

Ravesli [Ravesli](#)


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


Лямбда-выражения (анонимные функции) в C++

 [Дмитрий Бушуев](#) |

- [Уроки C++](#)

|

 Обновл. 8 Сен 2020 |

 18925

[14](#)

На этом уроке мы рассмотрим лямбда-выражения, их типы и использование в языке C++.

Оглавление:

1. [Зачем нужны лямбда-выражения?](#)
2. [Введение в лямбда-выражения](#)
3. [Тип лямбда-выражений](#)
4. [Общие/Обобщённые лямбды](#)
5. [Общие лямбды и статические переменные](#)
6. [Вывод возвращаемого типа и возвращаемые типы trailing](#)
7. [Функциональные объекты Стандартной библиотеки C++](#)
8. [Заключение](#)
9. [Тест](#)

Зачем нужны лямбда-выражения?

Рассмотрим следующий пример:

```
1 #include <algorithm>
2 #include <array>
3 #include <iostream>
4 #include <string_view>
5
6 static bool containsNut(std::string_view str) // static в данном контексте означает вн
```

```
7 {
8     // Функция std::string_view::find() возвращает std::string_view::npos, если она не на
9     // В противном случае, она возвращает индекс, где происходит вхождение подстроки в с
10    return (str.find("nut") != std::string_view::npos);
11 }
12
13 int main()
14 {
15     std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };
16
17     // std::find_if() принимает указатель на функцию
18     auto found{ std::find_if(arr.begin(), arr.end(), containsNut) };
19
20     if (found == arr.end())
21     {
22         std::cout << "No nuts\n";
23     }
24     else
25     {
26         std::cout << "Found " << *found << '\n';
27     }
28
29     return 0;
30 }
```

Этот код перебирает **массив** строк в поисках первого попавшегося элемента, который содержит подстроку nut. Таким образом, результат выполнения программы:

Found walnut

Хотя это и рабочий код, но мы можем его улучшить.

Проблема кроется в том, что функция std::find_if() требует **указатель на функцию** в качестве аргумента. Из-за этого мы вынуждены определить новую функцию, которая будет использована только один раз, дать ей имя и поместить её в глобальную область видимости (т.к. функции не могут быть вложенными!). При этом она будет настолько короткой, что быстрее и проще понять её смысл, посмотрев лишь на одну строку кода, нежели изучать описание этой функции и её имя.

Введение в лямбда-выражения

Лямбда-выражение (или просто «*лямбда*») в программировании позволяет определить анонимную функцию внутри другой функции. Возможность сделать функцию вложенной является очень важным преимуществом, так как позволяет избегать как захламления **пространства имен** лишними объектами, так и определить функцию как можно ближе к месту её первого использования.

Синтаксис лямбда-выражений является одним из самых странных в языке C++, и вам может потребоваться некоторое время, чтобы к нему привыкнуть.

Лямбда-выражения имеют следующий синтаксис:

```
[ captureClause ] ( параметры ) -> возвращаемыйТип
{
    стейтменты;
}
```

Поля `captureClause` и параметры могут быть пустыми, если они не требуются программисту.

Поле `возвращаемыйТип` является опциональным, и, если его нет, то будет использоваться вывод типа с помощью ключевого слова `auto`. Хотя мы ранее уже отмечали, что следует избегать использования вывода типа для возвращаемых значений функций, в данном контексте подобное использование допускается (поскольку обычно такие функции являются тривиальными).

Также обратите внимание, что лямбда-выражения не имеют имен, поэтому нам и не нужно будет их предоставлять. Из этого факта следует, что тривиальное определение лямбды может иметь следующий вид:

```
1 #include <iostream>
2
3 int main()
4 {
5     []() {}; // определяем лямбда-выражение без captureClause, параметров и возвращаемого
6
7     return 0;
8 }
```

Давайте перепишем предыдущий пример, но уже с использованием лямбда-выражений:

```
1 #include <algorithm>
2 #include <array>
3 #include <iostream>
4 #include <string_view>
5
6 int main()
7 {
8     std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };
9
10    // Определяем функцию непосредственно в том месте, где собираемся её использовать
11    auto found{ std::find_if(arr.begin(), arr.end(),
12                            [](std::string_view str) // вот наша лямбда, без поля captu
13                            {
14                                return (str.find("nut") != std::string_view::npos);
15                            }) };
16
17    if (found == arr.end())
18    {
19        std::cout << "No nuts\n";
20    }
```

```
21 | else
22 | {
23 |     std::cout << "Found " << *found << '\n';
24 | }
25 |
26 | return 0;
27 | }
```

При этом всё работает точно так же, как и в случае с указателем на функцию. Результат выполнения программы аналогичен:

Found walnut

Обратите внимание, насколько наша лямбда похожа на функцию `containsNut()`. Они обе имеют одинаковые параметры и тела функций. Отметим, что у лямбды отсутствует поле `captureClause` (детально о `captureClause` мы поговорим на следующем уроке), т.к. оно не нужно. Также для краткости мы пропустили синтаксис типа возвращаемого значения `trailing`, но из-за того, что `operator!=` возвращает значение типа `bool`, наша лямбда также будет возвращать логическое значение.

Тип лямбда-выражений

В примере, приведенном выше, мы определили лямбду прямо в том месте, где она была нам нужна. Такое использование лямбда-выражения иногда еще называют **функциональным литералом**.

Однако написание лямбды в той же строке, где она используется, иногда может затруднить чтение кода. Подобно тому, как мы можем инициализировать переменную с помощью литерала (или указателя на функцию) для использования в дальнейшем, так же мы можем инициализировать и лямбда-переменную с помощью лямбда-определения для её дальнейшего использования. Именованная лямбда вместе с удачным именем функции может облегчить чтение кода.

Например, в следующем фрагменте кода мы используем функцию `std::all_of()` для того, чтобы проверить, являются ли все элементы массива чётными:

```
1 | // Плохо: Мы должны прочитать лямбду, чтобы понять, что происходит
2 | return std::all_of(array.begin(), array.end(), [](int i){ return ((i % 2) == 0); });
```

Мы можем улучшить читабельность кода следующим образом:

```
1 | // Хорошо: Мы можем хранить лямбду в именованной переменной и передавать её в функцию в
2 | auto isEven{
3 |     [](int i)
4 |     {
5 |         return ((i % 2) == 0);
6 |     }
7 | };
8 |
9 | return std::all_of(array.begin(), array.end(), isEven);
```

Обратите внимание, как просто читается последняя строка кода: «... возвращаем все элементы массива, которые являются чётными ...».

Но какого типа является лямбда в `isEven`?

Оказывается, у лямбд нет типа, который мы могли бы явно использовать. Когда мы пишем лямбду, компилятор генерирует уникальный тип лямбды, который нам не виден.

Для продвинутых читателей: На самом деле, лямбды не являются функциями (что и помогает им избегать ограничений C++, которые накладываются на использование вложенных функций). Лямбды являются особым типом объектов, который называется **функтором**. **Функторы** — это объекты, содержащие [перегруженный оператор\(\)](#), который и делает их вызываемыми подобно обычным функциям.

Хотя мы не знаем тип лямбды, есть несколько способов её хранения для использования после определения. Если лямбда ничего не захватывает, то мы можем использовать обычный указатель на функцию. Как только лямбда что-либо захватывает, указатель на функцию больше не будет работать. Однако `std::function` может использоваться для лямбд, даже если они что-то захватывают:

```
1  #include <functional>
2
3  int main()
4  {
5      // Обычный указатель на функцию. Лямбда не может ничего захватить
6      double (*addNumbers1)(double, double){
7          [](double a, double b) {
8              return (a + b);
9          }
10     };
11
12     addNumbers1(1, 2);
13
14     // Используем std::function. Лямбда может захватывать переменные.
15     std::function addNumbers2{ // примечание: Если у вас не поддерживается C++17 и выше,
16         [](double a, double b) {
17             return (a + b);
18         }
19     };
20
21     addNumbers2(3, 4);
22
23     // Используем auto. Храним лямбду с её реальным типом
24     auto addNumbers3{
25         [](double a, double b) {
26             return (a + b);
27         }
28     };
29
30     addNumbers3(5, 6);
```

```
31 |  
32 |     return 0;  
33 | }
```

С помощью `auto` мы можем использовать фактический тип лямбды. При этом мы можем получить преимущество в виде отсутствия накладных расходов в сравнении с использованием `std::function`.

К сожалению, мы не всегда можем использовать `auto`. В тех случаях, когда фактический тип лямбды неизвестен (например, из-за того, что мы передаем лямбду в функцию в качестве параметра, и вызывающий объект сам определяет какого типа лямбда будет передана), мы не можем использовать `auto`. В таких случаях следует использовать `std::function`:

```
1  #include <functional>  
2  #include <iostream>  
3  
4  // Мы не знаем, чем будет fn. std::function работает с обычными функциями и лямбдами  
5  void repeat(int repetitions, const std::function<void(int)>& fn)  
6  {  
7      for (int i{ 0 }; i < repetitions; ++i)  
8      {  
9          fn(i);  
10     }  
11 }  
12  
13 int main()  
14 {  
15     repeat(3, [](int i) {  
16         std::cout << i << '\n';  
17     });  
18  
19     return 0;  
20 }
```

Результат выполнения программы:

```
0  
1  
2
```

Правило: Используйте `auto` при инициализации переменных с помощью лямбд и `std::function`, если вы не можете инициализировать переменную с помощью лямбд.

Общие/Обобщённые лямбды

По большей части лямбда-параметры работают так же, как и обычные параметры функций.

Одним примечательным исключением является то, что, начиная с [C++14](#), нам разрешено использовать `auto` с параметрами функций.

Примечание: В C++20 обычные функции также могут использовать `auto` с параметрами.

Если у лямбды есть один или несколько параметров `auto`, то компилятор определит необходимые типы параметров из вызовов лямбд-выражений.

Поскольку лямбды с одним или несколькими параметрами типа `auto` потенциально могут работать с большим количеством типов данных, то они называются **общими** (или «*обобщёнными*» от англ. «*generic lambdas*») **лямбдами**.

Рассмотрим использование общей лямбды на практике:

```
1  #include <algorithm>
2  #include <array>
3  #include <iostream>
4  #include <string_view>
5
6  int main()
7  {
8      std::array months{ // если у вас не поддерживается C++17, то используйте std::array<
9          "January",
10         "February",
11         "March",
12         "April",
13         "May",
14         "June",
15         "July",
16         "August",
17         "September",
18         "October",
19         "November",
20         "December"
21     };
22
23     // Поиск двух последовательных месяцев, которые начинаются с одинаковой буквы
24     auto sameLetter{ std::adjacent_find(months.begin(), months.end(),
25         [](const auto& a, const auto& b) {
26             return (a[0] == b[0]);
27         }) };
28
29     // Убеждаемся, что эти два месяца были найдены
30     if (sameLetter != months.end())
31     {
32         std::cout << *sameLetter << " and " << *std::next(sameLetter)
33             << " start with the same letter\n";
34     }
35 }
```

```
36 | return 0;  
37 | }
```

Результат выполнения программы:

June and July start with the same letter

В примере, приведенном выше, мы использовали auto-параметры для захвата наших строк с использованием [константной ссылки](#). Т.к. все строковые типы предоставляют доступ к своим отдельным символам через оператор [], то нам не нужно волноваться о том, передает ли пользователь в качестве параметра std::string, [строку C-style](#) или что-то другое. Это позволяет нам написать лямбду, которая могла бы принять любой из этих объектов, то есть, если позже мы изменим тип months, — нам не придется переписывать лямбду.

Однако auto не всегда является лучшим выбором. Рассмотрим следующую программу:

```
1  #include <algorithm>  
2  #include <array>  
3  #include <iostream>  
4  #include <string_view>  
5  
6  int main()  
7  {  
8      std::array months{ // если у вас не поддерживается C++17, то используйте std::array<  
9          "January",  
10         "February",  
11         "March",  
12         "April",  
13         "May",  
14         "June",  
15         "July",  
16         "August",  
17         "September",  
18         "October",  
19         "November",  
20         "December"  
21     };  
22  
23     // Подсчитываем количество месяцев с названиями в 5 букв  
24     auto fiveLetterMonths{ std::count_if(months.begin(), months.end(),  
25         [](std::string_view str) {  
26             return (str.length() == 5);  
27         }) };  
28  
29     std::cout << "There are " << fiveLetterMonths << " months with 5 letters\n";  
30  
31     return 0;  
32 }
```


Результат выполнения программы:

There are 2 months with 5 letters

В этом примере использование `auto` выводит тип `const char*`. Мы знаем, что со строками C-style трудно работать (кроме использования оператора `[]`). Поэтому в данном случае для нас предпочтительнее явно определить тип параметра, как `std::string_view`, который позволит нам работать с базовыми типами данных намного проще (например, мы можем запросить у представления значение длины строки, даже если пользователь передал в качестве аргумента массив C-style).

Общие лямбды и статические переменные

Следует иметь в виду, что для каждого отдельного типа, выводимого с помощью `auto`, будет сгенерирована уникальная лямбда. В следующем примере показано, как общая лямбда разделяется на две отдельные:

```
1  #include <algorithm>
2  #include <array>
3  #include <iostream>
4  #include <string_view>
5
6  int main()
7  {
8      // Выводим значение и подсчитываем, сколько раз будет вызван print
9      auto print{
10         [](auto value) {
11             static int callCount{ 0 };
12             std::cout << callCount++ << ": " << value << '\n';
13         }
14     };
15
16     print("hello"); // 0: hello
17     print("world"); // 1: world
18
19     print(1); // 0: 1
20     print(2); // 1: 2
21
22     print("ding dong"); // 2: ding dong
23
24     return 0;
25 }
```

Результат выполнения программы:

```
0: hello
1: world
0: 1
1: 2
2: ding dong
```

В примере, приведенном выше, мы определяем лямбду и затем вызываем её с двумя различными параметрами (строковым литералом и целочисленным типом). При этом генерируются две различные версии лямбды (одна с параметром строкового литерала, а другая — с параметром в виде целочисленного типа).

В большинстве случаев это не существенно. Однако, обратите внимание, что если в общей лямбде используются статические переменные, то эти переменные не являются общими для генерируемых лямбд.

Мы можем видеть это в вышеприведенном примере, где каждый тип (строковые литералы и целые числа) имеет свой собственный уникальный счет! Хотя мы написали лямбду один раз, были сгенерированы две лямбды, и у каждой есть своя версия `callCount`.

Если бы мы хотели, чтобы `callCount` был общим для лямбд, то нам пришлось бы объявить его вне лямбды и захватить его по ссылке, чтобы он мог быть изменен лямбдой.

Вывод возвращаемого типа и возвращаемые типы `trailing`

Если использовался вывод возвращаемого типа, то возвращаемый тип лямбды выводится из `стейтментов return` внутри лямбды. Если использовался вывод возвращаемого типа, то все возвращаемые `стейтменты` внутри лямбды должны возвращать значения одного и того же типа (иначе компилятор не будет знать, какой из них ему следует использовать). Например:

```
1  #include <iostream>
2
3  int main()
4  {
5      auto divide{ [](int x, int y, bool bInteger) { // примечание: Не указан тип возвращаемого значения
6          if (bInteger)
7              return x / y;
8          else
9              return static_cast<double>(x) / y; // ОШИБКА: Тип возвращаемого значения не совпадает
10     } };
11
12     std::cout << divide(3, 2, true) << '\n';
13     std::cout << divide(3, 2, false) << '\n';
14
15     return 0;
16 }
```

Это приведет к ошибке компиляции, так как тип возвращаемого значения первого `стейтмента return (int)` не совпадает с типом возвращаемого значения второго `стейтмента return (double)`.

В случае, когда у нас используются разные возвращаемые типы, у нас есть два варианта:

- ➔ выполнить явные преобразования в один тип;

➔ явно указать тип возвращаемого значения для лямбды и позволить компилятору выполнить неявные преобразования.

Второй вариант обычно является более предпочтительным:

```
1 #include <iostream>
2
3 int main()
4 {
5     // Примечание: Явно указываем тип double для возвращаемого значения
6     auto divide{ [](int x, int y, bool bInteger) -> double {
7         if (bInteger)
8             return x / y; // выполнится неявное преобразование в тип double
9         else
10            return static_cast<double>(x) / y;
11     } };
12
13     std::cout << divide(3, 2, true) << '\n';
14     std::cout << divide(3, 2, false) << '\n';
15
16     return 0;
17 }
```

Таким образом, если вы когда-либо решите изменить тип возвращаемого значения, вам (как правило) нужно будет изменить только тип возвращаемого значения лямбды и ничего внутри основной части.

Функциональные объекты Стандартной библиотеки C++

Для основных операций (например, сложения, вычитания или сравнения) вам не нужно писать свои собственные лямбды, потому что Стандартная библиотека C++ поставляется со многими базовыми вызываемыми объектами, которые вы можете использовать. Эти объекты определены в заголовочном файле `functional`. Например:

```
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 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 параметра в 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

Вместо преобразования функции `greater()` в лямбду, мы можем использовать `std::greater`:

```
1  #include <algorithm>
2  #include <array>
3  #include <iostream>
4  #include <functional> // для std::greater
5
6  int main()
7  {
8      std::array arr{ 13, 90, 99, 5, 40, 80 };
9
10     // Передаём std::greater в качестве параметра в std::sort()
11     std::sort(arr.begin(), arr.end(), std::greater{}); // примечание: Требуются фигурные
12
13     for (int i : arr)
14     {
15         std::cout << i << ' ';
16     }
17
18     std::cout << '\n';
19
20     return 0;
21 }
```

Результат выполнения программы:

99 90 80 40 13 5

Заключение

Лямбда-выражения и библиотека алгоритмов могут показаться излишне сложными по сравнению с обычными решениями, использующими циклы. Однако эта комбинация позволяет выполнять некоторые

очень мощные операции всего в несколько строчек кода и может быть куда читабельнее, чем ваши «самописные» циклы. Кроме того, библиотека алгоритмов обладает мощным и простым в использовании параллелизмом, который вы не получите при использовании циклов. Обновление исходного кода, использующего библиотечные функции, проще, чем обновление кода, использующего циклы.

Лямбды великолепны, но они не заменяют обычные функции для всех случаев. Используйте обычные функции для нетривиальных случаев.

Тест

Задание №1

Создайте структуру `Student`, которая будет хранить имя и баллы студента. Создайте массив студентов и используйте функцию `std::max_element()` для поиска студента с наибольшими баллами, а затем выведите на экран имя найденного студента. Функция `std::max_element()` принимает `begin` и `end` списка, и функцию с двумя параметрами, которая возвращает `true`, если первый аргумент меньше второго.

При использовании следующего массива:

```
1  std::array<Student, 8> arr{
2      { { "Albert", 3 },
3        { "Ben", 5 },
4        { "Christine", 2 },
5        { "Dan", 8 }, // Dan имеет больше всего баллов (8).
6        { "Enchilada", 4 },
7        { "Francis", 1 },
8        { "Greg", 3 },
9        { "Hagrid", 5 } }
10 };
```

Результатом выполнения вашей программы должно быть следующее:

Dan is the best student

Показать подсказку

```
1  #include <array>
2  #include <iostream>
3  #include <string>
4
5  struct Student
6  {
7      std::string name{};
8      int points{};
9  };
10
11 int main()
12 {
13     std::array<Student, 8> arr{
```

```
14     { { "Albert", 3 },
15       { "Ben", 5 },
16       { "Christine", 2 },
17       { "Dan", 8 },
18       { "Enchilada", 4 },
19       { "Francis", 1 },
20       { "Greg", 3 },
21       { "Hagrid", 5 } }
22 };
23
24 auto best{
25     std::max_element(arr.begin(), arr.end(), /* лямбда */
26 };
27
28 std::cout << best->name << " is the best student\n";
29
30 return 0;
31 }
```

Ответ №1

```
1  #include <array>
2  #include <iostream>
3  #include <string>
4
5  struct Student
6  {
7      std::string name{};
8      int points{};
9  };
10
11 int main()
12 {
13     std::array<Student, 8> arr{
14         { { "Albert", 3 },
15           { "Ben", 5 },
16           { "Christine", 2 },
17           { "Dan", 8 },
18           { "Enchilada", 4 },
19           { "Francis", 1 },
20           { "Greg", 3 },
21           { "Hagrid", 5 } }
22     };
23
24     auto best{
25         std::max_element(arr.begin(), arr.end(), [](const auto& a, const auto& b) {
26             return (a.points < b.points);
27         })
28     };
29 }
```

```
29  
30     std::cout << best->name << " is the best student\n";  
31  
32     return 0;  
33 }
```

Задание №2

Используйте `std::sort()` и лямбду в следующем коде для сортировки времен года по возрастанию средней температуры:

```
1  #include <algorithm>  
2  #include <array>  
3  #include <iostream>  
4  #include <string_view>  
5  
6  struct Season  
7  {  
8      std::string_view name{};  
9      double averageTemperature{};  
10 };  
11  
12 int main()  
13 {  
14     std::array<Season, 4> seasons{  
15         { "Spring", 285.0 },  
16         { "Summer", 296.0 },  
17         { "Fall", 288.0 },  
18         { "Winter", 263.0 } }  
19 };  
20  
21 /*  
22  * Используйте std::sort() здесь  
23  */  
24  
25 for (const auto& season : seasons)  
26 {  
27     std::cout << season.name << '\n';  
28 }  
29  
30 return 0;  
31 }
```

Результатом выполнения вашей программы должно быть следующее:

```
Winter  
Spring  
Fall  
Summer
```

Ответ №2

```
1  #include <algorithm>
2  #include <array>
3  #include <iostream>
4  #include <string_view>
5
6  struct Season
7  {
8      std::string_view name{};
9      double averageTemperature{};
10 };
11
12 int main()
13 {
14     std::array<Season, 4> seasons{
15         { { "Spring", 285.0 },
16           { "Summer", 296.0 },
17           { "Fall", 288.0 },
18           { "Winter", 263.0 } }
19     };
20
21     // Лямбде не нужно даже ничего захватывать, потому что порядок зависит только от эле
22     // Мы можем сравнить averageTemperature двух аргументов для сортировки массива
23     std::sort(seasons.begin(), seasons.end(),
24               [](const auto& a, const auto& b) {
25                   return (a.averageTemperature < b.averageTemperature);
26               });
27
28     for (const auto& season : seasons)
29     {
30         std::cout << season.name << '\n';
31     }
32
33     return 0;
34 }
```

Оценить статью:

 (66 оценок, среднее: 4,65 из 5)[← Урок №111. Эллипсис](#)[Лямбда-захваты в C++](#) 

Комментариев: 14



1. *Всеволод:*

[20 сентября 2020 в 19:40](#)

Объясните, пожалуйста, что значит в данной строке

`return (str.find("nut") != std::string_view::npos); =>`

`!= std::string_view::npos);` не пойму, что делает вторая половина этой строки

[Ответить](#)



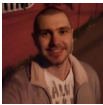
1. *Rock:*

[27 сентября 2020 в 23:29](#)

`find` возвращает итератор на нужную ячейку массива (0 или 2 или 6...)

но если не найдет ничего выдает -1 показывая что нету такого в массиве `npos` тоже что и -1 но пишут `npos` из за особенностей некоторых `ide` вот мы и получаем буловую логику если `find` выдаст что угодно кроме как -1 то нужно вернуть `true` и в переменную запишется то что нашла эта функция если уравнение будет `-1 == -1` то из за оператора не равно (`!=`) бул вернет `false` и в переменную не будет записана инфа и по итерации пойдет дальше

[Ответить](#)



2. *Дмитрий Бушуев:*

[28 сентября 2020 в 00:17](#)

`std::string_view::npos` — это специальная константа. Точное значение зависит от контекста, но обычно она используется как индикатор достижения конца строки в процессе некоторого поиска.

Функция `str.find("nut")` возвращает либо индекс первого элемента подстроки (если она найдена), либо — `npos` (если такой подстроки не нашлось) 😊

[Ответить](#)



2. *Владислав:*

[16 августа 2020 в 17:01](#)

Почему в первом задании теста `uniform`-инициализация происходит так:

```
1 std::array<Student, 8> arr{
2     { "Albert", 3},
3     { "Ben", 5},
4     { "Christine", 2},
5     { "Dan", 8},
6     { "Enchilada", 4},
```

```
7         {"Francis", 1},
8         {"Greg", 3},
9         {"Hagrid", 5}    }
10    };
```

а не так:

```
1    std::array<Student, 8> arr{
2        {"Albert", 3},
3        {"Ben", 5},
4        {"Christine", 2},
5        {"Dan", 8},
6        {"Enchilada", 4},
7        {"Francis", 1},
8        {"Greg", 3},
9        {"Hagrid", 5}
10    };
```

?

Попробовал в VS второй вариант, и действительно, студия ругается. Но почему, в таком случае, можно спокойно объявить с помощью `uniform-инициализации` массив `std::array` целых чисел следующим образом:

```
1    std::array<int, 4> arr{
2        1, 2, 3, 4
3    };
```

И студия не будет ругаться? Почему при объявлении структур таким образом она ругается (говорит, что слишком много значений инициализатора (компилятор)). Нелогично, на мой взгляд.

VS 2019, C++ 17

[Ответить](#)



1. *Владимир:*

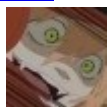
[4 сентября 2020 в 09:03](#)

Потому что ты невнимательный. В твоём примере массив из фундаментальных типов `int`, и он(`int`) принимает 1 параметр.

А в случае того же массива из структур `Student`, и он(`Student`) принимает 2 параметра.

Будь внимательней.

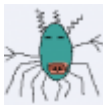
[Ответить](#)



1. *Иван:*

[23 октября 2020 в 14:23](#)

Вопрос про второй комплект угловых скобок. Причем здесь второй параметр?

[Ответить](#)1. *Мах:*[2 ноября 2020 в 01:11](#)

На сколько я понял, они здесь вместо знака равно, тоесть обычное присваивание.

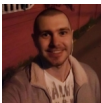
3. *VEX:*[31 июля 2020 в 13:26](#)

"Правило: Используйте auto при инициализации переменных с помощью лямбд и std::function, если вы не можете инициализировать переменную с помощью лямбд."

А мне кажется, что лучше всегда использовать std::function, тем более что после C++17 и тип возврата, и типы параметров указывать необязательно. На мой взгляд ключевое слово auto уже немного устарел 😊

[Ответить](#)4. *Viktor:*[5 июля 2020 в 15:11](#)

Здравствуйте) Если кому то не сложно, не могли бы вы объяснить предложение "Функция std::max_element() принимает begin и end списка и функцию с двумя параметрами, которая возвращает true, если первый аргумент меньше второго.". Я не совсем понимаю зачем в этой функции нужен третий параметр в виде лямбды. Заранее благодарю всех кто ответит.

[Ответить](#)1. *Дмитрий Бушуев:*[7 июля 2020 в 00:07](#)

Так а как вы собираетесь сравнивать двух студентов между собой? Это же составной тип данных, C++ ничего не знает про сравнение подобных переменных. Поэтому вы сами должны написать эту функцию (в виде лямбды) 😊

[Ответить](#)5. *Анастасия:*[5 мая 2020 в 12:32](#)

Я прочитала этот урок дважды с перерывом в неделю. И всё равно есть чувство, что я поняла не всё.

Во-первых, урок очень длинный.

Во-вторых, в примерах используется std::string_view, о котором до конца апреля в этих уроках ничего не было. Также используются алгоритмы STL, которые здесь также объяснялись скорее для

расширения кругозора.

В-третьих, кусок "Если лямбда ничего не захватывает, мы можем использовать обычный указатель на функцию. Как только лямбда захватывает что-либо, указатель на функцию больше не будет работать." не ясно, что значит, что лямбда что-то захватывает. Я подозреваю, что это связано с [Capture clause], который здесь никак не поясняется, но должен быть раскрыт в следующем уроке. В-четвёртых, главное правило данного урока "Используйте auto при инициализации переменных с помощью лямбд и std::function, если вы не можете инициализировать переменную с помощью лямбд." тоже не до конца раскрыто. Я прочитала пример, но до конца так и не поняла, в чём разница, и как это на практике разделять. Возможно, причина в том, что слово "лямбда" в данном уроке заменяет несколько разных смыслов, связанных с лямбда-выражениями. Скорей всего, речь идёт о том, что когда мы даём имя нашему лямбда-выражению, мы можем указать только тип auto. Или другие, более конкретные типы тоже возможны? Судя по тому, что я прочитала, нет, но ведь auto превращается в конкретный тип из return...

В-пятых, абзац "Если использовался вывод возвращаемого типа, то возвращаемый тип лямбды выводится из стейтментов return внутри лямбды. Если использовался вывод возвращаемого типа, то все возвращаемые стейтменты внутри лямбды должны возвращать значения одного и того же типа..." Два предложения, начинающихся с одного и того же условия "Если использовался вывод возвращаемого типа" почему бы тогда не переписать это как-то так, чтобы не путать читателя: Если использовался вывод возвращаемого типа:

- 1) возвращаемый тип лямбды выводится из стейтментов return внутри лямбды
- 2) все возвращаемые стейтменты внутри лямбды должны возвращать значения одного и того же типа

Хотя нет, даже так не понятно, что имеется в виду под "вывод" — печать на экране? В лямбда-выражении или в вызывающей функции?

В общем, здесь я попыталась проанализировать, что же заставило меня прокрастинировать с этим уроком целую неделю и почему до сих пор даже после второго прочтения у меня нет чувства, что "всё понятно". Хотя, судя по всему, тема несложная.

[Ответить](#)

1.  Юрий:
[5 мая 2020 в 15:54](#)

Здравствуйте, Анастасия.

1. Урок действительно больше обычного урока по C++, но посмотрите на уроки по OpenGL и тогда этот урок Вам не будет казаться таким уж большим 😊
2. Вместе с этим уроком были добавлены уроки по [std::string_view](#) и по [алгоритмам](#).
3. Есть отдельный урок и по [лямбда-захватам](#) (они же capture clause).

[Ответить](#)

2.  Дмитрий:
[31 июля 2020 в 14:56](#)

Тоже самое. Это самый сложный урок из всех, которые были до этого.

[Ответить](#)

6. Арбузик ❤️❤️❤️:

[8 марта 2020 в 18:06](#)

Круто!

[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