

Ravesli [Ravesli](#)


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


Урок №99. Передача по адресу

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 21 Сен 2020 |

 36227

[1](#)  15

Есть еще один способ передачи переменных в функцию в языке C++ — по адресу.

Оглавление:

1. [Передача по адресу](#)
2. [Передача по константному адресу](#)
3. [Передача адресов по ссылке](#)
4. [Существует только передача по значению](#)
5. [Плюсы и минусы передачи по адресу](#)

Передача по адресу

Передача аргументов по адресу — это передача адреса переменной-аргумента (а не исходной переменной). Поскольку аргумент является адресом, то параметром функции должен быть **указатель**. Затем функция сможет разыменовать этот указатель для доступа или изменения исходного значения. Вот пример функции, которая принимает параметр, передаваемый по адресу:

```
1 #include <iostream>
2
3 void boo(int *ptr)
4 {
5     *ptr = 7;
6 }
7
8 int main()
```

```
9 | {
10 |     int value = 4;
11 |
12 |     std::cout << "value = " << value << '\n';
13 |     boo(&value);
14 |     std::cout << "value = " << value << '\n';
15 |     return 0;
16 | }
```

Результат выполнения программы:

```
value = 4
value = 7
```

Как вы можете видеть, функция `boo()` изменила значение аргумента (переменную `value`) через параметр-указатель `ptr`. Передачу по адресу обычно используют с указателями на обычные массивы. Например, следующая функция выведет все значения массива:

```
1 | void printArray(int *array, int length)
2 | {
3 |     for (int index=0; index < length; ++index)
4 |         std::cout << array[index] << ' ';
5 | }
```

Вот пример программы, которая вызывает эту функцию:

```
1 | int main()
2 | {
3 |     int array[7] = { 9, 8, 6, 4, 3, 2, 1 }; // помните, что массивы распадаются в указатели
4 |     printArray(array, 7); // поэтому здесь array - это указатель на первый элемент массива
5 | }
```

Результат:

```
9 8 6 4 3 2 1
```

Помните, что фиксированные массивы распадаются в указатели при передаче в функцию, поэтому их длину нужно передавать в виде отдельного параметра. Перед разыменованием параметров, передаваемых по адресу, не лишним будет проверить — не являются ли они нулевыми указателями. Разыменование нулевого указателя приведет к сбою в программе. Вот функция `printArray()` с проверкой (обнаружением) нулевых указателей:

```
1 | #include <iostream>
2 |
3 | void printArray(int *array, int length)
4 | {
5 |     // Если пользователь передал нулевой указатель в качестве array
6 |     if (!array)
7 |         return;
8 | }
```

```
9   for (int index=0; index < length; ++index)
10      std::cout << array[index] << ' ';
11 }
12
13 int main()
14 {
15     int array[7] = { 9, 8, 6, 4, 3, 2, 1 };
16     printArray(array, 7);
17 }
```

Передача по константному адресу

Поскольку `printArray()` все равно не изменяет значения получаемых аргументов, то хорошей идеей будет сделать параметр `array` константным:

```
1  #include <iostream>
2
3  void printArray(const int *array, int length)
4  {
5      // Если пользователь передал нулевой указатель в качестве array
6      if (!array)
7          return;
8
9      for (int index=0; index < length; ++index)
10         std::cout << array[index] << ' ';
11 }
12
13 int main()
14 {
15     int array[7] = { 9, 8, 6, 4, 3, 2, 1 };
16     printArray(array, 7);
17 }
```

Так мы видим сразу, что `printArray()` не изменит переданный аргумент `array`. Когда вы передаете указатель в функцию по адресу, то значение этого указателя (адрес, на который он указывает) копируется из аргумента в параметр функции. Другими словами, он передается по значению! Если изменить значение параметра функции, то изменится только копия, исходный указатель-аргумент не будет изменен. Например:

```
1  #include <iostream>
2
3  void setToNull(int *tempPtr)
4  {
5      // Мы присваиваем tempPtr другое значение (мы не изменяем значение, на которое ука
6      tempPtr = nullptr; // используйте 0, если не поддерживается C++11
7  }
8
```

```
9 int main()
10 {
11     // Сначала мы присваиваем ptr адрес six, т.е. *ptr = 6
12     int six = 6;
13     int *ptr = &six;
14
15     // Здесь выведется 6
16     std::cout << *ptr << "\n";
17
18     // tempPtr получит копию ptr
19     setToNull(ptr);
20
21     // ptr до сих пор указывает на переменную six!
22
23     // Здесь выведется 6
24     if (ptr)
25         std::cout << *ptr << "\n";
26     else
27         std::cout << " ptr is null";
28
29     return 0;
30 }
```

В tempPtr копируется адрес указателя ptr. Несмотря на то, что мы изменили tempPtr на нулевой указатель (присвоили ему nullptr), это никак не повлияло на значение, на которое указывает ptr. Следовательно, результат выполнения программы:

6
6

Обратите внимание, хотя сам адрес передается по значению, вы все равно можете разыменовать его для изменения значения исходного аргумента. Запутано? Давайте проясним:

- ➔ При передаче аргумента по адресу в переменную-параметр функции копируется адрес из аргумента. В этот момент параметр функции и аргумент указывают на одно и то же значение.
- ➔ Если параметр функции затем разыменовать для изменения исходного значения, то это приведет к изменению значения, на которое указывает аргумент, поскольку параметр функции и аргумент указывают на одно и то же значение!
- ➔ Если параметру функции присвоить другой адрес, то это никак не повлияет на аргумент, поскольку параметр функции является копией, а изменение копии не приводит к изменению оригинала. После изменения адреса параметра функции, параметр функции и аргумент будут указывать на разные значения, поэтому разыменование параметра и дальнейшее его изменение никак не повлияют на значение, на которое указывает аргумент.

В следующей программе это всё хорошо проиллюстрировано:

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

```
3 void setToSeven(int *tempPtr)
4 {
5     *tempPtr = 7; // мы изменяем значение, на которое указывает tempPtr (и ptr тоже)
6 }
7
8 int main()
9 {
10     // Сначала мы присваиваем ptr адрес six, т.е. *ptr = 6
11     int six = 6;
12     int *ptr = &six;
13
14     // Здесь выведется 6
15     std::cout << *ptr << "\n";
16
17     // tempPtr получит копию ptr
18     setToSeven(ptr);
19
20     // tempPtr изменил значение, на которое указывал, на 7
21
22     // Здесь выведется 7
23     if (ptr)
24         std::cout << *ptr << "\n";
25     else
26         std::cout << " ptr is null";
27
28     return 0;
29 }
```

Результат выполнения программы:

```
6
7
```

Передача адресов по ссылке

Следует вопрос: «А что, если мы хотим изменить адрес, на который указывает аргумент, внутри функции?». Оказывается, это можно сделать очень легко. Вы можете просто передать адрес по ссылке. Синтаксис ссылки на указатель может показаться немного странным, но все же:

```
1 #include <iostream>
2
3 // tempPtr теперь является ссылкой на указатель, поэтому любые изменения tempPtr приве-
4 void setToNull(int *&tempPtr)
5 {
6     tempPtr = nullptr; // используйте 0, если не поддерживается C++11
7 }
8
9 int main()
```

```
10 {
11     // Сначала мы присваиваем ptr адрес six, т.е. *ptr = 6
12     int six = 6;
13     int *ptr = &six;
14
15     // Здесь выведется 6
16     std::cout << *ptr;
17
18     // tempPtr является ссылкой на ptr
19     setToNull(ptr);
20
21     // ptr было присвоено значение nullptr!
22
23     if (ptr)
24         std::cout << *ptr;
25     else
26         std::cout << " ptr is null";
27
28     return 0;
29 }
```

Результат выполнения программы:

6 ptr is null

Наконец, наша функция setToNull() действительно изменила значение ptr с &six на nullptr!

Существует только передача по значению

Теперь, когда вы понимаете основные различия между передачей по ссылке, по адресу и по значению, давайте немного поговорим о том, что находится «под капотом».

На уроке о [ссылках](#) мы упоминали, что ссылки на самом деле реализуются с помощью указателей. Это означает, что передача по ссылке является просто передачей по адресу. И чуть выше мы говорили, что передача по адресу на самом деле является передачей адреса по значению! Из этого следует, что C++ действительно передает всё по значению!

Плюсы и минусы передачи по адресу

Плюсы передачи по адресу:

- ✚ Передача по адресу позволяет функции изменить значение аргумента, что иногда полезно. В противном случае, используем const для гарантии того, что функция не изменит аргумент.
- ✚ Поскольку копирования аргументов не происходит, то скорость передачи по адресу достаточно высокая, даже если передавать большие [структуры](#) или классы.

✚ Мы можем вернуть сразу несколько значений из функции, используя [параметры вывода](#).

Минусы передачи по адресу:

- Все указатели нужно проверять, не являются ли они нулевыми. Попытка разыменовать нулевой указатель приведет к сбою в программе.
- Поскольку разыменовывание указателя выполняется медленнее, чем доступ к значению напрямую, то доступ к аргументам, переданным по адресу, выполняется также медленнее, чем доступ к аргументам, переданным по значению.

Когда использовать передачу по адресу:

- ➔ при передаче обычных массивов (если нет никаких проблем с тем, что массивы распадаются в указатели при передаче).

Когда не использовать передачу по адресу:

- ➔ при передаче структур или классов (используйте передачу по ссылке);
- ➔ при передаче фундаментальных типов данных (используйте передачу по значению).

Как вы можете видеть сами, передача по адресу и по ссылке имеют почти одинаковые преимущества и недостатки. Поскольку передача по ссылке обычно безопаснее, чем передача по адресу, то в большинстве случаев предпочтительнее использовать передачу по ссылке.

Правило: Используйте передачу по ссылке, вместо передачи по адресу, когда это возможно.

Оценить статью:

★★★★★ (205 оценок, среднее: 4,97 из 5)



← [Урок №98. Передача по ссылке](#)

[Урок №100. Возврат значений по ссылке, по адресу и по значению](#) ➔



Комментариев: 15



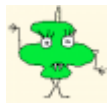
1. *Старый программист:*
[29 ноября 2019 в 14:50](#)

Цитата: Поскольку разыменование указателя выполняется медленнее, чем доступ к значению напрямую,

Это как?

"разыменование указателя" — это не более чем абстракция языка. Никакого машинного кода не генерируется. Как это может быть быстрее? Не думаю, что прямая адресация хоть как-то быстрее косвенно-регистровой. И та и другая команда выполняется за одинаковое количество тактов. Если смотреть трансляцию на RISC конвейер, то разницы найти не возможно, стадия MEM в обоих случаях выполняется за фиксированный такт.

[Ответить](#)



2. *Алексей:*

[22 августа 2019 в 17:34](#)

Немного запутали, но на практике вопросы уйдут.

[Ответить](#)



3. *Дмитрий:*

[15 июня 2019 в 23:23](#)

Я вот честно говоря тоже запутался как нужно передавать массив. В главе Урок №75 написано, что массив и без амперсанта и без указателя и так передается в функцию без копирования. Здесь написано, что нужно через адрес. При этом структуры и классы лучше через ссылку. Чем массив принципиально отличается от структуры? Не понятно, как правильно

[Ответить](#)



1. *Егор:*

[20 августа 2019 в 11:39](#)

Насколько я понял, фиксированный массив при передаче в функцию без ничего, приходит в неё в виде указателя на первый элемент, (ничего не копируется). Если же ты передаешь `std::array` или `std::vector`, то без ничего они полностью скопируются в аргумент функции (что затратно)

Поэтому лучше использовать адреса. А еще лучше ссылки.

[Ответить](#)



4. *Torgu:*

[13 июля 2018 в 09:32](#)

Так и не понял, зачем передавать массивы по адресу, когда безопаснее по ссылке?

[Ответить](#)



1. *Анастасия:*
[15 августа 2018 в 16:37](#)

Потому что массив при передаче в функцию распадается в указатель.

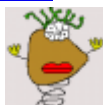
[Ответить](#)



1. *Torgu:*
[15 августа 2018 в 20:07](#)

так в чем собственно преимущество?

[Ответить](#)



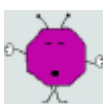
1. *Анастасия:*
[16 августа 2018 в 08:35](#)

Даже если передавать по ссылке, все равно массив распадется в указатель. Так в чем преимущество передачи массива по ссылке?



2. *Torgu:*
[17 августа 2018 в 09:28](#)

Здесь ограничение по глубине комментариев, поэтому могу ответить только себе. Итак, вы отвечаете вопросом на вопрос? То есть у указателей все-таки нет преимуществ перед ссылками?



2. *Александр:*
[24 февраля 2019 в 16:12](#)

по ссылке на что Вы собираетесь передавать массив?

Я так понимаю, что немного дальше автор нас будет знакомить с итераторами. Так вот, ссылка для массива — это простой итератор. И передача массива по ссылке унифицирует интерфейсы работы функций с различными контейнерами. Если Вы умеете обрабатывать массив, жонглируя исключительно ссылками (при обращении к элементам пользуетесь разыменованием, а не оператором `[]`), то в большинстве случаев можете тот же код применить и к другим контейнерам.

Кстати, желательно передавать в функцию не "указатель на начало массива + размер", а "указатель на начало и указатель на конец" (точнее, на элемент, который идет после последнего). Поначалу будет казаться, что это ненужное усложнение, но на самом деле так намного проще... да и многие стандартные функции обработки контейнеров принимают именно такой тип аргументов.

[Ответить](#)5. *Shom:*[10 мая 2018 в 13:48](#)

Здравствуйте! Извините за тупость, но вот тут:

```
1 void printArray(int *array, int length)
2 {
3     for (int index=0; index < length; ++index)
4         std::cout << array[index] << ' ';
5 }
```

мне не совсем понятно `array[index]`.

Ведь `array` — это указатель на первый элемент массива, то-есть, фактически, это адрес первого элемента и как к этому адресу пристыкуется `[index]`?

При следующих итерациях цикла, по моей логике, должен вновь получиться адрес первого элемента, но уже со следующим индексом в квадратных скобках.

[Ответить](#)1. *Юрий:*[10 мая 2018 в 17:24](#)

Привет.

`array` — это тип указателя на массив (`int *`), а `array[index]` — это уже набор значений определенного типа, например, `int`-ов. Определив `array[4]` и используя возле `array` квадратные скобки со значением внутри вы автоматически обращаетесь к `array`, как к набору `int`-ов, а не как к указателю на первый элемент. Не указывая квадратных скобок и значения в них вы можете относиться к `array` как к указателю на массив (на первый элемент).

Т.е. если вы добавляете `[index]` к `array`, то `array` используется в этом случае как тип данных (`int`), а не как указатель (`int *`). И при последующих итерациях цикла у вас получается не адрес первого элемента + следующий индекс в квадратных скобках, а адрес 2-ого значения из набора `int`-ов (т.е. из массива).

[Ответить](#)1. *Shom:*[10 мая 2018 в 18:05](#)

Всё понял, спасибо!

Но возник ещё один вопрос: память для массива всегда одним целым куском выделяется? Не может быть так, к примеру, что сначала идут первые сто адресов для массива, потом адрес памяти выделен под переменную (массив инициализировали позже переменной), а потом следующие адреса продолжают массив и при инструкции в цикле

```
1 | cout << *array++ << ", ";
```

эта переменная попадёт в вывод?

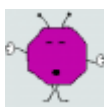
[Ответить](#)



1. *Юрий:*

[13 мая 2018 в 20:07](#)

Насколько я понял ваш вопрос, то память для массивов выделяется либо сразу (фиксированные массивы), либо по мере увеличения количества элементов (динамическое выделение памяти для динамических массивов). Может ли выделение памяти прерываться, как описали вы — не знаю.



2. *Александр:*

[24 февраля 2019 в 16:14](#)

при выделении памяти под массив (в том числе и под динамический) адреса элементов ГАРАНТИРОВАНО идут по порядку

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