Алгоритмы и структуры данных. STL

densine

25 октября 2025 г.

Содержание

1. Кратко

На этой лекции будут обсуждены различные фичи языка.

2. Ввод и вывод

2.1. Ускорение/оптимизация консоли

- ```
  #include <iostream>
  ios_base::sync_with_stdio(0);
  ```

  В C были стандартные команды работы со вводом и выводом – scanf(...) и printf(...). Они медленне cin и cout, и даже если ваша программа проходила по асимптотике, могла завилиться на этом, поэтому лучше это прописывать.

- ```
  cin.tie(0);
  ```

  Эта команда "откладывает", рассинхронизирует их, т.е. мы говорим C++ "cin и cout больше вместе не работают", и это ускоряет исполнения программы.

- ```
  cout << "Something" << endl;
  cout << "Something" << '\n';
  ```

  Как думаете, есть разница? Если выполните первый вариант, то "откладывание" вывода сбросится. endl не только переводит строку, но и принуждает к исполнению все отложенные операции. Так что и быстрее, и удобнее использовать символ перевода строки.

2.2. Ввод/вывод с файла

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);

int a, b;
cin >> a >> b;
cout << a + b << '\n';
```

Этот код буквально переназначает потоки ввода (stdin) и вывода (stdout) на потоки файлов input.txt и output.txt соответственно. Более современный подход:

```
#include <fstream>

ifstream fin("input.txt"); // Поток ввода из файла
ofstream fout("output.txt"); // Поток вывода в файл
int a, b;
fin >> a >> b;
fout << a + b << '\n';
```

2.3. Ввод n-ного количества чисел

```
int x;
int summary = 0;

while (cin >> x) {
    summary += x;
}

cout << summary << '\n';
```

2.4. Ввод до определенного символа

```
#include <string>

string s;
getline(cin, s /* , '\n' */);
cout << s << '\n';
```

Третьим аргументом можно передать сепаратор.
```

2.5. Количество знаков после запятой

```
#include <iomanip>

double x = 1.2345;
cout << fixed << setprecision(10) << x << '\n';
```

Такой способ вывода автоматически округляет до ближайшего целого А вдруг нам это не нужно? Тогда делаем так:

```
#include <sstream>

stringstream ss;
ss << x;

string s;
ss >> s;

int idx = s.size();
for (int i = 0; i < s.size(); i++) {
    if (s[i] == '.') {
        idx = i;
    }
}

for (int j = 0; j < min(idx + 4, (int)s.size()); j++) {
    cout << s[j];
}

cout << '\n';
```

Иначе говоря, мы превратили double в string с помощью эмуляции потока ввода/вывода через строку (stringstream) и дальше работали уже как со строкой.

3. Библиотека algorithm

- Сортировка выглядит так:

  ```
  #include <algorithm>
  #include <vector>
  #include <iostream>

  vector<int> a = {2, -3, 1, 1, 5, 6};
  sort(a.begin(), a.end());
  ```

  a.begin() и a.end() – итераторы. Может быть так, что вам понадобится итераторы в переменных:

```
vector<int>::iterator it = a.begin(); // Громоздко...
auto it = a.begin(); // auto автоматически подставляет тип

// Затем, что такое итератор? Можно сказать, что указатель.
// Поэтому можно так: it += 3;
cout << *it << '\n'; // Выведет 1
```

И с .begin(), и с .end() можно совершать такие манипуляции. .begin() указывает на начало массива, а end() на 1 элемент после конца. Также существует цикл for-each:

```
for (auto el : a) {
    cout << el << ' ';
}
cout << '\n';
```

- Бинарный поиск

```
cout << binary_search(
    a.begin(),
    a.begin() + a.size() - 1, 5) << '\n';
// Необходима сортированность подотрезка для его раобты!

// Также существуют такие функции:

// Минимальное число, большее x
auto it = upper_bound(a.begin(), a.end(), x);
// it == a.end() => такого числа нет
// иначе итератор на этот элемент

// Минимальное число, которое не меньше x
auto it = lower_bound(a.begin(), a.end(), x);
// Возвращемое работает также
```

- Сортировка по определенному правилу:

```
sort(a.begin(), a.end(), [](int a, int b) {
    if (a > b) return true;
    else return false;
});
```

Иначе говоря, мы определили лямбда-функцию (можно было задать ее заранее, важно чтобы было 2 аргумента и возвращало bool) и внутри задали, когда мы меняем элементы. Затем, в лямбда-функции можно "захватывать" переменные из окружения. Например, мы хотим отсортировать индексы:

```
vector<int> b = {0, 1, 2, 3, 4, 5};
sort(a.begin(), a.end(), [&](int x, int y){
    return a[x] < a[y];
});
```

Т.е. теперь индексы в b указывают на элементы в порядке возрастания в a.

4. Множества и словари!

4.1. Сортированное множество

```cpp
#include <set>

set<int> st;

// Добавление элементов
st.insert(3);
st.insert(6);
st.insert(-1);
st.insert(2);
st.insert(4);

for (auto el : st) cout << el << ' ';
cout << '\n';

// Удаление элементов
st.erase(3);
st.erase(-1);

itn x;
cin >> x;

auto it = st.find(x);

if (st == st.end()) { // Если число не нашлось
    cout << "-1\n";
} else {
    it++; // можно
    it--; // можно
    it += 2; // нельзя!
    cout << *it << '\n';
}

st[t]; // нельзя обращаться по индексу
```

Все операции с обычным set работают за $log(n)$. Также можно использовать функции бинарного поиска и все с ним связанные.

4.2. Словари

```cpp
#include <map>


map<int, int> mp;
map[10000000] = 1; // [ключ] = значение
```

```
cout << mp[10000000] << '\n';

// В качестве ключей и значений можно использовать
// любые сравниваемые типы.

// По факту map - это set из pair.
mp.insert({1010, 1});
```

4.3. Multiset

```
multiset<int> s;
s.insert(1);
s.insert(2);
s.insert(-1);
s.insert(1);

for (auto el : s) cout << el << ' ';
cout << '\n';
// Выведет -1 1 1 2

cout << s.count(1) << '\n'; // O(n == 1 + log(n))

// Удалятся все копии
s.erase(1);

// Удаление единственного элемента
s.erase(s.find(1));
```

4.4. Хэш-таблицы

Вместо set можно использовать unordered set, а вместо map – unordered map. Тогда операции обращения и поска выполняются за $O(1)$, но довольно долгую, а элементы неупорядочены.