STL - Стандартная библиотека шаблонов. Содержит набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Состоит из компонентов:

- Контейнеры
- Итераторы
- Адаптеры
- Алгоритмы
- Функциональный объект

Так как STL это библиотека шаблонов, то она позволяет использовать свои компоненты с пользовательскими типами, если они удовлетворяют требованиям шаблонов. И STL с каждым стандартом C++ расширяется, и пополняется новыми компонентами.

1. Контейнеры std::array и std::vector

Контейнеры, это объекты, позволяющие хранить данные в каком-то определенном виде. В STL есть последовательные контейнеры, такие как **std::array** и **std::vector**, и непоследовательные, такие **std::set** и **std::map**. Все контейнеры автоматически контролируют выделяемую память, и также очищают ее, если объект контейнера разрушен. STL позволяет работать через *итераторы* (будут рассмотрены позже), что делает работу с контейнерами унифицированной, и многие алгоритмы из библиотеки могут работать со стандартными контейнерами. Вот некоторые контейнеры, которые есть в STL:

- **std::array<T, N>** - массив фиксированный длины N типа T
- **std::vector<T>** - массив переменной длины типа T
- **std::list<T>** - список типа T. От std::vector отличается видом доступа к данным
- **std::map<Key, T>** - словарь с ключом тип Key и значением типа T
- **std::set<T>** - множество типа T. Все элементы имеют уникальные значения

std::array<T, N> - одномерный статический массив длины N, хранящий элементы типа данных T. Является оберткой над обычным статическим массивом вида T[N], поэтому сочетает в себе производительность чистых массивов в C-стиле и преимущества контейнеров STL. При создании объекта **std::array** память уже сразу выделена, и можно обращаться к элементам, то есть в нем что-то может

быть записано. Можно при создании сразу в фигурных скобках (списке инициализации) сразу задать элементы, которые будут помещены в контейнер. Если в списке инициализации указано меньше элементов, чем размер массива, то все элементы будут записаны в массив начиная с индекса 0. Начиная со стандарта C++17, на уровне языка была добавлена поддержка выведения (deducing) параметров шаблона. Поэтому, при создании **std::array** с помощью списка инициализации, компилятор сам определит тип элементов (они все должны быть одного типа) и размер массива.

Обращаться к элементам **std::array** можно используя оператор [ ], как у обычных массивов. В таком случае не будет проверки на диапазон, и если обратиться к элементу, индекс которого больше длины массива, то поведение не определено стандартом. Чтобы была проверка на диапазон, то нужно использовать метод **at**. Есть отдельные методы **front** и **back**, для получения первого и последнего элемента соответственно.

std::array как и многие другие контейнеры STL можно сравнивать используя операторы сравнения. Сравнение происходит в лексикографическом порядке по каждому элементу.

Для того, чтобы узнать размер массива, необходимо вызвать метод **size**. Данный метод есть почти у всех контейнеров STL.

std::vector<T> - динамический массив, размер которого можно менять во время жизни объекта. Вычислительная сложность доступа к элементам O(1), вставки и удаления в конец O(1), вставка и удаление не в конце O(n).

Создание и обращение к элементам аналогично **std::array** . Только при создании, не надо указывать в качестве параметра шаблона размер. При создании объекта, компилятор может вывести тип хранимых элементов из списка инициализации. Так как размер массива внутри **std::vector** не фиксирован, то при создании через список инициализации, будет проведено выделение памяти под поликчетсов элементов в списке.

Есть отдельные операции по добавлению в конец и удалению из конца одного элемента, это методы **push_back** и **pop_back** соответственно.

Помимо метода **size**, чтобы узнать длину массива, есть метод **capacity**, которые возвращает значение под сколько элементов выделена память. **std::vector** устроен так, что при добавлении элементов, если доступная память кончилась, то происходит расширение выделенной памяти сразу с

запасом, то есть при добавлении элементов по одному, не будет постоянного изменения размера выделенной памяти, а будет увеличение выделенной памяти примерно в 2 раза.

Для удаления всех элементов из массива используется метод **clear**, этот метод не уменьшает уже выделенную память, только меняет значение того, сколько элементов хранится.

Метод **resize** позволяет изменить количество текущих элементов. В случае если новый размер меньше старого, то элементы в конце будут считать удаленными. Если новый размер больше, то будет заполнение значениями по умолчанию, либо переданным a в качестве второго аргумента значением.

Метод **reserve** позволяет изменить количество выделенной памяти, может только увеличить выделенную память. Чтобы освободить неиспользуемую память, нужно вызвать метод **shrink_to_fit**.

Остальные методы по вставке и удалению элементов в **std::vector** будут рассмотрены после рассмотрения итераторов.

2. Итераторы

Итератор - поведенческий паттерн GOF. Предоставляет унифицированный подход к обходу коллекций, не раскрывая их внутренней структуры.

В STL все контейнеры возвращают итераторы с одинаковым интерфейсом, основные это операция разыменования *, чтобы получить элемент и операция инкремента ++ для перехода к следующему элементу. Таким образом, итератор можно рассматривать как указатель на какой-то элемент, и есть возможность перейти к следующему, вне зависимости от того, как элементы расположены в памяти: последовательно, в виде дерева или списка.

В тоже время, один и тот же контейнер можно обходить разным способом, используя один и тот же интерфейс. У почти всех контейнеров есть методы, которые возвращают итератор на начало и конец:

- begin() - возвращает итератор на начало
- end() - возвращает итератор на элемент после последнего
- rbegin() - возвращает итератор на конец
- rend() - возвращает итератор на элемент перед первым
- cbegin() и cend(), crbegin(), crend() - константные аналоги итераторов (нельзя менять элементы на которые они указывают)

Отдельно стоит отметить, что методы имеющие в названии слово end возвращают итератор на элемент после последнего. Попытка разыменовать такой итератор скорее всего приведет к ошибкам. И данный итератор нужен в первую очередь, чтобы была пройден весь контейнер, и дальше обходить не надо.

Методы, у которых в названии есть буква **r**, возвращают итераторы, которые при выполнении операции инкремента, будут идти от конца контейнера к началу.

Методы, у которых в названии есть буква **c**, возвращают константные итераторы. При помощи таких итераторов можно проходить по контейнеру, но изменять элементы нельзя, только читать их значение. Можно рассматривать это как указатель на константу.

Разные алгоритмы могут требовать от итераторов наличие операций кроме описанных выше. А также разные контейнеры могут возвращать итераторы с разными операциями. Поэтому в STL есть 6 категорий итераторов:

- **input_iterator_tag** – разыменуется как rvalue
- **output_iterator_tag** – разыменуется как lvalue
- **forward_iterator_tag** – проход только в одну сторону
- **bidirectional_iterator_tag** – проход в оба направления
- **random_access_iterator_tag** – можно переходить на произвольное кол-во элементов вперед и назад
- **contiguous_iterator_tag** (C++20) – тоже самое, что и random_access, но элементы в памяти расположены последовательно

То, какие операции есть у итераторов разных категорий представлено в таблице 1.

Таблица 1.

<table>
<tr><th colspan="4">Категория</th><th>Характеристика</th><th>Выражения</th></tr>
<tr><td colspan="4" rowspan="3">Все категории</td><td>Construct/Copy/Move/Delete</td><td>X b(a), b = a</td></tr>
<tr><td>Увеличение на 1</td><td>++a, a++</td></tr>
<tr><td>Разыменовывание</td><td>*a</td></tr>
<tr><td rowspan="8">Random</td><td rowspan="6">Bidirectional</td><td rowspan="4">Forward</td><td rowspan="2">Input</td><td>Сравнение на равенство</td><td>a == b, a != b</td></tr>
<tr><td>Разыменование как rvalue</td><td>*a, a->m</td></tr>
<tr><td>Output</td><td>Разыменование как lvalue</td><td>*a = o</td></tr>
<tr><td rowspan="5"></td><td>Множественный доступ</td><td>X a(b); ++a == ++b</td></tr>
<tr><td rowspan="4">Уменьшение на 1</td><td rowspan="4">--a, a--</td></tr>
<tr><td rowspan="4"></td></tr>
<tr><td>Можно использовать + и -</td><td>a + n, a -= n</td></tr>
<tr><td>Разыменование по индексу</td><td>a[n] == *(a + n)</td></tr>
<tr><td>Сравнение больше/меньше</td><td>a < b, a > b, a <= b, a >= b</td></tr>
</table>

Рассмотрим пример трех разных способов итерирования по вектору и вывод квадратов всех его значений (листинге 1).

Классический подход это обход массива также как в C. В цикле инициализируем переменную i, отвечающая за индекс в массиве. Выполняем цикл пока индекс меньше размера, увеличивая индекс на 1. В теле цикла происходит обращение к элементу вектора через операция [ ].

При использовании итераторов, инициализируется переменная **it** итератором на начало. Ключевое слово auto, вместо типа позволяет компилятору вывести тип данных на основании выражения справа от знака =. Цикл длится пока переменная it не станет равна итератору на конец, то есть не будет достигнут элемент после последнего. На каждом шаге операцией ++ увеличиваем переменную **it**, переходя к следующему элементу. В теле цикла доступ к элементу проводится путем разыменование итератора операцией *.

Автоматическое итерирование позволяет пройти по всему контейнеру. Сначала записывается тип и название переменной куда записывается элемент, затем после двоеточия контейнер, по которому нужно пройтись. В самом теле цикла уже доступ к элементу получать не надо, он записан в переменную.

Листинг 1.

```
std::vector<int> vec = {1,2,3,-4,5,-6};
 //"Классический" подход
 for(size_t i = 0; i < vec.size(); ++i)
   std::cout << vec[i] * vec[i] << ' ';

 //Использование итераторов
 for(auto it = vec.begin(); it != vec.end(); ++it)
   std::cout << (*it) * (*it) << ' ';

 //Автоматические итерирование
 for(int& elem: vec)
   std::cout << elem * elem << ' ';
```

При автоматическом итерировании можно пройти только от начала до конца контейнера, но запись получается краткой. При использовании итераторов, можно обходить контейнеры у которых нет операции взятия

индекса, или обойти в обратном порядке, или предварительно сместить итераторы и пройти не по всему контейнеру, но отсутствует доступ к индексу. Классический подход позволяет иметь информацию об индексе, а также контролировать шаг.

3. Контейнеры. std::vector продолжение, std::list, std::deque, std::map, std::set

Для вставки и удаления элементов в **std::vector** необходимо использовать итераторы. Для вставки элемента используется метод **insert**, куда передается итератор и значение, которое нужно вставить. Элемент вставляет **перед** элементом на который указывает итератор. Для удаления элементов используется метод **std::vector**, в который нужно передать итератор на элемент, который нужно удалить.

std::list<T> - является последовательным контейнером, и хранит элементы в виде двусвязного списка, то есть элементы в памяти будут расположены не последовательно. Во многом, работа с ним аналогична работе с **std::vector**, кроме произвольного доступа к элементу по индексу через оператор [ ] и метод **at**. Для вставки и удаления используются такие же методы **insert** и **erase,** а также методы вставки удаления в конец **push_back** и **pop_back**, и в начало **push_front** и **pop_front**.

std::forward_list<T> - последовательный контейнер, хранящий элементы в виде односвязного списка. Основные отличия от **std::list** это отсутствие методов **push_back** и **pop_back**, метод **insert_after** и **erase_after**, вставляют и удаляют элемент после элемента на который указывает итератор. Также, для данного контейнера нет обратного итератора.

std::deque<T> - последовательный контейнер, который хранит элементы в нескольких непрерывных участков памяти. Методы вставки и удаления аналогичны **std::list**, но сложность операций как у **std::vector**. В отличие от **std::vector** эффективнее выделяет память под новые элементы.

std::map<Key, T> - ассоциативный массив (словарь), который хранит элементы типа **T** по уникальным ключам типа **Key**. Хранение реализовано через красно-черное дерево, поэтому операции вставки, удаления и поиска имеются сложность O(log(n)). Итерирование происходит по возрастанию ключей. Метод **at** позволяет обратиться к элементу с проверкой наличия ключа в словаре. Обращение к элементам через оператор [ ], вернет элемент если указанный ключ существует. Если такой ключ отсутствует, то

будет создан элемент с таким ключом. При итерировании по **std::map**, итератор будет указывать на пару (ключ, значение), которые хранится в типе данных **std::pair**, где поле **first** - ключ, а поле **second** - значение. Элементы будут упорядочены в порядке возрастания ключей

std::unordered_map<Key, T> - аналогичен **std::map**, кроме того, что хранит данные в хеш таблице, у него отсутствует обратный итератор, и не гарантируется упорядоченность элементов.

std::set<T> - контейнер, который хранит только уникальные элементы в виде красно-черного дерева. Отсутствует методы для доступа к элементам. Для проверки наличия элемента в **std::set** используется метод **find**, который вернет итератор на элемент, если он будет найден, в ином случае будет возвращен итератор на конец. Методы **insert** и **erase** не принимают итераторы, а сразу значение, которые нужно вставить или удалить. Если попытаться вставить элемент, который уже есть в контейнере, то ничего не произойдет. Если попытаться удалить элемент, которого нет в контейнере, то ничего не произойдет. При итерированию по контейнеру элементы будут упорядочены по возрастанию.

std::unordered_set<T> - аналогичен **std::set**, но отсутствует обратный итератор, и не гарантируется упорядоченность элементов.

4. Адаптеры

Адаптеры - объекты, которые используют существующие компоненты, предоставляя альтернативный интерфейс.

Адаптеры контейнеров - используют существующий контейнер для хранения данных.

std::stack<T> - LIFO контейнер (стэк). По умолчанию внутри используют **std::deque**. В качестве второго аргумента шаблона требует контейнер у которого есть методы: **back, push_back, pop_back.** Из стандартных контейнеров также подходят **std::vector и std::list**

Методы:

- **top** - просмотреть верхний элемент
- **push** - поместить в стек
- **pop** - убрать из стека.

std::queue<T> - FIFO контейнер (очередь). По умолчанию использует **std::deque.** В качестве второго аргумента шаблона требует контейнер у которого есть методы: **back, front, push_back, pop_front.** Из стандартных контейнеров также подходит **std::list**

Методы:

- front - просмотреть начало очереди
- back - просмотреть конец очереди
- push - добавить в конец очереди
- pop - убрать из начала очереди.

std::priority_queue<T> - очередь с приоритетом. По умолчанию использует **std::vector**. В качестве второго аргумента шаблона требует контейнер у которого есть методы: **front, push_back, pop_back,** а также random_access_iterator**.** Из стандартных контейнеров также подходит **std::deque**

Методы:

- **top** - просмотреть начало очереди
- **push** - добавить в конец очереди
- **pop** - убрать из начала очереди

Адаптеры итераторов - также внутри содержат контейнеры, но имеют интерфейс итераторов. Необходимы для создания итератора, которые при выполнении операции присваивания, выполняет вставку в начало/конец/середину.

- **back_insert_iterator** - при присваивании ему, вызывает метод **push_back** у контейнера, создается функцией **back_inserter**
- **front_insert_iterator** - при присваивании ему, вызывает метод **push_front** у контейнера, создается функцией **front_inserter**
- **insert_iterator** - при присваивании ему, вызывает метод **insert** у контейнера, создается функцией **inserter**

Потоковые итераторы - позволяют работать с потом как с итератором.

i**stream_iterator<T>** - позволяет итерироваться по входному потоку. Параметр **T** определяет, какой тип данных считывается. В качестве аргумента конструктора передается поток для чтения. без аргумента, то итератор на конец.

std::ostream_iterator<T> - позволяет итерировать по выходному потоку. Параметр **T** определяет, какой тип данных выводится. В качестве

аргументов конструктора передается потом для записи и разделительную строку.

5. Алгоритмы

Стандартная библиотека предоставляет набор алгоритмов для работы с контейнерами следующих видов:

- for_each
- Поиск
- Копирование
- Обмен
- Преобразование
- Генерация значений
- Удаление
- Изменение порядка
- Случайный выбор значения
- Сортировка

Алгоритмы находятся в модуле **<algorithm>**. Так как STL предоставляет большой набор различных функций для работы с контейнерами, то будут рассмотрены только основные и широко-используемые.

Функция **for_each** применяет унарную функцию к диапазону итераторов. Применяется, когда надо вызвать функцию где каждый элементы будет аргументом этой функции. Если функция возвращает какое-то значение, то оно никуда не присваивается. Поэтому, если нужно изменять значения в контейнере, то унарная функция должна принимать в качестве аргументы ссылочный тип, либо в контейнере должен хранится указатель. **for_each** в качестве аргументов принимает итератор на начало диапазона, итератор на конец диапазона, и унарную функцию. Унарная функция может быть передана как функция, лямбда-выражение или функциональный объект (функтор). Есть аналогичная функция **for_each_n**, которая применяет функцию к первым n элементам (в таком случае функция принимает итератор на начало диапазона, n и унарную функцию). В STL, многие функции имеют альтернативу в виде с припиской _n, которая выполняет над n элементами, начиная с итератора.

Функции **all_of** / **any_of** / **none_of** принимают два итератора и предикат, и проверяют что все / хотя бы один / ни один элемент вернул **true** для для предиката. Функции принимают два итератора, начало и конец, а также предикат. Предикат - функция от 1 аргумента, которая возвращает булево значение (также может быть передана как лямбда-выражение и функтор).

Функция **find** ищет первый элемент совпадающий с переданным значением. Принимает два итератора на начало и конец диапазона, а также значение, которое необходимо найти. Функцич возвращают итератор на первый найденный элемент. Если не найден - то итератор на конец. Альтернативная функция **find_if** ищет первый элемент, для которого переданный предикат вместо значения вернет **true**. Как и с припиской _n, некоторые алгоритмы имеют приписку _if, такие функции принимает предикат, и выполняют алгоритм только для элементов, у которых предикат равен **true**.

Функция **copy** копирует диапазон значений в новое место используя итератор. Принимает три итератора: начало диапазон откуда копировать, конец диапазона откуда копировать, итератор куда вставлять копии элементов. Если 3-й аргумент итератор из того же контейнера, что и первые два, то поведение неопределено. В качестве третьего аргумента может быть итератор на элемент в каком-то контейнере (нужно убедиться, что не будет выхода за диапазон контейнер), может быть адаптер итератора для вставки в контейнер (безопасный вариант), потоковый выходной итератор (тогда данные будут выводится в поток). В качестве первых двух аргументов можно передать потоковый входной итератор, тогда данные будут сразу копироваться из входного потока сразу в контейнер. Версия **copy_if** скопирует только те элементы, для которых предикат истинный.

Функция **transform** применяет функцию в каждому элементу и записывает результат по итератору. Принимает три итератора, на начало и конец диапазона аргументов функции, итератор куда записывать результат, и функцию преобразования. Функция может проводить любые преобразования, даже к другому типу данных.

Функция **fill** заполняет диапазон заданным значением. Принимает 2 итератора на начало и конец диапазона, который необходимо заполнить, и само значение.

Функция **generate** заполняет диапазон согласно заданной функции. Принимает 2 итератора на начало и конец диапазона, который необходимо заполнить, и функцию, которая не принимает аргументы.

Функция **remove** удаляет все элементы равные заданному значению. Функция **remove_if** удаляет все элементы для которых предикат **true**. Функции принимают два итератора на начало и конец диапазона, из которого надо удалить элементы, и третьим аргументом само значения или предикат. Особенность реализации функции в том, что на самом деле функции не удаляют элементы из памяти, а лишь перемещают их в конец. В связи с этим, в качестве результата функции возвращают итератор, который является новым концом контейнера. Чтобы фактически удалить элементы из памяти, необходимо примять функцию **erase** самого контейнера.

Пример использования STL представлен в листинге 2. В данном коде: функция **change** принимает ссылку на **int** и делает его положительным, если оно отрицательное; **greaterThan10** предикат, который проверяет, значение **int**, если больше ли оно 10; **printStr** функция, которая преобразует число к строке, указывая порядок числа, в котором оно поступило функцию, и само значение. Шаблонная функция **example** в качестве аргументов принимает два шаблонных аргумента, предполагая, что будут переданы итераторы на начало и конец диапазона. В теле функции, сначала все отрицательные элементы диапазона преобразуются к положительным (**for_each** применяет функцию **change**). Далее происходит удаление всех элементов больше 10 (**remove_if** использует предикат **greaterThan10**). Оставшиеся элементы преобразуются в строку методом **transform** и функцией **printStr**, результат записывается в заранее созданный файловый поток. В функции main в функцию example передаются **std::vector** и **std::list**. За счет того, что функция **example** шаблонная, то можно передать итераторы контейнеров, которые имеют все необходимые операции для алгоритмов. Таким образом, получилась универсальная функция, выполняющая заданный набор действий.

```
#include<iostream>
#include<algorithm>
#include<vector>
#include<list>
#include<iterator>
#include<string>
#include<fstream>

void change(int& x){
  if(x < 0)
    x *= -1;
}
bool greaterThan10(int x){
  return x > 10;
}
std::string printStr(int x){
  static int num = 1;
  return std::string("Value Nº") + std::to_string(num++) + " is " +
std::to_string(x);
}

template<typename It>
void example(It begin, It end){
  std::for_each(begin, end, change);

  auto new_end = std::remove_if(begin, end, greaterThan10);

  std::ofstream file("output", std::ios_base::app);

  std::transform(begin, new_end,
                 std::ostream_iterator<std::string>(file,"\n"),
                 printStr);
}

int main(){
  std::list lst = {-5,2,-30,-4,-6,7,-20};
```

```
  std::vector vec = {-5,2,-30,-4,-6,7,-20};
  example(lst.begin(), lst.end());
  example(vec.begin(), vec.end());
}
```