C++ [4]

На этой лекции мы закончим разговор про C++. Пройденный материал за эти 4 занятия покрывает почти всё, что нужно знать про этот язык, для написания кода на олимпиадах. Естесственно, это только маленькое подмножество C++, но его нам сейчас вполне хватит.

Ранее мы с вами говорили про вектора и строки. Однако стандартная библиотека предоставляет довольно обширный набор контейнеров, достаточный для олимпиад. Сегодня мы о них и поговорим.

Очередь (queue) и стек (stack)

Иногда линейные контейнеры, в которых есть случайный доступ к элементам по индексу не всегда нужны. Иногда вам достаточно знать, в каком порядке элементы идут, и в этом порядке их брать. Для этого могут использоваться два контейнера.

FIFO (first in, first out)

Первый контейнер, который бывает полезен - это очередь *queue*. Лежит она, как обычно логично происходит, в заголовочном файле *queue*.

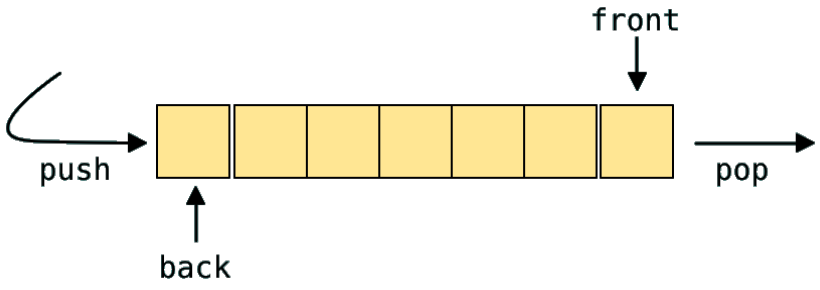

Рис. 1: Иллюстрация очереди

Что мы с ней можем делать? Чтобы создать очередь, например из интов, нужно написать "$queue < int > q$;". Здесь не нужно передавать никаких параметров конструктора, как это можно было делать в векторах или строках. Очереди нельзя задать изначальный размер, значения по умолчанию, она изначально инициализируется пустой.

Что ещё можно делать?

```
// методы, не изменяющие очередь
q.front(); // возвращает первый элемент
q.back(); // возвращает последний элемент
q.size(); // возвращает количество элементов
q.empty(); // возвращает пуста ли очередь (true/false)
// методы, изменяющие очередь
q.push(5); // добавление элемента 5 в конец
q.pop(); // удаление элемента из начала
```

Опять же, если вы делаете *pop* из пустой очереди, это ваша проблема. Лучше обрабатывайте такой случай руками.

LIFO (last in, first out)

Второй полезный контейнер - это стек *stack*. Лежит он в заголовочном файле *stack*. Хорошая аналогия для него - стопка книг. Вы можете положить книгу наверх стопки, видеть обложку самой верхней и забирать книгу опять же сверху.

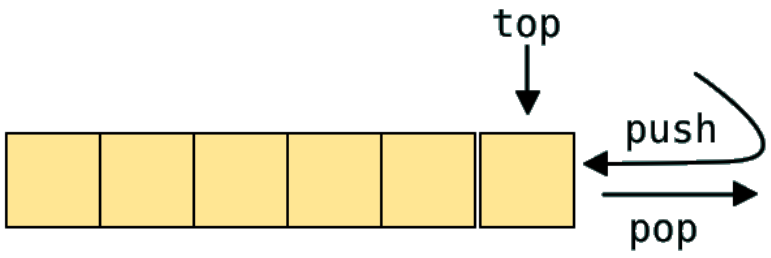

Рис. 2: Иллюстрация стека

Итого у стека есть следующий набор методов:

```
// методы, не изменяющие стек
s.top(); // возвращает элемент из головы стека
s.size(); // возвращает количество элементов
s.empty(); // возвращает пуста ли очередь (true/false)
// методы, изменяющие стек
s.push(5); // добавление элемента 5 в голову стека
s.pop(); // удаление элемента из головы
```

Инициализируется стек, как и очередь, без указания начальных параметров.

Каждая операция в этих структурах данных работает за $O(1)$, то есть операции требуют вне зависимости от количества элементов в структуре примерно одинаковое константное количество времени.

Дек (deque)

"Double ended queue"или дек - это очередь, в которой можно клать и забирать элементы с обеих сторон. Лежит в *deque*.

Что нам доступно для работы с деком?

```
// Методы, изменяющие дек
d.push_back(1); // добавить в конец
d.push_front(2); // добавить в начало
d.pop_back(); // удалить из конца
d.pop_front(); // удалить из начала
// Методы, не изменяющие
d.size();
d.empty();
```

Также внезапно, в противоречие всем стандартам, дек поддерживает обращение к элементам по индексу, как в векторах.

```
d[1] = 3;
cout << d[4] << endl;
```

Хоть все операции в деке работают также за $O(1)$, константное время, из-за большой вариативности операций (и особенностей реализации, с этим связанных), эта константа у него несколько больше чем в других структурах, соответственно работает он немного медленнее.

Множества (set)

То были простые, иногда полезные структурки. Перейдём к более полезным структуркам.

Что вам часто бывает нужно делать? Вам часто нужно поддерживать множество разных элементов. Например, вам нужно хранить числа или строки и смотреть, что они все различные. Для этого есть специальная структура, которая называется *set* и хранится в одноимённом заголовочном файле. Сет имитирует обычные математические множества.

Как внутри устроен *set*? Это сбалансированное бинарное дерево поиска (красно-чёрное в STL), структура, написанная на указателях. Эти детали нам сейчас не важны, о них вы узнаете когда-нибудь позже. Сейчас мы просто посмотрим, как оно примерно устроено.

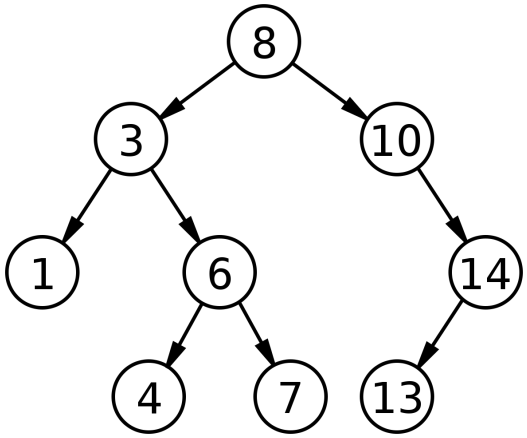

Рис. 3: Устройство сета

На иллюстрации выше показано, как в сете хранятся элементы. Пусть в нашем сете есть какой-то элемент x. Что мы точно можем про него сказать в таком дереве? Что все элементы в левом поддереве точно меньше чем x, а все элементы в правом поддереве точно больше чем x. Что это позволяет?

Это позволяет нам в сете эффективно искать. По сути вы делаете на такой структуре данных на указателях бинарный поиск. Начинаете с верхнего элемента и спрашиваете, меньше, равен или больше искомого этот элемент. В зависимости от ответа переходите в левое или правое поддерево. Принцип работы вы можете увидеть на картинке 4.

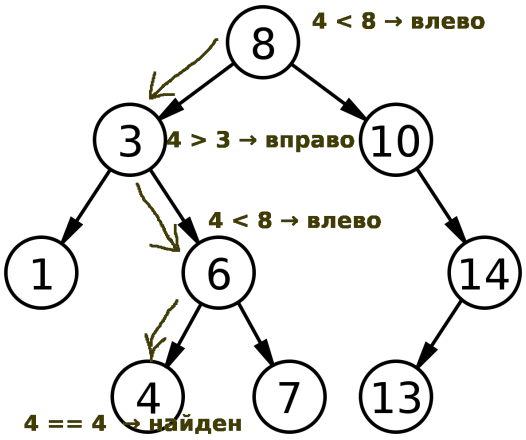

Рис. 4: Поиск элемента 4 в сете

Что нам гарантированно в сете? Что операции доступа, добавления, удаления элементов работают за $O(log\, n)$.

Давайте посмотрим на то, что мы можем делать с сетом.

```
set<int> s; // инициализация сета
s.insert(2); // добавление элемента
```

Если вы дважды добавали один и тот же элемент, в сете он всё равно будет храниться в единственном экземпляре, как и положено в математических множествах.

```
s.erase(2); // удаление элемента из сета
s.clear(); // удаление всех элементов
s.size();
s.empty();
```

Если вы хотите проверить, есть ли у вас в сете определённый элемент, тогда воспользуйтесь следующим методом, который для сета вернёт 0 или 1:

```
s.count(x); // сколько элементов x есть в сете
```

Так как у сета довольно сложная структура, не линейная, *lower_bound* и *upper_bound* на нём работают не так, как на векторах или массивах. Здесь они являются не функциями, принимающими итераторы, а методами, принимающими только искомый элемент.Чтобы получить итератор на какой-то элемент x нашего сета s, нужно написать следующее:

```
auto lower = s.lower_bound(x);
auto upper = s.upper_bound(x);
```

Чтобы посмотреть значение, на которое указывает итератор, нужно его разыменовать. Если итератор указывает на конец сета, что бывает когда мы ищем элемент больший максимального в сете, то нам нужно это обработать.

```
if (lower == s.end()) {
    cout << "Doesn't exists\n";
}
else {
     cout << *lower << endl; // звёздочка перед итератором
}
```

Также мы можем попытаться найти элемент с помощью метода $find()$.

```
auto it = s.find(x); // возвращает итератор на элемент
```

Пройтись по всем элементам сета можно с помощью range-based for:

```
for (int x: s) {
    cout << x << endl;
}
```

Очень полезное свойство - что минимальный элемент сета хранится в итераторе на начало сета.

```
cout << *s.begin() << endl; // выводим минимальный элемент
↪  сета
```

Здесь опять же ничего не гарантируется. Если сет пустой, то вам выведется что-то.

Если мы хотим взять последний элемент сета, можно взять итератор на конец и перейти к итератору на предыдущий элемент. В сете, опять же, из-за его нелинейности, передвигаться можно только на соседние итераторы. Делается это с помощью операторов $++$ и $--$.

```
cout << *--s.end() << endl;
```

Также все коллекции стандартной библиотеки поддерживают реверсивные итераторы *rbegin*() и *rend*() (reversed begin, reversed end). Таким образом, максимальный элемент можно найти таким способом:

```
cout << *s.rbegin() << endl;
```

Свои структуры в сете

Сет правильно работает со всеми стандартными типами. Но вдруг нам понадобится хранить множество своих структур?

Для того, чтобы сет правильно работал и с нашими структурами, нужно указать ему, как наши структуры будут упорядочены. Первый способ сделать это - перегрузить оператор сравнения $<$.

Посмотрим на примере кода со структурой даты.

```
#include <set>
#include <iostream>
struct date {
    int year, month, day;
};
bool operator<(const date& lhs, const date& rhs) {
    if (lhs.year != rhs.year) {
        return lhs.year < rhs.year;
    }
    if (lhs.month != rhs.month) {
        return lhs.month < rhs.month;
    }
    return lhs.day < rhs.day;
}
int main() {
    std::set<date> s;
    s.insert({2020, 10, 15});
    s.insert({2020, 10, 8});
    s.insert({2020, 10, 22});
    for (date x: s) {
        std::cout << x.day << '.' << x.month << '.' << x.year
        ↪  << std::endl;
    }
}
```

Передача компаратора в сет

Иногда вам всё-таки хочется два способа что-то сравнивать.

Например, в прошлой лекции был пример на сортировку по абсолютному значению. Допустим теперь мы хотим хранить их в множестве, напишем компаратор. Теперь нам нужно его как-то передать сету.

У сета есть ещё один параметр шаблона "что использовать в качестве компаратора". Он принимает не функцию, а структуру. У этой структуры должен присутствовать оператор круглые скобочки *operator*(). Подробнее можете почитать об этом на cppreference, но вообще вам это знать не обязательно, поэтому давайте просто посмотрим на код:

```cpp
#include <set>
#include <iostream>
#include <algorithm>

struct cmp_abs { // это называется функтор
    bool operator()(int lhs, int rhs) {
        return std::abs(lhs) < std::abs(rhs);
    }
};

int main() {
    std::set<int, cmp_abs> s;
    s.insert(1);
    // также мы можем объявить переменную этой структуры
    // и сравнивать элементы, используя её
    cmp_abs cmp;
    std::cout << cmp(1, 2);
}
```

Мультимножество (multiset)

Также в заголовочном файле *set* лежит структурка под названием *multiset*. Она позволяет хранить не просто различные значения, но и подсчитывать количество одинаковых вставленных элементов.

В мультисете почти всё работает, как в сете, кроме *erase*(). Чтобы удалить только один из имеющихся элемент x из мультисета, а не все сразу, в *erase*() нужно передать не x, а итератор на элемент x в мультисете, найденный, например, с помощью метода $find()$.

Мапы (map)

Сеты и мультисеты, конечно, очень полезные структуры. Но иногда нам хотелось бы хранить данные в формате ключ-значение. Если кто-то изучал питон, в нём за это отвечает dict, словари. В C++ за такое представление данных отвечает *map* из заголовочного файла *map*. По сути, это уже известный наш сет, где каждому элементу (далее говорим ключу) соотвествует ещё какое-то значение.

В мапе также любые операции поиска, добавления, удаления работают за логарифмическое время.

Чтобы создать *map*, в шаблоне ему нужно указать тип ключа и значения. При том, при упорядочивание, мапа смотрит только на ключ, значение в этом никак не участвует.

Так как мапа во многом эквивалентна сету, давайте сразу рассмотрим пример кода. Например, давайте посчитаем, сколько раз встречалась каждая строка из входных данных.

```cpp
#include <iostream>
#include <string>
#include <map>
int main() {
    std::map<string, int> counts;
    std::string tmp;
    while (std::cin >> tmp) {
        counts[tmp] += 1; // если значения с ключём tmp не было, при обращении counts[tmp] оно создастся и изначально будет стандартному значению типа, в случае инта нулю
    }
    // если вы хотите обратиться к элементу мапы,
    // то он будет храниться в паре ключ-значение
    for (const auto& kv: counts) { // строки могут быть длинными, поэтому передаём по ссылке
        // kv.first - ключ
        // kv.second - значение
        std::cout << kv.first << ' ' << kv.second << '\n';
    }
}
```

Очередь с приоритетом (priority_queue)

В заголовочном файле *queue* хранится также структура данных под названием очередь с приоритетом (по-другому куча). В чём её фишка? Она позволяет нам получать наибольший элемент не за $O(log\, n)$, как в сете и не за $O(n)$, как в обычном векторе, а за $O(1)$. Её устройство несколько похоже на устройство сета, только реализована она не на указателях, а на векторе, и условие там: выше находится элемент, который больше.

Посмотрим, что в ней есть:

```
priority_queue<int> pq; // привычная инициализация
pq.push(2); // элемент добавляется куда-то O(log n)
pq.pop(); // удаление наибольшего элемента O(log n)
int x = pq.top(); // возвращает наибольший элемент O(1)
```

Если вы хотите получить наименьшее значение, то числа можно добавлять со знаком минус. Для других же типов можно, как и в сете, передавать функтор. Только он здесь идёт третьим параметром шаблона. Вторым параметром нужно передавать контейнер, на котором эта структура реализована. Нам нужно оставить вектор с типом наших элементов.

```
priority_queue<int, vector<int>, cmp_abs> pq;
```

Список (list)

Ещё один контейнер в C++, который бывает полезен в реальных проектах, когда нужно хранить какие-то большие структуры – это list. Это как раз то, в чём в питоне хранятся какие-нибудь массивы данных. В олимпиадных задачах списки вряд ли получится где-то применять, лично мне такие задачи не встречались. К тому же почти все их операции для вставки, удаления и поиска работают за $O(n)$, что для нас слишком доллго. Поэтому его рассмотрение оставляю интересущимся: `https://en.cppreference.com/w/cpp/container/list`

Несколько слов по задачам на контестах

Иногда в задачах входные данные бывают такие длинные, что их не целесообразно давать на считывание. Потому что за секунду в C++ можно считать порядка 10^6 чисел (если не использовать самописное считывание), а что-то порядка 10^7 будет считываться целых 10 секунд - гораздо больше времени работы программы. Поэтому авторы иногда дают вам несколько входных значений, по которым вы будете сами генерировать входные данные, и потом уже работать с ними. Обычно в условии указывается функция для их генерации, вы должны будете использовать её.

Также иногда встречаются задачи под одной буквой, но пронумерованные. Их условия будут идентичными, различаться будут только размеры входных данных. Поэтому если вы придумаете решение для более сложной версии задачи, можете засылать его и в задачу с меньшими ограничениями.

Есть ещё просто совет. Если вы видете, что какую-то задачу люди решают, а у вас она не решена, при том вы сидите над задачей, которую пока никто не решил, и у вас она не очень получается, возможно вам стоит переключиться на ту, которую решают.

Итог

В этой лекции мы рассмотрели контейнеры C++, которыми будем в дальнейшем пользоваться на олимпиадах.

Также это была последняя лекция по основам языка C++. Дальше, конечно, будут появляться какие-то незатронутые моменты, но занятия уже будут посвящены конкретно олимпиадным темам.