

Ravesli [Ravesli](#)


- [Уроки по C++](#)
- [OpenGL](#)
- [SFML](#)
- [Qt5](#)
- [RegExр](#)
- [Ассемблер](#)
- [Купить .PDF](#)


Алгоритмы в Стандартной библиотеке C++

 [Дмитрий Бушуев](#) |

- [Уроки C++](#)

|

 Обновл. 20 Окт 2020 |

 15044

[12](#)

На этом уроке мы рассмотрим использование алгоритмов из Стандартной библиотеки C++.

Оглавление:

1. [Библиотека алгоритмов](#)
2. [Алгоритм `std::find\(\)` и поиск элемента по значению](#)
3. [Алгоритм `std::find_if\(\)` и поиск элемента с условием](#)
4. [Алгоритмы `std::count\(\)/std::count_if\(\)` и подсчет вхождений элемента](#)
5. [Алгоритм `std::sort\(\)` и пользовательская сортировка](#)
6. [Алгоритм `std::for_each\(\)` и все элементы контейнера](#)
7. [Порядок выполнения](#)
8. [Диапазоны в C++20](#)
9. [Заключение](#)

Библиотека алгоритмов

Новички обычно тратят довольно много времени на написание пользовательских циклов для выполнения относительно простых задач, таких как: сортировка, поиск или подсчет элементов [массивов](#). Эти циклы могут стать проблематичными как с точки зрения того, насколько легко в них можно сделать ошибку, так и с точки зрения общей надежности и удобства использования, т.к. данные циклы могут быть трудны для понимания.

Поскольку поиск, подсчет и сортировка являются очень распространенными операциями в программировании, то в состав Стандартной библиотеки C++ изначально уже включен большой набор функций, которые выполняют данные задачи всего в несколько строчек кода. В дополнение к этому, эти

функции уже предварительно протестированные, эффективные и имеют поддержку множества различных типов контейнеров. А некоторые из этих функций поддерживают и **распараллеливание** — возможность выделять несколько потоков ЦП для одной и той же задачи, чтобы выполнить её быстрее.

Функционал, предоставляемый библиотекой алгоритмов, обычно относится к одной из 3-х категорий:

- **Инспекторы** — используются для просмотра (без изменений) данных в контейнере (например, операции поиска или подсчета элементов).
- **Мутаторы** — используются для изменения данных в контейнере (например, операции сортировки или перестановки элементов).
- **Фасилитаторы** — используются для генерации результата на основе значений элементов данных (например, объекты, которые умножают значения, либо объекты, которые определяют, в каком порядке пары элементов должны быть отсортированы).

Данные алгоритмы расположены в библиотеке алгоритмов ([заголовочный файл](#) `algorithm`). На этом уроке мы рассмотрим некоторые из наиболее распространенных алгоритмов.

Примечание: Все эти алгоритмы используют [итераторы](#).

Алгоритм `std::find()` и поиск элемента по значению

Функция `std::find()` выполняет поиск первого вхождения заданного значения в контейнере. В качестве аргументов `std::find()` принимает 3 параметра:

- итератор для начального элемента в последовательности;
- итератор для конечного элемента в последовательности;
- значение для поиска.

В результате будет возвращен итератор, указывающий на элемент с искомым значением (если он найден) или конец контейнера (если такой элемент не найден). Например:

```
1 #include <algorithm>
2 #include <array>
3 #include <iostream>
4
5 int main()
6 {
7     std::array<int, 6> arr{ 13, 90, 99, 5, 40, 80 };
8
9     std::cout << "Enter a value to search for and replace with: ";
10    int search{};
11    int replace{};
12    std::cin >> search >> replace;
13
```

```
14 // Проверка пользовательского ввода должна быть здесь
15
16 // std::find() возвращает итератор, указывающий на найденный элемент (или на конец к
17 // Мы сохраним его в переменной, используя автоматический вывод типа итератора
18 auto found{ std::find(arr.begin(), arr.end(), search) };
19
20 // Алгоритмы, которые не нашли то, что искали, возвращают итератор, указывающий на к
21 // Мы можем получить доступ к этому итератору, используя метод end()
22 if (found == arr.end())
23 {
24     std::cout << "Could not find " << search << '\n';
25 }
26 else
27 {
28     // Перезаписываем найденный элемент
29     *found = replace;
30 }
31
32 for (int i : arr)
33 {
34     std::cout << i << ' ';
35 }
36
37 std::cout << '\n';
38
39 return 0;
40 }
```

Примечание: Для корректной работы всех примеров данного урока ваш компилятор должен поддерживать стандарт C++17. Детально о том, как использовать функционал C++17 вашей IDE, читайте [здесь](#).

Пример, в котором элемент найден:

```
Enter a value to search for and replace with: 5 234
13 90 99 234 40 80
```

Пример, в котором элемент не найден:

```
Enter a value to search for and replace with: 0 234
Could not find 0
13 90 99 5 40 80
```

Алгоритм `std::find_if()` и поиск элемента с условием

Иногда мы хотим увидеть, есть ли в контейнере значение, которое соответствует некоторому условию (например, строка, содержащая заданную подстроку).

В таких случаях функция `std::find_if()` будет идеальным помощником. Она работает аналогично функции `std::find()`, но вместо того, чтобы передавать значение для поиска, мы передаем вызываемый объект, например, указатель на функцию (или лямбду — об этом чуть позже), который проверяет, найдено ли совпадение. Функция `std::find_if()` будет вызывать этот объект для каждого элемента, пока не найдет искомый элемент (или в контейнере больше не останется элементов для проверки).

Вот пример, где мы используем функцию `std::find_if()`, чтобы проверить, содержат ли какие-либо элементы подстроку "nut":

```
1  #include <algorithm>
2  #include <array>
3  #include <iostream>
4  #include <string_view>
5
6  // Наша функция возвратит true, если элемент найден
7  bool containsNut(std::string_view str)
8  {
9      // std::string_view::find возвращает std::string_view::npos, если он не нашел подстроку
10     // В противном случае, он возвращает индекс, где происходит вхождение подстроки в строку
11     return (str.find("nut") != std::string_view::npos);
12 }
13
14 int main()
15 {
16     std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };
17
18     // Сканируем наш массив, чтобы посмотреть, содержат ли какие-либо элементы подстроку
19     auto found{ std::find_if(arr.begin(), arr.end(), containsNut) };
20
21     if (found == arr.end())
22     {
23         std::cout << "No nuts\n";
24     }
25     else
26     {
27         std::cout << "Found " << *found << '\n';
28     }
29
30     return 0;
31 }
```

Результат выполнения программы:

Found walnut

Если бы мы решали задачу, приведенную выше, обычным стандартным способом, то нам бы понадобилось, по крайней мере, два цикла (один для циклического перебора массива и один для

сравнения подстроки). Функции Стандартной библиотеки C++ позволяют сделать то же самое всего в несколько строчек кода!

Алгоритмы `std::count()`/`std::count_if()` и подсчет вхождений элемента

Функции `std::count()` и `std::count_if()` ищут все вхождения элемента или элемент, соответствующий заданным критериям.

В следующем примере мы посчитаем, сколько элементов содержит подстроку "nut":

```
1  #include <algorithm>
2  #include <array>
3  #include <iostream>
4  #include <string_view>
5
6  bool containsNut(std::string_view str)
7  {
8      return (str.find("nut") != std::string_view::npos);
9  }
10
11 int main()
12 {
13     std::array<std::string_view, 5> arr{ "apple", "banana", "walnut", "lemon", "peanut" };
14
15     auto nuts{ std::count_if(arr.begin(), arr.end(), containsNut) };
16
17     std::cout << "Counted " << nuts << " nut(s)\n";
18
19     return 0;
20 }
```

Результат выполнения программы:

Counted 2 nut(s)

Алгоритм `std::sort()` и пользовательская сортировка

Ранее мы использовали `std::sort()` для сортировки массива в порядке возрастания, но возможности `std::sort()` этим не ограничиваются. Есть версия `std::sort()`, которая принимает вспомогательную функцию в качестве третьего параметра, что позволяет выполнять сортировку так, как нам это захочется. Данная вспомогательная функция принимает два параметра для сравнения и возвращает `true`, если первый аргумент должен быть упорядочен перед вторым. По умолчанию, `std::sort()` сортирует элементы в порядке возрастания.

Давайте попробуем использовать `std::sort()` для сортировки массива в обратном порядке с помощью вспомогательной пользовательской функции для сравнения `greater()`:

```
1  #include <algorithm>
2  #include <array>
3  #include <iostream>
4
5  bool greater(int a, int b)
6  {
7      // Размещаем a перед b, если a больше, чем b
8      return (a > b);
9  }
10
11 int main()
12 {
13     std::array arr{ 13, 90, 99, 5, 40, 80 };
14
15     // Передаем greater в качестве аргумента в функцию std::sort()
16     std::sort(arr.begin(), arr.end(), greater);
17
18     for (int i : arr)
19     {
20         std::cout << i << ' ';
21     }
22
23     std::cout << '\n';
24
25     return 0;
26 }
```

Результат выполнения программы:

99 90 80 40 13 5

Опять же, вместо того, чтобы самостоятельно писать с нуля свои циклы/функции, мы можем отсортировать наш массив так, как нам нравится, с использованием всего нескольких строчек кода!

Совет: Поскольку сортировка в порядке убывания также очень распространена, то C++ предоставляет пользовательский тип `std::greater{}` для этой задачи (который находится в заголовочном файле `functional`). В примере, приведенном выше, мы можем заменить:

```
1 | std::sort(arr.begin(), arr.end(), greater); // вызов нашей функции greater
```

На:

```
1 | std::sort(arr.begin(), arr.end(), std::greater{}); // используем greater из Стандартной
```

Обратите внимание, что `std::greater{}` нуждается в фигурных скобках, потому что это не вызываемая функция, а тип данных, и для его использования нам нужно создать экземпляр данного типа. Фигурные

скобки создают [анонимный объект](#) данного типа (который затем передается в качестве аргумента в функцию `std::sort()`).

Алгоритм `std::for_each()` и все элементы контейнера

Функция `std::for_each()` принимает список в качестве входных данных и применяет пользовательскую функцию к каждому элементу этого списка. Это полезно, когда нам нужно выполнить одну и ту же операцию со всеми элементами списка.

Вот пример, где мы используем `std::for_each()` для удвоения всех чисел в массиве:

```
1  #include <algorithm>
2  #include <array>
3  #include <iostream>
4
5  void doubleNumber(int &i)
6  {
7      i *= 2;
8  }
9
10 int main()
11 {
12     std::array arr{ 1, 2, 3, 4 };
13
14     std::for_each(arr.begin(), arr.end(), doubleNumber);
15
16     for (int i : arr)
17     {
18         std::cout << i << ' ';
19     }
20
21     std::cout << '\n';
22
23     return 0;
24 }
```

Результат выполнения программы:

2 4 6 8

Новичкам данный способ может показаться ненужным алгоритмом, потому что эквивалентный код с использованием [цикла for](#) с явным указанием диапазона будет короче и проще. Но плюс `std::for_each()` состоит в том, что у нас есть возможность повторного использования тела цикла и применения распараллеливания при его обработке, что делает `std::for_each()` более подходящим инструментом для больших проектов с большим объемом данных.

Порядок выполнения

Обратите внимание, что большинство алгоритмов в библиотеке алгоритмов не гарантируют определенного порядка выполнения. Для использования таких алгоритмов вам нужно позаботиться о том, чтобы любые передаваемые функции не предполагали заданного порядка выполнения, так как порядок вызова этих функций может быть различным в зависимости от используемого компилятора.

Следующие алгоритмы гарантируют последовательное выполнение:

- `std::for_each()`
- `std::copy()`
- `std::copy_backward()`
- `std::move()`
- `std::move_backward()`

Совет: Если не указано иное, считайте, что для алгоритмов из Стандартной библиотеки C++ порядок выполнения является неопределенным. Алгоритмы, приведенные выше, дают гарантию последовательного выполнения.

Диапазоны в C++20

Необходимость для каждого алгоритма явно передавать `arr.begin()` и `arr.end()` может немного раздражать. Но в стандарте C++20 добавлен такой инструмент, как **диапазоны**, который позволит нам просто передавать `arr`. Благодаря этому мы сможем сделать наш код еще короче и читабельнее.

Заключение

Библиотека алгоритмов имеет массу полезных функций, которые могут сделать ваш код проще и надежнее. На этом уроке мы рассмотрели лишь небольшую часть алгоритмов, но, поскольку большинство из них работают схожим образом, как только вы разберетесь с некоторыми из них, вы сможете без больших трудностей использовать и оставшиеся функции.

Совет: Отдавайте предпочтение использованию функций из библиотеки алгоритмов, нежели самостоятельному написанию своего собственного функционала для выполнения данных задач.

Оценить статью:

★★★★★ (83 оценок, среднее: 4,88 из 5)



← [Введение в итераторы в C++](#)



Комментариев: 12

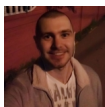


1. *Рустем:*

[4 ноября 2020 в 15:23](#)

Не пойму третий пункт этой главы, именно как решается эта задача с поиском "nut" `std::find_if()` и поиск элемента с условием.

[Ответить](#)



1. *Дмитрий Бушуев:*

[4 ноября 2020 в 16:59](#)

С какой конкретно строки начинается недопонимание? 😊

[Ответить](#)



1. *Рустем:*

[7 ноября 2020 в 15:51](#)

Не могу уловить связь в функции `main` со строки

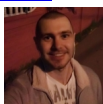
```
1 | auto found{ std::find_if(arr.begin(), arr.end(), containsNut) };
```

с функцией

```
1 | bool containsNut(std::string_view str)
2 | {
3 |     return (str.find("nut") != std::string_view::npos);
4 | }
```

И где эта строка `std::string_view::npos`???? для чего нужна?

[Ответить](#)



1. *Дмитрий Бушуев:*

[12 ноября 2020 в 06:50](#)

Начнём с последнего.

>>И где эта строка `std::string_view::npos`???? для чего нужна?

Если прочитать статью с самого начала, то во втором фрагменте кода есть вот такой комментарий: "std::string_view::find возвращает std::string_view::npos, если он не нашел подстроку. Или же возвращает индекс, где происходит вхождение подстроки в строку."

std::string_view::npos — это не строка, а константа, чем-то похожая на константу EOF (End of File) при работе с файлами. Когда она используется вместе со строками, то её значение воспринимается как "признак достижения конца строки". То есть, получается мы как бы говорим — "искать, пока не достигнем конца строки".

Например:

str.find("nut") != std::string_view::npos — выполнять поиск вхождения "nut" в заданной строке, пока не достигнем её конца. И тут возможны два варианта: мы либо найдем такую подстроку, и тогда str.find() вернет индекс, с которого начинается вхождение подстроки "nut"; либо str.find() вернет признак достижения конца строки (std::string_view::npos).

Далее, std::find_if(arr.begin(), arr.end(), containsNut). Ну с arr.begin() и arr.end() я думаю всё понятно, это обычные итераторы, указывающие на начало и конец строки, в рамках которой мы будем производить поиск. А containsNut — это "условие", по которому мы производим поиск. В данном случае в качестве условия мы передали функцию, которая (как написано выше), ищет вхождение подстроки "nut".



2. *KiberCyber:*

[16 июля 2020 в 23:32](#)

<Необходимость для каждого алгоритма явно передавать arr.begin() и arr.end() может немного раздражать. Но в стандарте C++20 добавлен такой инструмент, как диапазоны, который позволит нам просто передавать arr. Благодаря этому мы сможем сделать наш код ещё короче и читабельнее.> Так-то круто конечно, вот только как это сделать то? Ну, не сочтите прям совсем глупым, но погуглив я нашел лишь вот это: <https://habr.com/ru/company/otus/blog/456452/>, но мало что прояснилось(+/- ничего). Я пробовал не многое: только лишь заменить arr.begin() и arr.end() на просто arr, однако был вежливо послан компилятором со словами "candidate function template not viable: requires 3 arguments, but 2 were provided".

[Ответить](#)



1. *VEX:*

[23 июля 2020 в 13:10](#)

Если пользуетесь VS19, то почитайте вот это(главное меню — это правое окошко в VS19, где показаны все файлы вашего проекта)

[Ответить](#)



2. *Andrey:*

[7 октября 2020 в 08:12](#)

Просто вместо `std::find`, нужно написать `std::ranges::find`, и прописать массив без `begin()` и `end()`

```
1 | auto found{std::ranges::find(target_is, search)};
```

[Ответить](#)



3. Константин:

[1 июля 2020 в 22:40](#)

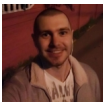
Может быть глупый вопрос, но почему `std::array` записывается как

```
1 | array arr {1, 2, 3}
```

А не `array<int, 3> arr {1, 2, 3}`?

Разве это будет работать?

[Ответить](#)



1. Дмитрий Бушув:

[7 июля 2020 в 00:25](#)

>>Может быть глупый вопрос, но почему `std::array` записывается как...

Начиная со C++17, в стандарт включен т.н. "Вывод Параметров Шаблонов Классов (Class Template Argument Deduction)".

Благодаря этому объявлять массив можно как:

```
1 | array arr {1, 2, 3}
```

>>Разве это будет работать?

Да, при условии, что ваш компилятор поддерживает стандарт C++17. Только что проверил это у себя (Qt 5.14.2 + MinGW 7.3.0), дописав в файл проекта параметр:

```
CONFIG += c++17 console
```

[Ответить](#)



1. Константин:

[7 июля 2020 в 17:21](#)

Спасибо большое

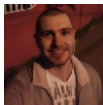
[Ответить](#)



4. Yerd:

[25 апреля 2020 в 16:25](#)

Почему все пользовательские функции, которые передаются в качестве параметров записываются в виде "func" нежели "func()"?

[Ответить](#)1. *Дмитрий Бушуев:*[26 апреля 2020 в 21:37](#)

Ответ на ваш вопрос находится здесь:

Урок №104. Указатели на функции (параграф — Передача функций в качестве аргументов другим функциям)

<https://ravesli.com/urok-104-ukazateli-na-funktsii/#toc-3>

[Ответить](#)

Добавить комментарий

Ваш E-mail не будет опубликован. Обязательные поля помечены *

Имя *

Email *

Комментарий






☐ Сохранить моё Имя и E-mail. Видеть комментарии, отправленные на модерацию

☐ Получать уведомления о новых комментариях по электронной почте. Вы можете [подписаться](#) без комментирования.

[TELEGRAM](#)  [КАНАЛ](#)

[ПАБЛИК](#) 

ТОП СТАТЬИ

-  [Словарь программиста. Сленг, который должен знать каждый кодер](#)
-  [Урок №1. Введение в программирование](#)
-  [70+ бесплатных ресурсов для изучения программирования](#)
-  [Урок №1: Введение в создание игры «SameGame» на C++/MFC](#)
-  [Урок №4. Установка IDE \(Интегрированной Среды Разработки\)](#)

- [Ravesli](#)
- - [О проекте/Контакты](#) -
- - [Пользовательское Соглашение](#) -
- - [Все статьи](#) -
- Copyright © 2015 - 2020