

Ravesli [Ravesli](#)


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


Введение в итераторы в C++

 [Дмитрий Бушуев](#) |

- [Уроки C++](#)

|

 Обновл. 16 Авг 2020 |

 14055

[↑](#)  6

На этом уроке мы рассмотрим тему использования итераторов в языке C++, а также связанные с этим нюансы.

Оглавление:

1. [Итерация по элементам структур данных](#)
2. [Итераторы в C++](#)
3. [Указатели в качестве итераторов](#)
4. [Итераторы Стандартной библиотеки C++](#)
5. [Итераторы и циклы for с явным указанием диапазона](#)
6. [«Висячие» итераторы](#)

Итерация по элементам структур данных

Итерация/перемещение по элементам [массива](#) (или какой-нибудь другой структуры) является довольно распространенным действием в программировании. Мы уже рассматривали множество различных способов выполнения данной задачи, а именно: с использованием циклов и индексов ([циклы for](#) и [while](#)), с помощью указателей и арифметики указателей, а также с помощью циклов for с явным указанием диапазона:

```
1 #include <array>
2 #include <iostream>
3
4 int main()
```

```
5 {
6 // Автоматическое определение типа как std::array<int, 7> (требуется поддержка станд
7 std::array data{ 0, 1, 2, 3, 4, 5, 6 }; // используйте std::array<int, 7>, если ваш
8 std::size_t length{ std::size(data) };
9
10 // Цикл while с использованием явного индекса
11 std::size_t index{ 0 };
12 while (index != length)
13 {
14     std::cout << data[index] << ' ';
15     ++index;
16 }
17 std::cout << '\n';
18
19 // Цикл for с использованием явного индекса
20 for (index = 0; index < length; ++index)
21 {
22     std::cout << data[index] << ' ';
23 }
24 std::cout << '\n';
25
26 // Цикл for с использованием указателей (обратите внимание, ptr не может быть const,
27 for (auto ptr{ &data[0] }; ptr != (&data[0] + length); ++ptr)
28 {
29     std::cout << *ptr << ' ';
30 }
31 std::cout << '\n';
32
33 // Цикл for с явным указанием диапазона
34 for (int i : data)
35 {
36     std::cout << i << ' ';
37 }
38 std::cout << '\n';
39
40 return 0;
41 }
```

Использование циклов с индексами в ситуациях, когда мы используем их только для доступа к элементам, требует написания большего количества кода, нежели могло бы быть.

При этом данный способ работает только в том случае, если контейнер (например, массив), содержащий данные, дает возможность прямого доступа к своим элементам (что делают массивы, но не делают некоторые другие типы контейнеров, например, списки).

Использование циклов с указателями и арифметикой указателей требует довольно большого объема теоретических знаний и может сбить с толку читателей, которые не знакомы с арифметикой указателей и не знают её правил. Да и сама арифметика указателей применима лишь том случае, если элементы

структуры данных расположены в памяти последовательно (что опять же верно для массивов, но не всегда выполняется для других типов данных, таких как списки, деревья, карты).

Примечание для продвинутых читателей: Указатели (без арифметики указателей) могут использоваться для перебора/итерации некоторых структур данных с непоследовательным расположением элементов. Например, в связном списке каждый элемент соединен указателем с предыдущим элементом, поэтому мы можем перебирать список, следуя по цепочке указателей.

Циклы `for` с явным указанием диапазона чуть более интересны, поскольку у них скрыт механизм перебора нашего контейнера, но при всем этом они всё равно могут быть применены к различным структурам данных (массивы, списки, деревья, карты и т.д.). «Как же они работают?» — спросите вы. Они используют итераторы.

Итераторы в C++

Итератор — это объект, разработанный специально для перебора элементов контейнера (например, значений массива или символов в строке), обеспечивающий во время перемещения по элементам доступ к каждому из них.

Контейнер может предоставлять различные типы итераторов. Например, контейнер на основе массива может предлагать прямой итератор, который проходит по массиву в прямом порядке, и реверсивный итератор, который проходит по массиву в обратном порядке.

После того, как итератор соответствующего типа создан, программист может использовать интерфейс, предоставляемый данным итератором, для перемещения по элементам контейнера или доступа к его элементам, не беспокоясь при этом о том, какой тип перебора элементов задействован или каким образом в контейнере хранятся данные. И, поскольку итераторы в языке C++ обычно используют один и тот же интерфейс как для перемещения по элементам контейнера (оператор `++` для перехода к следующему элементу), так и для доступа (оператор `*` для доступа к текущему элементу) к ним, итерации можно выполнять по разнообразным типам контейнеров, используя последовательный метод.

Указатели в качестве итераторов

Простейший пример итератора — это указатель, который (используя арифметику указателей) работает с последовательно расположенными элементами данных. Давайте снова рассмотрим пример перемещения по элементам массива, используя указатель и арифметику указателей:

```
1 #include <array>
2 #include <iostream>
3
4 int main()
5 {
6     std::array data{ 0, 1, 2, 3, 4, 5, 6 };
7
8     auto begin{ &data[0] };
9     // Обратите внимание, что здесь мы указываем на место, следующее за последним элементом
10    auto end{ begin + std::size(data) };
```

```
11
12 // Цикл for с использованием указателя
13 for (auto ptr{ begin }; ptr != end; ++ptr) // выполняем инкремент для перехода к сле
14 {
15     std::cout << *ptr << ' '; // разыменовываем указатель для получения текущего значе
16 }
17 std::cout << '\n';
18
19 return 0;
20 }
```

Результат выполнения программы:

0 1 2 3 4 5 6

В примере, приведенном выше, мы определили две переменные: `begin` (которая указывает на начало нашего контейнера) и `end` (которая указывает на конец нашего контейнера). Для массивов конечным маркером обычно является место в памяти, где мог находиться последний элемент, если бы контейнер содержал на один элемент больше.

Затем указатель перемещается между `begin` и `end`, при этом доступ к текущему элементу можно получить с помощью оператора разыменовывания.

Предупреждение: У вас может появиться соблазн вычислить конечную точку, используя оператор адреса (&) следующим образом:

```
1 int* end{ &array[std::size(array)] };
```

Но это приведет к неопределенному поведению, потому что `array[std::size(array)]` обращается к элементу, который находится за пределами массива.

Вместо этого следует использовать:

```
1 int* end{ &array[0] + std::size(array) };
```

Итераторы Стандартной библиотеки C++

Выполнение итераций является настолько распространенным действием, что все Стандартные библиотеки контейнеров обеспечивают прямую поддержку итераций. Вместо вычисления начальной и конечной точек вручную, мы можем просто попросить контейнер сделать это за нас, обратившись к функциям `begin()` и `end()`:

```
1 #include <iostream>
2 #include <array>
3
4 int main()
5 {
6     std::array array{ 1, 2, 3 };
```

```

7
8 // Просим наш массив указать нам начальную и конечную точки при помощи функций begin
9 auto begin{ array.begin() };
10 auto end{ array.end() };
11
12 for (auto p{ begin }; p != end; ++p) // выполняем инкремент для перехода к следующему
13 {
14     std::cout << *p << ' '; // разыменовываем указатель для получения текущего значения
15 }
16 std::cout << '\n';
17
18 return 0;
19 }

```

Результат выполнения программы:

1 2 3

В заголовочном файле `iterator` также содержатся две обобщенные функции (`std::begin()` и `std::end()`):

```

1 #include <array>
2 #include <iostream>
3 #include <iterator> // для std::begin() и std::end()
4
5 int main()
6 {
7     std::array array{ 1, 2, 3 };
8
9     // Используем std::begin() и std::end() для получения начальной и конечной точек array
10    auto begin{ std::begin(array) };
11    auto end{ std::end(array) };
12
13    for (auto p{ begin }; p != end; ++p) // выполняем инкремент для перехода к следующему
14    {
15        std::cout << *p << ' '; // разыменовываем указатель для получения текущего значения
16    }
17    std::cout << '\n';
18
19    return 0;
20 }

```

Результат выполнения аналогичен предыдущему результату:

1 2 3

Не стоит сейчас беспокоиться о типах итераторов. Сейчас важно понять лишь то, что всю работу по перемещению по контейнеру итератор берет на себя. Нам не обязательно знать детали того, как это происходит. Нам нужно знать следующие 4 вещи:

- начальная точка;
- конечная точка;
- оператор ++ для перемещения итератора к следующему элементу (или к концу);
- оператор * для получения значения текущего элемента.

Итераторы и циклы for с явным указанием диапазона

Все типы данных, которые имеют методы `begin()` и `end()` или используются с `std::begin()` и `std::end()`, могут быть задействованы в циклах `for` с явным указанием диапазона:

```
1 #include <array>
2 #include <iostream>
3
4 int main()
5 {
6     std::array array{ 1, 2, 3 };
7
8     // Данный цикл работает аналогично циклу, приведенному выше
9     for (int i : array)
10    {
11        std::cout << i << ' ';
12    }
13    std::cout << '\n';
14
15    return 0;
16 }
```

На самом деле, циклы `for` с явным указанием диапазона для осуществления итерации незаметно обращаются к вызовам функций `begin()` и `end()`. Тип данных [std::array](#) также имеет в своем арсенале методы `begin()` и `end()`, а значит и его мы можем использовать в циклах `for` с явным указанием диапазона. Массивы C-style с фиксированным размером также можно использовать с функциями `std::begin()` и `std::end()`. Однако с [динамическими массивами](#) данный способ не работает, так как для них не существует функции `std::end()` (из-за того, что отсутствует информация о длине массива).

Позже вы узнаете, как добавлять функционал к вашим типам данных так, чтобы их можно было использовать и с циклами `for` с явным указанием диапазона.

Циклы `for` с явным указанием диапазона используются не только при работе с итераторами. Они также могут быть задействованы вместе с `std::sort` и другими алгоритмами. Теперь, когда вы знаете, что это такое, вы можете заметить, что они довольно часто используются в Стандартной библиотеке C++.

«Висячие» итераторы

Подобно указателям и [ссылкам](#), итераторы также могут стать «висячими», если элементы, по которым выполняется итерация, изменяют свой адрес или уничтожаются. Когда такое происходит, то говорят, что итератор был **недействительным** (или произошла «*инвалидация итератора*»). Обращение к недействительному итератору порождает ошибку неопределенного поведения.

Некоторые операции, которые изменяют контейнеры (например, добавление элемента в [std::vector](#)), могут иметь побочный эффект, приводя к изменению адресов элементов контейнера. Когда такое происходит, текущие итераторы для этих элементов считаются недействительными. Хорошая справочная документация по C++ обязательно должна иметь информацию о том, какие операции с контейнерами могут привести или приведут к инвалидации итераторов (в качестве примера вот справочная информация по [«инвалидации итераторов» в std::vector](#)).

Вот пример подобной ситуации:

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector v { 1, 2, 3, 4, 5, 6, 7 };
7
8     auto it { v.begin() };
9
10    ++it; // двигаемся ко второму элементу
11    std::cout << *it << '\n'; // ок: выводится "2"
12
13    v.erase(it); // удаляем элемент, на который в данный момент указывает итератор
14
15    // erase() инвалидирует итераторы для стираемого элемента (и последующих элементов)
16    // поэтому теперь итератор "it" является недействительным
17
18    ++it; // неопределенное поведение
19    std::cout << *it << '\n'; // неопределенное поведение
20
21    return 0;
22 }
```

На следующем уроке мы рассмотрим алгоритмы Стандартной библиотеки C++.

Оценить статью:

★★★★★ (77 оценок, среднее: 4,84 из 5)



← [Урок №95. Введение в std::vector](#)



Комментариев: 6



1. *mojoRising:*
[6 декабря 2020 в 12:33](#)

есть такой вопрос ещё, `auto begin{ &data[0] }` — я так понимаю компилятор сам понимает что создаётся указатель? просто до этого момента для создания указателя необходимо было использовать `*`, а информации об автоматическом определении не было

[Ответить](#)



1. *Виталий:*
[7 декабря 2020 в 09:56](#)

`auto begin{ &data[0] }` это тоже что и `int *begin = &data[0];`

[Ответить](#)



2. *Максим:*
[11 сентября 2020 в 19:50](#)

В первом примере циклы `for` прописаны так:

```
1 // Цикл for с использованием явного индекса
2 for (index = 0; index < length; ++index)
3
4 // Цикл for с использованием указателей
5 for (auto ptr{ &data[0] }; ptr != (&data[0] + length); ++ptr)
```

и далее во всех примерах с указателями для определения завершения цикла используется оператор `"!="` (но не оператор `"<"`).

Подскажите, корректно ли в циклах `for` с указателями на массив использовать операторы `"<"`, `">"`? Или как-то неправильно сравнивать указатели друг с другом на предмет: кто больше, кто меньше?

На практике все работает.

Прикладываю код со сравнением указателей друг с другом в циклах:

```
1 // программа "обрезает" массив по краям
2
3 #include <iostream>
4 #include <array>
5 #include <iterator>
```



```

6
7 int main()
8 {
9     std::array array {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
10
11     std::cout << "Enter an edge number: ";
12     int edge;
13     std::cin >> edge;
14
15
16     std::cout << "\nfor:\n";
17
18     for (const *ptr {std::begin(array)}; ptr < std::end(array); ++ptr)
19     {
20         if(ptr >= (std::begin(array) + edge) && ptr < (std::end(array) - edge))
21             std::cout << *ptr << ' ';
22     }
23
24     std::cout << "\nforeach:\n";
25
26     for (const auto &element : array)
27     {
28         if (&element >= (array.begin() + edge) && &element < (array.end() - edge))
29             std::cout << element << ' ';
30     }
31     std::cout << '\n';
32
33     return 0;
34 }

```

Ответить



1. *Артурка:*

29 октября 2020 в 00:38

Так как в array используется random access iterator, то у него перегружены операторы типа:

1	<code>a < b</code>
2	<code>a > b</code>
3	<code>a <= b</code>
4	<code>a >= b</code>

но в том же std::forward_list таких перегрузок нету, но есть перегрузки типа:

1	<code>a == b</code>
2	<code>a != b</code>

По этому подход с последними операторами более универсальный.

Подробнее:

<http://www.cplusplus.com/reference/iterator/RandomAccessIterator/>

<http://www.cplusplus.com/reference/iterator/ForwardIterator/>

[Ответить](#)



3. *Александр:*

[6 мая 2020 в 16:53](#)

Что такое `std::size_t` и что такое `std::size`, где о них можно почитать? У вас в уроках нет никаких даже минимальных пояснений.

[Ответить](#)



1. *Steindvart:*

[10 мая 2020 в 15:03](#)

`size_t` — это универсально совместимый тип для хранения значений, которые как-либо связаны с адресацией в памяти (и, следуя из этого, с размерностью, индексацией массивов).

То есть, простыми словами говоря, это тип для беззнаковых целых чисел, которые связаны с адресацией памяти. Так как адрес не может быть:

1. Отрицательным
2. Дробным

Универсальный он потому, что его размер подстраивается под конкретную платформу. То есть размер адреса может быть разным на разных машинах: например, 32-битные и 64-битные системы. Где "битность" означает кол-во битов отведённые под представление адреса в памяти. Используя `size_t`, размер типа будет соответствовать размерам адреса платформы.

Как правило, он определяется в том или ином заголовочном файле в виде:

`typedef unsigned long size_t` или `typedef unsigned int size_t`, в зависимости от платформы.

`std::size()` — функция, которая возвращает размер контейнера или стандартного статического массива (так как его размер известен компилятору).

[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.

Отправить комментарий

[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