Стандратная библиотека шаблонов C++

Плотников Даниил Михайлович

Санкт-Петербургский государственный университет

Оглавление

Организационные моменты . 3

Ввод/Вывод 4

Работа с данными 6

Статические массивы 7

Динамические массивы 8

Итераторы 9

Заголовочный файл <algorithm> 13

Передача функций. Лямбды. 15

std::pair и std::tuple 21

Структуры данных . . . 25

std::string 26

std::set и std::unordered_set 28

std::map и std::unordered_map 33

FIFO и LIFO структуры 35

std::bitset 44

Прочее 45

Организационные моменты

- Про весь материал сегодняшней лекции можно почитать подробнее на cppreference.
- Попрактивовать применения структур данных можно на acmp

Ввод/Вывод

- C++ имеет высокий уровень обратной совместимости с C. В результате можно использовать как заголовочный файл `<iostream>`, так и `<stdio>`.
- При печати выгодно не писать символ по одному, а сразу целый буфер данных.
- Совокупность этих выктов вынуждает `cin` и `cout` синхронизировать буфер с `printf()` и `scanf()`, что приводит к замедлению работы. Для отключения этого поведения нужно в начале функции `main()` написать `std::sync_with_stdio(0)`.
- Так же входной и выходной буфер имеют определённую синхронизацию. Перед каждым вводом(`cin`) выходной буфер(`cout`) очищается. Это поведение так же можно отключить `std::cin.tie(0)`.

  Не забывайте про это поведение при работе с интерактивными задачами.
- Вызвать очиску буфера можно вручную `cout.flush()` / `cin.flush()`

  Когда вы пишете `std::endl` для переноса строки вы так же неявно вызываете очистку буфера. Это может оказаться критичным если в задаче много вывода. Рекомендуется писать `cout << "\n";`

Работа с данными

Статические массивы

- Для масива фиксированной длины можно использовать стандартную структуру `int a[n]`, где `n` - переменная, объявленная при помощи ключевого слова `const`, либо литерал.

  Некоторые компиляторы позволяют так создать массив с не известной на этапе компиляции длиной, но это поведение не является стандартом языка.
- Так же можно создать массив в стиле ооп при помощи структуры `std::array<int, n> a(value)`, где value - значение которым надо заполнить массив изначально.
- К `std::array` можно применять ряд методов:
  - `a.back()` - получить последний элемент массива;
  - `a.front()` - получить первый элемент массива;
  - `a.at(idx)` - получить элемент массива на позвиции `idx` с проверкой на вылезание за границу массива.
  - `a.size()` - получить размер массива.

Динамические массивы

Если же размер массива не известен на этапе компиляции, то можно создать вектор(динаимческий массив): `vector<int> v(n, value)`.
К вектору можно применять методы применимые к `std::array`, а так же:

- `v.push_back(value)` - добавить значение в конец массива. Асимптотика — $O(1)$.
- `v.emplace_back(value)` - добавить значение в конец массива. В отличии от `push_back()` создаёт объект сразу на месте и не вызывает `move()`. Почти не влияет, но иногда чуть быстрее. Асимптотика — $O(1)$.
- `v.insert(pos, value)` - добавить значение на позицию `pos`. Асимптотика — $O(n)$.
- `v.emplace(pos, value)` - добавить значение на позицию `pos`. Асимптотика — $O(n)$.
- `v.pop_back(value)` - удалить значение в конца массива.
- `v.erase(pos)` - удалить значение на позиции `pos`. Асимптотика — $O(n)$.

Итераторы

У нас есть 3 способа описать массив с практически одинаквыми интерфейсами взаимодействия. На них можно применять одни и те же алгоритмы при помощи одних и тех же функций, меняя лишь тип данных в аргументах. Но что если нам нужно оперировать разными структурами, но выполнять одну и ту же задачу? Например поиск элемента? Хотелось бы иметь одну и ту же функцию `find()` и для массива, и для списка и для дерева.
Для этого существую итераторы. Разберёмся с ними на примере `int a[n]`. Будет писать итератор подходящий для функции `find()`.
Рассмотим как выглядит поиск элемента для класического a[n].

- Как узнать где находится первый элемент массива? Это адрес массива `a`.
- Как узнать значение элемента массива? Предположим что за указателем `i` находится элемент массива `a`. Тогда `*i` — значение этого элемента.
- Как происходит доступ к следующему элементу массива? Предположим что за указателем `i` находится элемент массива `a`. Тогда следующий элемент массива это `i+1`.
- Как понять что мы дошли до конца массива? Все элементы в массиве хранятся последовательно. Значит следующий за последним элементом массива будет находиться в ячейка с адресом `a+n`.

```
int target; //искомый элемент
//...
for(int* i = a; i < a+n; ++i) {
  if(*i == target){
    return i;
  }
}
```

Попробуем абстрагироваться от конкретного указателя и представим такой тип данных:

- Мы можем узнать какой итератор соответствует первому элементу структуры при помощи функции `a.begin()`
- Мы можем получить данные лежашие за итератором вызвав оператор `*i`
- Мы можем получить следующий итератор при помощи функции `i.next()`
- Мы можем узнать какой итератор соответствует концу структуры при помощи функции `a.end()`.

Тогда наша функция будет выглядеть как:

```
for(iterator i = a.begin(); i != a.end(); i = i.next()) {
  if(*i == target){
    return i;
  }
}
```

Пользуясь уже реализованным классом `std::iterator` из c++ цикл будет выглядеть подобным образом:

```
for (std::vector<int>::iterator it = a.begin(); it != a.end(); ++it)
```

Или же для статического массива:

```
for (std::array<int, 5>::iterator it = a.begin(); it != a.end(); ++it)
```

Написание подобного является очень громоздким. Поэтому в c++ есть синтаксических сахар для класических ForEach циклов.

```
for (int i : a)
```

Тогда в переменной `i` у нас будет **копия** элемента. Если хотим изменять, то пишем `int& i`.

Иногда описание типа тоже является громоздким, тогда пишем:

```
for(auto& el : a)
```

Заголовочный файл `<algorithm>`

Существенная часть типовых алгоритмов уже реализованы в стандратной библиотехе шаблонов используя итераторы. Аргументами здесь является отрезок итераторов $[l, r)$

- `find(..., val)`, `find_if(..., val)`, `find_if_not(..., val)` – найти первое входжеие значения
- `find_last(..., val)`, ... – найти последнее вхождение значение
- `count(..., val)`, `count_if(..., val)` — посчитать количество вхождений значения
- `fill(..., val)` – заполнить значением
- `reverse(...)` – развернуть элементы
- `max_element(...)` – максимальный элемент
- `min_element(...)` – минимальный элемент

- `next_permutation(...)` – получить лексикографически следующую перестановку
- `accumulate(..., init)` — посчитать сумму элементов. Начальное значение – `init`
- `sort(...)` — отсоритровать
- `lower_bound(..., val)` — возвращает итератор первого элемента не меньшего значения в отсортированном массиве за $O(\log n)$
- `upper_bound(..., val)` — возвращает итератор первого элемента большего значения в отсортированном массиве за $O(\log n)$
- `binary_search(..., val)` — проверяет есть ли элемент со значением `val` в остортированно массиве за $O(\log n)$

Так же в этой библиотеке есть пару полезных функций по типу `max(a, b)`, `min(a,b)`.

Передача функций. Лямбды.

Иногда при работе с этими функциями требуется изменить операцию. Например вместо суммы получить произведение всех чисел. Функции для которых это логично принимают так же дополнительный аргумент(который впрочем можно опустить). Разберём этот пример.

```
int multiply(int lhs, int rhs) {
    return lhs*rhs;
}

int main(){
    vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int product = std::accumulate(v.begin(), v.end(), 1, multiply);
    cout << "product: " << product << '\n'; //product: 3628800
}
```

Иногда написание функции далеко от места её вызова может запутать. В таком случае можно использовать лямбда функции. Для этого примера это будет выглядеть так.

```
int main(){
    std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int product = std::accumulate(v.begin(), v.end(), 1,
      [](int lhs, int rhs){return lhs*rhs;}
    );
    std::cout << "product: " << product << '\n';
}
```

О ужас! `auto a = [](){};` это реальное выражение из языка c++... Давайте разбираться что это под капотом.

Пусть структура представляющая лямбда функцию будет хранить данные, к которым она имеет доступ из вне, и саму функцию. Давайте напишем такую структуру для этого примера:

```
int value = 42;
int& ref_to_value = value;

auto lambda = [value, &ref_to_value](int offset) {
  ref_to_value = 100;
  return value + offset;
};
```

Лямбда хочет получить доступ к переменным `value` и `ref_to_value`, вызвать функцию с агрументом `int offset` и телом:

```
  ref_to_value = 100;
  return value + offset;
```

Называется наша выдумка замыканием, и для примера выглядит подобным образом:

```
class __CompilerGeneratedClosureType {
private:
    int m_captured_value;
    int& m_captured_ref;

public:
    __CompilerGeneratedClosureType(int value, int& ref_to_value)
        : m_captured_value(value), m_captured_ref(ref_to_value) {
    }

    int operator()(int offset) const {
        m_captured_ref = 100;
        return m_captured_value + offset;
    }
};
```

И так, резюмируя страшный `[](){}`:

- внутри `[]` располагаются “захваты”. Можно задать дефолтное поведение если написать `[=]`, чтобы захватить все доступные переменные по значению, и `[&]`, чтобы захватить все значение по ссылке. Если есть исколючения то можно написать `[&, value]`.
- внутри `()` располагаются аргументы функции.
- внутри `{}` располагается тело функции.
- вызвать такую функцию можно так же как и обычную применив к объекту `operator()`

Особенно часто использование таких функций пригождается при сортировке. Изначально функция `sort()` сортирует массив по неубыванию, но нам может быть удобно сортировать по невозрастанию. Тогда хочется сделать так:

```
sort(a.begin(), a.end(),
  [](const auto &lhs, const auto &rhs){return lhs > rhs;}
)
```

Писать свою функцию вовсе не обязательно. Для этого есть итераторы обратного порядка:

```
sort(a.rbegin(), a.rend())
```

Так наша функция будет думать что массив изначально развёрнут и она ставит наименьший элемент на позицию `0`, однака фактически происходит наоборот.

`std::pair` и `std::tuple`

Множество алгоритмов требуют групировку данных для своей работы. Например, метод сканирующей прямой. Тогда можно написать свою структуру:

```
struct event{
  int moment;
  int type;
}
```

Чтобы использовать функции из заговочного файла `<algorithm>` потребуется реализовывать различные методы (например, `operator<` для сортировки), или обязаельно передавать функцию (например, аккумулятор для `accumulate()`). Чтобы этого не делать можно использовать уже реализованные класс `std::pair`.

В нашем примере этот класс будет выглядеть как `std::pair<int, int>`. Создать объект можно либо через функцию `std::make_pair(a,b)` либо через конструктор `{a,b}`

Объявить пару можно следующим образом:

```
std::pair<int, int> p;
p = {1,2};
```

Обратиться к переменным внутри пары можно следующими образам:

```
int f = p.first, s = p.second;
```

либо

```
auto [f, s] = p;
```

Последнее особенно удобно внутри циклов ForEach:

```
std::vector<std::pair<int, int>> v;
//...
for(auto [f,s] : v){
  //...
}
```

Но что если нам нужно сгрупировать не 2 переменные, а 4? Это будет выглядеть как `std::pair<int std::pair<int, std::pair<int, int>>>`. Разумеется так делать никто не хочет и не делает. Для такого есть класс `tuple<int, int, int int>`.

Аналогичная работа с ним:

```
std::tuple<int, int, int, int> t{1, 2, 3, 4};
int fi = std::get<0>(t);
int se = std::get<1>(t);
int th = std::get<2>(t);
int fo = std::get<3>(t);
auto [first, second, third, fourth] = t;
for(auto [a,b,c,d] : t) {
  //...
}
```

Для этих классов не реализованны операции по типу суммы, но есть сравнения при помощи загаловочного файла `<functional>` и объектов `std::greater<pair<int,int>>`, `std::less<...>` и т.д.

Применение будет выглядеть следующим образом:

```
std::sort(v.begin(), v.end(), std::greater<pair<int,int>>())
```

Для невозрастающей последовательности можно просто опустить аргумент функции.

Структуры данных

`std::string`

В языке C строки представляли собой не более чем массив из символов с `'\0'` в конце. Однако это не очень удобно во многом и тяжело унифицировать относительно других данных. Поэтому в c++ добавили структуру `std::string`. Для неё реализован объектно ориентированный интерфейс в виде методов:

- Сравнения: `==`, `!=`, `<`, `>`, `<=`, `>=`;
- Конкатенация: `+`, `+=`;
- Поиск и замена: `.find()`, `rfind()`, `replace()`;
- Подстроки: `substr()`;
- Доступ к элементам: `[]`, `.at()`, `.front()`, `.back()`
- Размер: `.size()`, `length()`, `capacity()`

Строки реализацию поиска элемента, псокольку мы можем искать не только один символ, но и подстроку, что требует других, более сложных алгоритмов

Обычно поиск выглядит вот так:

```
if(std::find(s.begin(), s.end(), 'a') != s.end()){
  //Ура! Нашли!
}
```

Но для строк так же можно написать:

```
if(s.find('a') != std::string::npos){
  //Ура! Нашли!
}
```

`std::set` и `std::unordered_set`

`std::set` представляет собой математическое множесто. Под капотом реализовано при помощи красно-чёрного дерева.

`std::unordered_set` представляет собой математическое множесто. Под капотом реализовано при помощи хэш таблицы.

Как можно заметить, это очень похожие структуры которые отличаются только реализацией под капотом и как результат деталями использования. Сейчас речь пойдёт об общем применении математического множества.

Пример использования множества:

```cpp
std::set<int> s;
//std::unordered_set<int> s;
s.insert({1, 2, 7, 2 42, 5, 64});
if(s.find(1) != s.end()){
    std::cout << "hooray!\n";
}
if(s.find(3) == s.end()){
    std::cout << "oh :(\n";
}
s.erase(7);
for(auto el : s){
    std::cout << el << " ";
}
```

Вывод для `std::set`

```
hooray!
oh :(
1 2 5 42 64
```

Вывод для `std::unordered_set`

```
hooray!
oh :(
64 5 42 2 1
```

Сравнение `std::set` и `std::unordered_set`

n - мощность множества на момент операции

Операция	Асимптотика в худшем случае		Асимптотика в среднем	
	std::set	std::unordered_set	std::set	std::unordered_set
Добавление	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
Удаление	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
Поиск	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$

Помимо разницы в асимтотиках, `std::set` так же выполняет одну дополнительную функцию – данные в нём упорядочены, в то время как для `std::unordered_set` расположение данных непредсказуемо.

Использование `std::unordered::set` может привести к неожиданному превышению времени выполнения из-за особенностей хэшей. Во избежании этого стоит пользоваться своим хэшем. Подробнее можно почитать в блоге на codeforces.

Как `std::set`, так и `std::multiset` при добавлении повторного элемента теряют его. Если это не поведение которые требуется в задаче, то следует использовать `std::multiset` и `std::unordered_multiset`.

```
std::multiset<int> s;
s.insert({1, 1, 2, 2, 5, 5, 64});
for(auto el : s){
    std::cout << el << " ";
}
std::cout << '\n';
s.erase(2);
if(s.find(2) == s.end()){
  std::cout << "oh :(\n";
}
s.extract(5);
for(auto el : s){
    std::cout << el << " ";
}
```

Вывод

```
1 1 2 2 5 5 64
oh :(
1 1 5 64
```

Если порядок элементов важен, и при этом он должен отличаться от стандартного, то можно передать компаратор в качестве аргумента шаблона при создании множества:

```
std::set<int, std::less<int>> s;
```

`std::map` и `std::unordered_map`

Иногда хочется иметь такой массив, в котором индексами будут являться не целые положительные числа, а, например, строки. Для такого используются ассоциативные массивы, словари или же математические отображения. Всё это выполняет роль одной и той же структуры.

В c++ эту функцию выполняют `std::map` и `std::unordered_map`. Отличия в них аналогичны отличиям `std::set` и `std::unordered_set`, так что это сравнение опустим и перейдём к применению. Точно так же сущестуют классы `std::multimap` и `std::unordered_map`, но лично мне всегда удобнее было хранить вектор значений.

Изменение порядка элементов в `std::map` тоже работает аналогичным образом.

```cpp
map<string, int> m;
m["three"] = 3;
m["seven"] = 7;

for(auto [key, value] : m){
    cout << key << "->" << value << '\n';
}
cout << '\n';

cout << m["three"] << ' ' << m["one"] << '\n';

string check[2] = {"present", "not present"};
cout << check[m.find("one") == m.end()] << '\n'
     << check[m.find("two") == m.end()];
```

Вывод

```
seven->7
three->3

3 0
present
not present
```

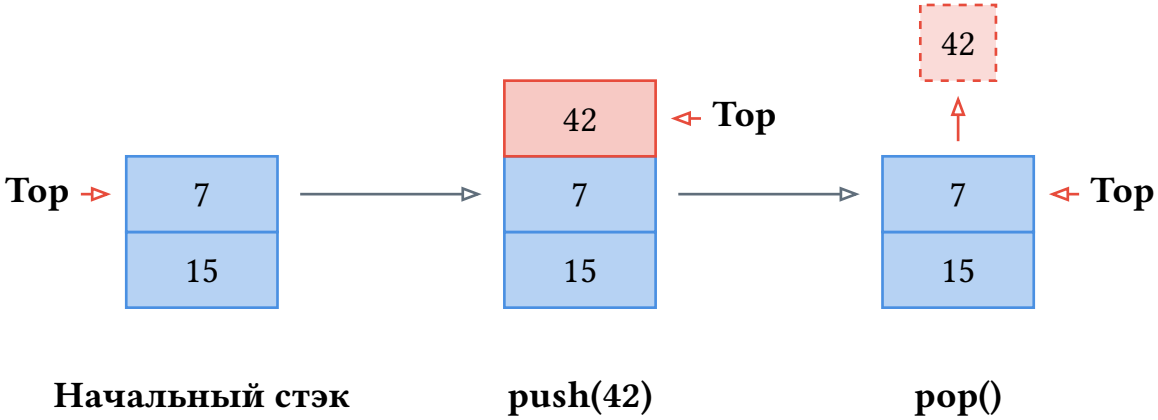
FIFO и LIFO структуры

Эти классы структуру в c++ представленны в достаточно удобном формате

- LIFO (Last In – First Out) – `std::stack`, `std::vector`, `std::deque`
- FIFO (First In – First Out) – `std::queue`, `std::deque`
- FIFO с приоритетом – `std::priority_queue`

Работа с ними выглядит довольно похоже. Разберёмся поочереди.

`std::stack` - класическая реализация стэка. Часто его любят представлять стопкой блинов. Вы можете положить блин наверх, и взять блин сверху. Для того чтобы взять блин из середины, требуется сначала снять все блины выше нашей цели.



Как илюстрация с предыдущего слайда будет выглядеть на c++:

```
std::stack<int> s;
s.push(15);
s.push(7);
std::cout << s.top() << '\n';
s.push(42);
std::cout << s.top() << '\n';
s.pop();
std::cout << s.top() << '\n';
```

Вывод

```
7
42
7
```

С точки зрения теории наиболее эффективный способ реализовать такую структуру это сипользовать список. Однако как, показывает практика, список почти никогда не является хорошим выбором. Интересные примеры по этому поводу написана Бьёрном Страустоупом(создателем языка c++), это даже упомниается на сайте iso cpp. Можно почитать тут и посмотреть тут.

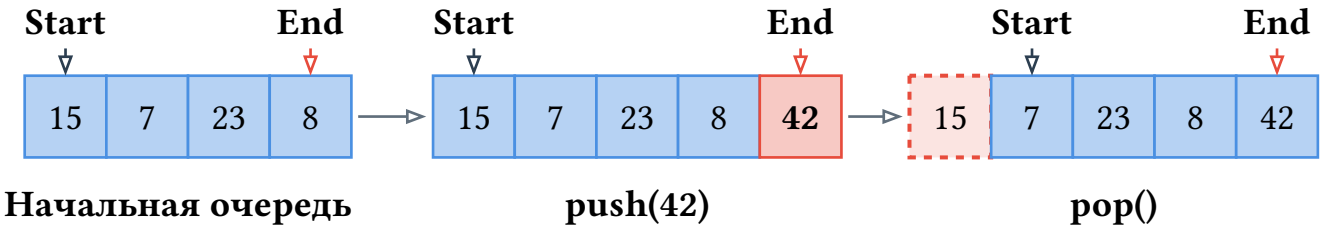
Под капотом это на самом деле ни что иное как `std::vector`. Нужно просто хранить указатель на конец стека отдельно от конца массива. Что на самом деле и так происходит. Что мешает использовать вектор вместо стека? Ничего.

```cpp
std::vector<int> v;
v.push_back(15);
v.push_back(7);
std::cout << v.back() << '\n';
v.push_back(42);
std::cout << v.back() << '\n';
v.pop_back();
std::cout << v.back() << '\n';
```

Вывод

```
7
42
7
```

`std::queue` - класическая реализация очереди. Очередь удобнее всего представлять в виде... Очереди на кассе в магазине. Кто первый пришёл, тот первый и купит товар.



Как илюстрация с предыдущего слайда будет выглядеть на c++:

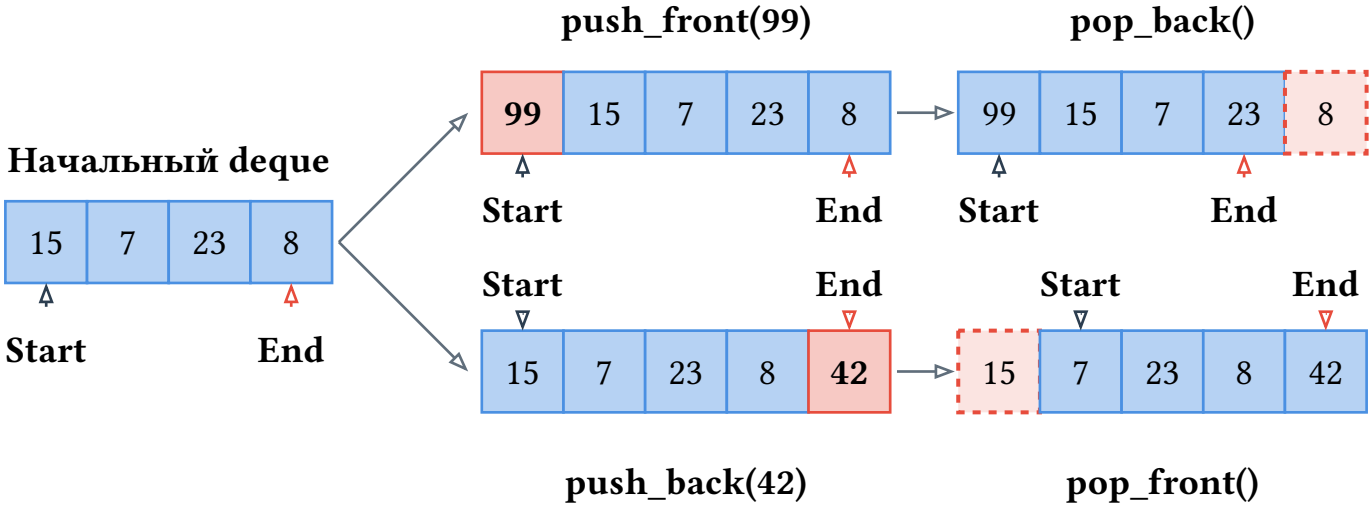
```cpp
std::queue<int> q;
q.push(15); q.push(7);
q.push(23); q.push(8);

std::cout << q.front() << " < > "
          << q.back() << '\n';
q.push(42);
std::cout << q.front() << " < > "
          << q.back() << '\n';
q.pop();
std::cout << q.front() << " < > "
          << q.back() << '\n';
```

Вывод

```
15 < > 8
15 < > 42
7 < > 42
```

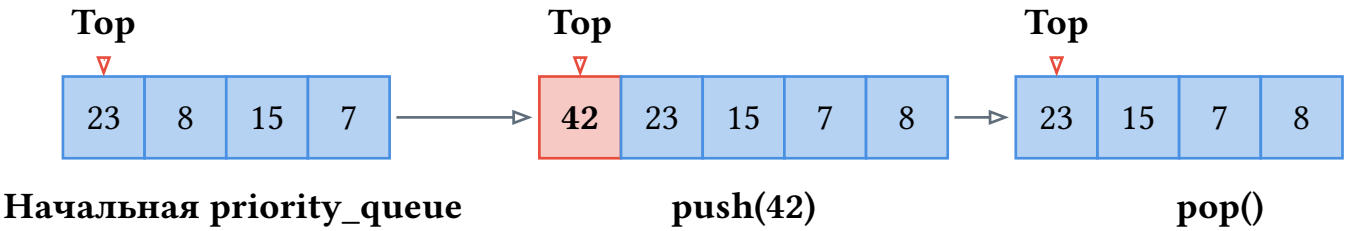
Ну может здесь нам удобно будет применить список? Нет, это тоже `std::vector`. Только здесь нужна небольшая обёртка в виде колцевого буфера, так что просто использовать вектор не так удобно.

`std::dequeue` - класическая реализация двусторонней очереди. Это обычная очередь, но мы можем ивлекать элементы не только сверху, но и справа.



Код выглядит аналогично очереди, так что его опустим.

`std::priority_queue` – приоритетная очередь. Это очередь в которой сначала обслуживается самый важный клиент. Под капотом представляет из себя max_heap. Значит что добавление и удаление работают за $O(\log n)$, а посмотреть наибольший элемент за $O(1)$. Данные не обязательно будут отсортированы на каждом шаге, но наибольший элемент всегда в начале.



Как илюстрация с предыдущего слайда будет выглядеть на c++:

```cpp
std::priority_queue<int> s;
s.push(23); s.push(8);
s.push(15); s.push(7);
std::cout << s.top() << '\n';
s.push(42);
std::cout << s.top() << '\n';
s.pop();
std::cout << s.top() << '\n';
```

Вывод

```
23
42
23
```

`std::bitset`

`std::bitset` представляет собой последовательность из битов фиксированой длины. К битам можно применять обычные логические и побитовые операции.

```cpp
std::bitset<4> b1{0xA};
std::cout << b1 << "\n";
b1 |= 0b0100; assert(b1 == 0b1110);
b1 &= 0b0011; assert(b1 == 0b0010);
b1 ^= std::bitset<4>{0b0110};
std::cout << b1 << "\n";
b1.reset(); assert(b1 == 0);
b1.set();
assert(b1.all() && b1.any() && !b1.none());
b1.flip(3);
b1[0] = false;
std::cout << b1 << "\n";
```

Вывод

```
1010
0100
0111
0110
```

Прочее

`std::vector` при помощи ряда функций из `<algorithm>`, о которых я умолчал ранее:

- `make_heap(begin, end)` – создать кучу на отрезке $[\text{begin}, \text{end})$
- `push_heap(value)` – добавить элемент сохранив свойства кучи;
- `pop_heap()` – удалить элемент сохранив свойства кучи;
- `sort_heap(begin, end)` – отсортировать элементы кучи. Работает чуть лучше обычной сортировки, но работает только на кучах;
- `is_heap(begin, end)` – проверить является ли отрезок кучей;
- `is_heap_until(begin, end)` – возвращает последний итератор на котором всё ещё выполняются свойста кучию

Чаще всего когда нужна куча используются `std::priority_queue`, либо `std::set`

Препроцессор c++ позволяет написать небольшие макросы для удобства написания кода. Ниже пару примеров из моего шаблона. Он использует не только препроцессор, но и другие возможности языка.

Макрос	Пример использования
`#define all(x) x.begin(), x.end()`	`sort(all(v))`
`#define YES cout << "YES\n"`	`if(flag) YES;`
`typedef long long ll`	`ll counter = 0;`

Можно написать `#define int long long` чтобы никогда не задумывать о переполнении, но это может привести к неожиданному певышению ограничения по памяти или ошибкам при работе с побитовым представлением

Всё что описано далее следует использовать только после дополнительного предварительного изучения поведения.

- `std::_Rb_tree` – голая реализация красно чёрного дерева
- `std::_Hashtable` – голая реализация хэш-таблицы
- Флаги компиляции которые могут ускорить выполнение програмы, при этом не требует изменение параметров запуска команды g++:
  - `#pragma GCC optimize("Ofast,no-stack-protector,unroll-loops,fast-math")`
  - `#pragma GCC target("avx,avx2,sse,sse2,sse3,sse3,sse4.1,sse4.2")`
  - `#pragma GCC target("popcnt,abm,mmx,tune=native")`
- `__int128` – целое число которое хранится в 128 битах