

Ravesli [Ravesli](#)


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


## Урок №111. Эллипсис

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 8 Сен 2020 |

 21526

[↑](#)  7

Во всех функциях, которые мы рассматривали до этого момента, количество их параметров должно было быть известно заранее (даже если это [параметры по умолчанию](#)). Однако есть несколько случаев, в которых полезно передать переменную, указывающую на количество параметров, в функцию. В языке C++ есть специальный объект, который позволяет это сделать — эллипсис.

Оглавление:

1. [Эллипсис](#)
2. [Почему эллипсис небезопасен?](#)
3. [Рекомендации по безопасному использованию эллипсиса](#)

## Эллипсис

Функции, использующие эллипсис, выглядят следующим образом:

тип\_возврата имя\_функции(список\_аргументов, ...)

список\_аргументов — это один или несколько обычных параметров функции. Обратите внимание, функции, которые используют эллипсис, должны иметь по крайней мере один параметр, который не является эллипсисом.

Эллипсис (англ. «*ellipsis*»), который представлен в виде многоточия ... в языке C++, всегда должен быть последним параметром в функции. О нем можно думать, как о [массиве](#), который содержит любые другие параметры, кроме тех, которые указаны в список\_аргументов.

Рассмотрим пример с использованием эллипсиса. Предположим, что нам нужно написать функцию, которая вычисляет среднее арифметическое переданных аргументов:

```
1  #include <iostream>
2  #include <cstdarg> // требуется для использования эллипсиса
3
4  // Эллипсис должен быть последним параметром.
5  // Переменная count - это количество переданных аргументов
6  double findAverage(int count, ...)
7  {
8      double sum = 0;
9
10     // Мы получаем доступ к эллипсису через va_list, поэтому объявляем переменную этого типа
11     va_list list;
12
13     // Инициализируем va_list, используя va_start. Первый параметр - это список, который мы используем
14     // Второй параметр - это последний параметр, который не является эллипсисом
15     va_start(list, count);
16
17     // Перебираем каждый из аргументов эллипсиса
18     for (int arg=0; arg < count; ++arg)
19         // Используем va_arg для получения параметров из эллипсиса.
20         // Первый параметр - это va_list, который мы используем.
21         // Второй параметр - это ожидаемый тип параметров
22         sum += va_arg(list, int);
23
24     // Выполняем очистку va_list, когда уже сделали всё необходимое
25     va_end(list);
26
27     return sum / count;
28 }
29
30 int main()
31 {
32     std::cout << findAverage(4, 1, 2, 3, 4) << '\n';
33     std::cout << findAverage(5, 1, 2, 3, 4, 5) << '\n';
34 }
```

Результат выполнения программы:

```
2.5
3
```

Как вы можете видеть, функция findAverage() принимает переменную count, которая указывает на количество передаваемых аргументов. Рассмотрим другие компоненты этого примера.

Во-первых, мы должны подключить [заголовочный файл](#) cstdarg. Этот заголовок определяет va\_list, va\_start и va\_end — макросы, необходимые для доступа к параметрам, которые являются частью эллипсиса.

Затем мы объявляем функцию, которая использует эллипсис. Помните, что `список_аргументов` должен быть представлен одним или несколькими фиксированными параметрами. Здесь мы передаем одно целочисленное значение, которое сообщает функции, сколько будет параметров.

Обратите внимание, в эллипсисе нет никаких имен переменных! Вместо этого мы получаем доступ к значениям через специальный тип — `va_list`. О `va_list` можно думать, как об [указателе](#), который указывает на массив с эллипсисом. Сначала мы объявляем переменную `va_list`, которую называем просто `list` для удобства использования.

Затем нам нужно, чтобы `list` указывал на параметры эллипсиса. Делается это с помощью `va_start()`, который имеет два параметра: `va_list` и имя последнего параметра, который не является эллипсисом. После того, как `va_start()` был вызван, `va_list` указывает на первый параметр из списка передаваемых аргументов.

Чтобы получить значение параметра, на который указывает `va_list`, нужно использовать `va_arg()`, который также имеет два параметра: `va_list` и тип данных параметра, к которому мы пытаемся получить доступ. Обратите внимание, с помощью `va_arg()` мы также переходим к следующему параметру `va_list`!

Наконец, когда мы уже всё сделали, нужно выполнить очистку: `va_end()` с параметром `va_list`.

## Почему эллипсис небезопасен?

Эллипсис предоставляет программисту большую гибкость для реализации функций, которые принимают переменную, указывающую на общее количество параметров. Однако эта гибкость имеет свои недостатки.

С обычными параметрами функции компилятор использует проверку типов для гарантирования того, что типы аргументов функции соответствуют типам параметров функции (или аргументы могут быть [неявно преобразованы](#) для дальнейшего соответствия). Это делается с целью предотвращения случаев, когда вы передадите в функцию целочисленное значение, тогда как она ожидает [строку](#) (или наоборот). Обратите внимание, параметры эллипсиса не имеют объявлений типа данных. При их использовании компилятор полностью пропускает проверку типов данных. Это означает, что можно отправить аргументы любого типа в многоточии, и компилятор не сможет предупредить вас, что это произошло. В конечном итоге, мы получим сбой или неверные результаты. При использовании эллипсиса вся ответственность ложится на caller, и от него зависит корректность переданных аргументов в функцию. Очевидно, что это является хорошей лазейкой для возникновения ошибок. Рассмотрим пример такой ошибки:

```
1 | std::cout << findAverage(6, 1.0, 2, 3, 4, 5, 6) << '\n';
```

Хотя на первый взгляд всё может показаться достаточно безвредным, но посмотрите на второй аргумент типа `double` — он должен быть типа `int`. Хотя всё скомпилируется без ошибок, но результат следующий:

1.78782e+08

Число не маленькое. Как это произошло?

Как мы уже знаем из предыдущих уроков, компьютер сохраняет все данные в виде последовательности бит. Тип переменной указывает компьютеру, как перевести эту последовательность бит в определенное (читабельное) значение. Однако в эллипсисе тип переменной отбрасывается. Следовательно, единственный способ получить нормальное значение обратно из эллипсиса — вручную указать `va_arg()`, каков ожидаемый тип параметра. Это то, что делает второй параметр в `va_arg()`. Если фактический тип параметра не соответствует ожидаемому типу параметра, то происходят плохие вещи.

В программе, приведенной выше, с помощью `va_arg()`, мы указали, что все параметры должны быть типа `int`. Следовательно, каждый вызов `va_arg()` будет возвращать последовательность бит, которая будет конвертирована в тип `int`.

В этом случае проблема заключается в том, что значение типа `double`, которое мы передали в качестве первого аргумента эллипсиса, занимает 8 байт, тогда как `va_arg(list, int)` возвращает только 4 байта данных при каждом вызове (тип `int` занимает 4 байта). Следовательно, первый вызов `va_arg` возвращает первую часть типа `double` (4 байта), а второй вызов `va_arg` возвращает вторую часть типа `double` (еще 4 байта). Итого, в результате получаем мусор.

Поскольку проверка типов пропущена, то компилятор даже не будет жаловаться, если мы сделаем что-то вообще дикое, например:

```
1 | int value = 8;  
2 | std::cout << findAverage(7, 1.0, 3, "Hello, world!", 'G', &value, &findAverage) <<
```

Верите или нет, но это действительно скомпилировалось без ошибок и выдало следующий результат на моем компьютере:

1.56805e+08

Этот результат подтверждает фразу: «Мусор на входе, мусор на выходе».

Мало того, что эллипсис отбрасывает тип параметров, он также отбрасывает и количество этих параметров. Это означает, что нам нужно будет самим разработать решение для отслеживания количества параметров, передаваемых в эллипсис. Как правило, это делается одним из следующих 3-х способов:

**Способ №1: Передать параметр-длину.** Нужно, чтобы один из фиксированных параметров, не входящих в эллипсис, отображал количество переданных параметров. Это решение использовалось в программе, приведенной выше. Однако даже здесь мы столкнемся с проблемами, например:

```
1 | std::cout << findAverage(6, 1, 2, 3, 4, 5) << '\n';
```

Результат на моем компьютере:

4.16667

Что случилось? Мы сообщили `findAverage()`, что собираемся передать 6 значений, но фактически передали только 5. Следовательно, с первыми пятью значениями, возвращаемыми `va_arg()` — всё ок. Но вот 6-е значение, которое возвращает `va_arg()` — это просто мусор из [стека](#), так как мы его не передавали. Следовательно, таков и результат. По крайней мере, здесь очевидно, что это значение является мусором. А вот рассмотрим более коварный случай:

```
1 | std::cout << findAverage(6, 1, 2, 3, 4, 5, 6, 7) << '\n';
```

Результат:

3.5

На первый взгляд всё корректно, но последнее число (7) в списке аргументов игнорируется, так как мы сообщили, что собираемся предоставить 6 параметров (а предоставили 7). Такие ошибки бывает довольно-таки трудно обнаружить.

**Способ №2: Использовать контрольное значение.** Контрольное значение — это специальное значение, которое используется для завершения цикла при его обнаружении. Например, нуль-терминатор используется в строках для обозначения конца строки. В эллипсисе контрольное значение передается последним из аргументов. Вот программа, приведенная выше, но уже с использованием контрольного значения -1:

```
1  #include <iostream>
2  #include <cstdarg> // требуется для использования эллипсиса
3
4  // Эллипсис должен быть последним параметром
5  double findAverage(int first, ...)
6  {
7      // Обработка первого значения
8      double sum = first;
9
10     // Мы получаем доступ к эллипсису через va_list, поэтому объявляем переменную этого типа
11     va_list list;
12
13     // Инициализируем va_list, используя va_start. Первый параметр - это список, который мы используем
14     // Второй параметр - это последний параметр, который не является эллипсисом
15     va_start(list, first);
16
17     int count = 1;
18     // Бесконечный цикл
19     while (1)
20     {
21         // Используем va_arg для получения параметров из эллипсиса.
22         // Первый параметр - это va_list, который мы используем.
23         // Второй параметр - это ожидаемый тип параметров
24         int arg = va_arg(list, int);
25
26         // Если текущий параметр является контрольным значением, то прекращаем выполнение цикла
27         if (arg == -1)
28             break;
29
30         sum += arg;
31         count++;
32     }
33
34     // Выполняем очистку va_list, когда уже сделали всё необходимое
35     va_end(list);
```

```

36
37     return sum / count;
38 }
39
40 int main()
41 {
42     std::cout << findAverage(1, 2, 3, 4, -1) << '\n';
43     std::cout << findAverage(1, 2, 3, 4, 5, -1) << '\n';
44 }

```

Обратите внимание, нам уже не нужно явно передавать длину в качестве первого параметра. Вместо этого мы передаем контрольное значение в качестве последнего параметра.

Однако здесь также есть нюансы. Во-первых, язык C++ требует, чтобы мы передавали хотя бы один фиксированный параметр. В предыдущем примере для этого использовалась переменная `count`. В этом примере первое значение является частью чисел, используемых в вычислении. Поэтому, вместо обработки первого значения в паре с другими параметрами эллипсиса, мы явно объявляем его как обычный параметр. Затем нам нужно это обработать внутри функции (мы присваиваем переменной `sum` значение `first`, а не `0`, как в предыдущей программе).

Во-вторых, требуется, чтобы пользователь передал контрольное значение последним в списке. Если пользователь забудет передать контрольное значение (или передаст неправильное), то функция будет циклически работать до тех пор, пока не дойдет до значения, которое будет соответствовать контрольному (которое не было указано), т.е. мусору (или произойдет сбой).

**Способ №3: Использовать строку-декодер.** Передайте строку-декодер в функцию, чтобы сообщить, как правильно интерпретировать параметры:

```

1  #include <iostream>
2  #include <string>
3  #include <cstdarg> // требуется для использования эллипсиса
4
5  // Эллипсис должен быть последним параметром
6  double findAverage(std::string decoder, ...)
7  {
8      double sum = 0;
9
10     // Мы получаем доступ к эллипсису через va_list, поэтому объявляем переменную этого
11     va_list list;
12
13     // Инициализируем va_list, используя va_start. Первый параметр - это список, который
14     // Второй параметр - это последний параметр, который не является эллипсисом
15     va_start(list, decoder);
16
17     int count = 0;
18     // Бесконечный цикл
19     while (1)
20     {
21         char codetype = decoder[count];

```

```
22     switch (codetype)
23     {
24     default:
25     case '\\0':
26         // Выполняем очистку va_list, когда уже сделали всё необходимое
27         va_end(list);
28         return sum / count;
29
30     case 'i':
31         sum += va_arg(list, int);
32         count++;
33         break;
34
35     case 'd':
36         sum += va_arg(list, double);
37         count++;
38         break;
39     }
40 }
41
42
43 int main()
44 {
45     std::cout << findAverage("iiii", 1, 2, 3, 4) << '\n';
46     std::cout << findAverage("iiii", 1, 2, 3, 4, 5) << '\n';
47     std::cout << findAverage("ididdi", 1, 2.2, 3, 3.5, 4.5, 5) << '\n';
48 }
```

В этом примере мы передаем строку, в которой указывается как количество передаваемых аргументов, так и их типы (`i = int`, `d = double`). Таким образом, мы можем работать с параметрами разных типов. Однако следует помнить, что если число или типы передаваемых параметров не будут в точности соответствовать тому, что указано в строке-декодере, то могут произойти плохие вещи.

## Рекомендации по безопасному использованию эллипсиса

Во-первых, если это возможно, не используйте эллипсис вообще! Часто доступны другие разумные решения, даже если они требуют немного больше работы и времени. Например, в функции `findAverage()` в вышеприведенной программе мы могли бы передать **динамически выделенный массив** целых чисел, вместо использования эллипсиса. Это бы обеспечило проверку типов (гарантируя, что `caller` не попытается сделать что-то бессмысленное), сохраняя при этом возможность передавать переменную длину, которая бы указывала на количество всех передаваемых значений.

Во-вторых, если вы используете эллипсис, не смешивайте разные типы аргументов в пределах вашего эллипсиса, если это возможно. Это уменьшит вероятность того, что `caller` случайно передаст данные не того типа, а `va_arg()` произведет результат-мусор.

В-третьих, использование параметра `count` или строки-декодера в качестве `список_аргументов` обычно безопаснее, чем использование контрольного значения. Это гарантирует, что цикл эллипсиса

будет завершен после четко определенного количества итераций.

Оценить статью:

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



[← Урок №110. Аргументы командной строки](#)

[Лямбда-выражения \(анонимные функции\) в C++](#)



## Комментариев: 7



1. *Валера:*

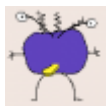
[28 июля 2020 в 19:47](#)

Аналог функции printf:

```
1 #include <iostream>
2 #include <cstdarg>
3
4 void printf2(const char* format, ...)
5 {
6     va_list list;
7
8     va_start(list, format);
9
10    int index = 0;
11
12    while (format[index] != '\0')
13    {
14        char code = format[index];
15
16        if (code != '%')
17        {
18            std::cout << format[index];
19        }
20        else if (code == '\\')
21        {
22            ++index;
23            switch (format[index])
```



```
24     {
25         case 'n': std::cout << "\n";
26             break;
27         default: std::cout << '\\\ ' << format[index] << " not supported";
28             break;
29     }
30 }
31 else
32 {
33     ++index;
34     switch (format[index])
35     {
36         case 'i': std::cout << va_arg(list, int);
37             break;
38         case 'f': std::cout << va_arg(list, float);
39             break;
40         case 'x': std::cout << std::hex << va_arg(list, int);
41             break;
42         case 's': std::cout << va_arg(list, char*);
43             break;
44         default: std::cout << format[index] << " type is not supported";
45             break;
46     }
47 }
48 ++index;
49 }
50 va_end(list);
51 }
52
53 int main()
54 {
55     printf2("str=%i\nstr2=%x\nstr3=%s", 5, 0xA, "Hello world");
56 }
```

[Ответить](#)

2. Евгений:

[28 мая 2020 в 11:24](#)

непонятно, как мы итерируем по аргументам через `va arg` — как мы получаем следующий элемент? второй, третий

[Ответить](#)

1. Максим:

[27 сентября 2020 в 19:15](#)

"Чтобы получить значение параметра, на который указывает `va_list`, нужно использовать `va_arg()`, который также имеет два параметра: `va_list` и тип данных параметра, к которому мы пытаемся получить доступ.

Обратите внимание, с помощью `va_arg()` мы также переходим к следующему параметру `va_list`!"

[Ответить](#)



3. *Алексей:*

[26 октября 2019 в 00:04](#)

Статья хорошая, но стоило бы ее продолжить и написать, что одним из самых частых примером эллипсиса является `printf`, `sprintf` и др. И что их можно сделать безопасными с помощью `__attribute__((format(printf, x, y)))`

[Ответить](#)



4. *Алексей:*

[10 июля 2019 в 15:30](#)

Я так понимаю, если эллипсис существует, следовательно он нужен для каких-то конкретных ситуаций, где динамические массивы использовать затруднительно или невозможно? Можете, пожалуйста привести примеры такого использования?

[Ответить](#)



1. *Евгений Павлов:*

[31 августа 2019 в 09:34](#)

Эллипсис появился еще до C++, еще до создания динамических массивов. В C++11 появилась чуть более безопасная версия эллипсиса: Вариативный шаблон.

[Ответить](#)



2. *Steindvart:*

[11 мая 2020 в 19:19](#)

Как верно было подмечено в комментарии выше — эллипсис был задолго до C++. А именно в языке Си. Там нет шаблонов, классов, перегрузки функций и прочих прелестей. Но наличие функций общего назначения, которые смогут работать с переменным количеством параметров было необходимо, поэтому и придумали такую конструкцию. Для обратной совместимости, C++ сохранил эту конструкцию.

Самый распространённый пример использования — это Сишные функции ввода-вывода:

```
int printf(const char *format, ...)
int sprintf(char *buf, const char *format, ...)
int scanf(const char *format, ...)
```

[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