

Ravesli [Ravesli](#)


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

Урок №75. Фиксированные массивы

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 15 Сен 2020 |

 42305

[15](#)

Этот урок является продолжением предыдущего урока о [массивах в языке C++](#).

Оглавление:

1. [Инициализация фиксированных массивов](#)
2. [Длина массива](#)
3. [Массивы и перечисления](#)
4. [Массивы и классы enum](#)
5. [Передача массивов в функции](#)
6. [Оператор sizeof и массивы](#)
7. [Определение длины фиксированного массива](#)
8. [Индексирование массива вне диапазона](#)
9. [Тест](#)

Инициализация фиксированных массивов

Элементы массива обрабатываются так же, как и обычные переменные, поэтому они не инициализируются при создании. Одним из способов инициализации массива является присваивание значений каждому элементу поочерёдно:

```
1 int array[5]; // массив содержит 5 простых чисел
2 array[0] = 4;
3 array[1] = 5;
4 array[2] = 8;
5
```

```
6 | array[3] = 9;  
   | array[4] = 12;
```

Однако это не совсем удобно, особенно когда массив большой.

К счастью, язык C++ поддерживает более удобный способ инициализации массивов с помощью **списка инициализаторов**. Следующий пример эквивалентен примеру выше:

```
1 | int array[5] = { 4, 5, 8, 9, 12 }; // используется список инициализаторов для инициализации
```

Если в этом списке инициализаторов больше, чем может содержать массив, то компилятор выдаст ошибку.

Однако, если в списке инициализаторов меньше, чем может содержать массив, то остальные элементы будут проинициализированы значением 0. Например:

```
1 | #include <iostream>  
2 |  
3 | int main()  
4 | {  
5 |     int array[5] = { 5, 7, 9 }; // инициализируем только первые 3 элемента  
6 |  
7 |     std::cout << array[0] << '\n';  
8 |     std::cout << array[1] << '\n';  
9 |     std::cout << array[2] << '\n';  
10 |    std::cout << array[3] << '\n';  
11 |    std::cout << array[4] << '\n';  
12 |  
13 |    return 0;  
14 | }
```

Результат выполнения программы:

```
5  
7  
9  
0  
0
```

Следовательно, чтобы инициализировать все элементы массива значением 0, нужно:

```
1 | // Инициализируем все элементы массива значением 0  
2 | int array[5] = { };
```

В C++11 вместо этого мы можем воспользоваться синтаксисом **uniform-инициализации**:

```
1 | int array[5] { 4, 5, 8, 9, 12 }; // используем uniform-инициализацию для инициализации
```

Длина массива

Если вы инициализируете фиксированный массив с помощью списка инициализаторов, то компилятор может определить длину массива вместо вас, и вам уже не потребуется её объявлять.

Следующие две строки выполняют одно и то же:

```
1 int array[5] = { 0, 1, 2, 3, 4 }; // явно указываем длину массива
2 int array[] = { 0, 1, 2, 3, 4 }; // список инициализаторов автоматически определит длину
```

Это не только сэкономит время, но также вам не придется обновлять длину массива, если вы захотите добавить или удалить элементы позже.

Массивы и перечисления

Одна из основных проблем при использовании массивов состоит в том, что целочисленные индексы не предоставляют никакой информации программисту об их значении. Рассмотрим класс из 5 учеников:

```
1 const int numberOfStudents(5);
2 int testScores[numberOfStudents];
3 testScores[3] = 65;
```

Кто представлен элементом `testScores[3]`? Непонятно!

Это можно решить, используя перечисление, в котором перечислители сопоставляются каждому из возможных индексов массива:

```
1 enum StudentNames
2 {
3     SMITH, // 0
4     ANDREW, // 1
5     IVAN, // 2
6     JOHN, // 3
7     ANTON, // 4
8     MAX_STUDENTS // 5
9 };
10
11 int main()
12 {
13     int testScores[MAX_STUDENTS]; // всего 5 студентов
14     testScores[JOHN] = 65;
15
16     return 0;
17 }
```

Вот теперь понятно, что представляет собой каждый из элементов массива. Обратите внимание, добавлен дополнительный перечислитель с именем `MAX_STUDENTS`. Он используется во время объявления массива для гарантирования того, что массив имеет корректную длину (она должна быть на

единицу больше самого большого индекса). Это полезно как для подсчета элементов, так и для возможности автоматического изменения длины массива, если добавить еще один перечислитель:

```
1 enum StudentNames
2 {
3     SMITH, // 0
4     ANDREW, // 1
5     IVAN, // 2
6     JOHN, // 3
7     ANTON, // 4
8     MISHA, // 5
9     MAX_STUDENTS // 6
10 };
11
12 int main()
13 {
14     int testScores[MAX_STUDENTS]; // всего 6 студентов
15     testScores[JOHN] = 65; // всё работает
16
17     return 0;
18 }
```

Обратите внимание, этот трюк работает только в том случае, если вы не изменяете значения перечислителей вручную!

Массивы и классы enum

Классы enum не имеют неявного преобразования в целочисленный тип, поэтому, если вы попытаете сделать следующее:

```
1 enum class StudentNames
2 {
3     SMITH, // 0
4     ANDREW, // 1
5     IVAN, // 2
6     JOHN, // 3
7     ANTON, // 4
8     MISHA, // 5
9     MAX_STUDENTS // 6
10 };
11
12 int main()
13 {
14     int testScores[StudentNames::MAX_STUDENTS]; // всего 6 студентов
15     testScores[StudentNames::JOHN] = 65;
16 }
```

То получите ошибку от компилятора. Это можно решить, используя [оператор static_cast](#) для конвертации перечислителя в целое число:

```
1 int main()
2 {
3     int testScores[static_cast<int>(StudentNames::MAX_STUDENTS)]; // всего 6 студентов
4     testScores[static_cast<int>(StudentNames::JOHN)] = 65;
5 }
```

Однако, это также не очень удобно, поэтому лучше использовать стандартное перечисление внутри [пространства имен](#):

```
1 namespace StudentNames
2 {
3     enum StudentNames
4     {
5         SMITH, // 0
6         ANDREW, // 1
7         IVAN, // 2
8         JOHN, // 3
9         ANTON, // 4
10        MISHA, // 5
11        MAX_STUDENTS // 6
12    };
13 }
14
15 int main()
16 {
17     int testScores[StudentNames::MAX_STUDENTS]; // всего 6 студентов
18     testScores[StudentNames::JOHN] = 65;
19 }
```

Передача массивов в функции

Хотя передача массива в функцию на первый взгляд выглядит так же, как передача обычной переменной, но «под капотом» C++ обрабатывает массивы несколько иначе.

Когда обычная переменная передается по значению, то C++ копирует значение аргумента в параметр функции. Поскольку параметр является копией, то изменение значения параметра не изменяет значение исходного аргумента.

Однако, поскольку копирование больших массивов — дело трудоёмкое, то C++ не копирует массив при его передаче в функцию. Вместо этого передается фактический массив. И здесь мы получаем побочный эффект, позволяющий функциям напрямую изменять значения элементов массива!

Следующий пример хорошо иллюстрирует эту концепцию:

```
1 #include <iostream>
2
```

```
3 void passValue(int value) // здесь value - это копия аргумента
4 {
5     value = 87; // изменения value здесь не повлияют на фактическую переменную value
6 }
7
8 void passArray(int array[5]) // здесь array - это фактический массив
9 {
10     array[0] = 10; // изменения array здесь изменят исходный массив array
11     array[1] = 8;
12     array[2] = 6;
13     array[3] = 4;
14     array[4] = 1;
15 }
16
17 int main()
18 {
19     int value = 1;
20     std::cout << "before passValue: " << value << "\n";
21     passValue(value);
22     std::cout << "after passValue: " << value << "\n";
23
24     int array[5] = { 1, 4, 6, 8, 10 };
25     std::cout << "before passArray: " << array[0] << " " << array[1] << " " << array[2] << " " << array[3] << " " << array[4] << "\n";
26     passArray(array);
27     std::cout << "after passArray: " << array[0] << " " << array[1] << " " << array[2] << " " << array[3] << " " << array[4] << "\n";
28
29     return 0;
30 }
```

Результат выполнения программы:

```
before passValue: 1
after passValue: 1
before passArray: 1 4 6 8 10
after passArray: 10 8 6 4 1
```

В примере, приведенном выше, значение переменной `value` не изменяется в функции `main()`, так как параметр `value` в функции `passValue()` был лишь копией фактической переменной `value`. Однако, поскольку массив в параметре функции `passArray()` является фактическим массивом, то `passArray()` напрямую изменяет значения его элементов!

Примечание: Если вы не хотите, чтобы функция изменяла значения элементов массива, переданного в нее в качестве параметра, то нужно сделать массив константным:

```
1 // Даже если array является фактическим массивом, внутри этой функции он должен рассма
2 void passArray(const int array[5])
3 {
4     // Поэтому каждая из следующих строк вызовет ошибку компиляции!
5     array[0] = 11;
```

```
6 |     array[1] = 7;  
7 |     array[2] = 5;  
8 |     array[3] = 3;  
9 |     array[4] = 2;  
10| }
```

Оператор sizeof и массивы

Оператор sizeof можно использовать и с массивами: он возвращает общий размер массива (длина массива умножена на размер одного элемента) в байтах. Обратите внимание, из-за того, как C++ передает массивы в функции, следующая операция не будет корректно выполнена с массивами, переданными в функции:

```
1 | #include <iostream>  
2 |  
3 | void printSize(int array[])  
4 | {  
5 |     std::cout << sizeof(array) << '\n'; // выводится размер указателя (об этом поговорим позже)  
6 | }  
7 |  
8 | int main()  
9 | {  
10|     int array[] = { 1, 3, 3, 4, 5, 9, 14, 17 };  
11|     std::cout << sizeof(array) << '\n'; // выводится размер массива  
12|     printSize(array);  
13|  
14|     return 0;  
15| }
```

Результат выполнения программы:

```
32  
4
```

По этой причине будьте осторожны при использовании оператора sizeof с массивами!

Определение длины фиксированного массива

Чтобы определить длину фиксированного массива, поделите размер всего массива на размер одного элемента массива:

```
1 | #include <iostream>  
2 |  
3 | int main()  
4 | {  
5 |     int array[] = { 1, 3, 3, 4, 5, 9, 14, 17 };  
6 | }
```

```
6 | std::cout << "The array has: " << sizeof(array) / sizeof(array[0]) << " elements\n"
7 |
8 | return 0;
9 | }
```

Результат выполнения программы:

The array has 8 elements

Как это работает? Во-первых, размер всего массива равен длине массива, умноженной на размер одного элемента. Формула: размер массива = длина массива * размер одного элемента.

Используя алгебру, мы можем изменить это уравнение: длина массива = размер массива / размер одного элемента. `sizeof(array)` — это размер массива, а `sizeof(array[0])` — это размер одного элемента массива. Соответственно, длина массива = `sizeof(array) / sizeof(array[0])`. Обычно используется нулевой элемент в качестве элемента массива в уравнении, так как только он является единственным элементом, который гарантированно существует в массиве, независимо от его длины.

Это работает только если массив фиксированной длины, и вы выполняете эту операцию в той же функции, в которой объявлен массив (мы поговорим больше об этом ограничении на следующих уроках).

На следующих уроках мы будем использовать термин «длина» для обозначения общего количества элементов в массиве, и термин «размер», когда речь будет идти о байтах.

Индексирование массива вне диапазона

Помните, что массив длиной N содержит элементы от 0 до N-1. Итак, что произойдет, если мы попытаемся получить доступ к индексу массива за пределами этого диапазона? Рассмотрим следующую программу:

```
1 | int main()
2 | {
3 |     int array[5]; // массив содержит 5 простых чисел
4 |     array[5] = 14;
5 |
6 |     return 0;
7 | }
```

Здесь наш массив имеет длину 5, но мы пытаемся записать значение в 6-й элемент (индекс 5).

Язык C++ не выполняет никаких проверок корректности вашего индекса. Таким образом, в вышеприведенном примере значение 14 будет помещено в ячейку памяти, где 6-й элемент существовал бы (если бы вообще был). Но, как вы уже догадались, это будет иметь свои последствия. Например, произойдет перезаписывание значения другой переменной или вообще сбой программы.

Хотя это происходит реже, но C++ также позволяет использовать отрицательный индекс, что тоже приведет к нежелательным результатам.

Правило: При использовании массивов убедитесь, что ваши индексы корректны и соответствуют диапазону вашего массива.

Тест

Задание №1

Объявите массив для хранения температуры (дробное число) каждого дня в году (всего 365 дней). Инициализируйте массив значением `0.0` для каждого дня.

Ответ №1

Примечание: Если размер не является ограничением, то вместо типа `float` лучше использовать тип `double`.

```
1 double temperature[365] = { 0.0 };
```

Задание №2

Создайте перечисление со следующими перечислителями: `chicken`, `lion`, `giraffe`, `elephant`, `duck` и `snake`. Поместите перечисление в пространство имен. Объявите массив, где элементами будут эти перечислители и, используя список инициализаторов, инициализируйте каждый элемент соответствующим количеством лап определенного животного. В функции `main()` выведите количество ног у слона, используя перечислитель.

Ответ №2

```
1 #include <iostream>
2
3 namespace Animals
4 {
5     enum Animals
6     {
7         CHICKEN,
8         LION,
9         GIRAFFE,
10        ELEPHANT,
11        DUCK,
12        SNAKE,
13        MAX_ANIMALS
14    };
15 }
16
17 int main()
18 {
19     int legs[Animals::MAX_ANIMALS] = { 2, 4, 4, 4, 2, 0 };
20
21     std::cout << "An elephant has " << legs[Animals::ELEPHANT] << " legs.\n";
22
23     return 0;
```

Оценить статью:

★★★★★ (238 оценок, среднее: 4,94 из 5)



[← Урок №74. Массивы](#)



[Урок №76. Массивы и циклы →](#)

Комментариев: 15



1. *Vicktoras:*

[24 июля 2020 в 17:02](#)

В первом задании написано: Объявите массив для хранения температуры (дробное число) каждого дня в году (всего 365 дней).

Но в ответе на первое задание указано 366 дней: `double temperature[365]`
Вед в уроке говорилось что массив начинается с нуля. Или я что-то не понял?

[Ответить](#)



1. *Павел:*

[27 июля 2020 в 19:46](#)

При объявлении массива в квадратных скобках указывается общее кол-во его ячеек, а вот уже сами ячейки нумеруются с нуля

[Ответить](#)



2. *Борис:*

[8 мая 2020 в 16:54](#)

По поводу выхода за границы массива:

- 1) В этом случае, как я понимаю, процесс может испортить только свою память? Порча (или чтение) памяти других процессов тут невозможно? Иначе это просто катастрофа.
- 2) Если (1) верно — это обеспечено самой операционкой (защищённая модель памяти и т. д.) или чем-то ещё? И будет ли безопасно на других ОС (не винда) и платформах?

В сети ясного ответа на это не нашёл.

[Ответить](#)



1. *Артём:*

[1 июля 2020 в 22:57](#)

Насколько я помню из видеокурсов по углубленному изучению языка C, то выделение памяти отдаётся на откуп ОС.

Любая программа (компилятор тоже программа) при запуске запрашивает у ОС память. Как правило, ОС выделяют не блок памяти под всю программу, а участками, поэтому ни о какой защищенной модели памяти здесь речи не идёт.

Если ваш выход за массив залез в вашу же программу — вам повезло.

[Ответить](#)

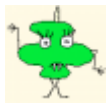


3. *Алексей:*

[26 июля 2019 в 16:56](#)

После всего — второе задание оказалось простым в исполнении, только текстом не приукрасил.

[Ответить](#)



4. *Алексей:*

[26 июля 2019 в 16:46](#)

Немного не понял первое — почему `double temp[365] = {0.0}`.

Я использовал `double temp[365] = { }`.

В том и ином случаи "0".

[Ответить](#)



5. *Анна:*

[27 февраля 2019 в 23:07](#)

Не подскажите, зачем нужно писать

`= {0.0}`, если все элементы по умолчанию, без присваивания значений равны 0.

нельзя ли просто написать

`double god[365];` например?

или это будет неправильно?

[Ответить](#)



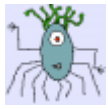
6. *Oleksiy:*

[27 августа 2018 в 11:26](#)

"Трюк: Чтобы определить длину фиксированного массива — разделите размер всего массива на размер одного элемента массива:"

В русском языке принято употреблять слово "размер" массива, когда речь идет о количестве элементов в нем? Разве "длина" в этом случае не будет более уместным термином? Размер больше ассоциируется с количеством байт.

[Ответить](#)



7. *Алексей:*

[28 марта 2018 в 15:54](#)

Вместо "трюка" — уравнения с `sizeof`, на мой взгляд, гораздо удобнее использовать более компактную встроенную функцию:

```
1 | std::size(array);
```

Она возвращает количество элементов в контейнере (массивы, `vector`, `string`)
Должна подключаться отдельным заголовком

```
1 | #include <iterator>
```

но в Visual Studio 2017 работает уже при

```
1 | #include<iostream>
```

[Ответить](#)



1. *Юрий:*

[29 марта 2018 в 18:27](#)

Да, можно использовать уже встроенную функцию. Встроенная функция даже удобнее, нежели трюк, указанный в статье. Но для общего развития знать не помешает.

[Ответить](#)



2. *ц3к32к:*

[7 мая 2018 в 20:46](#)

Нет, чувак, тогда выведется предупреждение: `<`: несоответствие типов со знаком и без знака

[Ответить](#)



8. *Антон:*

[4 марта 2018 в 11:47](#)

"Тест 1. Инициализируйте массив значением 0.0 для каждого дня."

```
"double temperature[365] = { 0.0 } ;"
```

Так ведь инициализируется только `temperature[0]`, значением 0.0. Остальные тоже будут 0, но не потому что мы так захотели. Нужно ведь через
`for (int count = 0; count < 365; ++count)`

```
temperature[count] = 0.0;  
ну либо double temperature[365] = { 0.0, 0.0, ...362раза..., 0.0 };  
или я ошибаюсь?
```

[Ответить](#)1. *Юрий:*[4 марта 2018 в 11:57](#)

Зачем использовать цикл или 365 раз писать 0.0, если массив инициализируется просто одним лишь { 0.0 } или {}. Есть действия, которые установлены уже по умолчанию, просто так прописывать то, что установлено по умолчанию — зачем? Вы просто напишите лишний код. В этом задании никакой программист писать 365 раз нули не будет, цикл здесь также не уместен. Другое дело уже, если нужно заполнить массив ненулевыми значениями.

[Ответить](#)1. *Антон:*[4 марта 2018 в 12:02](#)

Так я думал задание и состоит в том, чтобы каждому значению массива присвоить число, просто для примера взял 0 и так уж совпало, что это "значение по умолчанию" для элементов массива. Меня смутила фраза в задании "для каждого дня".

[Ответить](#)1. *Юрий:*[4 марта 2018 в 18:09](#)

В задании говорится именно о 0. Если бы было другое значение, то тогда бы уже использовался цикл.

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