КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 21 — Алгоритмы

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2021

Оглавление

Алгоритмы

```
#include <algorithm>
```

Заголовок **<algorithm>** определяет набор функций, специально предназначенных для использования в диапазонах элементов.

Диапазон — это любая последовательность объектов, к которой можно получить доступ через итераторы или указатели, например массив или экземпляр некоторых контейнеров STL.

Однако следует обратить внимание, что алгоритмы работают через итераторы непосредственно над значениями, никоим образом не влияя на структуру любого возможного контейнера. Они никогда не влияют ни на размер, ни на распределение памяти контейнера.

Шаблоны функций в <algorithm>

Немодифицирующие последовательные операции

`all_of()` — проверить, удовлетворяет ли условию все элементы в диапазоне
`any_of()` — проверить, удовлетворяет ли условию какой-либо элемент в диапазоне
`none_of()` — проверить, все ли элементы в диапазоне не удовлетворяют условию
`for_each()` — применить функцию к диапазону
`find()` — найти значение в диапазоне
`find_if()` — найти элемент в диапазоне, удовлетворяющий предикату
`find_if_not()` — найти элемент в диапазоне, не удовлетворяющий предикату
`find_end()` — найти последнюю подпоследовательность в диапазоне
`find_first_of()` — найти элемент из набора в диапазоне
`adjacent_find()` — найти равные соседние элементы в диапазоне
`count()` — подсчитать появление значения в диапазоне
`count_if()` — возвратить количество элементов в диапазоне, удовлетворяющих условию
`mismatch()` — возвратить первую позицию, где различаются два диапазона
`equal()` — проверить, равны ли элементы в двух диапазонах
`is_permutation()` — Проверить, является ли диапазон перестановкой другого
`search()` — искать диапазон появления подпоследовательности
`search_n()` — искать диапазон появления нескольких элементов

Модифицирующие последовательные операции

`copy()` — копировать диапазон элементов
`copy_n()` — копировать несколько элементов
`copy_if()` — копировать определенные элементы диапазона
`copy_backward()` — копировать диапазон элементов в обратном порядке
`move()` — переместить диапазон элементов
`move_backward()` — переместить диапазон элементов в обратном порядке
`swap()` — обменять значения двух элементов
`swap_ranges()` — обменять значения двух диапазонов
`iter_swap()` — обменять значения двух элементов, на котороые указывают итераторы
`transform()` — преобразовать диапазон
`replace()` — заменить значения в диапазоне
`replace_if()` — заменить в диапазоне значения, удовлетворяющие предикату
`replace_copy()` — копировать диапазон значений, заменяя элементы
`replace_copy_if()` — копировать диапазон значений, заменяя удовлетворяющие предикату элементы
`fill()` — заполнить диапазон значением
`fill_n()` — заполнить несколько значений, начиная с указанного
`generate()` — генерировать с помощью указанной функции значения для диапазона
`generate_n()` — генерировать с помощью указанной функции несколько значений
`remove()` — удалить значение из диапазона, заменяя его на неравное следующее
`remove_if()` — удалить удовлетворяющее предикату значение из диапазона, заменяя его ...
`remove_copy()` — копировать диапазон, пропуская значение

remove_copy_if() — копировать из диапазона, пропуская удовлетворяющие предикату
unique() — удалить дубликаты в диапазоне
unique_copy() — копировать диапазон, удаляя дубликаты
reverse() — обратить порядок следования элементов в диапазоне
reverse_copy() — копировать диапазон в обратном порядке
rotate() — циклически сдвинуть элементы в диапазоне
rotate_copy() — копировать, сдвигая циклически элементы в диапазоне
random_shuffle() — переставить элементы в диапазоне в случайном порядке
shuffle() — переставить элементы в диапазоне в случайном порядке, используя генератор

Секционирование
is_partitioned() — проверить, разделен ли диапазон в соответствии с предикатом
partition() — реорганизовать элементы в диапазоне в соответствии с предикатом
stable_partition() — реорганизовать элементы в диапазоне сохраняя порядок
partition_copy() — копировать диапазон, реорганизуя в в соответствии с предикатом
partition_point() — получить точку разделения в соответствии с предикатом

Сортировка

sort() — сортировать элементы в диапазоне
stable_sort() — сортировать элементы в диапазоне, сохраняя порядоку эквивалентных
partial_sort() — сортировать элементы в части диапазона
partial_sort_copy() — копировать часть элементов из диапазона и отсортировать

`is_sorted()` — проверить, отсортированы ли элементы в диапазоне
`is_sorted_until()` — найти первый элемент диапазоне, который не в порядке сортировки
`nth_element()` — сортировать элементы в диапазоне, сохраняя позицию указанного

Двоичный поиск (в секционированных/отсортированных диапазонах)

`lower_bound()` — найти первый элемент в диапазоне, который не меньше указанного
`upper_bound()` — найти первый элемент в диапазоне, который больше указанного
`equal_range()` — получить поддиапазон эквивалентных элементов
`binary_search()` — проверить, существует ли элемент в отсортированной последовательности

Слияние (в отсортированных диапазонах)

`merge()` — объединить отсортированные диапазоны
`inplace_merge()` — объединить два последовательных отсортированных диапазона
`includes()` — проверить, включает ли отсортированный диапазон другой отсортированный диапазон
`set_union()` — объединить два отсортированных диапазона без дублирования
`set_intersection()` — сформировать пересечение двух отсортированных диапазонов
`set_difference()` — сформировать разность (дополнение к пересечению)
`set_symmetric_difference()` — сформировать разность (дополнение к пересечению)

Куча

`push_heap()` — вставить элемент в диапазон кучи
`pop_heap()` — извлечь элемент из диапазона кучи
`make_heap()` — сделать кчу из диапазона
`sort_heap()` — сортировать элементы в куче
`is_heap()` — проверить, является ли диапазон кучей
`is_heap_until()` — найти первый элемент, который не в порядке кучи.

Min/max

`min()` — вернуть наименьший элемент
`max()` — вернуть наибольший элемент
`minmax()` — вернуть наименьший и наибольший элементы
`min_element()` — вернуть наименьший элемент из диапазона
`max_element()` — вернуть наибольший элемент из диапазона
`minmax_element()` — вернуть наименьший и наибольший элементы из диапазона

Прочие

`lexicographical_compare()` — лексикографическое сравнение «меньше-чем»
`next_permutation()` — преобразовать диапазон в следующую перестановку
`prev_permutation()` — преобразовать диапазон в предыдущую перестановку

all_of(), any_of(), none_of() — все, какой-либо или никакой не удовлетворяет

all_of() — проверить, удовлетворяет ли условию все элементы в диапазоне
any_of() — проверить, удовлетворяет ли условию какой-либо элемент в диапазоне
none_of() — проверить, все ли элементы в диапазоне не удовлетворяют условию

```
template <class InputIterator, class UnaryPredicate>
bool quantor(InputIterator first, InputIterator last, UnaryPredicate pred);
```

quantor — заместитель одного из **all_of**, **any_of()**, **none_of()**

pred — унарная функция, которая принимает элемент из диапазона в качестве аргумента и возвращает значение, конвертируемое в **bool**.

Возвращаемое значение указывает, соответствует ли элемент условию, проверенному этой функцией.

Функция не должна изменять свой аргумент.

pred может быть указателем на функцию или функциональным объектом.

Пример none_of()

```
#include <iostream>     // std::cout
#include <algorithm>    // std::none_of
#include <array>        // std::array

int main () {

  std::array<int, 8> foo = {1, 2, 4, 8, 16, 32, 64, 128};

  if (std::none_of(foo.begin(), foo.end(), [](int i){return i < 0;}))
    std::cout << "There are no negative elements in the range.\n";
  return 0;
}
```

Пример all_of()

```
#include <iostream>     // std::cout
#include <algorithm>    // std::all_of
#include <array>        // std::array

int main () {

  std::array<int, 8> foo = {3, 5, 7, 11, 13, 17, 19, 23};

  if ( std::all_of(foo.begin(), foo.end(), [](int i){return i%2;}) )
    std::cout << "All the elements are odd numbers.\n";
}
```

Функциональные объекты (функторы)

Функтор — это в значительной степени просто класс, определяющий оператор **( )**.

Это позволяет создавать объекты, которые «выглядят» как функция:

```
// Это функтор
struct add_x {
  add_x(int val) : x(val) {}  // Constructor
  int operator()(int y) const { return x + y; }
private:
  int x;
};
```

Как это можно использовать?

```
add_x add42(42);  // создать экземпляр класса функтора
int i = add42(8); // and "call" it
assert(i == 50);  // and it added 42 to its argument

std::vector<int> in; // положим, что содержит сколько-там значений)
std::vector<int> out(in.size());
// Передаем функтор std::transform(), которая будет его вызывать для ∀ элемента
// из входной последовательности и сохранять результат в выходной
std::transform(in.begin(), in.end(), out.begin(), add_x(1));
assert(out[i] == in[i] + 1); // проверим для всех i
```

В функторах есть несколько приятных моментов.

В отличие от обычных функций, они могут содержать состояние.

В приведенном выше примере создается функция, которая добавляет 42 к тому значению, которое ей передается. Но это значение 42 не жестко запрограммировано — оно было указано в качестве аргумента конструктора при создании экземпляра этого функтора.

Можно создать сколько угодно таких сумматоров, которые добавляли бы любое другое значение, простым вызовом конструктора с этим значением. Данный момент делает функторы легко настраиваемыми и потенциально более эффективными, нежели указатели на функции.

Все то же самое можно сделать с обычным указателем на функцию, за исключением того, что функторы могут быть «настроены» — если бы использовался указатель на функцию, пришлось бы написать функцию которая добавляла бы к своему аргументу ровно 1, в то время как функтор является более общим и добавляет все, чем его проинициализировали.

Функторы часто передаются в качестве аргументов другим функциям, таким как **`std::transform()`** или другим стандартным библиотечным алгоритмам.

В приведенном выше примере компилятор точно знает, какую функцию в **`std::transform()`** следует вызвать — он должен вызывать **`add_x::operator()`**. Это означает, что можно встроить этот вызов функции, что делает его столь же эффективным, как если функция вызвалась вручную для каждого значения вектора.

Если бы вместо этого был передан указатель на функцию, компилятор не смог бы сразу увидеть, на какую функцию он указывает, поэтому ему пришлось бы во время выполнения всякий раз разыменовать указатель, чтобы выполнить вызов.

Пример

```
class MyFunctor {
public:
    int operator()(int x) { return x*2;}
}

MyFunctor doubler;
int x = doubler(5);
```

Пример

```
class Matcher {
   int target;
public:
   Matcher(int m) : target(m) {}
   bool operator()(int x) { return x == target;}
}

Matcher is_five(5);

if (is_five(n)) { // то же самое, что и if (n == 5)
   ....
}
```

Лямбда-выражения в C++

C++ включает полезные общие функции, такие как **std::for_each()** и **std::transform()**, что может быть очень удобно. К сожалению, они также могут быть довольно громоздкими в использовании, особенно если функтор, который нужно применить, уникален для конкретной функции.

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

namespace {
  struct f {
    void operator()(int) {
      // do something
    }
  };
}

void func(std::vector<int>& v) {
  f f;
  std::for_each(v.begin(), v.end(), f);
}
```

Если **f** используется только один раз и только в этом конкретном месте, кажется излишним писать целый класс только для того, чтобы сделать что-то тривиальное и одноразовое.

В C ++ 03 можно написать что-то вроде следующего, чтобы функтор оставался локальным:

```
void func2(std::vector<int>& v) {
  struct {
    void operator()(int) {
       // делаем тут что-нибудь
    }
  } f;
  std::for_each(v.begin(), v.end(), f);
}
```

В C++11 появились лямбда-выражения, позволяющие написать встроенный анонимный функтор для замены **`struct f`**.

Использование лямбда-выражения может быть проще для чтения (оно хранит все в одном месте) и потенциально проще в обслуживании, например, в простейшей форме:

```
void func3(std::vector<int>& v) {
  std::for_each(v.begin(), v.end(), [](int) { /* делаем тут что-нибудь */ });
}
```

Лямбда-функции — это просто синтаксический сахар для анонимных функторов.

Типы возврата

В простых случаях тип возвращаемого значения лямбда-функции выводится автоматически, например:

```
void func4(std::vector<double>& v) {
  std::transform(v.begin(),
                 v.end(),
                 v.begin(),
                 [](double d) { return d < 0.00001 ? 0 : d; });
}
```

Однако, в случае более сложных лямбда-выражений, можно столкнуться со случаями, когда тип возвращаемого значения компилятором определен быть не может, например:

```
void func4(std::vector<double>& v) {

    std::transform(v.begin(), v.end(), v.begin(),
        [](double d) {
            if (d < 0.0001) {
                return 0;
            } else {
                return d;
            }
        });
}
```

Чтобы решить эту проблему, разрешено явно указать тип возвращаемого значения для лямбда-функции, используя **-> T**:

```
void func4(std::vector<double>& v) {

    std::transform(v.begin(), v.end(), v.begin(),
//      [](double d) {
        [](double d) -> double {
            if (d < 0.0001) {
                return 0;
            } else {
                return d;
            }
        });
}
```

«Захват» переменных

В вышеприведенных примерах не использовалось ничего, кроме того, что было передано внутрь лямбда-выражения. Однако, внутри лямбды можно также использовать и другие переменные. Чтобы получить доступ к другим переменным, можно использовать список захвата (выражения в **[]**), например:

```
void func5(std::vector<double>& v, const double& epsilon) {

    std::transform(v.begin(), v.end(), v.begin(),
        [epsilon](double d) -> double {
            if (d < epsilon) {
                return 0;
            } else {
                return d;
            }
        });
}
```

Сгенерированный **operator()** по умолчанию является константным, что подразумевает, что захваты, при доступе к ним по умолчанию будут, константными.

Это приводит к тому, что каждый вызов с одним и тем же входом будет давать один и тот же результат, однако можно пометить лямбду, как изменяемую, чтобы запросить, чтобы сгенерированный **operator()** не был константным.

Более формально

В C++ лямбда-функция определяется следующим образом

```
[] () {} // базовая лямбда
```

или во всей красе

```
[] () mutable -> T {} // T - это возвращаемый тип, но без throw()
```

[] — это список захвата, **()** — список аргументов и **{}** — тело функции.

Список захвата

Можно захватывать как ссылку, так и значение, которые можно указать с помощью **&** и **=**, соответственно, а также смешивать тип захвата для переменных, указывая их в виде списка, разделенного запятыми **[x, &y, ...]**:

[&epsilon]	захват по ссылке;
[&]	захват всех переменных, которые используются в лямбде, по ссылке;
[=]	захват всех переменных, которые используются в лямбде, по значению;
[&, epsilon]	захват переменных, как в **[&]**, за исключением **epsilon**, которое по значению;
[=, &epsilon]	захват переменных, как в **[=]**, за исключением **epsilon**, которое по ссылке.

Список аргументов

Список аргументов такой же, как и в любой другой функции C++.

Тело функции

Код, который будет выполняться при фактическом вызове лямбды.

Вывод типа возврата

Если лямбда-выражение имеет только один оператор возврата, тип возврата можно опустить и он имеет неявный тип **`decltype(return_statement)`**.

Изменяемая

Если лямбда помечена как изменяемая (например, **`[] () mutable {}`**), разрешается изменять те значения, которые были захвачены по значению.

Случаи применения

Библиотека, определенная стандартом ISO, сильно выигрывает от лямбда-выражений и повышает удобство использования, поскольку теперь пользователям не нужно загромождать свой код небольшими функторами.

Пример использования

```
int x = 4;
auto y = [&r = x, x = x + 1]()->int {      // r == 4, x == 5
            r += 2;                        // r == 6
            return x + 2;                  // возвращает 7
         }();  // Значение ::x становится 6, а y инициализируется в 7.
```

Обобщенные лямбды

Лямбда-выражения теперь могут быть универсальными

```
auto lambda = [](auto x, auto y) {return x + y;};
```

Здесь **auto** было бы эквивалентно **T**, если бы **T** был аргументом шаблона типа где-то в окружающей области видимости.

Улучшенный вывод типа возврата

C++14 позволяет выводить типы возвращаемого значения для каждой функции и не ограничивает ее функциями с выходом в форме **return expression;**.

Это также распространяется и на лямбды.

Лямбда-выражения обычно используются для инкапсуляции алгоритмов, чтобы их можно было передать другой функции.

Однако можно выполнить лямбду сразу после определения:

```
[&](){ ... код... }(); // лямбда-выражение выполняется сразу
```

Это функционально эквивалентно

```
{... код ...} // простой блок кода
```

Такое поведение делает лямбда-выражения мощным инструментом для рефакторинга сложных функций.

Сначала секция кода заворачивается в лямбда-функцию, как показано выше.

Затем постепенно может выполняться процесс явной параметризации с промежуточным тестированием после каждого шага.

После того, как блок кода будет полностью параметризован (будет удален символ & в списке захвата), можно переместить код во внешнее расположение и сделать его нормальной функцией.

Так же можно использовать лямбда-выражения для инициализации переменных на основе результата выполнения алгоритма.

```
int a = [](int b) { int r = 1; while (b > 0) r *= b--; return r; }(5); // 5!
```

Лямбда-выражения также позволяют создавать именованные вложенные функции, что может быть удобным способом избежать дублирования логики.

Использование именованных лямбда-выражений также имеет тенденцию быть немного проще для глаз (по сравнению с анонимными встроенными лямбдами) при передаче нетривиальной функции в качестве параметра другой функции.

```
auto algorithm =
[&](double x, double m, double b) -> double {
   return m * x + b;
};

int a = algorithm(1, 2, 3);
int b = algorithm(4, 5, 6);
```

Если такое решение приводит к значительным накладным расходам на инициализацию функционального объекта, можно переписать все это как обычную функцию.

for_each() – применить функцию к диапазону

```
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function fn);
```

Применяет функцию **fn** к каждому из элементов в диапазоне **[first,last)**.

fn – унарная функция, которая принимает в качестве аргумента элемент из диапазона.

Она может быть указатель на функцию или функциональный объект, котороый может быть сконструирован перемещением.

Возвращаемое значение, если оно есть, игнорируется.

Пример `for_each()`

```
#include <iostream>     // std::cout
#include <algorithm>    // std::for_each
#include <vector>       // std::vector

void function(int i) {  // функция
  std::cout << ' ' << i;
}

struct myclass {            // тип функционального объекта
  void operator() (int i) {std::cout << ' ' << i;}
} functor;

int main () {

  std::vector<int> myvector = {10, 20, 30};

  std::cout << "myvector contains:";
  for_each(myvector.begin(), myvector.end(), function);
  std::cout << '\n';
  // or:
  std::cout << "myvector contains:";
  for_each(myvector.begin(), myvector.end(), functor);
  std::cout << '\n';

  return 0;
}
```

transform() – преобразовать диапазон

(1) Унарная операция

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first1, InputIterator last1,
                         OutputIterator result, UnaryOperation op);
```

(2) Бинарная операция

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator,
          class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2,
                         OutputIterator result,
                         BinaryOperation binary_op);
```

Диапазон преобразования

Операция применяется последовательно к элементам одного (1) или двух (2) диапазонов, а результат сохраняется в диапазоне, который начинается с **`result`**.

Возвращаемое значение

Итератор, указывающий на элемент, следующий за последним элементом, записанным в результирующей последовательности.

(1) Унарная операция

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first1, InputIterator last1,
                         OutputIterator result, UnaryOperation op);
```

Применяет **op** к каждому из элементов в диапазоне **[first1, last1)** и сохраняет значение, возвращаемое каждой операцией в диапазоне, который начинается с **result**.

op — унарная функция, которая принимает в качестве аргумента один элемент типа, на который указывает **InputIterator**, и возвращает некоторое значение результата, конвертируемое в тип, на который указывает **OutputIterator**.

op может быть указателем на функцию или функциональным объектом.

(2) Бинарная операция

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator,
          class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2,
                         OutputIterator result,
                         BinaryOperation binary_op);
```

Вызывает **binary_op**, используя каждый из элементов в диапазоне **[first1, last1)** в качестве первого аргумента и соответствующий аргумент в диапазоне, который начинается с **first2**, в качестве второго аргумента.

binary_op — бинарная функция, которая принимает два элемента в качестве аргумента (по одному из каждой из двух последовательностей) и возвращает некоторое значение результата, конвертируемое в тип, на который указывает **OutputIterator**.

binary_op может быть указателем на функцию или функциональным объектом.

Значение, возвращаемое каждым вызовом **binary_op**, сохраняется в диапазоне, который начинается с **result**.

Функция позволяет целевому диапазону быть таким же, как один из входных диапазонов для выполнения преобразований на месте.

Ни **op**, ни **binary_op** не должны напрямую изменять элементы, переданные в качестве аргументов, — они косвенно изменяются алгоритмом с использованием возвращаемого значения, если для **result** указан тот же диапазон.

Пример алгоритма `transform`

```
#include <iostream>     // std::cout
#include <algorithm>    // std::transform
#include <vector>       // std::vector
#include <functional>   // std::plus

int op_increase(int i) {return ++i;}

int main () {

  std::vector<int> foo = {10, 20, 30, 40, 50};
  std::vector<int> bar;
  bar.resize(foo.size());                          // allocate space

  std::transform(foo.begin(), foo.end(), bar.begin(), op_increase);
                                                   // bar: 11 21 31 41 51
  // std::plus adds together its two arguments:
  std::transform(foo.begin(), foo.end(), bar.begin(), foo.begin(),
                 std::plus<int>());                // foo: 21 41 61 81 101

  std::cout << "foo contains:";
  for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
}
```

std::plus – класс функтора сложения

```
template <class T> struct plus;
```

Класс бинарного функционального объекта (функтора), вызов которого возвращает результат сложения двух своих аргументов (типа, как возвращет оператор **+**).

Как правило, функциональные объекты являются экземплярами класса с определенной функцией-членом **operator ()**. Эта функция-член позволяет использовать объект с тем же синтаксисом, что и вызов функции. Он определяется с таким же поведением, как:

C++98

```
template <class T> struct plus : binary_function <T, T, T> {
  T operator() (const T& x, const T& y) const {return x + y;}
};
```

C++11

```
template <class T> struct plus {
  T operator() (const T& x, const T& y) const {return x + y;}
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef T result_type;
};
```

std::sort – сортировка элементов в диапазоне

(1) По умолчанию

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

(2) Кастомизированная

```
template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Сортирует элементы в диапазоне **[first,last)** в порядке возрастания.

Элементы сравниваются с помощью **operator<** для первой версии и **comp** для второй.

Не гарантируется, что эквивалентные элементы сохранят свой первоначальный относительный порядок (Для этого есть **stable_sort( )**).

comp — бинарная функция, которая принимает два элемента из диапазона в качестве аргументов и возвращает значение, конвертируемое в **bool**.

Возвращаемое значение указывает, считается ли элемент, переданный в качестве первого аргумента, предшествующим второму в конкретном строгом слабом порядке, который он определяет. Функция не должна изменять ни один из своих аргументов.

comp может быть указателем на функцию или функциональным объектом.

Пример sort()

```
#include <iostream>     // std::cout
#include <algorithm>    // std::sort
#include <vector>       // std::vector

bool func(int i, int j) {return (i < j);}

struct {
  bool operator() (int i, int j) {return (i < j);}
} functor;

int main () {

  int myints[] = {32, 71, 12, 45, 26, 80, 53, 33};
  std::vector<int> vect(myints, myints + 8);     // 32 71 12 45 26 80 53 33
  // сравнение по умлчанию (operator<):
  std::sort(vect.begin(), vect.begin() + 4);     // 12 32 45 71 26 80 53 33
  // функция в качестве comp
  std::sort(vect.begin() + 4, vect.end(), func); // 12 32 45 71 26 33 53 80
  // функтор в качестве comp
  std::sort(vect.begin(), vect.end(), functor);  // 12 26 32 33 45 53 71 80

  std::cout << "myvector contains:";
  for(auto it = vect.begin(); it != vect.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
}
```

std::stable_sort – сортировка с сохранением порядка эквивалентных элементов

```
template <class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Сортирует элементы в диапазоне **[first,last)** в порядке возрастания, как и **sort**, но **stable_sort** сохраняет относительный порядок элементов с эквивалентными значениями.

Элементы сравниваются с помощью **operator<** для первой версии и **comp** для второй.

Пример сортировки stable_sort()

```
#include <iostream>     // std::cout
#include <algorithm>    // std::stable_sort
#include <vector>       // std::vector

bool compare_as_ints(double i, double j) {
  return (int(i) < int(j));
}

int main () {

  double array[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
  std::vector<double> myvector;
  myvector.assign(array, array + 8);
  std::cout << "сравнение по умолчанию: ";
  std::stable_sort(myvector.begin(), myvector.end());
  for(auto it = myvector.begin(); it != myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  myvector.assign(array, array + 8);
  std::cout << "сравнение 'compare_as_ints' :";
  std::stable_sort(myvector.begin(), myvector.end(), compare_as_ints);
  for (auto it = myvector.begin(); it != myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
}
```

```
#include <iostream>     // std::cout
#include <algorithm>    // std::stable_sort
#include <vector>       // std::vector

int main () {

  double array[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
  std::vector<double> myvector;
  myvector.assign(array, array + 8);
  std::cout << "Сравнение по умолчанию: ";
  std::stable_sort(myvector.begin(), myvector.end());
  for(auto it = myvector.begin(); it != myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  myvector.assign(array, array + 8);
  std::cout << "Сравнение как 'ints' :";
  std::stable_sort(myvector.begin(), myvector.end(),
                   [](double x, double y) {return (int(x) < int(y));});

  for (auto it = myvector.begin(); it != myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
}
```

Сравнение по умолчанию: 1.32 1.41 1.62 1.73 2.58 2.72 3.14 4.67

Сравнение как 'ints' : 1.41 1.73 1.32 1.62 2.72 2.58 3.14 4.67