алгоритмах поиска и сортировки. Алгоритмы STL определены в заголовочных файлах `<algorithm>` и `<numeric>`. Далее будут обзорно рассмотрены наиболее широко использующиеся из них.

Неизменяющие алгоритмы

▶ `std::all_of` / `std::any_of` / `std::none_of`

Проверяют, выполняется ли предикат для элементов контейнера.

```
bool IsOdd(int x) {
     return x & 1;
}

int main() {
     std::vector<int> a{2, 4, 3, 8};
     std::cout << std::boolalpha << std::all_of(a.begin(), a.end(),
IsOdd); // false
     std::cout << std::boolalpha << std::any_of(a.begin(), a.end(),
IsOdd); // true
}
```

▶ `std::find` / `std::find_if` / `std::find_if_not`

Возвращают итератор на первый элемент диапазона, который удовлетворяет определенному критерию. `std::find` сравнивает элементы с переданным образцом; `std::find_if` и `std::find_if_not` проверяют их на выполнение предиката. Если такого элемента не найдено, возвращается итератор на конец диапазона.

```
bool IsOdd(int x) {
     return x & 1;
}

int main() {
     std::vector<int> a{2, 4, 3, 8};
     std::cout << *std::find(a.begin(), a.end(), 4); // 4
     std::cout << *std::find_if(a.begin(), a.end(), IsOdd); // 3
     std::cout << *std::find_if_not(a.begin(), a.end(), IsOdd); // 2
}
```

▶ `std::count` / `std::count_if`

Подсчитывают количество элементов некоторого диапазона, которые удовлетворяет определенному критерию. `std::count` сравнивает элементы с переданным образцом; `std::count_if` проверяет их на выполнение предиката.

```
bool IsOdd(int x) {
     return x & 1;
}

int main() {
     std::vector<int> a{2, 2, 3, 8};
     std::cout << std::count(a.begin(), a.end(), 2); // 2
```

```
    std::cout << std::count_if(a.begin(), a.end(), IsOdd); // 1
}
```

▶ `std::equal` / `std::mismatch`

Проверяет два диапазона на поэлементное равенство. `std::equal` возвращает результат проверки; `std::mismatch` возвращает пару итераторов на несовпавшие элементы.

```
int main() {
    std::vector<int> a{2, 2, 3, 6};
    std::vector<int> b{2, 3, 7};
    std::cout << std::equal(a.begin() + 1, a.begin() + 3,
                            b.begin(), b.begin() + 2);
    // output: true
    std::cout << std::equal(a.begin() + 1, a.begin() + 4,
                            b.begin(), b.begin() + 3);
    // output: false
    auto diff_elements = std::mismatch(a.begin() + 1, a.begin() + 4,
                                       b.begin(), b.begin() + 3);
    std::cout << *diff_elements.first << ' ' << *diff_elements.second;
    // output: 6 7
}
```

▶ `std::search`

Ищет подпоследовательность `[s_first, s_last)` в последовательности `[first, last)` и возвращает итератор на ее начало. Возвращает `last`, если подпоследовательность не найдена. Поддерживает параметризацию бинарным предикатом, который используется для сравнения.

```
bool P(int x, int y) {
    return x != y;
}

int main() {
    std::vector<int> a{3, 2, 3, 4, 5, 6};
    std::vector<int> b{3, 4, 5};
    std::cout << *std::search(a.begin(), a.end(), b.begin(), b.end());
// 3
    std::cout << *std::search(a.begin(), a.end(), b.begin(), b.end(),
P); // 2
    // Примечание. Передаваемый компаратор используется для сравнения
    // элементов именно на равенство (каждый равен каждому),
```

```
    // поэтому когда мы передаём компаратор P, std::search проверяет
    // условие «каждый не равен каждому»
}
```

Начиная с C++17, `std::serach` может также быть параметризован функторами `std::default_searcher`, `std::boyer_moore_searcher` и `std::boyer_moore_horspool_searcher`, которые определяют алгоритм поиска: алгоритм по умолчанию (обычно линейный поиск), алгоритм Бойера-Мура или алгоритм Бойера-Мура-Хорспула.

Примечание. Следующие поисковые алгоритмы работают только на диапазонах, монотонно упорядоченных относительно некоторого предиката.

▶ `std::lower_bound`

Возвращает итератор на первый элемент, *не меньший* заданного значения на некотором диапазоне. Возвращает итератор на конец диапазона, если такой элемент не найден. Поддерживает параметризацию пользовательским компаратором.

```
int main() {
    std::vector<int> a{1, 2, 7, 19, 150};
    std::cout << *std::lower_bound(a.begin(), a.end(), 19); // 19
}
```

▶ `std::upper_bound`

Возвращает итератор на первый элемент, *больший* заданного значения на некотором диапазоне. Возвращает итератор на конец диапазона, если такой элемент не найден. Поддерживает параметризацию пользовательским компаратором.

```
int main() {
    std::vector<int> a{1, 2, 7, 19, 150};
    std::cout << *std::upper_bound(a.begin(), a.end(), 25); // 150
}
```

▶ `std::binary_search`

Проверяет, встречается ли элемент в диапазоне, используя бинарный поиск. Впрочем, если переданный итератор не поддерживает произвольный доступ, алгоритмическая сложность остаётся линейной. Возвращает `true` или `false`. Поддерживает параметризацию пользовательским компаратором.

```cpp
int main() {
    std::vector<int> a{1, 2, 7, 19, 150};
    std::cout << std::binary_search(a.begin(), a.end(), 7); // true
}
```

Изменяющие алгоритмы

▶ `std::for_each`

Применяет операцию ко всем элементам из диапазона, практически заменяя собой цикл `for`. Может как изменять, так и не изменять состояние контейнера.

```cpp
void Print(int x) {
    std::cout << x << ' ';
}

int main() {
    std::vector<int> a{3, 1, 0, 4};
    std::for_each(a.begin(), a.end(), Print); // 3 1 0 4
}
```

▶ `std::copy` / `std::copy_if` / `std::copy_backward`

Копирует элементы из одного диапазона в другой (возможно, с условием или в обратном порядке). Если диапазоны пересекаются, поведение не определено.

```cpp
int main() {
    std::vector<int> a{1, 0, 0, 0, 1};
    std::vector<int> b{3, 1, 7};
    std::copy(b.begin(), b.end(), a.begin() + 1); // a = {1, 3, 1, 7, 1}
}
```

▶ `std::transform`

Применяет операцию к каждому элементу из диапазона и сохраняет результат в другой диапазон.

```cpp
int Twice(int& x) {
    return x * 2;
}

int main() {
    std::vector<int> a{1, 0, 0, 0, 1};
```

```
    std::vector<int> b{3, 1, 7};
    std::transform(b.begin(), b.end(), a.begin(), Twice);
    // a = {1, 6, 2, 14, 1};
}
```

▶ `std::replace` / `std::replace_if`

Заменяет значение всех элементов, удовлетворяющих некоторому критерию, на новое. `std::replace` сравнивает элементы с переданным образцом; `std::replace_if` проверяет их на выполнение предиката.

```
bool IsOdd(int x) {
    return x & 1;
}

int main() {
    std::vector<int> a{1, 7, 4, 4, 6};
    std::replace(a.begin(), a.end(), 4, 7); // 1 7 7 7 6
    std::replace_if(a.begin(), a.end(), IsOdd, 3); // 3 3 3 3 6
}
```

▶ `std::fill`

Заполняет диапазон заданным значением.

```
int main() {
    std::vector<int> a(5);
    std::fill(a.begin(), a.end(), 3); // 3 3 3 3 3
}
```

▶ `std::generate`

Присваивает диапазону результаты последовательных вызовов функции.

```
int counter = 0;

int Iota() {
    return ++counter;
}

int main() {
    std::vector<int> a(5);
```

```
    std::generate(a.begin(), a.end(), Iota); // 1 2 3 4 5
}
```

▶ `std::remove` / `std::remove_if`

Стирает из диапазона элементы, удовлетворяющие некоторому критерию. `std::remove` сравнивает элементы с переданным образцом; `std::remove_if` проверяет их на выполнение предиката.

```
bool IsOdd(int x) {
    return x & 1;
}

int main() {
    std::vector<int> a{1, 7, 4, 4, 6};
    std::remove(a.begin(), a.end(), 7); // 1 4 4 6 6
    std::remove_if(a.begin(), a.end(), IsOdd); // 4 4 6 6 6
}
```

Важно, что эти функции не удаляют элементы физически — они лишь переупорядочивают элементы так, чтобы не удалённые занимали младшие позиции. Фактическое удаление требует дополнительных методов — например, вызова `.erase()`. Для этого даже отводится соответствующая идиома Erase-Remove:

```
v.erase(std::remove(v.begin(), v.end(), item), v.end());
```

▶ `std::unique`

Удаляет последовательно идущие дубликаты элементов в заданном диапазоне. Поддерживает параметризацию бинарным предикатом, использующимся для сравнения.

```
int main() {
    std::vector<int> a{1, 7, 4, 4, 4, 6, 6};
    std::unique(a.begin(), a.end()); // 1 7 4 6 6 6 6
}
```

К `std::unique` так же применима идиома Erase-Remove.

▶ `std::reverse`

Инвертирует порядок элементов в заданном диапазоне.

```cpp
int main() {
    std::vector<int> a{1, 2, 3, 4, 5};
    std::reverse(a.begin(), a.end()); // 5 4 3 2 1
}
```

▶ `std::rotate`

Циклически сдвигает диапазон `[first, last)` так, чтобы элементы `[first, middle)` оказались после элементов `[middle, last)`.

```cpp
int main() {
    std::vector<int> a{1, 2, 3, 4, 5};
    std::rotate(a.begin(), a.begin() + 2, a.end()); // 3 4 5 1 2
}
```

▶ `std::random_shuffle` (`std::shuffle`)

Случайным образом перемешивает элементы из диапазона.
Начиная с C++17, Стандарт отказался от `std::random_shuffle` в пользу `std::shuffle`, основное отличие которого в том, что пользователь обязан явно предоставить собственный генератор случайных чисел.

```cpp
int main() {
    std::vector<int> a{1, 2, 3, 4, 5};
    std::mt19937 rng(3104);
    std::shuffle(a.begin(), a.end(), rng); // 4 5 3 2 1
}
```

▶ `std::sample`

Случайным образом выбирает `n` элементов из одного диапазона и помещает их в другой.

```cpp
int main() {
    std::vector<int> a{1, 2, 3, 4, 5};
    std::vector<int> b{1, 0, 0, 0, 1};
    std::mt19937 rng(52);
    std::sample(a.begin(), a.end(), b.begin(), 3, rng); // 1 2 4 5 1
}
```

▶ `std::merge`

Сливает два упорядоченных диапазона в один упорядоченный диапазон. Поддерживает параметризацию компаратором.

```
int main() {
    std::vector<int> a{1, 3, 5};
    std::vector<int> b{2, 4, 6};
    std::vector<int> res(6);
    std::merge(a.begin(), a.end(), b.begin(), b.end(), res.begin());
    // res = {1, 2, 3, 4, 5, 6}
}
```

Переупорядочивающие алгоритмы

▶ `std::partition` / `std::stable_partition`

Разбивает все элементы из диапазона на две группы так, что все элементы, для которых верен некоторый предикат, предшествуют тем, для которых он ложен. `std::stable_partition`, помимо прочего, сохраняет исходный относительный порядок равных элементов.

```
bool IsOdd(int x) {
    return x & 1;
}

int main() {
    std::vector<int> a{1, 2, 3, 4, 5};
    std::partition(a.begin(), a.end(), IsOdd); // 1 5 3 4 2
}
```

▶ `std::sort` / `std::stable_sort`

Сортирует элементы диапазона согласно некоторому компаратору, который по умолчанию равен `std::less` (*см. Функциональные объекты*). В процессе сортировки происходит $O(n \log n)$ обращений к компаратору.
`std::stable_sort`, помимо прочего, сохраняет исходный относительный порядок равных элементов.

```
int main() {
    std::vector<int> a{1, 5, 3, 4, 2};
    std::sort(a.begin(), a.end()); // 1 2 3 4 5
}
```

▶ `std::nth_element`

Переупорядочивает элементы диапазона так, что n-й элемент стоит на той позиции, которую занимал бы в остортированном массиве, а все остальные расположены относительно него. Поддерживат параметризацию пользовательским компаратором.

```
int main() {
      std::vector<int> a{1, 5, 100, 7, 18, 92, 503};
      std::nth_element(a.begin(), a.begin() + 3, a.end());
      // a = {7 5 1 18 92 100 503}
      // 4-й элемент на своей позиции, слева — меньшие, справа — бо̀льшие;
      // порядок всех, кроме 4-го, может быть любым
}
```

▶ `std::next_permutation` / `std::prev_permutation`

Генерирует лексикографически следующую и предыдущую перестановки.

```
int main() {
     std::vector<int> a{1, 2, 3, 4, 5};
     std::next_permutation(a.begin(), a.end()); // 1 2 3 5 4
     std::next_permutation(a.begin(), a.end()); // 1 2 4 3 5
     std::next_permutation(a.begin(), a.end()); // 1 2 4 5 3
     // ...
}
```

Теоретико-множественные алгоритмы

Примечание. Алгоритмы в этой категории работают только на упорядоченных диапазонах.

▶ `std::includes`

Проверяет, содержит ли один диапазон все элементы другого.

```
int main() {
     std::vector<int> a{1, 2, 3, 4, 5};
     std::vector<int> b{1, 3, 4};
     std::vector<int> c{1, 4, 6};
     std::cout << std::includes(a.begin(), a.end(), b.begin(), b.end());
     // true
     std::cout << std::includes(a.begin(), a.end(), c.begin(), c.end());
     // false
}
```

▶ `std::set_union`

Вычисляет объединение двух множеств.

```
int main() {
     std::vector<int> a{1, 2, 3};
     std::vector<int> b{1, 3, 4};
     std::vector<int> res;
     std::set_union(a.begin(), a.end(), b.begin(), b.end(),
std::back_inserter(res));
     // res = {1, 2, 3, 4}
}
```

▶ `std::set_intersection`

Вычисляет пересечение двух множеств.

```
int main() {
     std::vector<int> a{1, 2, 3};
     std::vector<int> b{1, 3, 4};
     std::vector<int> res;
     std::set_intersection(a.begin(), a.end(), b.begin(), b.end(),
std::back_inserter(res));
     // res = {1, 3}
}
```

▶ `std::set_difference`

Вычисляет разность двух множеств.

```
int main() {
     std::vector<int> a{1, 2, 3};
     std::vector<int> b{1, 3, 4};
     std::vector<int> res;
     std::set_difference(a.begin(), a.end(), b.begin(), b.end(),
std::back_inserter(res));
     // res = {2}
}
```

▶ `std::set_symmetric_difference`

Вычисляет симметрическую разность двух множеств.

```
int main() {
     std::vector<int> a{1, 2, 3};
     std::vector<int> b{1, 3, 4};
```

```
    std::vector<int> res;
    std::set_symmetric_difference(a.begin(), a.end(), b.begin(),
b.end(), std::back_inserter(res));
    // res = {2, 4}
}
```

Числовые алгоритмы (`<numeric>`)

▶ `std::accumulate` / `std::reduce` (C++17)

Вычисляет сумму некоторого изначального значения и всех элементов диапазона. В качестве «суммы» может выступать также и произвольная пользовательская операция.
`std::reduce` несёт ту же семантику, но не гарантирует порядок вычислений.

```
int Product(int x, int y) {
    return x * y;
}

int main() {
    std::vector<int> a{1, 2, 3, 4, 5};
    std::cout << std::accumulate(a.begin(), a.end(), 0); // 15
    std::cout << std::accumulate(a.begin(), a.end(), 1, Product); // 120
}
```

▶ `std::inner_product`

Вычисляет скалярное произведение (сумму попарных произведений) двух диапазонов и прибавляет результат к некоторому изначальному значению. Обе операции, образующие скалярное произведение, могут быть параметризированы.

```
int main() {
    std::vector<int> a{1, 2, 3, 4};
    std::vector<int> b{2, 5, 6, 1};
    std::cout << std::inner_product(a.begin(), a.end(), b.begin(), 0);
    // 34
}
```

▶ `std::partial_sum`

Вычисляет префиксные суммы заданного диапазона элементов. Операция может быть параметризирована.

```
int main() {
     std::vector<int> a{1, 2, 3, 4, 5};
     std::vector<int> b;
     std::partial_sum(a.begin(), a.end(), std::back_inserter(b));
     // b = {1, 3, 6, 10, 15}
}
```

▶ `std::adjacent_difference`

Вычисляет попарные разности заданного диапазона элементов. Операция может быть параметризирована.

```
int main() {
     std::vector<int> a{1, 2, 4, 7, 11};
     std::vector<int> b;
     std::adjacent_difference(a.begin(), a.end(), std::back_inserter(b));
     // b = {1, 1, 2, 3, 4}
}
```

▶ `std::iota`

Заполняет диапазон последовательно возрастающими значениями, начиная с некоторого изначального. Не может быть параметризована.

```
int main() {
     std::vector<int> a(5);
     std::iota(a.begin(), a.end(), 1); // 1 2 3 4 5
}
```

Функциональные объекты

Содержимое `<functional>`

Как было упомянуто, STL также предоставляет ряд функциональных объектов, которые могут быть полезны, в частности, для параметризации алгоритмов — например, функтор можно использовать в качестве компаратора в `std::sort` или операции в `std::accumulate`. Они определены в заголовочном файле `<functional>` и, за исключением нескольких особо интересных образцов, представляют собой простейшие арифметические или булевы операции.

▶ `std::plus` / `std::minus` / `std::multiplies` / `std::divides` / `std::modulus` / `std::negate`

Трудно поверить, но эти функторы реализуют операции сложения, вычитания, умножения, деления, остатка от деления и отрицания соответственно.

```
int main() {
     std::vector<int> a{1, 2, 3, 4, 5};
     std::cout << std::accumulate(a.begin(), a.end(), 1,
std::multiplies());
}
```

▶ `std::equal_to` / `std::not_equal_to` / `std::greater` / `std::less` / `std::greater_equal` / `std::less_equal`

Совершенно невероятно, но эти функторы реализуют операции равенства, неравенства, строгого сравнения и нестрогого сравнения соответственно.

```
int main() {
     std::vector<int> a{1, 2, 3, 4, 5};
     std::sort(a.begin(), a.end(), std::greater()); // 5 4 3 2 1
}
```

▶ `std::logical_and` / `std::logical_or` / `std::logical_not`

Топ 10 фактов, в которые невозможно поверить: эти функторы реализуют операции конъюнкции, дизъюнкции и отрицания соответственно.

```
int main() {
     // я даже пример придумывать не буду
}
```

▶ `std::bit_and` / `std::bit_or` / `std::bit_xor` / `std::bit_not` (C++14)

Не расстраивайтесь, если это будет за гранью вашего понимания. Эти функторы реализуют побитовые операции конъюнкции, дизъюнкции, симметрической разности и отрицания соответственно.

```
int main() {
     std::vector<int> a{1, 2, 4, 8};
     std::cout << std::accumulate(a.begin(), a.end(), 0, std::bit_xor());
     // 15
}
```

▶ `std::not_fn` (C++17)

Возвращает отрицание результата выполнения переданной функции.

```
bool IsOdd(int x) {
     return x & 1;
}

int main() {
     std::vector<int> a{2, 4, 6, 8};
     std::cout << std::all_of(a.begin(), a.end(), std::not_fn(IsOdd)); //
true
}
```

▶ `std::hash` (C++11)

Встроенная хэш-функция. Базовый шаблон этого функтора живет именно в `<functional>`, хотя его специализации для конкретных типов могут быть разбросаны по самым различным библиотекам. `std::hash` использует алгоритмы достаточно умные, чтобы здесь их не разбирать.

```
int main() {
     std::hash<std::string> hashik;
     std::string a = "orange";
     std::string b = "orange";
     std::string c = "tail";
     std::cout << std::boolalpha << (hashik(a) == hashik(b)); // true
     std::cout << std::boolalpha << (hashik(a) == hashik(c)); // false
}
```

Заголовок `<functional>` также предоставляет несколько классов-обёрток, среди которых, в частности, можно выделить класс `std::function`, служащий полиморфной обёрткой над семейством функций с одинаковой сигнатурой:

```
void Foo1() {
     std::cout << "I'm Foo1!\n";
}

void Foo2() {
     std::cout << "I'm Foo2!\n";
}

int main() {
     std::vector<std::function<void()>> a{Foo1, Foo2};
```

```
    for (auto foo : a) foo();
}
```

`std::bind`

Другим примечательным объектом, который привносит заголовок `<function>`, является функция `std::bind`, позволяющая связать с функциональным объектом конкретные аргументы, которые он принимает. Для этого используются:

- константы;
- заполнители `_1`, `_2`, `_3` и т.д., которые отвечают за первый, второй, третий и т.д. аргументы, передаваемые при вызове; определены в пространстве имён `std::placeholders`;
- ссылки, обёрнутые в `std::ref` / `std::cref`.

Так, например, с помощью `std::bind` можно записать функцию, возвращающую квадрат целого числа:

```
auto Square = std::bind(std::multiplies<int>(), std::placeholders::_1,
std::placeholders::_1);
```

Интересно также, что, по-видимому, оператор `()` в функторе, полученном при помощи `std::bind`, реализован с использованием вариадических шаблонов. Это позволяет, в частности, не указывать аргументы, проигнорированные при задании сигнатуры: наш функтор `Square` может принимать единственное число, тогда как исходный `std::multiply` требовал два.

Использование `std::bind` достаточно красиво с точки зрения единообразия кода в STL, однако со временем от него отошли в пользу лямбда-функций, которые предоставляют сильно более надёжный, эффективный и понятный интерфейс, позволяющий реализовать ту же семантику.

Источники:

1. А. П. Хвастунов — Лекции по основам программирования на C++, 2 семестр, 2026
2. GeeksForGeeks — Standard Template Library in C++ (https://www.geeksforgeeks.org/cpp/the-c-standard-template-library-stl/)
3. Wikipedia — Standard Template Library (https://en.wikipedia.org/wiki/Standard_Template_Library)

4. cppreference.com — Container named requirements (https://en.cppreference.com/w/cpp/iterator/concepts.html#Container)
5. cppreference.com — Iterator library (https://en.cppreference.com/w/cpp/iterator.html)
6. cppreference.com — iterator_traits (https://en.cppreference.com/w/cpp/iterator/iterator_traits.html)
7. cppreference.com — Algorithm library (https://en.cppreference.com/w/cpp/algorithm.html)
8. Wikibooks — Erase-Remove Idiom (https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Erase-Remove)
9. cppreference.com — Standard library header `<functional>` (https://en.cppreference.com/w/cpp/header/functional.html)