Оглавление

[Общие понятия 2](#_Toc130420929)

[Vector 3](#_Toc130420930)

[Итераторы 5](#_Toc130420931)

[List 8](#_Toc130420932)

[Postfix vs Prefix (iterators) 9](#_Toc130420933)

# Общие понятия

STL – Стандартная библиотека шаблонов

Большинство контейнеров STL – реализация динамических структур данных

Набор стандартных решений для частоиспользуемых задач

Контейнер – набор однотипных элементов

Основные методы у каждого из контейнеров одинаковые: иногда некоторые методы добавляется, некоторые убираются в зависимости от типа

# Vector

Вектор – улучшенный динамический массив

Не нужно заботится о выделении и освобождении памяти. Всё происходит автоматически.

Выделяется новая память с запасом, когда добавляется новый элемент. Можно сказать, что вектор резервирует память под данные, которые будут занесены в будущем. (происходит при достижении максимального числа элементов в capacity )

Создание вектора выглядит следующим образом:

**std::vector <тип данных> название\_вектора**

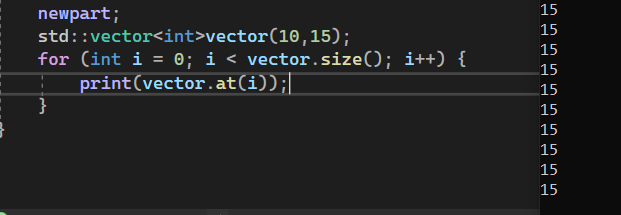


Как и с обычным динамическим массивом можно инициализировать его следующим образом



Если нужно инициализировать вектор числами X K-раз, то можно использовать следующую конструкцию

**std::vector<тип данных> имя\_вектора(K, X)**



**Методы вектора:**

* vector.push\_back(x) – добавление x в конце массива
* vector.pop\_back() – удаление последнего элемента
* vector.size() – возвращает количество элементов массива
* vector.capacity() – возвращает зарезервированные ячейки памяти
* vector.reserve(x) – резервирует x ячеек памяти.
* vector.shrink\_to\_fit() – убирает зарезервированную область памяти.
* vector.at – аналогично vector[x], только добавляет ограничение, чтобы не выйти из области доступной памяти (проверяет границы), но работает немного медленнее
* vector.clear() – очищаем массив
* vector.empty() – «пустой ли массив?» да = 1, нет = 0.
* vector.resize() – изменяет размер контейнера
* vector.insert(iterator,num) – вставка элемента по определенному индексу.
* vector.erase(iterator) – удаление i-го элемента
* vector.end() – указывает на элемент после конца вектора

# Итераторы

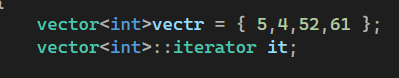
Итератор – сущности, которые нужны для взаимодействия с контейнерами STL

Поведение итераторов похоже на поведение указателей (объектная оболочка над указателями)

При работе с итераторами используется арифметика указателей

В большинстве контейнеров stl оператор [ ] не перегружен, поэтому необходимо использовать итераторы (в векторе он перегружен)

Для создания итератора необходимо создать объект такого типа, который использует наш контейнер. Например:



* + - * vector<int> – тип данных
      * it – имя оператора
        + Проще использовать кл.слово **auto**

Однако такой итератор будет бесполезен. Необходимо еще связать его с нашим вектором.

Например, нам надо пробежаться по всем элементам. Тогда вызываем следующий итератор



Функция .begin() вернет как раз таки итератор с типом данных нашего вектора. Именно по этому важно, чтобы тип данных и тип контейнера у нашего объекта и итератора совпадал.

Если говорить про наш случай, то он вернет **указатель** **на начало** **массива**.

Вывод нашего массива на экран (!не использовать квадратные скобки!)



если сократить, то будет выглядеть так



**Константный итератор**



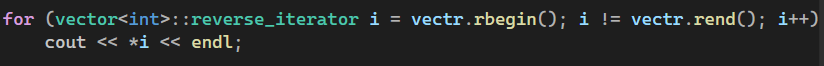
реализуется также, как и обычный итератор, но командой **const\_iterator**

В цепочке наследования обычный итератор находится после константного итератора, поэтому конфликта не возникнет, если присваивать константному итератору обычный. (**нельзя наоборот**)

если дописать в начале метода “c”, то вернется константный итератор



**Обратный итератор**



Всё, что сказано для константных итераторов справедливо и для реверснуты

**Функция advance**

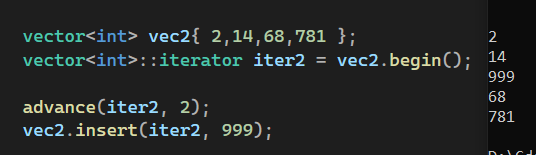
В некоторых типах контейнеров мы не можем перескакивать между данными, т.е. мы не можем прибавить 2 (по арифметики указателей) к итератору и получить второй элемент.

Для решения вышесказанной проблемы используется метод **advance**, который принимает первым параметр итератор, который нужно сдвинуть, а второй параметр – это шаг, на сколько нужно сдвинуть.

На самом деле это всего лишь надстройка над арифметикой указателей.

 **==** 

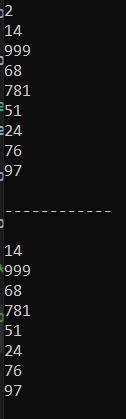
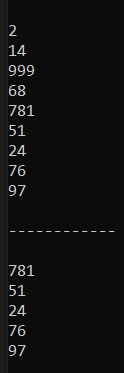
Вставка на второй место второго элемента значения 999 (остальные смещаются вправо. Замены значения не происходит)



*Бонусом прикреплю сюда удаление первого элемента*



До/после



Также работает с диапазонами. До/после ->



# List

List (список) – реализация библиотеки stl двусвязного списка (**см С++(DDS)**).

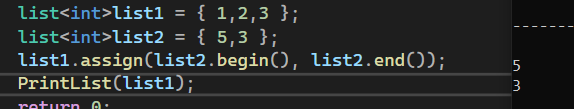
Очень быстро работает с **удалением и добавлением** элементов, но затруднена и зммедлена итерация данных.

Можно итерироваться **только** при помощи **итераторов**

Как и вектор является **шаблонным**

**Методы:**

* list.push\_back(x) – добавление в конец элемента х
* list.push\_front(x) – добавление в начало элемента x
* list.pop\_back() – удаление последнего элемента
* list.pop\_front() – удаление первого элемента (**!ломает итератор!**)
* list.insert(index,data) – вставляет data на место index
* list.erase(iterator) – удаляет элемент на который указывает iterator
* list.remove(data) – удаляет элемент со значением data
* advance(iterator,num) – смещает итератор на num элементов
* list.assgn(k,data) – удаляет прежние элементы листа и добавляет k раз значения data (так можно копировать листы, указав через итераторы начало и конец)



* list.sort() – сортировка элементов
* list.reverse() – переворачивает лист
* list.size() – выводит размер листа
* list.unique() – удаляет последовательные одинавые элементы (оставит 1 подобный элемент)
* list.clear() – удаление всех значений в листе

# Postfix vs Prefix (iterators)

Postfix: (отличия от обычного нет)

* Создается временная переменная, которая ссылается на наш объект
* Прибавляется (отнимается) единица к самому объекту
* Возвращается временная переменная **до** изменения

Prefix:

* Прибавляется едиинца к нашему объекту
* Возвращается ссылка на наш объект.

Префикс работает быстрее. Лучше использовать его.

Пример, где можно ошибиться при написании.



# forward list

**Односвязный** список

У него нет методов пушбэк и попбэк. В основном используется пушфрон и попфронт (операции в начале листа). Если нам нужно проводить операции в конце списка, используем обычный лист

**методы:**

* fl.before\_begin() – область памяти **перед** самым первым элементов листа