Содержание

4 Итераторы, алгоритмы, контейнеры **2**

4.1 Введение в итераторы . . . 2

4.1.1 Введение в итераторы . . . 2

4.1.2 Концепция полуинтервалов итераторов . . . 4

4.1.3 Итераторы множеств и словарей . . . 6

4.1.4 Продвинутое итерирование по контейнерам . . . 7

4.2 Использование итераторов в алгоритмах и контейнерах . . . 8

4.2.1 Использование итераторов в методах контейнеров . . . 8

4.2.2 Ипользование итераторов в алгоритмах . . . 9

4.2.3 Обратные итераторы . . . 10

4.2.4 Алгоритмы, возвращающие набор элементов . . . 11

4.2.5 Итераторы inserter и back_inserter . . . 13

4.2.6 Отличия итераторов векторов и множеств . . . 13

4.2.7 Категории итераторов, документации . . . 14

4.3 Очередь, дек и алгоритмы поиска . . . 14

4.3.1 Стек, очередь и дек . . . 14

4.3.2 Алгоритмы поиска . . . 16

4.3.3 Анализ распространённых ошибок . . . 18

Неделя 4

Итераторы, алгоритмы, контейнеры

4.1 Введение в итераторы

4.1.1 Введение в итераторы

Рассмотрим задачу. Пусть у нас есть вектор строк с языками программирования:

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

using namespace std;

int main() {
  vector<string> langs =
    {"Python", "C++", "C", "Java", "C#"};

  return 0;
}
```

Зададимся целью найти в нашем векторе язык, начинающийся с буквы 'C' или сказать, что такого языка нет. Можно написать цикл for и проверять первую букву каждого языка, но чтобы знать, нашёлся язык или нет, нам нужно хранить флажок типа bool и с каждым шагом все больше шанс ошибиться. Но существует стандартный алгоритм find_if(), принимающий начало диапазона, конец диапазона и лямбда-функцию, которая ищет нужный язык. Дописываем строчку:

```
#include <algorithm> //подключаем модуль с алгоритмами
...
auto result = find_if(
    begin(langs), end(langs),
    [](const string& lang) { //лямбда-функция по языкам
      return lang[0] == 'C'; //возвращает true, если нулевая буква = 'C'
    });
cout << *result; //вызываем * от find_if, чтобы обратиться к найденному элементу
//C++
```

Видим, что нашли первый подходящий язык. Этот результат можно сохранить в какую-нибудь строчку:

```
string& ref = *result; //сохраняем в строку
cout << ref << endl;  //выводим
```

Причём, поскольку *result неконстантная ссылка, мы по ней можем поменять элемент. Например:

```
string& ref = *result; //сохраняем в строку C++
ref = "D++";
cout << *result << endl;
//D++
```

А если у нас в векторе лежат не строки, а более сложные структуры:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Lang { //у каждого языка программирования есть название и возраст
  string name;
  int age;
};
int main() {
  vector<Lang> langs = //тоже поменяли на Lang
    {{"Python", 26},
   { "C++", 34},
   {"C", 45},
   { "Java", 22},
   { "C#", 17}};

  auto result = find_if(
    begin(langs), end(langs),
    [](const Lang& lang) { //лямбда-функция немного меняется
    return lang.name[0] == 'C';
  });
  if (result == end(langs)) {//если результат выводит ссылку на конец диапазона
    cout << "Not found";// значит, элемент не найден
  } else {       //иначе мы нашли элемент
  cout << (*result).age << endl;  // выводим возвраст первого попавшегося языка на C
  }
  return 0;
}
//34
```

Если результат не нашёлся, то result указывает на конец диапазона. Это значит, что мы можем спокойно проверить на наличие. Кроме того, обращаться можно короче:

```
cout << result->age << endl; //более удобная форма
```

А для языков на букву 'D' получим: Not Found. И begin(), end() и result указывают на какую-то позицию в контейнере. Все типы, указывающие на какую-то позицию контейнера называются **итераторами**. А операция * над итератором называется **разыменованием итератора**.
Выведем выведем начало контейнера:

```
cout << begin(langs)->name << " " << langs.begin()->age << endl;
//Python 26
```

А вот end(langs) уже указывает не на последний элемент контейнера, а на конец. Заметим, что к началу можно обращаться методом langs.begin().

4.1.2 Концепция полуинтервалов итераторов

Итераторы есть не только у вектора, но и у любого контейнера. Например, у строки:

```
#include <string>
...
string lang = langs[1].name;
auto it = begin(lang);
cout << *it;
//C
```

begin() у строки указывает на её первый символ. Пока что мы с их помощью только получали элемент контейнера, но из название следует, что с их помощью можно итерироваться по контейнеру. Допишем:

```
++it;// получим следующий символ
cout << *it
//C+
```

Теперь мы вывели сначала нулевой элемент, а затем первый. Давайте с помощью цикла for проитерируемся по вектору langs:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Lang {
  string name;
  int age;
};
int main() {
  vector<Lang> langs = //тоже поменяли на Lang
    {{"Python", 26},
    { "C++", 34},
    {"C", 45},
    { "Java", 22},
    { "C#", 17}};
  for (auto it = begin(langs); it !=end(langs); ++it) {
    cout << it-> name << " ";
  }
  return 0;
}
//Python C++ C Java C#
```

Таким образом мы вывели все языки. Когда итератор показывал на begin(langs), выводится первый элемент. Итератор двигается вправо, проверяет, что он не достиг конца и т.д. Выводим последний элемент и сдвигаем итератор на end. Т.е. end(langs) это итератор сразу за последним элементом. Получается полуинтервал $[begin\ ,\ end)$.
Для 5 элементов получаем 5 элементов и end. А вот пустой диапазон представляется из равных итераторов begin и end. Попробуем обраться к полю end(langs). Код компилируется, но падает, потому что там либо чужая память, либо вообще ничего не лежит.
Напишем универсальную функцию PrintRange, принимающую два итератора на Lang:

```
using LangIt = vector<string>::iterator; //укорачиваем запись
void PrintRange(
        LangIt range_begin, //начало
        LangIt range_end) { //конец
  for (auto it = range_begin; it !=range_end; ++it) {
    cout << *it << " "; //мы не хотим переопределять вывод структур
  }
}
int main() {
  vector<string> langs =  {"Python", "C++", "C", "Java", "C#"};

  PrintRange(begin(langs), end(langs));
  return 0;
}
//Python C++ C Java C#
```

Всё снова работает. Вспомним, что итераторы есть для разных контейнеров, и хотелось бы написать универсальный итератор. Цикл for уже достаточно универсален. Напишем шаблонную функцию:

```
template <typename It> //написали шаблонную функцию
void PrintRange(
  It range_begin, //поменяли здесь на шаблон итератора
  It range_end) { //и тут
  for (auto it = range_begin; it !=range_end; ++it){
    cout << *it << " ";
  }
}
...
PrintRange(begin(langs), end(langs)); //проходимся по контейнеру с языками
PrintRange(begin(langs[0]), end(langs[0]));//проходимся по контейнеру с python
//Python C++ C Java C# P y t h o n
```

Теперь, используя концепцию полуинтервалов, попробуем вывести половину векторов поотдельности, например, все языки строго до "C" и отдельно все языки начиная с него:

```
auto border = find(begin(langs),end(langs), "C"); //раздел
PrintRange(begin(langs), border); //Python, C++
PrintRange(border, end(langs)); //C, Java, C#
```

Мы просто разбили по нужному языку наш полуитервал на два диапазона.
Заключение:

- Итератор - способ задать позицию в контейнере;
- begin(c) и end(c) - границы полуинтервала $[begin, end)$;
- Алгоритмы часо принимают пару итераторов, образующую полуинтервал;
- С помощью шаблонов легко написать свой универсальный алгоритм, например, PrintRange.

4.1.3 Итераторы множеств и словарей

Давайте посмотрим, как работает итератор от других контейнеров. Увидим, что, по сути, они работают точно так же. Например, множество. Давайте возьмем наш вектор языков и заменим его на множество языков.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <set> //подключили множества
using namespace std;
template <typename It> //написали шаблонную функцию
void PrintRange(
  It range_begin, //поменяли здесь на шаблон итератора
  It range_end) { //и тут
  for (auto it = range_begin; it !=range_end; ++it) {
    cout << *it << " ";
  }
}

int main() {
  set<string> langs = {"Python","C++","C","Java","C#"};
  PrintRange(begin(langs), end(langs));
  //тоже выводит названия но в алфавитном порядке
  auto it = langs.find("C++");
  PrintRange(begin(langs), it);
  //выведем все языки до C++, в отсортированном порядке
  return 0;
}
//C C# C++ Java Python C C#
```

Получим вывод сначала всех строк, а потом строк, которые лексикографически меньше C++
Теперь рассмотрим словарь. Но для вызова PrintRange должен быть определён оператор <<

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map> //подключили словари
using namespace std;
void PrintRange...
int main() {
  map<string, int> langs = //словарь Имя-возраст
    {{"Python", 26},
    { "C++", 34},
    {"C", 45},
    { "Java", 22},
    { "C#", 17}};
  return 0;
}
```

Какой тип у элементов словаря? Когда вы итерировались по нему range based for-ом, этот тип был парой. И здесь *it это пара. Переопределять оператор вывода для словаря мы уже умеем.

Создадим функцию PrintMapRange для вывода словаря:

```
void PrintMapRange( It range_begin,It range_end){
  for (auto it = range_begin; it !=range_end; ++it){
  cout << it->first << '/' << it->second << " ";
  }
}
...
  auto it = langs.find("C++");
  PrintMapRange(begin(langs), it); //вызовем наш
  return 0;
}
//C/45 C#/17
```

Видим, что вывелись нужные нам языки и их возврast

4.1.4 Продвинутое итерирование по контейнерам

Продемонстрируем, что итераторы универсальнее чем range-based for, что их можно использовать в гораздо более широком числе случаев. Вернёмся к множеству языков. Сделаем множество строк с языками. И с помощью итераторов выведем множество в обратном порядке с помощью –it

```
#include <iostream>
#include <vector>
#include <algorythm>
using namespace std;
int main() {
  vector<string> langs = {"Python","C++","C","Java","C#"};
  auto it = end(langs);
  while (it != begin(langs)){
    --it;
    cout << *it << " ";
  }
}
//C# Java C C++ Python
```

Всё работает. Итерирование в обратную сторону работает так: проверяем, что it !=begin, двигаем его влево и уже можем вывести последний элемент. И т.д. пока не доходим до начала. Мы вывели вектор в обратном порядке. Кроме того, тот же код можно переписать в виде:

```
while (it != begin(langs)) { //заметим, что нельзя делать -it от it=begin
  --it;
  cout << *it << " ";
}
```

Опасные операции над итераторами:

- *end(c) - обращаться к элементу после последнего;
- auto it = end(c); ++it; - смотреть на элемент после end;
- auto it = begin(c); –it; - смотреть на элемент до begin.

Итераторы очень похожи на ссылки, но есть разница.
Отличия итераторов от ссылок:

- Итераторы могут указывать "в никуда" - на end. Ссылка всегда привязана к чему-то
- Итераторы можно перемещать на другие элементы:

```
vector<int> numbers = {1, 3, 5};
auto it = numbers.begin();
++it; //it указывает на 3
int& ref = numbers.front();
++ref; //теперь numbers[0] == 2
```

В итоге мы узнали, что у всех контейнеров есть итераторы, и чем они отличаются от ссылок. И итераторы предоставляют универсальный способ обхода контейнеров.

4.2 Использование итераторов в алгоритмах и контейнерах

4.2.1 Использование итераторов в методах контейнеров

Где мы раньше встречали итераторы?

1. В алгоритмах, например sort, count, count_if, reverse, find_if;
2. Конструкторы вектора и множества;
3. Метод find у множества.

Рассмотрим методы контейнеров на примере верктора строк.

```
vector<string> langs = {"Python","C++","C","Java","C#"};
auto it = find(begin(langs), end(langs), "C++"); //получаем позицию C++
langs.erase(it); //метод удаления элемента из контейнера по итератору it
langs.insert(begin(langs), "C++") //вставка перед итератором элемента C++
langs.erase(it, end(langs)); //удалили все с C++ и до конца
//C# Java C C++ Python
```

У вектора длины 5 есть 6 позиций для вставки от v.insert(begin(), value) до v.insert(end(), value). Вставка в произвольное место вектора:

- Вставка диапазона

```
v.insert(it, range\_begin, range\_end)
//вставит диапазон [range_begin ,range_end ) в позицию it
```

- Вставка элемента несколько раз

```
v.insert(it,{1, 2, 3})
//вставляет 1, 2, 3 в позицию it
```

- Вставка набора элементов

```
v.insert(it, range\_begin, range\_end)
//вставит диапазон [range_begin ,range_end ) в позицию it
```

4.2.2 Ипользование итераторов в алгоритмах

Рассмотрим алгоритмы, которые можно делать над векторами.

```
vector<string> langs = {"Python","C++","C","Java","C#"};
remove_if(begin(langs), end(langs),
  [](const string& lang){  //лямбда-функция, по которой удаляем

  return lang[0] == 'C' //хотим удалить все языки, начинающиеся на C
});
PrintRange(begin(langs), end(langs));
//Java Python C C#
```

Но у нас не удалились C и C#. Оказывается даже у стандартных алгоритмов C++ итераторы — это настолько общая концепция, что они не позволяют обратиться к исходным контейнерам. Поэтому алгоритм remove_if может лишь подсказать вам, помочь вам удалить что-то из контейнера. Он помог. Как он помог? Всё, что должно в контейнере остаться, перебросил в начало этого контейнера. И он вернул новый конец вашего вектора, то, что должно стать его концом. И теперь ваша задача сделать то, что вернул remove_if, новым концом вектора. Как это можно сделать? Например, с помощью метода erase.

```
langs.erase(it, end(langs));//удаляем начиная с итератора и до конца
//Java Python
```

Ещё раз: remove_if ничего не удаляет. Он лишь помогает нам удалить, потому что общие алгоритмы не могут влиять на контейнер (например, изменять его размер).

```
vector<string> langs = {"Python","C++","C++","Java","C++"};
auto it=unique(begin(langs),end(langs)); //перемещаем подряд идущие повторы в конец
langs.erase(it, end(langs)) //удаляем повторяющиеся, которые выкинуты в конец
PrintRange(begin(langs), end(langs));
//Python C++ Java C++
```

Таким образом мы удалили подряд идущие дубликаты элементов. Получается, что можно оставить в векторе только уникальные элементы, сначала их отсортировав, потом удалив повторы, то есть сначала sort, потом unique, и даже не нужно создавать их множество для этого, так получается экономнее.
Еще есть алгоритм нахождения минимума в массиве:

```
vector<string> langs = {"Python","C++","C","Java","C#"};
auto it = min_element(begin(langs), end(langs)); //кладём в итератор мин. элемент
cout << *it << endl;  //min_element может вернуть end(langs), только если он пуст
//C
```

```
auto it = max_element(begin(langs), end(langs)); // максимальный элемент
cout << *it << endl;
//Python
```

```
auto p = minmax_element(begin(langs), end(langs)); //пара - min и max в контейнере
cout << *p.first << ' ' << *p.second << endl;
//C Python
```

Таким образом, мы умеем получать минимальный и максимальный элементы в векторе. Но имеют ли смысл такие алгоритмы на множестве? Ответ - нет. На множестве (set) нам достаточно взять

it =begin(langs) для получения минимального элемента и it = end(langs); –it; для получения максимального.
Теперь попробуем вызвать remove для элементов множества:

```
set<string> langs = {"Python","C++","C","Java","C#"};
remove(begin(langs), end(langs), "C");
```

Но оно не скомпилировалось и выдало много ошибок. Вспомним, что делает remove - он переставляет элементы в конец для удаления. А множество не даёт переставлять элементы.
Но есть, например, алгоритм проверяющий какое-то свойство для всего диапазона

```
set<string> langs = {"Python","C++","C","Java","C#"};
all\_of(begin(langs), end(langs), [](const string& lang) {
  return lang[0] >= 'A' && lang[0] <= 'Z'; //все названия с большой буквы
});
//1
```

Все элементы начинаются с заглавной английской буквы и алгоритм выдал 1. Если изменить что-то на малую, начнёт выводить 0. Что мы узнали?

1. Методы вектора, позволяющие вставить в конкретное место вектора или удалить: insert, erase.

2. Алгоритмы

   - remove_if - выкидывает в конец вектора элементы для удаления;
   - unique - выкидывает в конец идущие подряд дублирований элементов;
   - min_element, max_element, minmax_element - min, max элементы;
   - all_off - проверка условия для всех элементов.

4.2.3 Обратные итераторы

Бывают не только обычные итераторы. Например, вывод в обратном порядке можно сделать так:

```
vector<string> langs = {"Python","C++","C","Java","C#"};
PrintRange(rbegin(langs), rend(langs)); //обратные итераторы
//C# Java C C++ Python
```

rbegin() и rend() от слов reverse - перевёрнутый. Это **обратные итераторы**. Причём *rbegin(langs) = "C#" , а вот *rend() не скомпилируется.

```
vector<string> langs = {"Python","C++","C","Java","C#"};
auto it = rbegin(langs); //итератор на последний элемент
cout << *it << " "; //выводим
++it;
cout << *it << " "; //сдвинули итератор и вывели предпоследний элемент
//C# Java
```

Если begin и end имеют тип vector<string>::iterator, то rbegin и rend - vector<string>::reverse_iterator.

Python	C++	C	Java	C#	
	Python	C++	C	Java	C#

begin указывает на Python, а end за C#
а вот rbegin уже на C# и rend перед Python
Передадим обратные итераторы в наши алгоритмы

```
auto result = find_if(
      rbegin(langs), rend(langs),
      [](const string& lang) { //лямбда-функция по языкам
      return lang[0] == 'C'; //возвращает true, если нулевая буква = 'C'
});
//C#
```

Раньше find_if возвращал первый подходящий элемент, а теперь последний (первый для диапазона обратных итераторов).
Вызовем sort от обратных итераторов и посмотрим на результат

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map> //подключили словари
using namespace std;
template <typename It> //написали шаблонную функцию
void PrintRange(
  It range_begin, //поменяли здесь на шаблон итератора
  It range_end){ //и тут
  for (auto it = range_begin; it !=range_end; ++it){
    cout << *it << " ";
  }
}

int main() {
  vector<string> langs = {"Python","C++","C","Java","C#"};
  sort(rbegin(langs), rend(langs)); //сортируем по убыванию
  PrintRange(begin(langs), end(langs));
  return 0;
}
//Python Java C++ C# C
```

reverse для обратных операторов работает как и для прямых (переворачивает).
В итоге обратные итераторы упрощают итерирование по контейнеру в обратную сторону и могут быть переданы в алгоритмы. A sort(rbegin(langs), rend(langs)); - простой способ сортировки вектора по убыванию.

4.2.4 Алгоритмы, возвращающие набор элементов

Мы рассматривали алгоритм remove_if, который позволяет удалить элемент по какому-то критерию в массиве, в диапазоне элементов в векторе. А что если мы хотим эти элементы не удалить, а аккуратно отложить, например, в конец вектора? Для этого есть алгоритм partition. Я вызвал алгоритм partition от диапазона, который я хочу разбить на две части, и передаю критерий, по которому я хочу разбить.

```
vector<string> langs = {"Python","C++","C","Java","C#"};
auto it = partition(begin(langs), end(langs), [](const string& lang){
    return lang[0] == 'C'; //делим по принципу "оканчивается или не оканчивается на C"
  });
PrintRange(begin(langs), end(langs));
//C# C++ C Java Python
```

Сначала идут все языки, которые начинаются с буквы C в каком-то порядке, как получилось: C#, C++ и C, а потом все остальные языки. То есть все элементы, которые удовлетворяют критерию, для которых λ-функция возвращает true, перемещаются в начало диапазона. Причём мы можем сохранить результат (последний элемент, который удовлетворяет условию) partition в итератор it. Если мы хотим переложить какие-то элементы из одного вектора в другой, используем алгоритм copy_if:

```
//исправим PrintRange, чтобы выводил через запятую
...
vector<string> langs = {"Python","C++","C","Java","C#"};
vector<string> c_langs(langs.size()); //вектор, куда мы копируем, должен быть
   объявлен
auto it = copy_if(begin(langs), end(langs), begin(c_langs),
    [](const string& lang) {
  return lang[0] == 'C';
});
PrintRange(begin(c_langs), end(c_langs));
//C++, C, C#, , ,
```

Заметим, что размер остался равным 5. Итераторы не могут менять размер векторов. Но copy_if копирует подходящие под условие элементы и возвращает в it указатель на новый конец вектора, в который копировали.
Теперь поговорим о set-ах. Что можно делать с множествами в математике? Объединять, пересекать, вычитать. Рассмотрим алгоритмы C++, которые помогают делать это же с множествами здесь

```
//исправим PrintRange, чтобы выводил через запятую
set<int> a = {1, 8 ,3}
set<int> b = {3, 6 ,8}
vector<int> v(a.size()); //вектор для хранения результата размера как set a
auto it = set_intersection(begin(a), end(a), begin(b), end(b), begin(v));
//intersection принимает два полуинтервала и итератор, куда сохранять результат
PrintRange(begin(v), end(v));
//3, 8, 0,
```

В итоге мы имеем пересечение множеств и 0, который дополняет до размера a. Сам set_interselection возвращает итератор за концом итогового пересечения. Т.е.

```
PrintRange(begin(v), it);
//3, 8,
```

Замечание: если мы не укажем явно размер вектора, в который помещаем результат, то код упадёт.

4.2.5 Итераторы inserter и back_inserter

Рассмотрим более удобный способ.

```
#include <iterator>
...
vector<string> langs = {"Python","C++","C","Java","C#"};
vector<string> c_langs; //о размере уже не беспокоимся
copy_if(begin(langs), end(langs),
    back_inserter,  //специальный итератор, вставляющий в конец
    [](const string& lang){ //привычная лямбда функция
  return lang[0] == 'C'; //снова выделяем только начинающиеся на C
  });
PrintRange(begin(c_langs), end(c_langs));
//C++, C, C#,
```

Вектор имеет нужный размер и не содержит лишних элементов. Итераторы умеют делать лишь ограниченный набор действий: * (ссылка на конкретный элемент), ++, – и сравнение итераторов. А итератор back_inserter (если с ним происходят перечиленные операции) делает push-back в контейнер, к которому он относится.
Аналогично с алгоритмом set_intersection у множества есть просто insert:

```
set<int> a = {1, 8 ,3}
set<int> b = {3, 6 ,8}
set<int> res;
auto it = set_intersection(begin(a), end(a), begin(b), end(b),
    inserter(res, end(res)) ); //итератор для вставки в множество
PrintRange(begin(res), end(res));
```

inserter возвращает специальный итератор, который вставляет элемент в множество

4.2.6 Отличия итераторов векторов и множеств

Рассмотрим задачу: Найти в векторе строк язык, который начинается с буквы 'C', и найти позицию элемента в векторе(а не итератор).

```
vector<string> langs = {"Python","C++","C","Java","C#"};
vector<string> c_langs(langs.size());
auto it = find_if(begin(langs), end(langs),
  [](const string& lang) {
    return lang[0] == 'C';
});
cout << it - begin(langs) << endl; //таким образом получаем номер элемента в векторе
PrintRange(begin(c_langs), end(c_langs));
//1
```

Для получения номера элемента достаточно из полученного с помощью find_if вычесть итератор начала диапазона. Получим 1. Действительно, у "C++" в векторе номер 1. Таким образом, **итераторы вектора можно вычитать друг из друга**. Но такое не сработает для итераторов множества т.к. для них не определена операция минус. Аналогично итераторы вектора можно сравнивать не только = и !=, но и < и >. А для множеств - только равно и не равно.
Попробуем для множества чисел вывести все элементы строго большие его самого.

```
set<int> s = {1, 6, 8, 9};
auto it = s.find(6);
++it;
PrintRange(it, end(s));
//8, 9,
```

Но часто бывает, что it нельзя изменить и поэтому нельзя писать it+1. Но этого можно избежать операцией next(it). По-сути она прибавляет 1 с помощью ++.

```
set<int> s = {1, 6, 8, 9};
auto it = s.find(6);
PrintRange(next(it), end(s));
//8, 9,
```

Точно так же работает функция prev, только вычитает один.

4.2.7 Категории итераторов, документации

Научимся читать документацию по языку C++ с помощью сайта ru.cppreference.com. Сейчас нас интересует документация по алгоритмам. Посмотрим на unique_copy.

```
template< class InputIt, class OutputIt >
ForwardIt unique_copy(InputIt first, InputIt last, OutputIt d_first );
```

Видим, что это шаблонная функция, которая принимает InputIt first, InputIt last, OutputIt d_first. Категории итераторов в документации:

- **Input:** итераторы, из которых можно читать (итераторы любых контейнеров). Но не подходят inserter и back_inserter;
- **Forward, Bidir:** обычные итераторы, из которых можно читать (кроме set и map, если по forward итератору что-то меняется);
- **Random:** итераторы, к которым можно прибавлять число или вычитать друг из друга (только итераторы векторов и строк);
- **Output:** итераторы, в которые можно писать (итераторы векторов и строк, inserter и back_inserter).

Например, partial_sort принимает Random итераторы, потому что он сортирует (переставляет элементы) и пользуется функцией.

4.3 Очередь, дек и алгоритмы поиска

4.3.1 Стек, очередь и дек

Рассмотрим новый контейнер: **очередь**. Очередь бывает из людей или запросов. Новые приходят в конец и удаляются из начала (или наоборот). Её можно реализовать с помощью вектора:

```
v.push_back(x); //добавляем новый
v.erase(begin(v)); //удаляем с помощью удаления из начала вектора, что очень долго
```

Элементы вектора хранятся подряд, и поэтому удаление из начала вектора будет работать за длину вектора, потому что он будет переставлять все элементы в начало, чтобы заполнить полученную пустоту: удалили 0ый, переместили 1ый на 0ое место, 2ой на 1ое и т.д.
Для работы с очередью есть специальный контейнер - **deque** (double-ended queue)

- это двусторонняя очередь;
- #include <deque>;
- Быстрые операции:

```
d.push_back(x)     //добавление в конец
d.pop_back(x)      //удаление из конца
d.push_front(x)    //добавление в начало
d.pop_front(x)     //удаление из начала
d[i]          //обращение к элементу по индексу
```

На коде представленном ниже продемонстрируем скорость работы deque:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
  int n = 80000;
  vector<int> v(n);
  while(!v.empty()) {
    v.erase(begin(v));
  }
  cout << "Empty!" << endl;
  return 0;
}
```

Т.е. мы сделали примерно $80000^2/2$ операций (т.к. каждый раз удаляли и перемещали).

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
int main() {
  int n = 80000;
  deque<int> v(n);
  while(!v.empty()) {
    v.erase(begin(v));
  }
  cout << "Empty!" << endl;
  return 0;
}
```

Сработало моментально. И даже pop_front тоже сработает сразу. Дек умеет больше, но, как следствие, он менее эффективен. Если нужно работать с двумя концами, используйте дек. Но если хватает вектора, используйте вектор.
Разберём ещё одну структуру: **очередь** (queue)

- если нужна только очередь, используйте queue;
- основана на деке, но работает немного быстрее;
- #include <queue>;
- Умеет совсем немного:

```
q.push(x), q.pop(x) //вставляем в начало и удаляем из конца
q.front(), q.back() //ссылки на первый и последний элементы очереди
q.size(), q.empty() //размер и проверка на пустоту
```

Кроме того существует **стек** (stack)

- позволяет лишь добавлять в конец и удалять из конца;
- #include <stack>;
- Как вектор, но умеет меньше

```
st.push(x), st.pop(x) //вставляем в конец и удаляем из конца
st.front(), st.back() //ссылки на первый элемент
st.size(), st.empty() //размер и проверка на пустоту
```

4.3.2 Алгоритмы поиска

Рассмотрим специальный класс методов контейнеров и алгоритмов. Алгоритмы поиска. Мы с ними уже сталкивались:

- Поиск по вектору и множеству
- Подсчёт количества:

```
count(begin(v), end(v), x)
s.count(x)
```

- поиск

```
find(begin(v), end(v), x)
s.find(x)
```

Рассмотрим задачу поиска элементов в контейнерах:

1. Где будем искать?
   - Не отсортированный вектор (или строка);
   - Отсортированный вектор;
   - Множество (или словарь).
2. Что будем искать и проверять?
   - Проверить существование;
   - Проверить существование и найти первое вхождение;
   - Найти первый элемент, больший или равный данному;
   - Найти первый элемент, больший данного;

- Подсчитать количество;
- Перебрать все.

Как осуществляется поиск в неотсортированном векторе?

- поиск конкретного элемента:

```
find(begin(v), end(v), x)
```

- элемент по какому-то условию (больше, меньше, больше или равен)

```
find_if(begin(v), end(v), [](int y) {...})
```

- посчитать количество

```
count(begin(v), end(v), x)
```

- перебор всего с помощью цикла и find.

Например, выведем позиции всех пробелов в строке:

```
for (auto it = find(begin(s), end(s), ' ');
    it != end(s);
    it = find(next(it), end(s), ' ')){ //переходим в цикле к следующему пробелу
  cout << it - begin(s) << ""; //next(it) эквивалентен it+1
}
```

В отсортированном векторе поиск можно осуществить быстрее с помощью бинарного поиска. Количество операций равно $\log_2(N)$ - двоичный логарифм числа элементов. Столько же работает поиск во множестве и словаре.
Отсюда следствие: если вы просто хотите быстро искать по набору элементов, но не хотите добавлять новые или удалять какие-то, вам достаточно отсортированного вектора. Это будет оптимальнее, чем, если вы используете множество. В отсортированном векторе можно искать так:

- проверка на существование

```
binary_search(begin(v), end(v), x)
```

- первый больший или равный данному

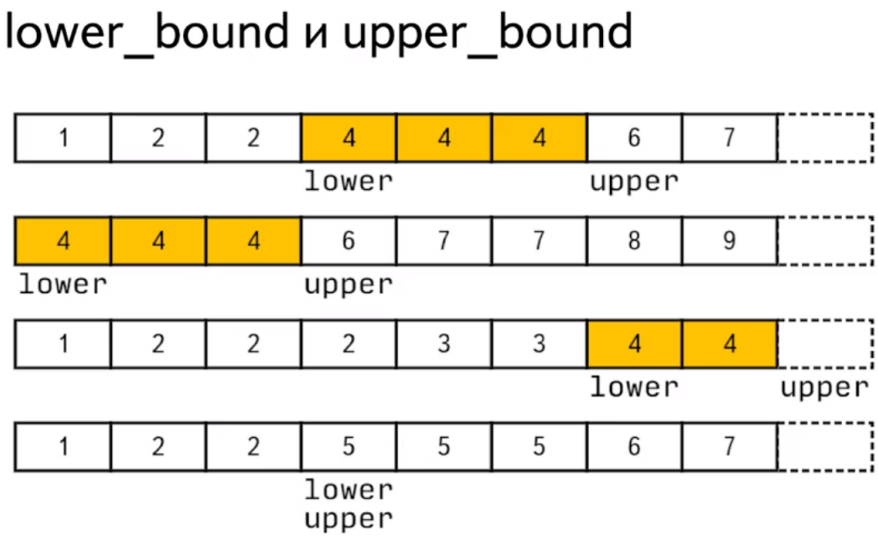
```
lower_bound(begin(v), end(v), x)
```

- первый элемент, больший данного

```
upper_bound(begin(v), end(v), x)
```

- диапазон элементов, равных данному (аналог minmax)

```
equal_range(begin(v), end(v), x) ==
make_pair(lower_bound(...), upper_bound(...))
```



Еще про equal_range.

- Если элемент есть, то $equal\_range = [lower\_bound, upper\_bound)$ - диапазон всех вхождений;
- Если же элемента нет, то $lower\_bound == upper\_bound$ - позиция, куда можно вставить элемент без нарушения порядка сортировки;
- Количество вхождений $== upper\_bound - lower\_bound$;
- А перебрать все элементы, равные данному, можно просто проитерировавшись от lower_bound до upper_bound.

Поиск во множестве мы уже знаем:

- s.count(x)
- s.find(x)
- s.lower_bound(x)
- s.upper_bound(x)
- s.equal_range(x)

4.3.3 Анализ распространённых ошибок

Первый пример распространённых ошибок: вычитание итераторов множества

```
int main() {
  set<int> s = {1, 2, 7};
  end(s) - begin(s);
  return 0;
}
//no match for 'operator-'
```

это ошибка простая, а вот если мы возьмём алгоритм, принимающий Random (например, partial_sort) итераторы и передадим ему итераторы множества

```
int main() {
  set<int> s = {1, 2, 7};
  partial_sort(begin(s), end(s), end(s))
  return 0;
}
//no match for 'operator-', 'operator+', 'operator<'
```

Все по той же причине. Итератор множества не Random итератор, по нему нельзя сравнивать или перемещать элементы.
Теперь попробуем вызвать remove

```
int main() {
  set<int> s = {1, 2, 7};
  remove(begin(s), end(s), 0);
  return 0;
}
//assignment of read-only location ...
```

Т.е. присваивание в итератор, содержимое которого нельзя менять. Ссылка под итераторами константная. Теперь одна из самых страшных ошибок (не ловится на этапе компиляции): Передача диапазона, у которого итераторы от разных контейнеров

```
int main() {
  vector<int> s1 = {1, 2, 7};
  vector<int> s2 = {2, 7};
  sort(begin(s1), end(s2)); //диапазон от начала одного вектора, до конца другого
  return 0;
}
```

Запускаем код и программа упала. В обратном порядке она, скорее всего, зациклится.
Проверить это можно закоментировав код, и если он заработает, проблема в нём.
Если же мы передадим итераторы разных типов, то :

```
int main() {
  vector<int> s1 = {1, 2, 7};
  vector<int> s2 = {2, 7};
  sort(begin(s1), rend(s2));
  return 0;
}
//deduced conflicting types for parameter random access iterator
```

Итераторы должны иметь один тип. А у нас в данном случае тип разный и не получается вызвать функцию sort.