# Библиотека алгоритмов

М. А. Ложников

18 марта 2019 г.

**Это предварительная невыверенная версия. Читайте на свой страх и риск.**

## 1 Лямбда-выражения (C++11)

Рассмотрим задачу сортировки диапазона элементов пузырьком. Требуется написать функцию, которая принимает два итератора и правило сравнения. В языке C в качестве правила сравнения можно было бы передать указатель на функцию.

---

```
1  // Определяем функцию сравнения.
2  int CompareInt(int lhs, int rhs) {
3    if (lhs < rhs)
4      return -1;
5    else if (lhs == rhs)
6      return 0;
7
8    return 1;
9  }
10 // Вариант на C. Для простоты функция сортирует только массивы целых чисел.
11 void CSort(int* data, int count, int (*compare)(int lhs, int rhs)) {
12   int i, j;
13
14   // Для простоты рассмотрим этот далеко не самый оптимальный вариант.
15   for (i = 0; i < count; i++) {
16     for (j = 0; j < count - 1; j++) {
17       if (compare(data[j + 1], data[j])) {
18         int tmp = data[i];
19         data[i] = data[i + 1];
20         data[i + 1] = tmp;
21       }
22     }
23   }
24 }
25
26 // Пример использования:
27 int main() {
28   int data[10] = { /* заполняем чем-нибудь */};
29
30   CSort(data, 10, CompareInt);
31   return 0;
32 }
```

В языке C++ кроме того можно передать в качестве сравнения экземпляр класса, в котором, например, переопределена операция функционального вызова `()`. Тип переданного аргумента можно сделать шаблоном.

```
#include <vector>
#include <utility>

class IntComparator {
 public:
  /* Возвращает true, если левый аргумент меньше правого. Аргументы в компараторе
     лучше принимать по ссылкам, чтобы избежать копирования. Для int'ов не актуально.
     В данном случае лучше было бы принимать по значению в силу быстрого копирования
     переменных типа int. */
  bool operator()(const int& lhs, const int& rhs) const {
    return lhs < rhs;
  }
};

// IteratorType --- тип итератора, Compare --- тип класса для сравнения
template<typename IteratorType, typename Compare>
void CppSort(IteratorType rangeBegin, IteratorType rangeEnd, Compare compare) {
  // Для простоты рассмотрим этот далеко не самый оптимальный вариант.
  for (IteratorType it = rangeBegin; it != rangeEnd; ++it) {
    for (IteratorType inner = rangeBegin; std::next(inner) != rangeEnd; ++inner) {
      // Вызываем операцию () у объекта compare
      if (compare(std::next(inner), inner))
        std::swap(*std::next(inner), *inner);
    }
  }
}

int main() {
  // Пример использования
  std::vector<int> vec = { /* заполняем чем-нибудь */};
  IntComparator comparator;
  CppSort(vec.begin(), vec.end(), comparator);

  // Можно ещё короче, здесь незачем создавать лишнюю переменную.
  CppSort(vec.begin(), vec.end(), IntComparator());
  return 0;
}
```

В примере выше шаблонная функция `std::swap()`, определённая в заголовочном файле `<utility>`, переставляет местами свои аргументы. Отметим, что благодаря тому, что мы использовали операцию функционального вызова для сравнения, синтаксис вызова компаратора в предложенных функциях сортировки очень похож. Более того, в `CppSort()` можно третьим аргументом передать указатель на функцию, принимающую два элемента сортируемого контейнера (возможно, по константным ссылкам) и возвращающую `bool`.

Очень часто бывает нужно отсортировать элементы массива/контейнера только один раз. Если критерий сравнения нетривиальный, то придётся писать операцию сравнения “меньше” < или передавать в функцию компаратор. Если критерий сравнения идёт в разрез с операцией сравнения “меньше”, которая была определена ранее по совершенно другим причинам, то так или иначе придётся писать компаратор и передавать его в функцию. В результате критерий сравнения может оказаться определён довольно далеко от того места, где он применяется. Как правило это приводит к необходимости листать исходники и затрудняет читабельность кода.

Для борьбы с этой проблемой в C++11 появились лямбда-выражения. Их можно рассматривать как объекты классов с переопределёнными операциями функционального вызова () (или если не понятно, то просто как обычные функции), которые определяются прямо в том месте, где они используются. Рассмотрим пример

```
void SortExample() {
  /* Синтаксис создания лямбда выражения следующий:
    квадратный скобки, потом список аргументов в круглых скобках, затем идёт тело
    функции. Тип возвращаемого значения лямбда выражения определяется автоматически. Тип
    данных самого лямбда-выражения зависит от реализации компилятора. Таким образом, для
    сохранения лямбды в переменную нужно использовать слово auto для автоматического
    определения типа данных переменной при инициализации. */
  auto compareFunction = [](int lhs, int rhs) {
    return lhs < rhs;
  };

  /* Теперь compareFunction можно использовать как функцию. Она принимает два
    целочисленных аргумента и возвращает bool поскольку возвращаемое значение
    (lhs < rhs) имеет тип bool. */
  std::vector<int> vec = {1, 3, 2, 5, 8, 2, 1, 22, 3, 45};
  CppSort(vec.begin(), vec.end(), compareFunction);

  // Можно было и не заводить отдельную переменную
  CppSort(vec.begin(), vec.end(), [](int lhs, int rhs) {
    return lhs < rhs;
  });
}
```

В квадратных скобках можно передавать список локальных переменных, которые в результате станут видны внутри лямбда-функции. Передавать переменные можно либо по ссылке либо по значению. Переданные по значению переменные копируются, и на них “навешивается” константность, то есть их нельзя модифицировать внутри лямбда-функции. Рассмотрим пример.

```
int a = 3;
double b = 5;
std::vector<int> vec = {1, 2, 3};

// Переменные a и b передаются по значению
auto lambda1 = [a, b](double x) {
  if (x < a)
    return 1.0;     // Все return должны возвращать один и тот же тип данных.
  else if (x > b)   // В противном случае будет ошибка компиляции.
    return 2.0;
```

```
  return 0.0;
};

// Переменная a передаются по значению, а vec --- по ссылке
auto lambda2 = [a, &v](double x) { /* ... */ };

// Все переменные передаются по ссылке
auto lambda3 = [&](double x) { /* ... */ };

// Все переменные передаются по значению
auto lambda4 = [=](double x) { /* ... */ };

// Все переменные передаются по значению кроме vec, которая передаётся по ссылке
auto lambda5 = [=, &vec](double x) { /* ... */ };

// Все переменные передаются по ссылке кроме b, которая передаётся по значению
auto lambda6 = [&, b](double x) { /* ... */ };
```

Передавать в лямбду можно сколько угодно переменных, доступных в данной области видимости. Правило передачи по-умолчанию пишется перед исключениями из правил.

Отметим, что всё то же самое можно было бы сделать при помощи обычного класса с переопределённой операцией функционального вызова. Например,

```
class ComplicatedRule {
 public:
  ComplicatedRule(std::vector<int>& vec, int x1, int x2) :
    vec(vec),
    x1(x1),
    x2(x2)
  { }

  int operator()(int x) {
    if (x < x1)
      return vec[0];
    else if (x > x2)
      return vec[2];

    return vec[1];
  }
 private:
  std::vector<int>& vec;
  const int x1;
  const int x2;
};
```

Такой класс соответствует следующему объявлению лямбда-функции:

```
std::vector<int> vec = {1, 2, 3}; // Заполним чем-нибудь
int x1 = 10;
```

```
int x2 = 20;
auto rule = [x1, x2, &vec](int x) {
  if (x < x1)
    return vec[0];
  else if (x > x2)
    return vec[2];

  return vec[1];
};
```

Операция функционального вызова () будет работать для предложенных объекта класса и лямбда-функции одинаково. Лямбда-выражения передают в функцию по значению, тип соответствующего аргумента функции делают шаблоном.

# 2 Библиотека алгоритмов

В данном разделе поговорим о некоторых шаблонных функциях, определённых в заголовочном файле <algorithm> и реализующих некоторые стандартные алгоритмы.

## 2.1 Немодифицирующие последовательные операции

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

int main() {
  std::vector<int> vec = {1, 4, 7, 43, 8, 2, 8, 1, 4, 6, 2, 2};

  /* Функция считает количество элементов, равных третьему аргументу функции
    в диапазоне, заданном первыми двумя аргументами --- полуинтервалом, определённым
    итераторами. */
  std::size_t num = std::count(vec.begin(), vec.end(), 1);

  /* То же самое, только третьим аргументом задаётся предикат --- условие поиска,
    например, посредством лямбды, принимающий элемент контейнера по значению или
    константной ссылке (если элемент тяжёлый) и возвращающей true, если элемент должен
    быть посчитан. */
  num = std::count_if(vec.begin(), vec.end(), [] (int x) {
    if (x > 2 && x < 10)  // Считаем количество элементов контейнера в интервале (2, 10)
      return true;
    return false;
  });

  std::vector<int>::iterator it;

  /* Функция возвращает на первый найденный элемент, равный третьему аргументу в
    полуинтервале, определяемом первыми двумя аргументами --- итераторами контейнера. */
  it = std::find(vec.begin(), vec.end(), 8); // it указывает на vec[4]

  /* То же самое, только третьим аргументом задаётся условие поиска, например,
```

```
29      посредством лямбды, принимающий элемент контейнера по значению или константной
30      ссылке (если элемент тяжёлый) и возвращающей true, если элемент подходит. */
31    it = std::find_if(vec.begin(), vec.end(), [](int x) { // Теперь it указывает на vec[3]
32      if(x > 10)       // Найдём первый элемент, больший 10
33        return true;
34      return false;
35    });
36  }
```

Кроме того, есть функция `std::find_if_not()`, отличающаяся от `std::find_if()` тем, что она возвращает итератор на первый элемент, для которого предикат вернул `false`.

Отметим, что если бы вектор в примере был бы константным, то его методы `begin()` и `end()` возвращали бы константный итератор (`const_iterator`), а поэтому и функции `std::find()` и `std::find_if()` возвращали бы константный итератор.

Приведённые функции работают для любых контейнеров, итераторы которых предоставляют хотя бы прямой последовательный доступ. Для корректной работы второй аргумент должен выводиться из первого последовательным инкрементом.

Заметим, что если в семейство функций `std::find()` подставить обратные итераторы, например, `rbegin()` и `rend()`, то подходящий элемент будет искаться с конца.

Сложность данных алгоритмов линейно зависит от длины полуинтервала.

## 2.2 Модифицирующие последовательные операции

```
1  std::vector<int> from = {1, 4, 87, 23, 5, 42, 87, 342, 87, 23, 35, 7};
2  /* В выходном векторе обязательно должно лежать необходимое для вывода количество
3    элементов. */
4  std::vector<int> to(10);
5
6  /* Функция копирует полуинтервал, определённый первыми двумя аргументами на позицию,
7    определяемую третьим аргументом. Третий параметр может быть итератором контейнера
8    другого типа (например, списка). Причём промежуток, в который запишутся данные должен
9    состоять из валидных итераторов, иными словами количество элементов в контейнере для
10   вывода должно позволять разместить указанный промежуток без увеличения количества
11   элементов в контейнере. */
12 // Копируем первые 5 элементов из from в начало to. Размер to должен быть не меньше 5.
13 std::copy(from.begin(), from.begin() + 5, to.begin());
14
15 /* То же самое, только четвёртым аргументом добавляется предикат --- условие
16   копирования. Если предикат возвращает true, то элемент будет скопирован.
17   Предикат принимает элемент исходного контейнера по значению или константной
18   ссылке (для тяжёлых элементов). */
19 // Копируем все числа, большие 9 из from в начало to. Размер to должен быть не меньше 8.
20 std::copy(from.begin(), from.end(), to.begin(), [](int elem) {
21   return elem >= 10;
22 });
23
24 /* Функция предназначена для "удаления" элементов. На самом деле она переставляет все
25   элементы полуинтервала, не равные третьему аргументу, в начало полуинтервала,
26   определённого первыми двумя аргументами. Физически функция не изменяет размер
27   контейнера. Функция возвращает итератор на новый конец полуинтервала. Начиная с этого
```

```
28   итератора значения в векторе будут не определены, это зависит от реализации
29   стандартной библиотеки. Функция сохраняет относительный порядок оставшихся элементов.
30   */
31 std::vector<int>::iterator it;
32 // Удали все элементы, равные 87.
33 it = std::remove_if(from.begin(), from.end(), 87);
34 /* Теперь в полуинтервале [from.begin(), it) содержатся элементы {1, 4, 23, 5, 42, 342,
35    87, 23, 35, 7}. Для физического удаления оставшихся элементов [it, from.end()) можно
36   использовать метод вектора erase(). */
37 from.erase(it, from.end()); // Теперь вектор равен {1, 4, 23, 5, 42, 342, 87, 23, 35, 7}
38
39 /* То же самое, только четвёртым аргументом добавляется предикат --- условие
40   "удаления". Если предикат возвращает true, то элемент будет "удфлён". */
41 // "Удалим" все цифры из вектора
42 it = std::remove_if(from.begin(), from.end(), to.begin(), [](int elem) {
43   return elem < 10;
44 });
45
46 from.erase(it, from.end()); // Физически удаляем лишние элементы.
47
48 /* Функция копирует элементы из полуинтервала, определённого первыми двумя аргументами,
49   на позицию начиная с третьего аргумента, пропуская элементы, равные четвёртому
50   аргументу. Третий аргумент может быть итератором контейнера другого типа (например,
51   итератором списка). Точно также, как и в функции std::copy(), в выходном контейнере
52   должно быть достаточно места. Функция возвращает итератор выходного контейнера,
53   стоящий после последнего добавленного элемента. */
54 /* Скопируем все элементы из from в начало вектора to кроме элементов, равных 23.
55   В векторе to должно быть место хотя бы под 4 элемента. */
56 it = std::remove_copy(from.begin(), from.end(), to.begin(), 23);
57 // В полуинтервале [to.begin(), it) лежат элементы {42, 342, 87, 35}.
58
59 /* То же самое, только с предикатом. Копируются все элементы кроме тех, для которых
60   предикат вернул true. */
61 // Скопируем все числа кроме трёхзначных
62 it = std::remove_copy_if(from.begin(), from.end(), to.begin(), [](int elem) {
63   return elem >= 100;
64 });
65 // В полуинтервале [to.begin(), it) лежат элементы {23, 42, 87, 23, 35}.
66
67 /* Функция заменяет элементы, равные третьему аргументу, на элементы, равные четвёртому
68   в полуинтервале, определённом первыми двумя аргументами. */
69 // Заменим все элементы, равные 23 на 11.
70 std::replace(vec.begin(), vec.end(), 23, 11);
71
72 /* То же самое только с унарным предикатом, возвращающим true, если элемент нужно
73   заменить. */
74 // Заменяем все элементы, большие 40, на 10.
75 std::replace_if(from.begin(), from.end(), [] (int elem) {
76   return elem > 40;
77 }, 10);
78
```

```
79 /* Есть аналогичные функции, записывающие результат в другой промежуток. Функция
80   возвращает итератор выходного контейнера, указывающий на элемент после последнего
81   добавленного. */
82 /* Копируем все элементы из from в to, заменяя 11 на 15.
83 it = std::replace_copy(from.begin(), from.end(), to.begin(), 11, 15);
84 // В полуинтервале [to.begin(), it) лежат элементы {15, 10, 10, 15, 35}.
85
86 /* То же самое только с унарным предикатом, возвращающим true, если элемент нужно
87   заменить. */
88 /* Копируем все элементы из from в to, заменяя элементы из (12, 20) на 17.
89 it = std::replace_copy_if(from.begin(), from.end(), to.begin() [] (int elem) {
90   return elem > 12 && elem < 20;
91 }, 17);
92 // В полуинтервале [to.begin(), it) лежат элементы {17, 10, 10, 17, 35}.
93
94 /* Функция переставляет элементы полуинтервала в обратном порядке. */
95 std::reverse(from.begin(), from.end());
96
97 /* Функция копирует элементы полуинтервала в другой промежуток исключая
98   последовательные одинаковые элементы. Функция возвращает итератор выходного
99   контейнера, указывающий на элемент после последнего добавленного.*/
100 it = std::unique_copy(from.begin(), from.end(), to.begin());
101
102 /* Функция "удаляет" одинаковые подряд идущие элементы. На самом деле она переставляет
103   оставшиеся элементы в начало промежутка. Физически удаления не происходит. */
104 it = std::unique(from.begin(), from.end());
105 from.erase(it, from.end()); // Физически удаляем лишние элементы.
```

Приведённые функции (кроме `std::reverse()`) работают для любых контейнеров, итераторы которых предоставляют хотя бы прямой последовательный доступ. Функция `std::reverse()` требует итератор, предоставляющий как прямой, так и обратный последовательный доступ. Для корректной работы второй аргумент должен выводиться из первого последовательным инкрементом.

Итераторы контейнера хранят лишь минимальную информацию, необходимую для итерирования по контейнеру и для чтения и/или изменения текущего элемента. Поэтому они не могут добавить или удалить элемент из контейнера. Таким образом, приведённые алгоритмы могут лишь модифицировать уже существующие элементы контейнера. Именно поэтому функция `std::copy()` требует, чтобы выходной промежуток целиком принадлежал выходному контейнеру, а функция `std::remove_if()` только переставляет элементы, не изменяя размер контейнера.

Отметим, что промежутки можно задавать посредством обратных итераторов. В этом случае приведённые алгоритмы будут перебирать входной промежуток в обратном порядке.

Сложность данных алгоритмов линейно зависит от длины полуинтервала.

## 2.3 Алгоритмы разбиения

Алгоритмы данного подраздела предназначены для разбиения промежутка на две части по какому-либо признаку. В каждый из этих алгоритмов передаётся унарный предикат, который принимает элемент контейнера по значению или константной ссылке (для тяжёлых объектов) и возвращающий `true`, если элемент должен располагаться в промежутке раньше тех элементов, для которых данный предикат возвращает `false`.

```cpp
std::vector<int> vec = {1, 3, 5, 7, 9, 2, 4, 6, 12};
/* Функция std::is_partitioned() возвращает true, если элементы полуинтервала,
  для которых предикат вернул true стоят перед элементами, для которых он вернул false.
  Полуинтервал определяется первыми двумя аргументами, а предикат~---третьим. */
bool isPartitioned = std::is_partitioned(vec.begin(), vec.end(), [] (int elem) {
  return elem % 2 == 1; // В данном случае нечётные элементы стоят перед чётными
});

/* Функция std::partition_point() возвращает итератор на элемент, следующий за
  последним элементом, удовлетворяющим предикату. Для корректной работы полуинтервал
  должен удовлетворять разбиению. Части разбиения могут быть пустыми. */
std::vector<int>::iterator it =
    std::partition_point(vec.begin(), vec.end(), [] (int elem) {
  return elem % 2 == 1; // В данном случае нечётные элементы стоят перед чётными
});
// Теперь it указывает на vec[4] = 2

/* Функция std::stable_partition() разбивает промежуток по определённому правилу,
  причём относительный порядок элементов каждой группы сохраняется. */
std::stable_partition(vec.begin(), vec.end(), [](int elem) {
  return elem % 3 == 0; // Переставим в начало элементы, делящиеся на три.
});
// Теперь в векторе лежат элементы {3, 9, 6, 12, 1, 5, 7, 2, 4};

/* Функция std::partition() разбивает промежуток по определённому правилу,
  без сохранения относительного порядка элементов каждой группы. */
std::stable_partition(vec.begin(), vec.end(), [](int elem) {
  return elem % 2 == 0; // Переставим в начало элементы, делящиеся на два.
});
```

Предположим, длина промежутка равна $N$. Алгоритм `std::is_partitioned()` имеет сложность не более $N$ вычислений предиката. Он требует итераторы, предоставляющие хотя бы прямой последовательный доступ на чтение.

Алгоритм `std::partition_point()` делает $O(\log N)$ вычислений предиката, однако, для итераторов не предоставляющих произвольный доступ делается $O(N)$ инкрементов. Он требует итераторы, предоставляющие хотя бы прямой последовательный доступ на чтение.

Алгоритм `std::stable_partition()` делает ровно $N$ вычислений предиката и не более $N \log N$ перестановок (может обойтись и $O(N)$ перестановками если достаточно лишней памяти). Он требует итераторы, предоставляющие хотя бы двусторонний последовательный доступ на чтение и запись.

Алгоритм `std::partiton()` делает ровно $N$ вычислений предиката и не более $N/2$ перестановок если итераторы предоставляют хотя бы двусторонний последовательный доступ на чтение и запись. В случае только прямого доступа делается не более $N$ перестановок.

## 2.4 Алгоритмы сортировки

Алгоритмы данного раздела помимо набора итераторов принимают некоторое правило для сравнения элементов, называемое компаратором. Компаратор может быть задан, например, лямбда-выражением, принимающем два элемента контейнера по значению или по константным ссылкам (для тяжёлых объектов) и возвращающего `true`, если первый аргумент меньше первого. Если для

сортируемых элементов определена операция сравнения “меньше” (<), то компаратор в алгоритм можно не передавать.

```
#include <vector>
#include <algorithm>
#include <string>

struct StringInt {
  std::string str;
  int num;
};
int main() {
  // Не определена операция сравнения
  std::vector<StringInt> vecStruct = {
      {"Sep", 30}, {"Oct", 31}, {"Nov", 30}, {"Dec", 31}
  };
 // Операция сравнения определена
  std::vector<std::pair<std::string, int>> vecPair = {
      {"Jan", 31}, {"Feb", 28}, {"Mar", 31}, {"Apr", 30}, {"May", 31}
  };

  /* Функция проверяет, отсортирован ли промежуток по возрастанию. */
  bool isSorted = std::is_sorted(vecStruct.begin(), vecStruct.end(),
      [](const StringInt& lhs, const StringInt& rhs) {
    if (lhs.str < rhs.str)
      return true;
    else if (lhs.str > rhs.str)
      return false;
    return lhs.num < rhs.num;
  });

  /* Если определена подходящая операция сравнения "меньше", то компаратор можно
    не передавать. */
  isSorted = std::is_sorted(vecPair.begin(), vecPair.end());

  /* Функция ищет итератор на конец наибольшего отсортированного по возрастанию
    полуинтервала, начиная от начала промежутка. Третьим аргументом можно передать
    в функцию произвольный компаратор. */
  std::vector<std::pair<std::string, int>>::iterator it =
      std::is_sorted_until(vecPair.begin(), vecPair.end());
  // В полуинтервале [vecPair.begin(), it) все элементы отсортированы.

  /* Функция сортирует промежуток по возрастанию. Третьим аргументом можно передать
    произвольный компаратор. Сохранение относительного порядка равных элементов
    не гарантируется. */
  std::sort(vecPair.begin(), vecPair.end());

  /* Функция сортирует промежуток по возрастанию, причём гарантируется сохранение
    относительного порядка равных элементов. Если определена операция сравнения "меньше"
    и она подходит по смыслу, то компаратор можно не передавать. */
```

```
  std::stable_sort(vecStruct.begin(), vecStruct.end(),
      [](const StringInt& lhs, const StringInt& rhs) {
    return lhs.num < rhs.num;
  });
  // Теперь vecStruct = {{"Sep", 30}, {"Nov", 30}, {"Oct", 31}, {"Dec", 31}}

  /* Первый аргумент (для краткости first) определяет начало исходного полуинтервала,
    третий (last) --- конец исходного полуинтервала [first, last). Второй аргумент
    (middle) определяет некоторую позицию между ними. Функция сортирует промежуток
    [first, last) таким образом, что в левом полуинтервале [first, middle)] лежат
    middle - first отсортированных по возрастанию наименьших элементов полного
    полуинтервала [first, last). Порядок элементов в правой части исходного
    полуинтервала []middle, last) не определён. Четвёртым аргументом можно передать
    произвольный компаратор. */
  it = vecPair.begin() + 3;
  std::partial_sort(vecPair.begin(), it, vecPair.end());
  /* Теперь в полуинтервале [vecPair.begin(), it) лежат в порядке возрастания три
    элемента исходного промежутка [vecPair.begin(),  vecPair.end()). */

  /* Первый аргумент (для краткости first) и третий аргумент (last) задают
    полуинтервал [first, last). Второй аргумент (nth) задаёт позицию между ними.
    Функция сортирует промежуток [first, last) таким образом, что
      1) в позиции nth стоит элемент, который стоял бы в ней, если бы промежуток
         [first, last) был отсортирован по возрастанию.
      2) все элементы в промежутке [first, nth) меньше или равны элементам
      промежутка (nth, last). Относительный порядок элементов в каждой части
    исходного полуинтервала не определён. Четвёртым аргументом можно передать
    произвольный компаратор. */
  // Найдём медиану вектора vecPair
  it = vecPair.begin() + vecPair.size() / 2;
  std::nth_element(vecPair.begin(), it, vecPair.end());
  std::cout << "Median is equal to " << it->first << " " << it->second << std::endl;
  return 0;
}
```

## 2.5 Алгоритмы поиска на отсортированных промежутках

В этом подразделе будем предполагать, что полуинтервал [first, last), определённый двумя итераторами отсортирован по возрастанию.

```
std::vector<int> vec = {1, 2, 3, 3, 9, 10, 11, 12, 13, 14};

/* Функция возвращает итератор на первый элемент полуинтервала [first, last),
  определяемого первым (для краткости first) и вторым (last) аргументами, который
  больше или равен третьего аргумента (value). Если такого элемента нет, то
  возвращается итератор на конец промежутка last. Четвёртым аргументом можно
  передать произвольный компаратор. */
std::vector<int>::iterator it = std::lower_bound(vec.begin(), vec.end(), 10);
// it указывает на vec[5] = 10

```

```
/* Функция возвращает итератор на первый элемент полуинтервала, больший третьего
  аргумента или итератор на конец полуинтервала в случае отсутствия такого элемента. */
it = std::upper_bound(vec.begin(), vec.end(), 10);
// it указывает на vec[6] = 11

/* Пример. Найдём последний элемент, меньший 10. Для этого передадим обратные итераторы
  и заменим оператор "меньше" на "больше". Заметим, что такой промежуток отсортирован
  по возрастанию относительно переданного компаратора. */
it = std::upper_bound(vec.rbegin(), vec.rend(), 10, [](int lhs, int rhs) {
  return lhs > rhs;
});
 it указывает на vec[4] = 9

/* Функция проверяет, присутствует ли элемент в промежутке и возвращает true, если
  присутствует. Четвёртым аргументом можно передать произвольный компаратор. */
bool hasElement = std::binary_search(vec.begin(), vec.end(), 12); // hasElement = true

using It = std::vector<int>::iterator; // Введём для краткости

/* Функция возвращает границы максимального полуинтервала, содержащего указанное
  значение. В поле first пары запишется итератор на первый элемент, больше или равный
  указанного значения или итератор на конец промежутка, если таковых не нашлось.
  В поле second пары записывается итератор на первый элемент, больший указанного
  значения или итератор на конец промежутка, если такого элемента не нашлось.*/
std::pair<It, It> equalRange = std::equal_range(vec.begin(), vec.end(), 3);
// equalRange.first указывает на vec[2], а equalRange.second на vec[4]
```

### 2.5.1 Использование алгоритмов поиска для классов std::map и std::set

Напомним, что итераторы не предоставляют доступ к внутренней структуре контейнера, поэтому для некоторых контейнеров стандартные алгоритмы могут оказаться крайне неэффективны.

Например, рассмотрим контейнеры, основанные на деревьях, а именно `std::map` и `std::set`. Поскольку их итераторы образуют промежуток, отсортированный по возрастанию ключа, то для них можно применять алгоритмы поиска на отсортированном промежутке. Однако, итераторы этих контейнеров предоставляют только последовательный доступ, следовательно количество инкрементов итератора будет линейно зависеть от длины промежутка $N$. Отметим, что один инкремент итератора может вызвать до $O(\log N)$ перемещений по узлам дерева.

Выходом из подобной ситуации является использование специальных алгоритмов, учитывающих структуру контейнера. Как правило они являются одноимёнными методами соответствующего класса.

```
std::map<std::string, int> stringToInt = {
  {"one", 1}, {"two", 2}, {"three", 3}, {"frour", 4}, {"five", 5}
};

std::set<std::string> stringSet = {
  "one", "two", "three", "four", "five"
};

// Объявим для краткости
```

```
using MapIt = std::map<std::string, int>::iterator;
using SetIt = std::set<std::string>::iterator;

/* Метод ищет элемент по ключу и возвращает итератор на этот элемент или итератор
   на конец контейнера, если такого элемента не нашлось. */
MapIt mapIt = stringToInt.find("two");
SetIt setIt = stringSet.find("two");

if (mapIt == stringToInt.end())
  std::cout << "Can't find element!" << std::endl;

/* Метод ищет наибольший промежуток с ключами, равными указанному. Работает аналогично
   функции std::equal_range(). */
std::pair<MapIt, MapIt> equalRange = stringToInt.equal_range("two");
std::pair<SetIt, SetIt> equalRange = stringSet.equal_range("two");

/* Метод возвращает итератор на элемент с ключом, большим или равным указанному
   или итератор на конец контейнера, если такового элемента не нашлось. */
mapIt = stringToInt.lower_bound("three");
setIt = stringSet.lower_bound("three");

/* Метод возвращает итератор на элемент с ключом, большим указанного или итератор на
   конец контейнера, если такового элемента не нашлось. */
mapIt = stringToInt.upper_bound("three");
setIt = stringSet.upper_bound("three");
```

## 2.6 Алгоритмы поиска максимального/минимального элементов

```
std::vector<int> vec = {1, 54, 34, 67, 123, 5, 12, 86, 23, 78, 23};

using VecIt = std::vector<int>::iterator; // Для краткости

/* Функция возвращает итератор на максимальный элемент промежутка или итератор на
   конец, если промежуток пустой. Третьим аргументом можно передать произвольный
   компаратор. */
VecIt it = std::max_element(vec.begin(), vec.end());

/* Функция возвращает итератор на минимальный элемент промежутка или итератор на
   конец, если промежуток пустой. Третьим аргументом можно передать произвольный
   компаратор. */
VecIt it = std::min_element(vec.begin(), vec.end());

/* Возвращает пару итераторов. В поле first итератор на минимальный элемент,
   в поле second итератор на максимальный элемент. В случае пустого промежутка запишет
   итератор на конец промежутка в оба поля. Третьим аргументом можно передать
   произвольный компаратор. */
std::pair<VecIt, VecIt> pairOfIt = std::minmax_element(vec.begin(), vec.end());
```

## 2.7 Численные алгоритмы

Все алгоритмы данного подраздела содержатся в заголовочном файле `<numeric>`.

```
std::vector<int> vec(10);

/* Функция заполняет промежуток последовательными значениями, начиная с указанного. */
std::iota(vec.begin(), vec.end(), -3);

/* Функция считает сумму элементов. Начальное значение инициализируется третьим
  аргументом. */
int res = std::accumulate(vec.begin(), vec.end(), 0);

std::map<std::string, int> stringToInt = {
  {"one", 1}, {"two", 2}, {"three", 3}, {"frour", 4}, {"five", 5}
};

/* Можно задать правило сложения, первый аргумент которого имеет тип, соответствующий
  начальному значению, а второй аргумент соответствует типу данных, полученному при
  разыменовании итератора. */
res = std::accumulate(stringToInt.begin(), stringToInt.end(), 0,
    [](int lhs, const std::pair<const std::string, int>& rhs) {
  return lhs + rhs.second;
});
```

# 3 Библиотека итераторов

Здесь поговорим о дополнительных классах и функциях, определённых в заголовочном файле `<iterator>`.

## 3.1 Адаптеры итераторов

Адаптеры итераторов предназначены для того, чтобы решить проблему, связанную с тем, что обычные итераторы контейнера не могут изменять его размер. Эти адаптеры можно передавать в алгоритмы вместо итераторов на начало выходного промежутка. Как правило они содержат дополнительную информацию, позволяющую добавлять элементы в контейнер, например, ссылку на него.

Существуют адаптеры для вставки элемента в конец контейнера (`std::back_insert_iterator`), в начало контейнера (`std::front_insert_iterator`), а также в произвольную позицию контейнера (`std::insert_iterator`). Для их создания используются функции `std::back_inserter()`, `std::front_inserter()` и `std::inserter()` соответственно.

Принцип работы этих адаптеров можно понимать так: их операция разыменования (`*`) возвращает сам адаптер, а присваивание (`=`) вызывает у контейнера метод `push_back()`, `push_front()` или `insert()` соответственно. Таким образом, предложенные адаптеры можно создавать только для тех контейнеров, у которых есть соответствующие методы.

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

```
5
6  // Оператор для вывода содержимого контейнера в поток
7  template<typename Container>
8  std::ostream& operator<<(std::ostream& out, const Container& container) {
9    for (auto& elem : container)
10     out << elem << " ";
11   return out;
12 }
13
14 int main() {
15   std::vector<int> vecFrom = {1, 2, 3, 4, 5, 6};
16   std::vector<int> vectorTo;
17   std::deque<int> dequeTo;
18   std::list<int> listTo = {10, 20, 30};
19   std::set<int> setTo;
20
21   /* Шаблонная функция std::back_inserter() принимает контейнер, куда нужно вставлять
22     элементы и возвращает итератор std::back_insert_iterator. */
23   // Вставка в конец вектора vectorTo
24   std::copy(vecFrom.begin(), vecFrom.end(), std::back_inserter(vectorTo));
25
26   std::cout << "vectorTo = " << vectorTo << std::endl;
27
28   /* Шаблонная функция std::front_inserter() принимает контейнер, куда нужно вставлять
29     элементы и возвращает итератор std::front_insert_iterator. */
30   // Вставка в начало дека dequeTo
31   std::copy(vecFrom.begin(), vecFrom.end(), std::front_inserter(dequeTo));
32
33   std::cout << "dequeTo = " << dequeTo << std::endl;
34
35   /* Шаблонная функция std::inserter() принимает контейнер, куда нужно вставлять
36     элементы и позицию, на которую нужно вставить элемент и возвращает итератор
37     std::insert_iterator. */
38   // Вставка между первым и вторым элементами списка listTo
39   std::copy(listTo.begin(), listTo.end(),
40             std::inserter(listTo, std::next(listTo.begin())));
41
42   std::cout << "listTo = " << listTo << std::endl;
43
44   /* Поскольку множество само определяет, куда нужно вставить элемент, то второй
45     параметр функции std::inserter() игнорируется и может быть любым итератором
46     контейнера. */
47   // Вставка в множество
48   std::copy(setTo.begin(), setTo.end(), std::inserter(setTo, setTo.end()));
49
50   std::cout << "setTo = " << setTo << std::endl;
51   return 0;
52 }
```

## 3.2 Потоковые итераторы

Эти итераторы позволяют читать элементы контейнеров из произвольного потока, а также записывать элементы контейнера в произвольный поток.

Для чтения элементов из потока используют шаблон `std::istream_iterator` с одним обязательным шаблонным параметром — типом данных элемента, который получится при разыменовании итератора. Конструктор по-умолчанию возвращает аналог итератора на конец промежутка. Если в конструктор передать поток, то он вернёт аналог итератора на начало промежутка.

Для вывода элементов в поток используют шаблон `std::ostream_iterator` с одним обязательным шаблонным параметром — типом данных элемента, который получится при разыменовании итератора. Он принимает два аргумента: первый — поток, в который будут выводиться элементы и второй необязательный аргумент — строку, которая будет выводиться после каждого элемента. Если разделитель не указать, то элементы будут выводиться слитно. Данный шаблон может использоваться как итератор на выходной промежуток в таких функциях как, например, `std::copy()`.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <sstream>

int main() {
  std::vector<int> vec;
  std::stringstream strm;

  // Запишем числа в поток
  strm << "1  5 2 7 234 7 123 76 2 45";

  // Добавим все элементы в вектор при помощи функции std::copy().
  std::copy(std::istream_iterator<int>(strm), // Итератор на начало промежутка
            std::istream_iterator<int>(),     // Итератор на конец промежутка
            std::back_inserter(vec));

  // Выведем через запятую и пробел элементы вектора в стандартный поток вывода.
  std::copy(vec.begin(), vec.end(), std::ostream_iterator<int>(std::cout, ", "));

  // В поток выведется "1, 5, 2, 7, 234, 7, 123, 76, 2, 45, "
  return 0;
}
```

# Список литературы

[1] https://en.cppreference.com/w/cpp/algorithm

[2] https://en.cppreference.com/w/cpp/language/lambda