

Ravesli [Ravesli](#)


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


## Урок №100. Возврат значений по ссылке, по адресу и по значению

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 5 Окт 2020 |

 45958

[16](#)

На 3-х предыдущих уроках мы узнали о передаче аргументов в функции [по значению](#), [по ссылке](#) и [по адресу](#). На этом уроке мы рассмотрим возврат значений обратно из функции в вызывающий объект всеми этими тремя способами.

Возврат значений (с помощью оператора return) работает почти так же, как и передача значений в функцию. Все те же плюсы и минусы. Основное отличие состоит в том, что поток данных движется уже в противоположную сторону. Однако здесь есть еще один нюанс — [локальные переменные](#), которые выходят из [области видимости](#) и уничтожаются, когда функция завершает свое выполнение.

Оглавление:

1. [Возврат по значению](#)
2. [Возврат по адресу](#)
3. [Возврат по ссылке](#)
4. [Смешивание возвращаемых значений и ссылок](#)
5. [Заключение](#)
6. [Тест](#)

### Возврат по значению

**Возврат по значению** — это самый простой и безопасный тип возврата. При возврате по значению, копия возвращаемого значения передается обратно в caller. Как и в случае с передачей по значению, вы можете возвращать [литералы](#) (например, 7), переменные (например, x) или выражения (например, x + 2), что делает этот способ очень гибким.

Еще одним преимуществом является то, что вы можете возвращать переменные (или выражения), в вычислении которых задействованы и локальные переменные, объявленные в теле самой функции. При этом, можно не беспокоиться о проблемах, которые могут возникнуть с областью видимости. Поскольку переменные вычисляются до того, как функция производит возврат значения, то здесь не должно быть никаких проблем с областью видимости этих переменных, когда заканчивается блок, в котором они объявлены. Например:

```
1 int doubleValue(int a)
2 {
3     int value = a * 3;
4     return value; // копия value возвращается здесь
5 } // value выходит из области видимости здесь
```

Возврат по значению идеально подходит для возврата переменных, которые были объявлены внутри функции, или для возврата аргументов функции, которые были переданы по значению. Однако, подобно передаче по значению, возврат по значению медленный при работе со [структурами](#) и классами.

**Когда использовать возврат по значению:**

- при возврате переменных, которые были объявлены внутри функции;
- при возврате аргументов функции, которые были переданы в функцию по значению.

**Когда не использовать возврат по значению:**

- при возврате [стандартных массивов](#) или [указателей](#) (используйте возврат по адресу);
- при возврате больших структур или классов (используйте возврат по ссылке).

## Возврат по адресу

**Возврат по адресу** — это возврат адреса переменной обратно в caller. Подобно передаче по адресу, возврат по адресу может возвращать только адрес переменной. Литералы и выражения возвращать нельзя, так как они не имеют адресов. Поскольку при возврате по адресу просто копируется адрес из функции в caller, то этот процесс также очень быстрый.

Тем не менее, этот способ имеет один недостаток, который отсутствует при возврате по значению: если вы попытаетесь вернуть адрес локальной переменной, то получите неожиданные результаты. Например:

```
1 int* doubleValue(int a)
2 {
3     int value = a * 3;
4     return &value; // value возвращается по адресу здесь
5 } // value уничтожается здесь
```

Как вы можете видеть, `value` уничтожается сразу после того, как её адрес возвращается в caller. Конечным результатом будет то, что caller получит адрес освобожденной памяти ([висячий указатель](#)),

что, несомненно, вызовет проблемы. Это одна из самых распространенных ошибок, которую делают новички. Большинство современных компиляторов выдадут предупреждение (а не ошибку), если программист попытается вернуть локальную переменную по адресу. Однако есть несколько способов обмануть компилятор, чтобы сделать что-то «плохое», не генерируя при этом предупреждения, поэтому вся ответственность лежит на программисте, который должен гарантировать, что возвращаемый адрес будет корректен.

Возврат по адресу часто используется для возврата динамически выделенной памяти обратно в caller:

```
1 int* allocateArray(int size)
2 {
3     return new int[size];
4 }
5
6 int main()
7 {
8     int *array = allocateArray(20);
9
10    // Делаем что-нибудь с array
11
12    delete[] array;
13    return 0;
14 }
```

Здесь не возникнет никаких проблем, так как динамически выделенная память не выходит из области видимости в конце блока, в котором объявлена, и все еще будет существовать, когда адрес будет возвращаться в caller.

**Когда использовать возврат по адресу:**

- при возврате динамически выделенной памяти;
- при возврате аргументов функции, которые были переданы по адресу.

**Когда не использовать возврат по адресу:**

- при возврате переменных, которые были объявлены внутри функции (используйте возврат по значению);
- при возврате большой структуры или класса, который был передан по ссылке (используйте возврат по ссылке).

## Возврат по ссылке

Подобно передаче по ссылке, значения, возвращаемые по ссылке, должны быть переменными (вы не сможете вернуть ссылку на литерал или выражение). При **возврате по ссылке** в caller возвращается ссылка на переменную. Затем caller может её использовать для продолжения изменения переменной, что может быть иногда полезно. Этот способ также очень быстрый и при возврате больших структур или классов.

Однако, как и при возврате по адресу, вы не должны возвращать локальные переменные по ссылке. Рассмотрим следующий фрагмент кода:

```
1 int& doubleValue(int a)
2 {
3     int value = a * 3;
4     return value; // value возвращается по ссылке здесь
5 } // value уничтожается здесь
```

В программе, приведенной выше, возвращается ссылка на переменную `value`, которая уничтожится, когда функция завершит свое выполнение. Это означает, что `caller` получит ссылку на мусор. К счастью, ваш компилятор, вероятнее всего, выдаст предупреждение или ошибку, если вы попытаетесь это сделать.

Возврат по ссылке обычно используется для возврата аргументов, переданных в функцию по ссылке. В следующем примере мы возвращаем (по ссылке) элемент массива, который был передан в функцию по ссылке:

```
1 #include <iostream>
2 #include <array>
3
4 // Возвращаем ссылку на элемент массива по индексу index
5 int& getElement(std::array<int, 20> &array, int index)
6 {
7     // Мы знаем, что array[index] не уничтожится, когда мы будем возвращать данные в C
8     // Так что здесь не должно быть никаких проблем с возвратом по ссылке
9     return array[index];
10 }
11
12 int main()
13 {
14     std::array<int, 20> array;
15
16     // Присваиваем элементу массива под индексом 15 значение 7
17     getElement(array, 15) = 7;
18
19     std::cout << array[15] << '\n';
20
21     return 0;
22 }
```

Результат выполнения программы:

7

Когда мы вызываем `getElement(array, 15)`, то `getElement()` возвращает ссылку на элемент массива под индексом 15, а затем `main()` использует эту ссылку для присваивания этому элементу значения 7.

Хотя этот пример непрактичен, так как мы можем напрямую обратиться к 15 элементу массива, но как только мы будем рассматривать классы, то вы обнаружите гораздо больше применений для возврата

значений по ссылке.

### Когда использовать возврат по ссылке:

- при возврате ссылки-параметра;
- при возврате элемента массива, который был передан в функцию;
- при возврате большой структуры или класса, который не уничтожается в конце функции (например, тот, который был передан в функцию).

### Когда не использовать возврат по ссылке:

- при возврате переменных, которые были объявлены внутри функции (используйте возврат по значению);
- при возврате стандартного массива или значения указателя (используйте возврат по адресу).

## Смешивание возвращаемых значений и ссылок

Хотя функция может возвращать как значение, так и ссылку, caller может неправильно это интерпретировать. Посмотрим, что произойдет при смешивании возвращаемых значений и ссылок на значения:

```
1  int returnByValue()  
2  {  
3      return 7;  
4  }  
5  
6  int& returnByReference()  
7  {  
8      static int y = 7; // static гарантирует то, что переменная y не уничтожится, когда  
9      return y;  
10 }  
11  
12 int main()  
13 {  
14     int value = returnByReference(); // случай A: всё хорошо, обрабатывается как возвращаемое значение  
15     int &ref = returnByValue(); // случай B: ошибка компилятора, так как 7 - это r-value  
16     const int &cref = returnByValue(); // случай C: всё хорошо, время жизни возвращаемого значения  
17 }
```

В случае A мы присваиваем ссылке возвращаемого значения переменную, которая сама не является ссылкой. Поскольку `value` не является ссылкой, то возвращаемое значение просто копируется в `value` так, как если бы `returnByReference()` был возвратом по значению.

В случае B мы пытаемся инициализировать ссылку `ref` копией возвращаемого значения функции `returnByValue()`. Однако, поскольку возвращаемое значение не имеет адреса (это **r-value**), мы получим

ошибку компиляции.

В случае C мы пытаемся инициализировать **константную ссылку** `cref` копией возвращаемого значения функции `returnByValue()`. Поскольку константные ссылки могут быть инициализированы с помощью `r-values`, то здесь не должно быть никаких проблем. Обычно `r-values` уничтожаются в конце выражения, в котором они созданы, однако, при привязке к константной ссылке, время жизни `r-value` (в данном случае, возвращаемого значения функции) продлевается в соответствии со временем жизни ссылки (в данном случае, `cref`).

## Заключение

В большинстве случаев идеальным вариантом для использования является возврат по значению. Это также самый гибкий и безопасный способ возврата данных обратно в вызывающий объект. Однако возврат по ссылке или по адресу также может быть полезен при работе с динамически выделенной памятью. При использовании возврата по ссылке или по адресу убедитесь, что вы не возвращаете ссылку или адрес локальной переменной, которая выйдет из области видимости, когда функция завершит свое выполнение!

## Тест

Напишите **прототипы** для каждой из следующих функций. Используйте наиболее подходящие параметры и типы возврата (по значению, по адресу или по ссылке). Используйте `const`, когда это необходимо.

### Задание №1

Функция `sumTo()`, которая принимает целочисленный параметр, а возвращает сумму всех чисел между 1 и числом, которое ввел пользователь.

#### Ответ №1

```
1 | int sumTo(const int value);
```

### Задание №2

Функция `printAnimalName()`, которая принимает структуру `Animal` в качестве параметра.

#### Ответ №2

```
1 | void printAnimalName(const Animal &animal);
```

### Задание №3

Функция `minmax()`, которая принимает два целых числа в качестве входных данных, а возвращает наименьшее и наибольшее числа в качестве отдельных параметров.

**Подсказка:** Используйте параметры вывода.

### Ответ №3

```
1 void minmax(const int a, const int b, int &minOut, int &maxOut);
```

### Задание №4

Функция `getIndexOfLargestValue()`, которая принимает целочисленный массив (как указатель) и его размер, а возвращает индекс наибольшего элемента массива.

### Ответ №4

```
1 int getIndexOfLargestValue(const int *array, const int length);
```

### Задание №5

Функция `getElement()`, которая принимает целочисленный массив (как указатель) и индекс и возвращает элемент массива по этому индексу (не копию элемента). Предполагается, что индекс корректен, а возвращаемое значение — константное.

### Ответ №5

```
1 const int& getElement(const int *array, const int index);
```

Оценить статью:



(174 оценок, среднее: 4,89 из 5)



[← Урок №99. Передача по адресу](#)

[Урок №101. Встроенные функции](#)



## Комментариев: 16



1. *Дмитрий:*

[23 ноября 2020 в 21:09](#)

Как на счет перегрузки операторов? Возможен ли возврат по указателю например для оператора +?  
Будет ли легальным следующий код:

```
1 char * MyString::operator+(const char *string)
2 {
3     char *temp = new char[a_length + StrLen(string) + 1];
4     memcpy(temp, a_data_, a_length);
```

```
5 |     memcpy(temp + a_length, string, StrLen(string) + 1);  
6 |     return temp;  
7 | }
```

[Ответить](#)2. *Sergey:*[23 октября 2020 в 22:33](#)

Во втором тесте, VS(2019) при компиляции ругался когда была такая запись

```
1 | void printAnimalName(const Animal &animal);
```

а когда добавил

```
1 | void printAnimalName(const struct Animal &animal);
```

сразу скомпилировался. Почему?

[Ответить](#)1. *Артурка:*[29 октября 2020 в 00:57](#)

Перед использованием структуры Animal требуется ее объявить до использования:

```
1 | struct Animal;  
2 | void printAnimalName(const Animal& animal);
```

Иначе компилятор не будет знать что такой тип данных существует.

[Ответить](#)3. *Константин:*[1 сентября 2019 в 20:18](#)

В случае смешивания возвращаемых значений очень хочется потрогать "за вымя" оператор static в функции int& returnByReference().

Вот как он возвращает r-values без адреса в случае функции времени исполнения программы. На какой такой мнимой оси будут сидеть r-values ?

[Ответить](#)1. *Nikita:*[10 сентября 2019 в 22:20](#)

Так наша static переменная получается глобальной, потому у нее есть свой конкретный адрес, который не уничтожается при выходе из функции. Разве не так?



[Ответить](#)

4. Константин:

[1 сентября 2019 в 19:30](#)

Здравствуйте, огромное спасибо за уроки, очень доступно, последовательно и ясно.

Вопрос:

Если мы передали в функцию объект да ещё и расположенный в динамической памяти по ссылке либо по адресу, зачем его возвращать? Можно сделать функцию вида:

```
1 void someFunc (usreClass * somBody)
2 {
3   somBody.weigt = 3;           // на этом всё, что задумывало
4 }
```

и ни чего не возвращать. Можно в коде делать так и не париться с возвратами? (за возможные ошибки извините)

[Ответить](#)

1. bash:

[21 февраля 2020 в 11:57](#)

Во-первых, ошибка в селекторе. При передаче класса или структуры по указателю, селектор должен быть косвенный.

Во-вторых, если это класс, то не факт, что это поле public и можно к нему обратиться.

[Ответить](#)

5. Дмитрий:

[7 июля 2019 в 15:06](#)

Понимающие люди, поясните пожалуйста:

1. Почему и с какой целью перед названием функции ставится "&" при передаче по ссылке и "\*" при возврате по адресу? (это подразумевает, что все параметры этой функции принимаются по ссылке / адресу соответственно, я правильно понимаю?)

2. Почему перед прототипом функции ставится const? (на примере 5-го задания)

[Ответить](#)

1. Анастасия:

[12 июля 2019 в 21:52](#)

1. Почему и с какой целью перед названием функции ставится "&" при передаче по ссылке и "\*" при возврате по адресу?

это синтаксис ссылки и указателя соответственно. То есть если их не ставить, то результат функции будет передаваться по значению. В каком случае что использовать — объясняет

данный урок.

2. Почему перед прототипом функции ставится `const`? (на примере 5-го задания)

в 5-м задании сказано, что возвращаемое значение должно быть константным, поэтому перед типом результата ставится `const`

[Ответить](#)



6. *Юлиана:*

[24 июня 2019 в 08:21](#)

У меня тоже вопрос. Почему нельзя возвращать по значению большие структуры или классы? И массивы? (Хотя про массивы я начинаю догадываться: потому что имя массива — это указатель на его начало, а если возвращать по значению указатель, то мы вернем тупо адрес, а не то, на что этот адрес ссылается), верно?

[Ответить](#)



1. *Анастасия:*

[12 июля 2019 в 21:47](#)

потому что, как уже было подмечено, чтобы вернуть что-то по значению, это что-то по сути копируется для возврата в вызывающую функцию. Так как структуры и классы — это, как правило, довольно объёмные штуки, лучше их не копировать, а передавать по ссылке. Если их в функции не надо менять, то ставить перед ними `const`

[Ответить](#)



7. *Andrey:*

[30 октября 2018 в 11:06](#)

Первое: Больше спасибо за Вашу работу.

Второе:

Вопрос накопился. 😊

1 задание: зачем создавать копию `value` когда можно туда передать ссылку, все равно же константа?

```
1 | int sumTo(const int &value);
```

3 задание: вопрос такой же как и в первом:

```
1 | void minmax(const &a, const &b, int &minOut, int &maxOut);
```

так же легче?

в 4 и 5 ссылкой можно так же передавать длину и индекс массива?

[Ответить](#)



1. *Danila:*

[27 декабря 2018 в 10:05](#)

Целочисленное значение будет передано быстрее по значению, чем по указателю, т.к. при передаче по ссылке используется неявное взятие адреса и разыменование внутри функции.

Передача по значению:

`mov rcx, 10` — помещаем в регистр значение

`call sumTo` — вызываем функцию

Передача по ссылке:

`mov dword ptr ss:[rbp-4], A` — помещаем значение в стек

`lea rcx, qword ptr ss:[rbp-4]` — получаем адрес в стеке — указатель/ссылка

`mov rcx, rcx` — помещаем этот адрес в регистр для передачи параметра

`call sumTo` — вызываем функцию

[Ответить](#)



1. *Uraut:*

[21 июня 2019 в 05:38](#)

На самом деле разницы в скорости особой нет как передавать: по значению, по адресу. При вызове функции все параметры функции, задом на перед, укладываются в стек, а только потом идет вызов `call`.

операции `mov`, `lea` — 3 тактовые. `call`, `push`, `pop` — 2 такта.

Перед вызовом функции вы при любом раскладе будете стек загружать. Нет разницы что вы кладете значение или адрес.

Любая переменная хранится в оперативе и ассемблер в любом случае лезет к ней по адресу (абсолютный:относительный). Ему все равно что забирать значение или адрес, по тактам это одинаково.

Для максимальной скорости надо больше регистры общего назначения задействовать, они на уровне кэш памяти работают и стараться гонять значения между ними (`eax`, `ebx`, `edx`, `ecx`, `esi`, `edi` и т.д.), и стек (хоть это и оперативная память, а не кеш), но доступ к стеку быстрее (за счет архитектуры микропроцессора: `pop` и `push` это 2 тактовые операции) чем просто области памяти

[Ответить](#)



8. *Oleksiy:*

[10 сентября 2018 в 11:08](#)

Выскажу свой опыт в понимании данной темы. В английском языке все просто: мы можем передать данные тремя способами: тупо по значению (`by value`), по указателю (`by pointer`) и по адресу (`by address`). Когда же читаешь тему по русски, то в сознании возникает каша, т.к. мозг не улавливает, в каком смысле используется слово "значение". Это слово имеет столько вариантов интерпретации (в отличии от английского), что затрудняет восприятие темы.

[Ответить](#)



1. *Nikita:*

[10 сентября 2019 в 22:43](#)

А это каши не вызывает?

"по указателю (by pointer) и по адресу (by address)"

Какая разница если указатель это и есть адрес? Или это типа ссылка? (опять таки на адрес)

[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