**Сортировка**

Сегодня мы будем изучать разные алгоритмы, которые есть в стандартной библиотеке C++. Они помогают быстрее писать программы. В основном мы будем использовать библиотеку algorithm.

Начнём с сортировки данных. Мы уже умеем делать её вручную, с помощью сортировки выбором, а также структуры multiset. Сегодня мы освоим ещё один, более быстрый и простой метод.

Итак, возьмём последовательность чисел, которую нужно считать, упорядочить и вывести. Вот как решается эта задача:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main() {

    int n;

    cin >> n;

    vector <int> a(n);

    for (int i = 0; i < n; i++) {

        cin >> a[i];

    }

    sort(a.begin(), a.end());

    for (auto now : a) {

        cout << now << " ";

    }

    return 0;

}

В этой программе есть новая функция sort. Она принимает на вход два параметра: начало и конец сортируемой области. В нашем случае это начало вектора (begin) и его конец (end).Функцию sort можно применять для векторов, которые состоят из сравнимых величин: целых и вещественных чисел, строк и чего-угодно другого, для чего можно ввести операцию «меньше».

**Сортировка пар**

Очень часто требуется отсортировать пары элементов. Мы уже столкнулись с парами во время изучения словарей, а сейчас остановимся на них подробнее. Для их использования нужно подключить библиотеку utility.

Например, мы хотим отсортировать пары «число — номер в исходной последовательности». В качестве ответа нужно вывести позиции отсортированных элементов в исходном массиве.

#include <iostream>

#include <vector>

#include <algorithm>

#include <utility>

using namespace std;

int main() {

    int n;

    cin >> n;

    vector <pair <int, int>> a(n);

    for (int i = 0; i < n; i++) {

        int temp;

        cin >> temp;

        a[i] = {temp, i}; // создание пары значение - номер

    }

    sort(a.begin(), a.end());

    for (auto now : a) {

        cout << now.second << " ";

    }

    return 0;

}

Чтобы создать пару, пишем ключевое слово pair, затем в треугольных скобках указываем через запятую два типа (для полей first и second соответственно). Ещё один интересный трюк — это создание пары из двух значений. Достаточно в фигурных скобках написать оба значения через запятую, чтобы получилась. Отличие от предыдущей задачи состоит в том, что мы выводим только поля second — номера элементов в исходной последовательности.

При сравнении двух пар сначала сравниваются первые элементы, а если они равны — то вторые. Как время: сначала по часам, затем по минутам.

**Сортировка по убыванию**

Сортировать по убыванию очень просто — нужно сначала выстроить массив по возрастанию, а потом его развернуть. Для этого используется функция reverse. В ней задаются те же параметры, что и у sort — начало и конец области.

**Структуры**

Не все сложные объекты можно описать с помощью пар. Чтобы описывать объекты, которые характеризуются несколькими разными значениями, нужны структуры. Фактически, структура — это новый тип переменных. Сначала нужно описать структуру, а после этого можно создавать переменные, вектора и прочие элементы такого типа. Описание структуры должно следовать сразу после using namespace std и находиться вне функций. Например, мы хотим описать человека при помощи двух характеристик: рост и имя. Структура создается так:

struct man {

    int height;

    string name;

};

Сначала пишем ключевое слово struct, затем — название структуры, а в фигурных скобках перечислим поля структуры вместе с указанием типа каждого поля. Обратите внимание на точку с запятой после фигурной скобки — она обязательно нужна.

Посмотрим полное решение задачи, в которой нужно считать информацию о людях, упорядочить их по росту и вывести имена.

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

using namespace std;

struct man {

    int height;

    string name;

};

bool cmp(man a, man b) {

    return a.height < b.height;

}

int main() {

    int n;

    cin >> n;

    vector <man> a(n);

    for (int i = 0; i < n; i++) {

        int temp;

        string s\_temp;

        cin >> temp >> s\_temp;

        man struct\_temp; // временная структура

        struct\_temp.height = temp;

        struct\_temp.name = s\_temp;

        a[i] = struct\_temp; // создание пары значение - номер

    }

    sort(a.begin(), a.end(), cmp);

    for (auto now : a) {

        cout << now.name << endl;

    }

    return 0;

}

В этой программе, во-первых, мы научились создавать переменные для описания людей, а также вектор из таких переменных. Во-вторых, теперь мы можем обращаться к отдельным полям структуры. Как и в парах, это делается через точку, то есть first и second в паре — это тоже поля структуры. И, наконец, мы научились сортировать структуры. Для этого в программе используется функция sort с тремя параметрами. Первые два из них обозначают начало и конец сортируемой области, а третий — имя функции для сравнения. В нашем случае это функция cmp, которая принимает на вход две наших структуры и возаращает истину, если первая «меньше» второй, или ложь в обратном случае. Этой функции достаточно для сортировки. Если заменить в ней знак «меньше» на «больше», то массив будет отсортирован по убыванию.

**Устойчивая сортировка**

Сортировка называется устойчивой, если она сохраняет взаимный порядок одинаковых элементов. Если Вася и Петя одного роста, но в исходной последовательности Вася стоял раньше Пети, то после устойчивой сортировки Вася также должен идти перед Петей.

При использовании sort взаимный порядок одинаковых элементов может нарушиться (Петя окажется перед Васей). Чтобы этого не произошло, нужно использовать функцию устойчивой сортировки stable\_sort. Она принимает те же параметры, что и sort, но работает немного медленнее.

**Медиана**

В статистике широко используется понятие «среднее значение». Например, есть три человека с зарплатами 3 миллиона, 300 рублей и 0 рублей. Средняя зарплата получится в 1 миллион 100 рублей. Ещё одна характеристика, которая полезна для оценки значений в последовательности, называется «медиана». Это значение элемента, который стоит на среднем месте в упорядоченной последовательности. В нашем примере с зарплатой медиана равна 300 рублям. Если количество элементов чётное, то медиан получается две.

Чтобы найти медианное значение, нужно отсортировать массив и взять элемент, стоящий в середине. Существуют и другие, более быстрые способы поиска медианы.

**Перестановки**

Необходимость получить все перестановки в последовательности и выбрать среди них наилучшую возникает очень часто. В C++ есть встроенная функция, которая позволяет из последовательности, содержащей перестановку чисел, получить следующую перестановку. Например, за перестановкой 1, 2, 3 идёт перестановка 1, 3, 2, затем 2, 1, 3 и так далее. Функция называется next\_permutation и принимает на вход два параметра: начало и конец последовательности, содержащей перестановку. Next\_permutation изменяет порядок значений в последовательности, а также возвращает логическое значение. Обычно это истина — кроме случая, когда все элементы уже стоят в убывающем порядке, и следующей перестановки просто нет. Тогда функция переставляет элементы в возрастающем порядке и возвращает ложь.

Например, такая программа сначала заполняет последовательность из N чисел числами от 1 до N по возрастанию, а затем генерирует все возможные перестановки и выводит их:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void print(vector <int> a) { // функция вывода вектора

    for (auto now : a) {

        cout << now << " ";

    }

    cout << endl;

}

int main() {

    int n;

    cin >> n;

    vector <int> a(n);

    for (int i = 1; i <= n; i++) {

        a[i – 1] = i;

    }

    while (next\_permutation(a.begin(), a.end()) {

        print(a);

    }

    return 0;

}

**Вместо заключения**

На этом наш курс заканчивается — я надеюсь, вам понравилось. С тем практическим опытом, который вы получили, будет намного легче двигаться дальше, продолжая своё обучение в университете, на курсах или самостоятельно. Спасибо за совместную работу!

**Алгоритмы в Стандартной библиотеке С++**

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

**Оглавление:**

1. [**Библиотека алгоритмов**](https://ravesli.com/algoritmy-v-standartnoj-biblioteke-s/#toc-0)
2. [**Алгоритм std::find() и поиск элемента по значению**](https://ravesli.com/algoritmy-v-standartnoj-biblioteke-s/#toc-1)
3. [**Алгоритм std::find\_if() и поиск элемента с условием**](https://ravesli.com/algoritmy-v-standartnoj-biblioteke-s/#toc-2)
4. [**Алгоритмы std::count()/std::count\_if() и подсчет вхождений элемента**](https://ravesli.com/algoritmy-v-standartnoj-biblioteke-s/#toc-3)
5. [**Алгоритм std::sort() и пользовательская сортировка**](https://ravesli.com/algoritmy-v-standartnoj-biblioteke-s/#toc-4)
6. [**Алгоритм std::for\_each() и все элементы контейнера**](https://ravesli.com/algoritmy-v-standartnoj-biblioteke-s/#toc-5)
7. [**Порядок выполнения**](https://ravesli.com/algoritmy-v-standartnoj-biblioteke-s/#toc-6)
8. [**Диапазоны в C++20**](https://ravesli.com/algoritmy-v-standartnoj-biblioteke-s/#toc-7)
9. [**Заключение**](https://ravesli.com/algoritmy-v-standartnoj-biblioteke-s/#toc-8)

**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**](https://ravesli.com/urok-74-massivy-chast-1/). 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 трех 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**](https://ravesli.com/urok-21-zagolovochnye-fajly/) algorithm). На этом уроке мы рассмотрим некоторые из наиболее распространенных алгоритмов.

***Примечание*:** Все эти алгоритмы используют [**итераторы**](https://ravesli.com/vvedenie-v-iteratory-v-s/).

**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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | #include <algorithm>  #include <array>  #include <iostream>    int main()  {    std::array<int, 6> arr{ 13, 90, 99, 5, 40, 80 };      std::cout << "Enter a value to search for and replace with: ";    int search{};    int replace{};    std::cin >> search >> replace;      // Проверка пользовательского ввода должна быть здесь      // std::find() возвращает итератор, указывающий на найденный элемент (или на конец контейнера).    // Мы сохраним его в переменной, используя автоматический вывод типа итератора    auto found{ std::find(arr.begin(), arr.end(), search) };      // Алгоритмы, которые не нашли то, что искали, возвращают итератор, указывающий на конец контейнера.    // Мы можем получить доступ к этому итератору, используя метод end()    if (found == arr.end())    {      std::cout << "Could not find " << search << '\n';    }    else    {      // Перезаписываем найденный элемент      \*found = replace;    }      for (int i : arr)    {      std::cout << i << ' ';    }      std::cout << '\n';      return 0;  } |

***Примечание:*** Для корректной работы всех примеров данного урока ваш компилятор должен поддерживать стандарт С++17. Детально о том, как использовать функционал C++17 вашей IDE, читайте [**здесь**](https://ravesli.com/konfiguratsiya-kompilyatora-vybor-standarta-yazyka-s/).

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

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

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

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{} (который находится в заголовочном файле functional) для этой задачи. 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 библиотеки С++ |

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**](https://ravesli.com/urok-127-anonimnye-obekty/) 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #include <algorithm>  #include <array>  #include <iostream>    void doubleNumber(int &i)  {    i \*= 2;  }    int main()  {    std::array arr{ 1, 2, 3, 4 };      std::for\_each(arr.begin(), arr.end(), doubleNumber);      for (int i : arr)    {      std::cout << i << ' ';    }      std::cout << '\n';      return 0;  } |

Результат выполнения программы:

2 4 6 8

Новичкам данный способ может показаться ненужным алгоритмом, потому что эквивалентный код с использованием [**цикла for**](https://ravesli.com/urok-69-tsikl-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 С++ 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.