Оглавление

[Общие понятия 2](#_Toc131758228)

[Vector 3](#_Toc131758229)

[Итераторы 5](#_Toc131758230)

[List 8](#_Toc131758231)

[Postfix vs Prefix (iterators) 9](#_Toc131758232)

[forward list 10](#_Toc131758233)

[Array 12](#_Toc131758234)

[Операторы сравнения 13](#_Toc131758235)

[SET-MULTISET 14](#_Toc131758236)

[Deque 16](#_Toc131758237)

# 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

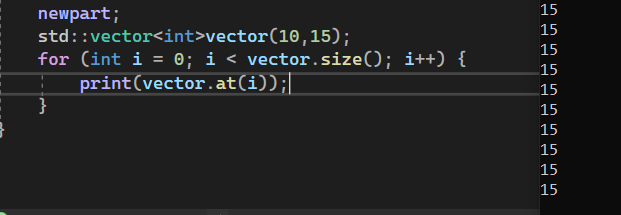


Как и с обычным динамическим массивом можно инициализировать его следующим образом



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

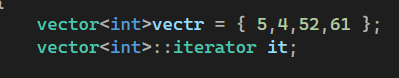
Итератор – сущности, которые нужны для взаимодействия с контейнерами STL

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



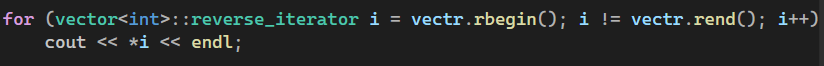
реализуется также, как и обычный итератор, но командой **const\_iterator**

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

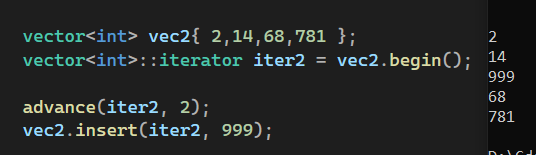
В некоторых типах контейнеров мы не можем перескакивать между данными, т.е. мы не можем прибавить 2 (по арифметики указателей) к итератору и получить второй элемент.

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.

 == 

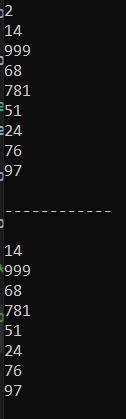
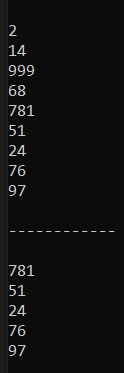
Вставка на второй место второго элемента значения 999 (остальные смещаются вправо. Замены значения не происходит)



Удаление первого элемента



До/после



Также работает с диапазонами. До/после ->



# List

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

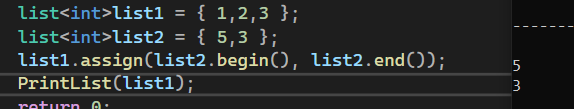
Очень быстро работает с **удалением и добавлением** элементов, но затруднена и зммедлена итерация данных.

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

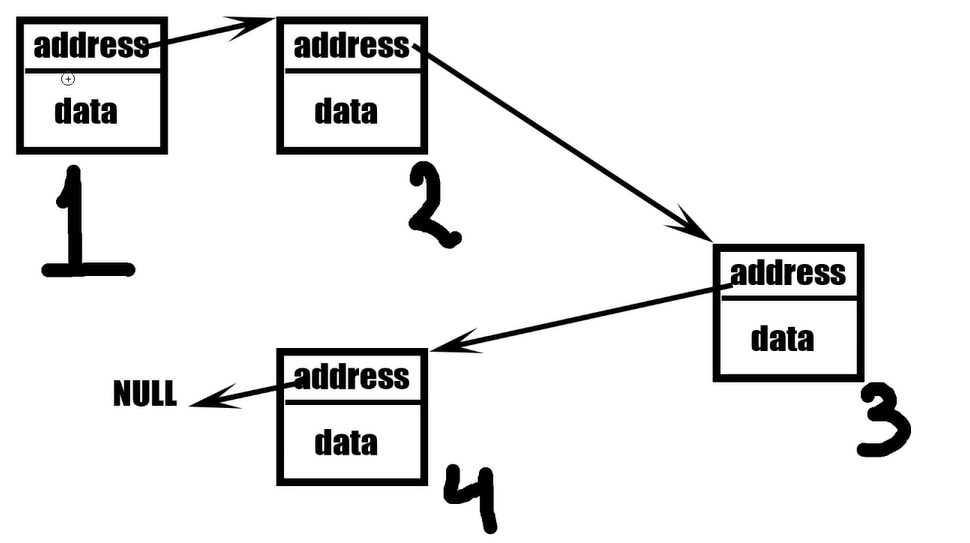
iterator++; сработает

--iterator; не сработает

отличительные методы:

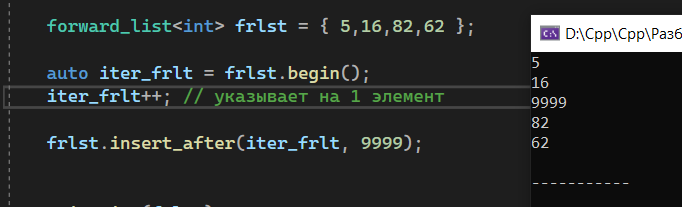
* fl.before\_begin() – область памяти **перед** самым первым элементов листа
* fl.insert\_after()
* fl.insert\_after
* fl.emplace\_after
* fl.emplace\_front

Данные методы связаны с тем, что односвязный список не хранит ссылку на предыдущее своё значение, поэтому необходимо проводить операции с последующей ячейкой



То есть, чтобы вставить наше значение на позицию 2, мы должны будем взаимодействовать с первым элементом, так как он знает адрес прежнего элемента с индексом 2. Мы сперва убирать связь между 1 и 2, в адрес у первой ячейки вставляем адрес адрес нашей ячейки, а адрес, который хранился там раннее, мы записываем в поле адреса нашей текущей ячейки

Пример insert\_after() приведен ниже



before\_begin() нужен для взаимодействия с первым (нулевым) элементом списка

# Array

Как и вектор является **обёрткой для массива**, но только не для динамического, а для **статического**

Синтаксис выглядит следующим образом



Нельзя добавить или удалить элементы. Даже метода сортировки нет.

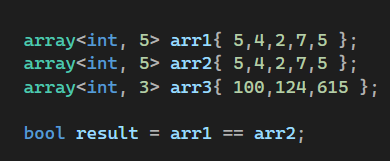
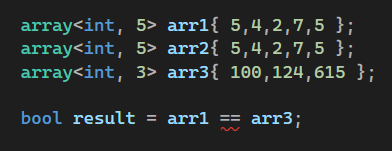
Важные методы:

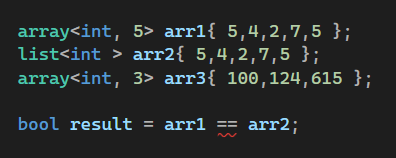
* arr.fill(x) – заполнение массива элементами х
* arr.front() – доступ к последнему элементу массива
* arr.back() – доступ к последнего элементу массива

# Операторы сравнения

Условия, что и с обычными контейнерами

* Должно совпадать кол-во элементов
* Должен быть один тип данных
* Есть критерий по которому сравнивать (в интах по величине элементов)
* Должен быть перегружен оператор сравнения
* **Нельзя** сравнивать разные типы контейнеров(лист и вектор, например)





Операции такие же, как и с обычными контейнерами

* > / >=
* < / <=
* ==

# SET-MULTISET

Ассоциативные контейнеры.

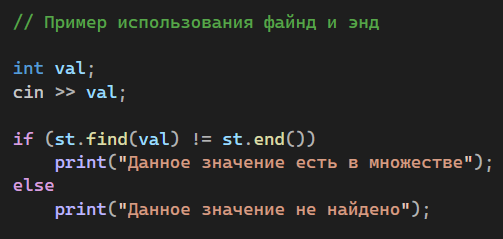
Реализованы на основе **бинарного дерева** -> все элементы упорядочены и не может хранить два одинаковых значения (ошибку не вызовет, но не будет добавлен новый экземпляр. В **multiset можно добавить**)

Нет методов пушбэк, пушфронт, нет перегруженных [ ]

Контейнер **set** **не может изменить** конкретное число, поэтому нам надо сперва **удалить** старый элемент, а потом **добавить** новый.

Методы set (почти все подходят и для мультисет):

* set.insert(x) – добавление х в наш set (не нужен итератор, т.к. бинарное дерево/ возвращает 2 элемента: сам x и булевую переменную, которая будет true, если элемент добавился успешно и false, если он не добавился)
* set.erase(x) – удаление x из нашего сета (если х нет в set, не будет ошикби)
* set.begin()
* set.swap()
* set.find(x) – ищет элемент со значением х и возвращает его итератор. если не найдет, то будет указывать на end



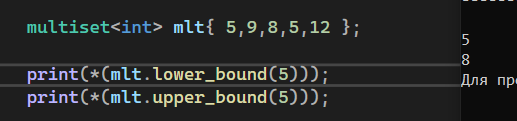
* set.clear()

Отличие **сета** от **мультисета** уже было сказано: в сете все значения уникальны, а в мультисете могут быть дубликаты.

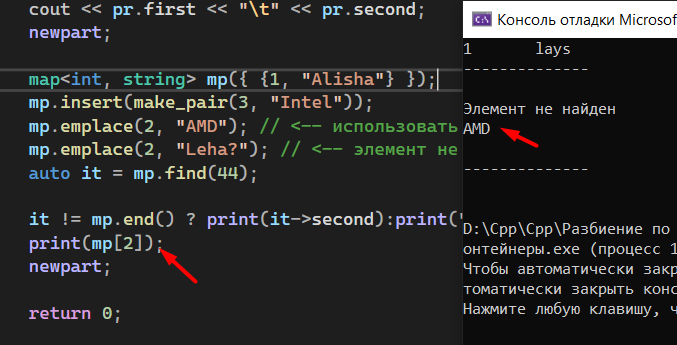
**Методы**:

* multiset.lower\_bound(x) – найдет первый элемент со значением х
* multiset.upper\_bound(y) – вернет следующий элемент после y по возрастанию
* multiset.equal\_range(x) – возвращает диапазон значений от lower\_bound до upper\_bound (x,y)
* map.at(x) - работает также, как и оператор [ ], но не добавляет значение, если оно не находится
* map.erase(x) – удаление ключа Х и его пары.

Пример работы баундов в мультисете



Перегружен оператор [ ] – ищется по ключу и выводит его пару



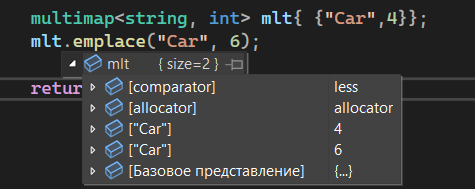
оператор [ ], если не находит элемент, то добавляет его в множество

mp[3] = 5; // создали элемент 3, которого не было в массиве

mp[3] = 100; // назначили элементу 3 значение 100

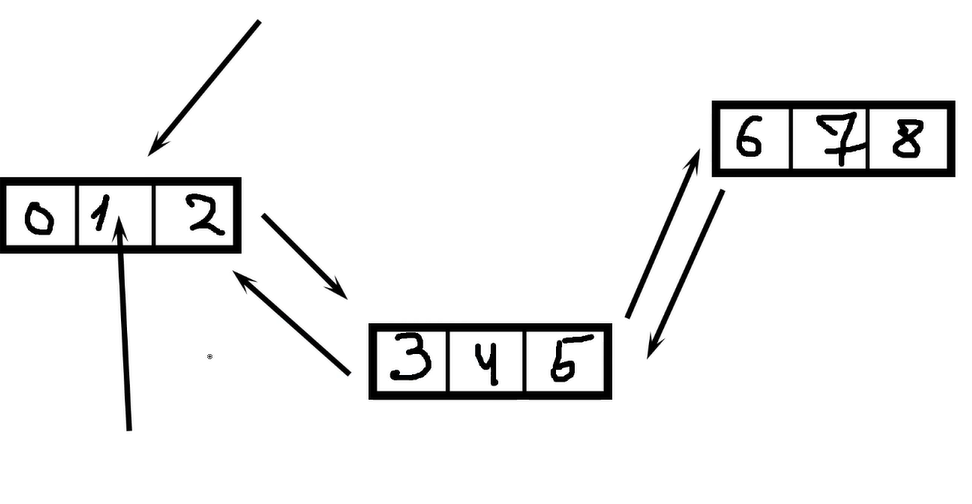
метод map.at() работает также, но не добавляет значение, если оно не находится

**Multimap –** может хранить два и более одинаковых ключа. Не перегружен оператор [ ] и метод at



# Deque

Двустороняя очередь (см. С++(DSS) файл) – гибрид вектора и двусвязного списка (свусвязный список векторов)



Был сделан, чтобы сгладить недостатки списка с доступок к элементам

По маленьким векторам быстро итерируемся, но замедляемся при переходе между массивчиками (между элементами 2 и 3, 5 и 6 в нашем случае)

Можем добавлять элементы в начало не беспокоясь о том, что весь массив будет копироваться. Также тут опускается операция в векторе, которая, когда кончается капасити, выделяет новую область памяти, куда копирует старые значения.

В нашем случае будет добавляться маленький вектор в начале и в конце. Чуть медленее, чем лист, но быстрее намного, чем вектор.

Логику довольно сложно реализовать. Затрачивает больше ресурсов.



# Map|Multimap (словарь)

Как и SET контейнер MAP упорядоченный и имеет структуру бинарного дерева

В отличии от SET, который хранит одно значение и строится в виде структуры бинарного дерева логически, map и multimap хранят пару: ключ-значение.

Map – хранит уникальные

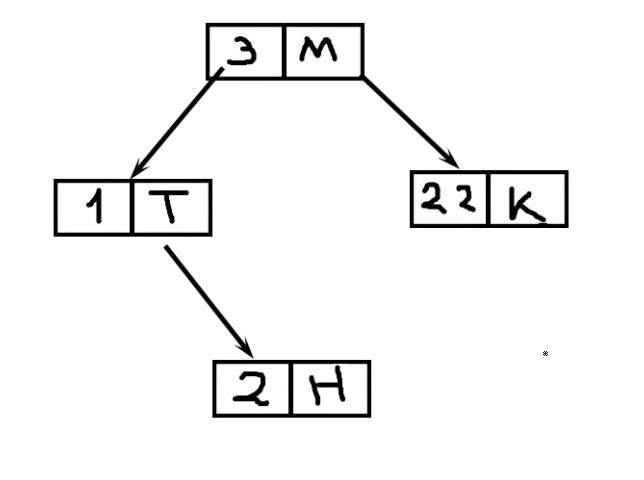
Multimap – хранит дубли

Упорядочивание этого контейнера осуществляется на основе ключа.

Map состоят из типов **pair,** задача которой связать 2 элемента

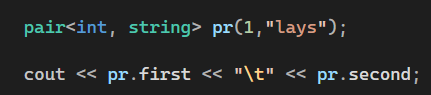
Упорядочивание идёт **по ключу.** Второй элемент идет как бы дополнением.

*Пример* построения бинарного дерева структуры map:



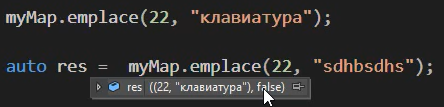
Методы пары (**pair**):

* pair.first
* pair.second
* make\_pair(x,y) – создает pair(x,y)

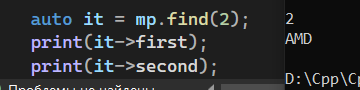


**Методы map:**

* map.emplace(x,y) – вставка в словарь пары значений x:y
  + Если такая пара уже имеется возвращает итератор со значениями ключ:false



* map.find(x) – поиск по ключу. Возвращает *итератор*, который содержит пару ключ-значение



(если *не смог найти* код ниже выведет ошибку, т.к. он вернет mp.end() – итератор, который возвращает кусок памяти за контейнером)

