# Повторение.

# Что будет? Как будет?

- Будет 5 лабораторных работ (*предположительно*):
  1) Методы поиска пути в графе (+ бфс-дфс-жадные алгоритмы)
  2) Максимальный поток в сети
  3) Поиск с возвратом (+динамическое программирование)
  4) Префикс-функция
  5) Конечноавтоматный Ахо-Корасик

  Бонус: немного современного C++
- Основные парадигмы и идеи при построении алгоритмов
- Без строгих доказательств, разбор некоторых задач
- Если хватит времени:
  - Список с пропусками
  - Хеш-таблицы и Фильтр Блюма

# Главный вопросы – как сдать лабу?

- В первую очередь – читаемость.
- Использование ООП – необязательно. НО! Следует использовать стратегию RAII для работы с ресурсами.
- Языки. Основные примеры, детали реализации будут на C++. Поэтому рекомендуется C++, можно также Python, остальное – по согласованию.
- Форматирование кода – единый стиль. *Google style-guide* или *а-ля golang* (C++).
- Знание и понимание исходного кода – обязательно. Есть возможность списать, но при этом не понимаете исходников? Лучше напишите самостоятельно, даже если получится хуже, чем у соседа.
- Знание и понимание изучаемых и используемых концепций, алгоритмов, структур данных и операций с ними – обязательно. Успели накидать работающий код, но при этом не разобрались с теорией? Лучше сдайте в следующий раз.
- Программы пишутся для людей, не для компьютеров

# Ещё чуть-чуть про читаемость кода

- Активно занимаемся декомпозицией логики, не стесняемся создавать функции

```
if ((str[1] ⩽ '9' && str[1] ⩾ '0') || (str[1] ⩽ 'Z' && str[1] ⩾ 'A') || (str[1] ⩽ 'z' && str[1] ⩾ 'a'))
```

```
if (isLetter(str[1]) || isNumber(str[1])) {
```

- Не экономим на спичках
- Явное лучше чем неявное
- Константное лучше, чем неконстантное
- Держим в голове: преждевременная оптимизация – корень всех зол.
- Не используем механизм исключений в стандартной бизнес-логике

# Ещё чуть-чуть про читаемость кода

- Максимально ограничиваем область видимости переменных.
- Максимально ограничиваем время жизни переменных.
- Никаких глобальных переменных! *(кроме констант или конфигураций запуска).*
- Не экономим на названиях переменных. Выбираем осмысленные имена.

```
int recordsToHadle;
float avg;
size_t deletedRecords;

avg = getAverage(taxiDB);

// много кода ...

while (recordsToHadle ≠ 0) {
    // ...
    recordsToHadle--;
}

// много кода ...

deletedRecords = countOldRecords(taxiDB);
```

```
float avg = getAverage(taxiDB);

// много кода ...

int recordsToHadle = findNumberOfRecordsToHandle(taxiDB);
while (recordsToHadle ≠ 0) {
    // ...
    recordsToHadle--;
}

// много кода ...

size_t deletedRecords = countOldRecords(taxiDB);
```

# Обзор контейнеров STL

Контейнеры + алгоритмы

- Стандартная библиотека **шаблонов** - универсальность
- Требования к скорости выполнения задач в реализациях комбинаций структур и алгоритмов описаны в стандарте языка C++
- Ожидаемые структуры и оценки работы прямиков из CS
  (в отличие от python-контейнеров)
- Общая методология и подход к созданию программ
- Используя стандартное решение мы избавляем от головной боли читателей нашего кода: обмен мыслями между разработчиками происходи быстрее
- Если где-то закралась ошибка, то у нас уже как минимум нет надобности проверять корректность STL – бородатые инженеры на зарплате уже всё проверили и оптимизировали до нас

# Ещё чуть-чуть про читаемость кода

- Активно изучаем и используем в возможности STL.

Задача: просуммировать числа в векторе. Как решаем?

## Колхоз

```
std::vector<int> numbers {1, 2, 3, 4, 5};
int summ = 0;
for (int i = 0; i < numbers.size(); i++) {
    summ += numbers[i];
}
```

## Уже лучше

```
std::vector<int> numbers {1, 2, 3, 4, 5};
int summ = 0;
for (const auto &elem : numbers) {
    summ += elem;
}
```

## Блистательно

```
std::vector<int> numbers {1, 2, 3, 4, 5};
int summ = std::accumulate(std::rbegin(numbers),
                           std::rend(numbers),
                           0);
```

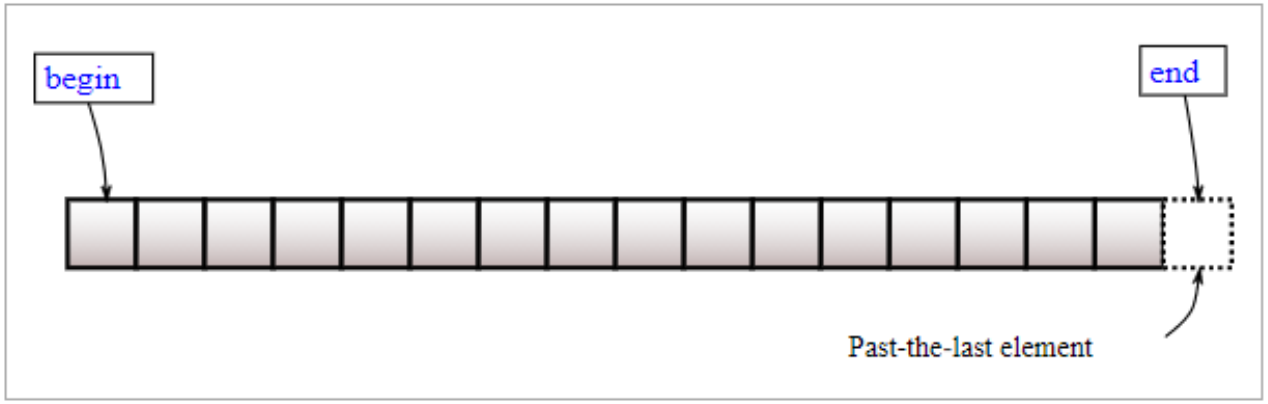
## Преисполнился в сознании

```
std::vector<int> numbers {1, 2, 3, 4, 5};
int summ {0};
std::for_each(numbers.rbegin(), numbers.rend(),
              [&summ] (int elem) {
                  summ += elem;
});
```

# Линейные структуры данных (SequenceContainer)

Динамический массив – **std::vector / python - list**

- Добавление в конец – амортизированная O(1) (что за амортизация?)
- Добавление/удаление в общем случае – ужасная O(n) (почему ужасная?).
- Доступ по порядковому номеру элемента – O(1). Fast RandomAccessIterator.
- Важность. Применение метода reserve(). Отличие от метода resize().
- Единственный смежный контейнер (не считая **std::array**) – меньше кеш-промахов при итеративной обработке данных, нет фрагментации данных.




Изменение концов: push/pop_back/front.
Изменение общий случай: insert, erase.

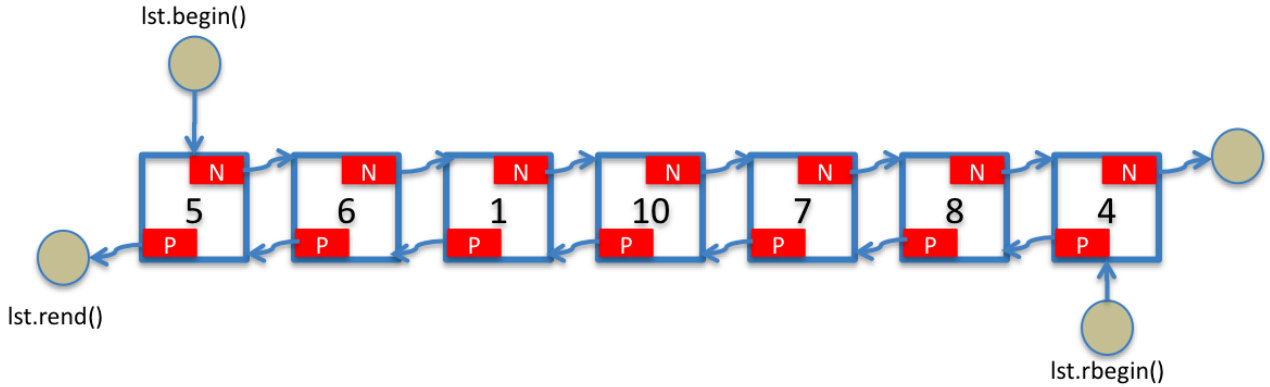
# Линейные структуры данных (SequenceContainer)

Список – **std::list**. Односвязный список – **std::forward_list**.

- Просто изменение структуры списка по концам – O(1) без подвохов.
- Доступ напрямую – O(1).
- Доступ к элементу по индексу – O(n) + костыль.

Стоит отметить, что при активном заполнении контейнера перемещаемыми данными – использование вектора предпочтительнее списка. Это происходит за счёт уменьшения количества системных вызовов при выделениях динамической памяти.

- В python на основе списка реализована структура deque



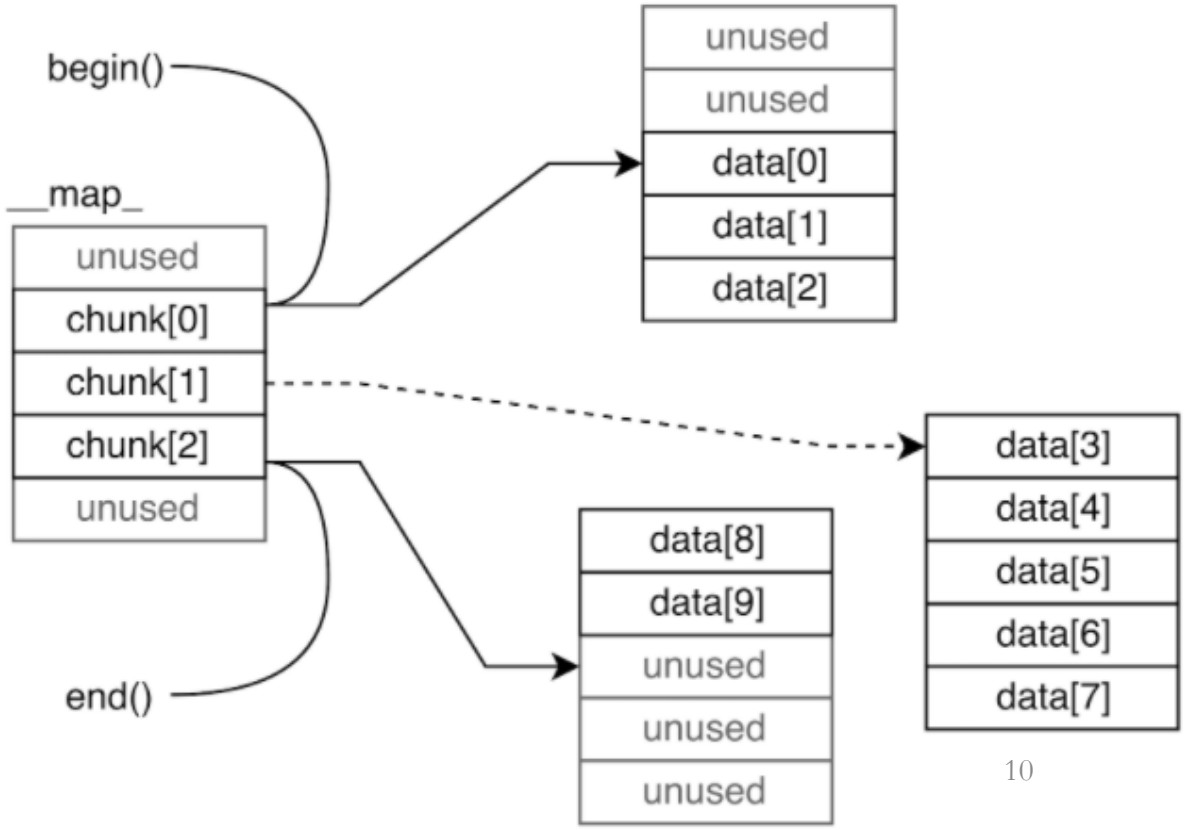
# Линейные структуры данных (SequenceContainer)

Дек – **std::deque** (двусторонняя очередь). **Python – deque.**

- Изменение контейнера по обоим концам – O(1) без подвохов.
- Доступ по индексу – O(1) с маленьким подвохом.
- Добавление и удаление в общем – O(n).
- Реализация?

**STL** – список массивов

**Python** – двусвязный список



# Линейные структуры данных. Что в итоге.

Список – **std::list**. Односвязный список – **std::forward_list**.

- Просто изменение структуры списка по концам – O(1) без подвохов.
- Доступ напрямую – O(1).
- Доступ к элементу по индексу – O(n) + костыль.

Стоит отметить, что при активном заполнении контейнера перемещаемыми данными – использование вектора предпочтительнее списка. Это происходит за счёт уменьшения количество системных вызовов при выделениях динамической памяти.

**Trade-offs / usage notes**

| | |
|---|---|
| std::array | Fast access but fixed number of elements |
| std::vector | Fast access but mostly inefficient insertions/deletions |
| std::list<br>std::forward_list | Efficient insertion/deletion in the middle of the sequence |
| std::deque | Efficient insertion/deletion at the beginning and at the end of the sequence |

# Контейнеры-адаптеры

Мы можем выбрать базовый контейнер, поверх которого реализуется требуемая функциональность. Всего таких адаптеров 3 штуки:

- Стек (LIFO)– **std::stack**. (поверх std::deque / std::vector / std::list).
- Очередь (FIFO) – **std::queue**. (поверх std::deque / std::list).
- Очередь с приоритетом – **std::priority_queue**. (поверх std::vector/ std::deque).

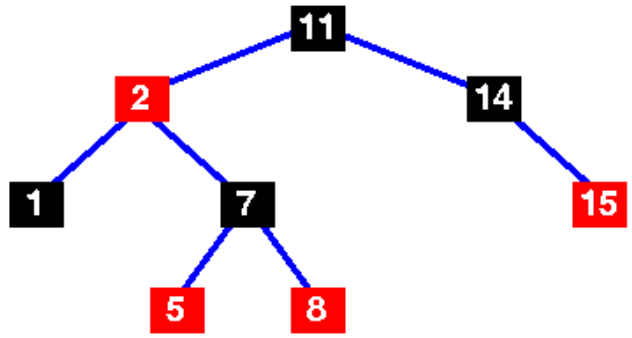
**Python:** Очередь с приоритетом – heapq, stack/queue – ручная адапатция list/deque

- Быстро извлекаем максимум/минимум – O(1).
- При этом жертвуем временем вставки и удаления – O(log N).
- Реализуется поверх двоичной кучи (пирамиды).

Удобно использовать, если не требуется менять приоритеты уже лежащих в очереди элементов. Иначе можно использовать «сырые» **std::make_heap**, **std::push_heap**, …

Пригодится во работе №2. Там же и разберём подробнее.

# Ассоциативные контейнеры - 1




**std::set** – хранит упорядоченный набор ключей.

- Упорядоченное множество уникальных элементов.
- Каждый ключ уникален (иначе используем std::multiset).
- Внутри лежит красно-черное дерево.
- Следовательно: поиск, удаление, вставка – O(log N).
- Для работы требуется компаратор <.

**std::map** – хранит упорядоченный набор пар ключ-значение.

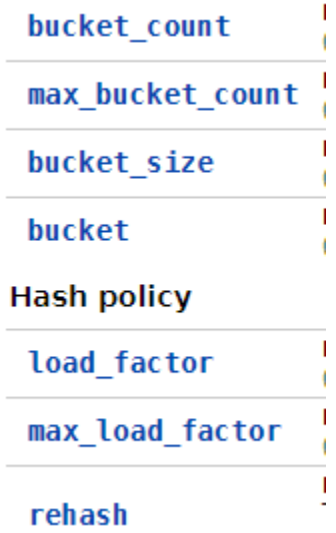
- Упорядоченный словарь. За порядок отвечают ключи.
- Каждый ключ уникален (иначе используем std::multimap).
- Остальное – идентично std::set.

Что нужно учитывать при использовании: в std-python нет затраты на хранение и динамическую природу памяти для новых узлов.

# Ассоциативные контейнеры - 2

**std::unordered_set** – хранит неупорядоченный набор ключей.

- Является ***множеством*** в классическом его понимании (python **set**).
- Каждый ключ уникален (иначе используем std::unordered_multiset).
- Внутри лежит хеш-таблица. Причём кишками наружу.
- Следовательно: поиск, удаление, вставка – avg-O(1).
- Ключи внутри контейнера менять нельзя.




**std::unordered_map** – хранит неупорядоченный набор пар ключ-значение.

- Является ***словарём*** в классическом его понимании (python **dict {}**).
- Каждый ключ уникален (иначе используем std::unordered_multimap).

Аккуратно работает с итераторами при изменении контейнеров.

# Вместо заключения

- Инвестиция времени в изучение основных алгоритмов и структур данных – с лихвой окупится
- Решение и анализ алгоритмических задач – хорошая тренировка как программирования на языке, так программирования с использованием языка
- Лучше знать и понимать причины, чем наизусть помнить их следствия
- В бою: сначала думаем о необходимости оптимизаций
- Выигрывая в скорости, проигрываем в памяти или в сложности
- Простите за PW