* [Понятие контейнера и итератора в STL](https://foxford.ru/lessons/32265/conspects/1)
* [Алгоритмы из библиотеки algorithm STL](https://foxford.ru/lessons/32265/conspects/2)
* **Понятие контейнера и итератора в STL**
* **Понятие Контейнера и итератора**
* Контейнер — это класс STL, реализующий функциональность некоторой структуры данных, то есть хранилища нескольких элементов. Примеры разных контейнеров: vector, stack, queue, deque, string, set, map и т.д.
* Различные контейнеры имеют различные способы доступа к элементом. Например, vector и deque предоставляют так называемый "произвольный доступ" ("random access"), позволяющий работать с любым элементом контейнера, обращаясь к нему по индексу, между тем как stack и queue позволяют обращаться только к крайним элементам контейнера.
* Для обращения к элементам контейнеров существует понятие итератора. Итератор является обобщением идеи доступа к элементу по индексу и обобщением указателей языка C. Можно рассматривать итераторы, как "умные" указатели.
* **Объявление и использование итераторов**
* Итератор - это специальный класс, связанный с соответствующим классом контейнера.
* Если, например, имеется контейнер
* **vector<int>**
* , то итератор, которым можно "бегать" по контейнеру будет объявляться так (it - имя, которое мы даем итератору):
* **vector<int>::iterator it;**
* У контейнеров есть два метода, которые возвращают итератор на начало контейнера (метод begin()) и итератор на фиктивный элемент, следующий за концом контейнера (метод end()).Основные операции, которые можно выполнять с любыми итераторами:
* == - проверка двух итераторов на равенство.
* != - проверка двух итераторов на неравенство.
* ++ - инкремент (увеличение итератора), то есть переход к следующему элементу контейнера.
* — - декремент (уменьшение итератора), то есть переход к предыдущему элементу контейнера.
* Операторы являются "указателями", то есть чтобы получить доступ к значению элемента, на который указывает итератор, его нужно разыменовать при помощи унарного оператора "\*".
* Пример вывода всех элементов контейнера при помощи итератора:
* **for (vector<int>::iterator it = a.begin(); it != a.end(); ++it)**
* **cout << \*it << " ";**
* Здесь мы объявляем итератор, присваиваем ему значение, которое возвращает метод begin(), то есть становимся в начало вектора, затем увеличиваем итератор, пока не выйдем на фиктивный элемент в конце вектора, который возвращает метод end(), при выводе значения нужно разыменовывать итератор при помощи операции "\*".
* **Итераторы контейнера vector**
* Вектор - это контейнер, элементы которого хранятся в памяти последовательно, и индексируются начиная с 0. Поэтому итераторы векторов поддерживают дополнительную функциональность.
* К итератору вектора можно прибавлять целое число k, что означает перемещение на k элементов. Если значение k<0, то перемещение осуществляется в сторону начала вектора.
* Таким образом, чтобы получить итератор на k-й элемент вектора от начала, можно взять итератор, который вернет метод begin() и прибавить к нему k.
* Эта особенность итераторов широко используется в разных алгоритмах. Например, алгоритм сортировки sort() получает два итератора - на первый элемент и на элемент, следующий за последним. Для сортировки вектора a обычно делают такой вызов:
* **sort(a.begin(), a.end())**
* Но используя операции сложения итератора с числом можно задать произвольный фрагмент вектора для сортировки. Например, чтобы отсортировать весь вектор, не трогая последний элемент:
* **sort(a.begin(), a.end() - 1)**
* Чтобы отсортировать вектор. не трогая первый и последний элемент:
* **sort(a.begin() + 1, a.end() - 1)**
* Чтобы отсортировать фрагмент вектора из 10 элементов, начиная с элемента с индексом 3:
* **sort(a.begin() + 3, a.begin() + 13)**
* Чтобы отсортировать 10 последних элементов вектора:
* **sort(a.end() - 10, a.end())**
* Из одного итератора можно вычитать другой итератор. Например, разница между итераторами begin()+7 и begin()+2 будет равна числу 5. А разница между итераторами end() и begin() будет равна количеству элементов в векторе.
* Итераторы вектора можно сравнивать при помощи неравеств <, >, <=, >=, которые будут возвращать true или false в зависимости от того, какой элемент находится раньше (имеет меньший адрес.
* У вектора есть методы erase и insert, которые позволяют удалять и вставлять элементы в вектор, используя итераторы.
* **erase**
* Метод erase удаляет один элемент или последовательность элемента из вектора. Для удаления одного элемента нужно дать итератор на этот элемент. Например, для удаление первого элемента вектора:
* **a.erase(a.begin());**
* а для удаления последовательности передается два итератора - на первый элемент и на элемент, следующий за последним элементом. То есть вызов:
* **a.erase(a.begin() + k1, a.begin() + k2);**
* удалит k2 - k1 элемент начиная от a[k1] (включительно) и до a[k2] (не включительно).
* **insert**
* Метод insert позволяет вставить в середину вектора один элемент, несколько равных элементов, или фрагмент этого же и другого вектора.
* Первый параметр метода insert() - итератор, указывающий позицию для вставки.
* Остальные параметры могут быть следующими.
* Если указан еще один параметр, то вставляется одно значение, равное этому. Например:
* **a.insert(a.begin() + 5, val)**
* вставляет значение val в элемент a[5] вектора. То, что ранее было в элементе a[5] и далее сдвигается вправо.
* Метод insert с тремя параметрами insert(pos, n, val) вставляет n значений, равных val.
* А метод insert с тремя параметрами insert(pos, it1, it2) вставляет в позицию pos фрагмент вектора начиная с итератора it1 до итератора it2 (разумееется, не включая it2).
* Пример такого использования, который удваивает вектор:
* **a.insert(a.end(), a.begin(), a.end())**
* **reverse\_iterator**
* У вектора и некоторых других контейнеров есть понятие reverse\_iterator - это итератор, который движется в обратном порядке.
* Метод rbegin() возвращает reverse\_iterator на последний элемент контейнера. Метод rend() возвращает reverse\_iterator на фиктивный элемент, перед первым элементом контейнера.
* Инкремент reverse\_iterator приводит к движению к началу контейнера.
* Пример использования reverse\_iterator для вывода элементов контейнера в обратном порядке:
* **for (vector<int>::reverse\_iterator it = a.rbegin(); it != a.rend(); ++it)**
* **cout << \*it << " ";**
* **auto-тип в C++11**
* В новом стандарте языка C++ 2011 года (называется C++11) появилось понятие auto-типа. В этом случае не требуется объявлять тип переменной явно, можно указать, что переменная имеет тип auto и проинициализировать переменную значением. В этом случае компилятор сам определит тип переменной (автоматически) исходя из типа значения, которым она проинициализирована.
* Например,
* **auto x = 1; // переменная x будет типа int**
* **auto y = 1.0; // переменная y будет типа double**
* Как правило auto-типы используются для итераторов, например, можно писать цикл так:
* **for (auto it = a.begin(); it != a.end(); ++it)**
* **Цикл по значению контейнера в C++11**
* В С++11 появилась возможность органи-зации range-based циклов (то, что называется циклом "foreach"), когда переменная принимает последовательно все значения из данного контейнера.
* Например, если объявить
* **vector<int> a;**
* то вывести все его элементы можно при помощи цикла:
* **for (int elem: a)**
* **cout << elem << " ";**
* Или можно использовать auto-тип:
* **for (auto elem: a)**
* **cout << elem << " ";**
* В данном случае elem будет принимать все значения из контейнера a, который может быть вектором, множеством, деком и т.д. Но чтобы модицифицировать элементы такого контейнера при помощи цикла нужно сделать цикл, в котором переменной цикла была бы ссылка на элемент контейнера, а не значение. Это можно сделать так (все элементы контейнера увеличиваются на 1):
* **for (auto & elem: a)**
* **++elem;**
* **Алгоритмы из библиотеки algorithm STL**
* **Общие подходы**
* Заголовочный файл algorithm содержит много полезных алгоритмов.
* Большинство из этих алгоритмов принимают в качестве параметра два итератора, будем обозначать их first и last. В этом случае алгоритм работает с элементами контейнера от first включительно до last невключительно. Чаще всего в качестве first используется метод begin(), а в качестве last - метод end(), в этом случае алгоритм применяется ко всему контейнеру. Например,
* **sort(a.begin(), a.end());**
* Используя операции "+" и "-" для итераторов можно применять алгоритмы не для всего контейнера, а для части.
* Можно передавать также reverse\_iteraror, наиболее употребительный способ использования - это сортировка в обратном порядке при помощи:
* **sort(a.rbegin(), e.rend());**
* **Алгоритмы поиска**
* **find**
* Алгоритм
* **find(first, last, val)**
* осуществляет линейный поиск значения
* **val**
* от итератора
* **first**
* до итератора
* **last**
* . Элементы просматриваются последовательность, возвращается итератор на первый найденный элемент. Если элемент
* **val**
* не будет найден, то возвращается значение итератора
* **last**
* .
* **binary\_search**
* Алгоритм
* **binary\_search(first, last, val)**
* осуществляет двоичный поиск значения
* **val**
* . Контейнер должен быть упорядочен. Возвращается значение типа
* **bool**
* , то есть
* **true**
* или
* **false**
* в зависимости от того, есть ли такой элемент в контейнере.
* **lower\_bound**
* Алгоритм
* **lower\_bound(first, last, val)**
* осуществляет двоичный поиск значения
* **val**
* и возвращает итератор
* **res**
* на первый элемент, который не меньше, чем
* **val**
* , то есть
* **\*res>=val**
* , a
* **\*(res-1)<val**
* . Если все элементы контейнера (начиная с
* **first**
* ) будут не меньше, чем
* **val**
* , то будет возвращено значение
* **first**
* . Если в контейнере все элементы меньше
* **val**
* , то возвращается значение
* **last**
* .
* **upper\_bound**
* Алгоритм
* **upper\_bound(first, last, val)**
* осуществляет двоичный поиск значения
* **val**
* и возвращает итератор
* **res**
* на первый элемент, который строго больше, чем
* **val**
* , то есть
* **\*res>val**
* , a
* **\*(res-1)<=val**
* . Если же все элементы контейнера (начиная с
* **first**
* ) будут больше, чем
* **val**
* , то будет возвращено значение
* **first**
* . Если в контейнере все элементы меньше или равны
* **val**
* , то возвращается значение
* **last**
* .
* **Алгоритмы сортировки, разворота, сдвига**
* **sort**
* **sort(first, last)**
* - упорядочивает элементы контейнера по неубыванию.
* **stable\_sort**
* **sort(first, last)**
* - упорядочивает элементы контейнера по неубыванию, при этом равные элементы не переставляются (так называемая "устойчивая сортировка").
* **reverse**
* **reverse(first, last)**
* - разворачивает фрагмент контейнера в обратном порядке, переставляя элементы, равноудаленные от концов.
* **rotate**
* **reverse(first, n\_first, last)**
* - осуществляет циклический сдвиг фрагмента контейнера. Элемент, на который указывает итератор
* **n\_first**
* становится первым элементом (то есть переходит на место элемента
* **first**
* ), элемент
* **n\_first+1**
* - вторым и т.д.
* **Перестановки**
* **next\_permutation**
* **next\_permutation(first, last)**
* - переставляет элементы так, чтобы получилась следующая в лексикографическом порядке перестановка. Можно применять не только к векторам, но и к строкам (как и многие другие алгоритмы). Метод возвращает
* **true**
* , если удалось построить следующую в лексикографическом порядке перестановку. Если же первоначальная перестановка уже была максимальной в лексикографическом порядке, то метод генерирует минимальную в лексикографическом порядке перестановку и возвращает
* **false**
* .
* Например, вывести все перестановки в лексикографическом порядке можно так:
* **do**
* **{**
* **for (auto x: a)**
* **cout << x;**
* **cout << endl;**
* **}**
* **while(next\_permutation(a.begin(), a.end());**
* **prev\_permutation**
* **prev\_permutation(first, last)**
* - переставляет элементы так, чтобы получилась предыдущая в лексикографическом порядке перестановка. Можно применять не только к векторам, но и к строкам (как и многие другие алгоритмы). Метод возвращает
* **true**
* , если удалось построить предыдущую в лексикографическом порядке перестановку. Если же первоначальная перестановка уже была минимальной в лексикографическом порядке, то метод генерирует максимальную в лексикографическом порядке перестановку и возвращает
* **false**
* .
* **random\_shuffle**
* **random\_shuffle(first, last)**
* - делает случайную перестановку элементов контейнера.
* **Минимальные и максимальные элементы, подсчет**
* **min\_element**
* Алгоритм
* **min\_element(first, last)**
* находит минимальный элемент в контейнере и возвращает итератор на этот элемент. Если есть несколько элементов, равных минимальному, возвращается значение первого из них.
* **max\_element**
* **max\_element(first, last)**
* возвращает итератор на наибольший элемент. Если есть несколько элементов, равных наибольшему - то на первый из них.
* **count**
* **count(first, last, val)**
* — подсчитывает сколько элементов контейнера равны значению
* **val**
* .