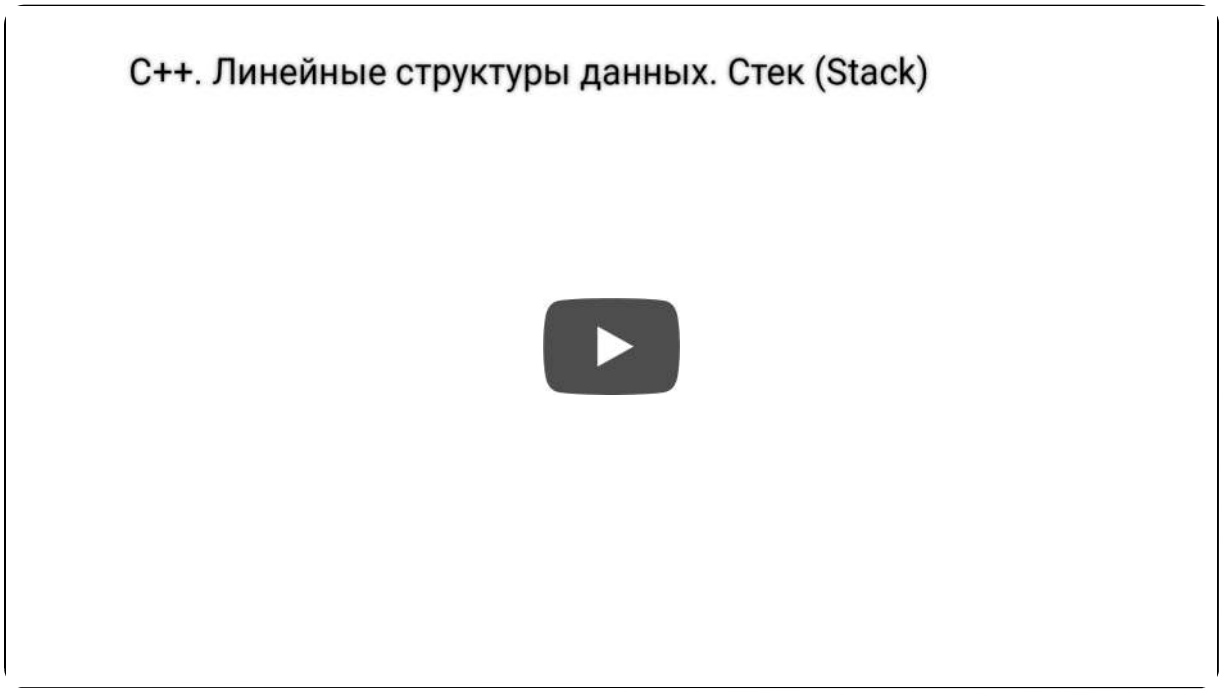
Стек



Стек

Кроме стандартных структур данных: список, множество, ассоциативный массив (словарь), в программировании используется и ряд других структур. Особенность каждой из них состоит в способе хранения данных и алгоритме доступа к элементам структуры. При использовании каждой структуры есть определённые ограничения — то, что структура "не умеет делать быстро". Например, операция добавления элемента в конец списка выполняется быстро, а удаление первого элемента выполняется за длину списка. Познакомимся с несколькими структурами, которые иногда называют "линейными структурами данных", потому что элементы в таких структурах данных хранятся в виде последовательности элементов, один за другим. Например, список в языке Python и вектор в языке C++ также являются линейной структурой данных, у которой есть операция доступа к i-му по счёту элементу, и эта операция выполняется быстро. А вот такие операции, как удаление элементов из начала или вставка элемента в середину списка или вектора, выполняются долго.

Стек (stack) — это линейная структура данных, в которой элементы добавляются и удаляются только с одного конца — вершины стека. Можно

представить себе стек как стопку тарелок. Вы можете положить новую тарелку на вершину стопки, можете снять верхнюю тарелку со стопки, но только верхнюю. Если вам нужно добраться до нижней тарелки, то есть до самого первого элемента стека, вы должны убрать из стопки все тарелки, причём одну за другой.

То есть если мы положим в стек последовательно числа $1, 2, 3$, а потом удалим из стека последний элемент, то в стеке останутся числа 1 и 2. Если мы затем положим в стек числа 4 и 5 и будем последовательно удалять из стека элементы, то элементы будут удаляться в порядке $5, 4, 2, 1$.

Таким образом, стек поддерживает следующие операции:

- Добавить элемент в конец стека (push)
- Узнать значение последнего элемента стека (top)
- Удалить последний элемент из стека (pop)
- Узнать размер стека

Все эти операции в стеке выполняются за $O(1)$.

Для реализации стека на C++ подойдёт контейнер vector, так как vector поддерживает операции добавления элемента в конец со средней сложностью $O(1)$ и удаления последнего элемента за $O(1)$ всегда.

Операция `push_back(x)` эквивалентна операции добавления элемента в конец стека. Операция `pop_back()` эквивалентна удалению последнего элемента. Также узнать размер стека нам поможет метод `size()`. Для того чтобы узнать значение последнего элемента, нужно обратиться к элементу `s[s.size() - 1]`, но у вектора есть метод `back()`, который возвращает ссылку на последний элемент, удобней использовать его.

Обратите внимание, что вызов методов `pop_back()` и `back()` на пустом векторе является ошибкой, в языке C++ это *Undefined behaviour*, то есть поведение программы после этого не определено. Это может быть как ошибкой исполнения программы, так и продолжением работы программы с произвольными значениями, которые вернули эти методы.

Стек с поддержкой минимального элемента

Давайте реализуем стек, который будет содержать ещё одну операцию —

"узнать значение наименьшего элемента во всём стеке". Это так называемый "стек с минимумом". Эта задача решается следующим образом: мы будем хранить два стека: в одном будут сами значения, а в другом стеке для каждого элемента будет храниться минимальное значение во всём стеке, когда мы добавили этот элемент.

Например, пусть мы положили в стек число 5. Оно же будет наименьшим элементом. Теперь положим в стек число 4. Наименьшим элементом во всём стеке теперь стало 4. Положим это число во второй стек. Теперь добавим в стек число 7. Наименьший элемент в стеке до этого — это 4, мы добавили большее число, поэтому значение минимума осталось равно 4, положим его во второй стек. Теперь добавим в стек число 2. Это число меньше минимума во всём стеке, поэтому во второй стек добавим число 2.

Если теперь из первого стека удалять элементы и одновременно удалять элементы из второго стека, то на вершине второго стека всегда будет наименьшее значение в первом стеке.

Реализация функций

```
vector <int> stack, stack_min;
void push(int elem) {
    stack.push_back(elem);
    if (stack_min.empty() || stack_min.back() > elem)
        stack_min.push(elem);
    else
        stack_min.push(stack_min.back());
}

void top() {
    return stack.back();
}

void get_min() {
    return stack_min.back();
}

void pop() {
```

```
    stack.pop_back();
    stack_min.pop_back();
}
```

Задача о правильной скобочной последовательности

Пусть нам дана последовательность из трёх видов скобок: $()$, $[]$, $\{\}$. Необходимо проверить, является ли она правильной.

Правильной является последовательность, которая может возникнуть при записи некоторого арифметического выражения.

Пример правильной последовательности: $([]())[]$

Примеры неправильных последовательностей:

- $(()$
- $(\}$
- $)($
- $([)]$

Формально правильная скобочная последовательность определяется таким образом:

- Пустая строка является правильной скобочной последовательностью;
- Правильная скобочная последовательность, взятая в скобки одного типа, — правильная скобочная последовательность;
- Правильная скобочная последовательность, к которой приписана слева или справа правильная скобочная последовательность, — тоже правильная скобочная последовательность.

Можно придумать наивный алгоритм проверки скобочной последовательности на правильность. В правильной скобочной последовательности всегда есть пара скобок, внутри которой нет других скобок. Это пара из открывающей и закрывающей скобки одного вида, идущих рядом. Удалим их. Опять найдём такую пару и снова удалим. Будем продолжать этот процесс, пока есть такие пары. Если мы в итоге смогли удалить все скобки, то есть все скобки разбились на пары открывающих и закрывающих, то последовательность правильная. Если же на каком-то шаге

мы не смогли найти пару соседних подходящих скобок, но скобки ещё остались, то последовательность неправильная.

Но такой алгоритм при наивной реализации будет иметь сложность $O(n^2)$. Действительно, давайте рассмотрим последовательность, в которой сначала идут $n/2$ открывающих скобок одного типа, потом $n/2$ закрывающих скобок того же типа. Эта последовательность будет правильной. Но каждый раз мы будем удалять две скобки из середины нашей строки, что будет выполняться за $O(n)$, поскольку необходимо сдвигать половину элементов строки. Это удаление элементов из середины будет выполняться $n/2$ раз, поэтому общая сложность будет $O(n^2)$.

Можно улучшить этот алгоритм, если использовать стек. Закрывающая скобка должна быть парной к последней открывающей. То есть если мы будем хранить последовательность из открывающих скобок, то закрывающая скобка удаляет последнюю открывающую, если она того же вида, в противном случае скобочная последовательность неправильная. Это означает, что последовательность открывающих скобок, которые не были ещё закрыты, представляет собой стек. Если мы встретили открывающую скобку, то она добавляется в конец стека. Если мы встретили закрывающую скобку, то должны выполняться следующие условия: стек не пуст и на вершине стека хранится скобка, парная данной закрывающей скобке. Если после обработки всех скобок стек оказался пуст, то последовательность правильная.

В каком случае последовательность будет неправильной? Если для очередной закрывающей скобки не нашлось парной открывающей, то есть если мы встретили закрывающую скобку, то либо стек пуст, либо на вершине его находится скобка другого вида. Или если мы для какой-то открывающей скобки не нашли закрывающую, это означает, что после рассмотрения всех скобок стек оказался не пуст.

Сложность алгоритма $O(n)$, так как каждую скобку мы кладём в стек или удаляем из стека не более одного раза, а данные операции на стеке работают за $O(1)$.