

Ravesli [Ravesli](#)

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

Урок №107. Рекурсия и Числа Фибоначчи

👤 [Юрий](#) |

- [Уроки C++](#)

|

🖊 Обновл. 27 Сен 2020 |

👁 40115

📄 14

На этом уроке мы рассмотрим, что такое рекурсия в языке C++, зачем её использовать, а также рассмотрим последовательность Фибоначчи и факториал целого числа.

Оглавление:

1. [Рекурсия](#)
2. [Условие завершения рекурсии](#)
3. [Рекурсивные алгоритмы](#)
4. [Числа Фибоначчи](#)
5. [Рекурсия vs. Итерации](#)
6. [Тест](#)

Рекурсия

Рекурсивная функция (или просто *«рекурсия»*) в языке C++ — это функция, которая вызывает сама себя. Например:

```
1 #include <iostream>
2
3 void countOut(int count)
4 {
5     std::cout << "push " << count << '\n';
6     countOut(count-1); // функция countOut() вызывает рекурсивно сама себя
7 }
8
9 int main()
10 {
11     countOut(4);
12 }
```

```
13 | return 0;  
14 | }
```

При вызове функции `countOut(4)` на экран выведется `push 4`, а затем вызывается `countOut(3)`. `countOut(3)` выведет `push 3` и вызывает `countOut(2)`. Последовательность вызова `countOut(n)` других функций `countOut(n-1)` повторяется бесконечное количество раз (аналог бесконечного цикла). Попробуйте запустить у себя.

На [уроке о стеке и куче в C++](#) мы узнали, что при каждом вызове функции, определенные данные помещаются в стек вызовов. Поскольку функция `countOut()` никогда ничего не возвращает (она просто снова вызывает `countOut()`), то данные этой функции никогда не вытягиваются из стека! Следовательно, в какой-то момент, память стека закончится и произойдет [переполнение стека](#).

Условие завершения рекурсии

Рекурсивные вызовы функций работают точно так же, как и обычные вызовы функций. Однако, программа, приведенная выше, иллюстрирует наиболее важное отличие простых функций от рекурсивных: вы должны указать условие завершения рекурсии, в противном случае — функция будет выполняться «бесконечно» (фактически до тех пор, пока не закончится память в стеке вызовов).

Условие завершения рекурсии — это условие, которое, при его выполнении, остановит вызов рекурсивной функции самой себя. В этом условии обычно используется [оператор if](#).

Вот пример функции, приведенной выше, но уже с условием завершения рекурсии (и еще с одним дополнительным выводом текста):

```
1 | #include <iostream>  
2 |  
3 | void countOut(int count)  
4 | {  
5 |     std::cout << "push " << count << '\n';  
6 |  
7 |     if (count > 1) // условие завершения  
8 |         countOut(count-1);  
9 |  
10 |    std::cout << "pop " << count << '\n';  
11 | }  
12 |  
13 | int main()  
14 | {  
15 |     countOut(4);  
16 |     return 0;  
17 | }
```

Когда мы запустим эту программу, то `countOut()` начнет выводить:

```
push 4  
push 3  
push 2  
push 1
```

Если сейчас посмотреть на стек вызовов, то увидим следующее:

```
countOut(1)
countOut(2)
countOut(3)
countOut(4)
main()
```

Из-за условия завершения, `countOut(1)` не вызовет `countOut(0)`: условие `if` не выполнится, и поэтому выведется `pop 1` и `countOut(1)` завершит свое выполнение. На этом этапе `countOut(1)` вытягивается из стека, и управление возвращается к `countOut(2)`. `countOut(2)` возобновляет выполнение в точке после вызова `countOut(1)`, и поэтому выведется `pop 2`, а затем `countOut(2)` завершится. Рекурсивные вызовы функций `countOut()` постепенно вытягиваются из стека до тех пор, пока не будут удалены все экземпляры `countOut()`.

Таким образом, результат выполнения программы, приведенной выше:

```
push 4
push 3
push 2
push 1
pop 1
pop 2
pop 3
pop 4
```

Стоит отметить, что `push` выводится в порядке убывания, а `pop` — в порядке возрастания. Дело в том, что `push` выводится до вызова рекурсивной функции, а `pop` выполняется (выводится) после вызова рекурсивной функции, когда все экземпляры `countOut()` вытягиваются из стека (это происходит в порядке, обратном тому, в котором эти экземпляры были введены в стек).

Теперь, когда мы обсудили основной механизм вызова рекурсивных функций, давайте взглянем на несколько другой тип рекурсии, который более распространен:

```
1 // Возвращаем сумму всех чисел между 1 и value
2 int sumCount(int value)
3 {
4     if (value <= 0)
5         return 0; // базовый случай (условие завершения)
6     else if (value == 1)
7         return 1; // базовый случай (условие завершения)
8     else
9         return sumCount(value - 1) + value; // рекурсивный вызов функции
10 }
```

Рассмотреть рекурсию с первого взгляда на код не так уж и легко. Лучшим вариантом будет посмотреть, что произойдет при вызове рекурсивной функции с определенным значением. Например, посмотрим, что произойдет при вызове вышеприведенной функции с `value = 4`:

`sumCount(4)`. $4 > 1$, поэтому возвращается `sumCount(3) + 4`
`sumCount(3)`. $3 > 1$, поэтому возвращается `sumCount(2) + 3`
`sumCount(2)`. $2 > 1$, поэтому возвращается `sumCount(1) + 2`
`sumCount(1)`. $1 = 1$, поэтому возвращается 1. Это условие завершения рекурсии

Теперь посмотрим на стек вызовов:

`sumCount(1)` возвращает 1
`sumCount(2)` возвращает `sumCount(1) + 2`, т.е. $1 + 2 = 3$

`sumCount(3)` возвращает `sumCount(2) + 3`, т.е. $3 + 3 = 6$
`sumCount(4)` возвращает `sumCount(3) + 4`, т.е. $6 + 4 = 10$

На этом этапе уже легче увидеть, что мы просто добавляем числа между 1 и значением, которое предоставил caller. На практике рекомендуется указывать [комментарии](#) возле рекурсивных функций, дабы облегчить жизнь не только себе, но, возможно, и другим людям, которые будут смотреть ваш код.

Рекурсивные алгоритмы

Рекурсивные функции обычно решают проблему, сначала найдя решение для подмножеств проблемы (рекурсивно), а затем модифицируя это «подрешение», дабы добраться уже до верного решения. В вышеприведенном примере, алгоритм `sumCount(value)` сначала решает `sumCount(value-1)`, а затем добавляет значение `value`, чтобы найти решение для `sumCount(value)`.

Во многих рекурсивных алгоритмах некоторые данные ввода производят предсказуемые данные вывода. Например, `sumCount(1)` имеет предсказуемый вывод 1 (вы можете легко это вычислить и проверить самостоятельно). Случай, когда алгоритм при определенных данных ввода производит предсказуемые данные вывода, называется **базовым случаем**. Базовые случаи работают как условия для завершения выполнения алгоритма. Их часто можно идентифицировать, рассматривая результаты вывода для следующих значений ввода: 0, 1, «» или null.

Числа Фибоначчи

Одним из наиболее известных математических рекурсивных алгоритмов является последовательность Фибоначчи. Последовательность Фибоначчи можно увидеть даже в природе: ветвление деревьев, спираль раковин, плоды ананаса, разворачивающийся папоротник и т.д.

Спираль Фибоначчи выглядит следующим образом:

Каждое из чисел Фибоначчи — это длина горизонтальной стороны квадрата, в которой находится данное число. Математически числа Фибоначчи определяются следующим образом:

$F(n) = 0$, если $n = 0$

1, если $n = 1$

$f(n-1) + f(n-2)$, если $n > 1$

Следовательно, довольно просто написать рекурсивную функцию для вычисления n-го числа Фибоначчи:

```
1  #include <iostream>
2
3  int fibonacci(int number)
4  {
5      if (number == 0)
6          return 0; // базовый случай (условие завершения)
7      if (number == 1)
8          return 1; // базовый случай (условие завершения)
9      return fibonacci(number-1) + fibonacci(number-2);
10 }
11
12 // Выводим первые 13 чисел Фибоначчи
13 int main()
14 {
15     for (int count=0; count < 13; ++count)
16         std::cout << fibonacci(count) << " ";
17
18     return 0;
19 }
```

Результат выполнения программы:

0 1 1 2 3 5 8 13 21 34 55 89 144

Заметили? Это те же числа, что и в спирали Фибоначчи.

Рекурсия vs. Итерации

Наиболее популярный вопрос, который задают о рекурсивных функциях: «Зачем использовать рекурсивную функцию, если задание можно выполнить и с помощью итераций (используя цикл for или цикл while)?».

Оказывается, вы всегда можете решить рекурсивную проблему итеративно. Однако, для нетривиальных случаев, рекурсивная версия часто бывает намного проще как для написания, так и для чтения. Например, функцию вычисления n -го числа Фибоначчи можно написать и с помощью итераций, но это будет сложнее! (Попробуйте!)

Итеративные функции (те, которые используют циклы `for` или `while`) почти всегда более эффективны, чем их рекурсивные аналоги. Это связано с тем, что каждый раз, при вызове функции, расходуется определенное количество ресурсов, которое тратится на добавление и вытягивание фреймов из стека. Итеративные функции расходуют намного меньше этих ресурсов.

Это не значит, что итеративные функции всегда являются лучшим вариантом. Иногда рекурсивная реализация может быть чище и проще, а некоторые дополнительные расходы могут быть более чем оправданы, сведя к минимуму трудности при будущей поддержке кода, особенно, если алгоритм не требует слишком много времени для поиска решения.

В общем, рекурсия является хорошим выбором, если выполняется большинство из следующих утверждений:

- рекурсивный код намного проще реализовать;
- глубина рекурсии может быть ограничена;
- итеративная версия алгоритма требует управления стеком данных;
- это не критическая часть кода, которая напрямую влияет на производительность программы.

Совет: Если рекурсивный алгоритм проще реализовать, то имеет смысл начать с рекурсии, а затем уже оптимизировать код в итеративный алгоритм.

Правило: Рекомендуется использовать итерацию, вместо рекурсии, но в тех случаях, когда это действительно практичнее.

Тест

Задание №1

Факториал целого числа N определяется как умножение всех чисел между 1 и N ($0! = 1$). Напишите рекурсивную функцию `factorial()`, которая возвращает факториал ввода. Протестируйте её с помощью первых 8 чисел.

Подсказка: Помните, что $x * y = y * x$, поэтому умножение всех чисел между 1 и N — это то же самое, что и умножение всех чисел между N и 1.

Ответ №1

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

```
2
3 int factorial(int n)
4 {
5     if (n < 1)
6         return 1;
7     else
8         return factorial(n - 1) * n;
9 }
10
11 int main()
12 {
13     for (int count = 0; count < 8; ++count)
14         std::cout << factorial(count) << '\n';
15 }
```

Задание №2

Напишите рекурсивную функцию, которая принимает целое число в качестве входных данных и возвращает сумму всех чисел этого значения (например, $482 = 4 + 8 + 2 = 14$). Протестируйте вашу программу, используя число 83569 (результатом должно быть 31).

Ответ №2

```
1 #include <iostream>
2
3 int sumNumbers(int x)
4 {
5     if (x < 10)
6         return x;
7     else
8         return sumNumbers(x / 10) + x % 10;
9 }
10
11 int main()
12 {
13     std::cout << sumNumbers(83569) << std::endl;
14 }
```

Задание №3

Это уже немного сложнее. Напишите программу, которая просит пользователя ввести целое число, а затем использует рекурсивную функцию для вывода бинарного представления этого числа (см. [урок №44](#)). Предполагается, что число, которое введет пользователь, является положительным.

Подсказка: Используя способ №1 для конвертации чисел из десятичной системы в двоичную, вам нужно будет выводить биты «снизу вверх» (т.е. в обратном порядке), для этого ваш стейтмент вывода должен находиться *после* вызова рекурсии.

Ответ №3

```
1 #include <iostream>
2
3 void printBinary(int x)
4 {
```

```

5 // Условие завершения
6 if (x == 0)
7     return;
8
9 // Рекурсия к следующему биту
10 printBinary(x / 2);
11
12 // Выводим остаток (в обратном порядке)
13 std::cout << x % 2;
14 }
15
16 int main()
17 {
18     int x;
19     std::cout << "Enter an integer: ";
20     std::cin >> x;
21
22     printBinary(x);
23 }

```

Задание №4

Используя программу из задания №3, обработайте случай, когда пользователь ввел 0 или отрицательное число, например:

```
Enter an integer: -14  
11111111111111111111111111110010
```

Подсказка: Вы можете конвертировать отрицательное целое число в положительное, используя [оператор `static_cast`](#) для конвертации в `unsigned int`.

Ответ №4

```

1  #include <iostream>
2
3  void printBinaryDigits(unsigned int n)
4  {
5      // Условие завершения
6      if (n == 0)
7          return;
8
9      printBinaryDigits(n / 2);
10
11     std::cout << n % 2;
12 }
13
14 void printBinary(int n)
15 {
16     if (n == 0)
17         std::cout << '0'; // выводим "0", если n == 0
18     else
19         printBinaryDigits(static_cast<unsigned int>(n));
20 }
21

```



```
22 int main()
23 {
24     int x;
25     std::cout << "Enter an integer: ";
26     std::cin >> x;
27
28     printBinary(x);
29 }
```

Оценить статью:

★★★★★ (135 оценок, среднее: 4,93 из 5)



[← Урок №106. Ёмкость вектора](#)

[Урок №108. Обработка ошибок, сегг и exit\(\)](#)



Комментариев: 14



1. Yaroslav:

[15 июля 2020 в 21:03](#)

Числа Фибоначчи итерация.

```
1  #include <iostream>
2
3  int fibonacci(int number)
4  {
5      int count = 0;
6      int fibon = 0;
7      int fibon1 = 1;
8      int fibon2 = 1;
9
10     while(count < number)
11     {
12         if (count < 2)
13         {
14             fibon = 1;
15         }
16         else if(number >= 2)
17         {
18             fibon = fibon1 + fibon2;
19             fibon1 = fibon2;
20             fibon2 = fibon;
21         }
22         ++count;
23     }
```

```

24
25     return fibon;
26 }
27
28 int main()
29 {
30     for (int count=0; count < 13; ++count)
31         std::cout << fibonacci(count) << " ";
32     return 0;
33 }

```

[Ответить](#)



2. *cybersatori:*
[8 марта 2020 в 16:00](#)

Да уж, просто пару часов собирал функцию которая у вас в одну строчку во втором задании, зато подружился с отладчиком Visual Studio)

```

1 long summator(long n){
2     long x = n;
3     int dec = 10;
4     int plus = 1;
5     int sumNum = 0;
6
7
8     while (x != 0) {
9         int firstNum = 0;
10        while (x > dec && x%dec != 0 && x != 0) {
11            x -= plus;
12            firstNum++;
13        }
14        while (x < dec && x != 0) {
15            x -= plus;
16            firstNum++;
17        }
18
19        dec*= 10;
20        plus*= 10;
21        sumNum += firstNum;
22    }
23
24    return sumNum;
25
26 }

```

[Ответить](#)



3. *Дима:*
[11 августа 2019 в 20:27](#)

Вопрос, почему когда вызываешь функцию, то числа Фибоначчи после 8 символов идут в периоде, а когда рекурсию то целыми числами.

```
1 using namespace std;
2 int boo(double b)
3 {
4     if (b <= 0)
5         return 0;
6     else if (b == 1)
7         return 1;
8     else
9         return boo(b - 1) + boo(b-2);
10 }
11
12 int main()
13 {
14     //Рекурсия
15     //for (double count = 0; count < 113; ++count)
16     //    cout << boo(count) << "\n";
17
18     //Функция
19     double a = 1;
20     double b = 0;
21     double c = 0;
22     for (size_t i = 0; i < 113; i++)
23     {
24         c = a + b;
25         b = a;
26         a = c;
27         cout << c << endl;
28     }
29
30
31     return 0;
32 }
```

[Ответить](#)



4. Yeti:

[5 июля 2019 в 15:29](#)

4 задание.

Вроде все выполняется как надо _(ツ)_/

```
1 #include <iostream>
2
3 void factorial(int n)
4 {
5     if (n > 1)
6         factorial(n / 2);
7
8     std::cout << n % 2 ;
9 }
```

```
10
11 int main()
12 {
13     std::cout << "Enter a value: ";
14     int value = 0;
15     std::cin >> value;
16
17     if (value < 0)
18         value = -value;
19
20     factorial(value);
21
22     return 0;
23 }
```

[Ответить](#)

1. Nikita:

[21 октября 2019 в 13:59](#)

Ну отлично. Единственное что в идеале ещё добавить, так это случай когда вместо числа, введут символ.

[Ответить](#)

2. Viktor:

[2 июля 2020 в 19:36](#)

То же самое сделал)) Вообще не очень понимаю пока что как в дальнейшем использовать рекурсию.

[Ответить](#)

5. ZiFIR:

[8 марта 2019 в 16:29](#)

Немного доработал код из 1 задания.

```
1  #include <iostream>
2
3  using std::cout;
4
5  int factorial (int Number)
6  {
7      if (Number < 1)
8          return 1;
9      else
10         return factorial(Number - 1) * Number;
11 }
12
13 int main()
14 {
15     for (int x = 1; x <= 7; ++x)
```

```

16         cout << x << "!" << " = " << factorial(x) << "\n";
17
18     system("pause");
19     return 0;
20 }

```

/*Output:

```

1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
*/

```

[Ответить](#)



1. *SuRprizZe:*

[13 апреля 2019 в 22:35](#)

Сделал номер 4 вот так, все работает вывод такой же...

```

1  #include "pch.h"
2  #include <iostream>
3
4  void printBinary(int x)
5  {
6      // условие завершения
7      if (x == 0)
8          return;
9
10     // рекурсия к следующему биту
11     printBinary(static_cast<unsigned int>(x) / 2);
12
13     // выводим остаток (в обратном порядке)
14     std::cout << static_cast<unsigned int>(x) % 2;
15 }
16
17 int main()
18 {
19     int x;
20     std::cout << "Enter an integer: ";
21     std::cin >> x;
22
23     printBinary(x);
24 }

```

[Ответить](#)



6. *Александр:*

[1 марта 2019 в 14:16](#)

Приведенная рекурсивная реализация вычислений чисел Фибоначчи безумно медленная. При том, что глубина вызовов не превышает номера числа, но самих вызовов производится очень много.

Я бы на примере этой функции советовал очень аккуратно относиться к рекурсивным функциям, которые вызывают более одного своего экземпляра (эта проблема легко лечится, но само ее возникновение не так очевидно для начинающих)

По поводу цикл vs рекурсия не совсем согласен с тем, что "любую рекурсию можно реализовать циклом". Наоборот верно — "любой цикл можно заменить рекурсией"

Если пытаться оставить ту же идею и не менять ее принципиально, то в общем случае рекурсия циклом не заменяется. Иногда приходится радикально менять подход для того, чтобы переписать функцию итеративно. Это уже не "замена рекурсии на цикл", а "придумывание принципиально иного алгоритма"

На цикл однозначно и всегда заменяется "хвостовая рекурсия"

Очень простой пример: вывод последовательности в обратном порядке (последовательность читаем до 0). рекурсивное решение:

```
1 void rec(std::istream &cin, std::ostream &cout) {
2     int a;
3     cin >> a;
4     if(a) rec(cin, cout);
5     cout << a << " ";
6 }
```

Эту задачу легко решить итеративно, но это будет совершенно иной подход. Прочитать последовательность в какой-нить массив и затем вывести в обратном порядке.

ЗЫ Спорить о том, как лучше нет смысла. Пример приведен исключительно, чтобы показать, что "тупо в лоб" рекурсия на цикл не заменяется. Можно рассмотреть более сложные примеры из комбинаторики, для которых рекурсия простая и очевидная, а реализация решения итеративно просто выносит мозг.

[Ответить](#)



1. *Fakovka999:*

[13 августа 2020 в 00:31](#)

Соглашусь с вами, делал свою функцию — альтернативу "itos" (int в string)

```
1 void rev(unsigned int i, std::string& s)
2 {
3     if (i > 0)
4     {
5         rev(i / 10, s);
6         s += char(0x30 | (i % 10));
7     }
8 }
```

такая запись для меня оказалась на много удобней и эстетичней, чем решение ее в лоб итерацией, а это значит перебрать массив в обратном порядке. Как не крути это разные подходы к одной задаче. Но как не крути в конечном итоге когда весь основной код будет готов, можно попробовать и другие методы.

[Ответить](#)



7. *Александр:*

[1 марта 2019 в 14:04](#)

@@@

Поскольку функция countOut() никогда ничего не возвращает (она просто снова вызывает countOut()), то данные этой функции никогда не вытягиваются из стека!

@@@

Нет... данные не извлекаются из стека не потому, что функция ничего не возвращает... можно переписать ее код так:

```
1 void countOut(int count) {
2     std::cout << "push " << count << '\n';
3     if(count) countOut(count-1);
4 }
```

и она все еще ничего не будет возвращать (да и как она что-то может вернуть, если она void?)

корректней сказать, что ни одна из функций не завершается, так как она ожидает завершения вызванной ей копии. Ну или как-то в этом роде...

Оно то понятно, что имеется в виду под "не возвращает", но для этого "понятно" нужно уже что-то знать, а тексты вроде как для новичков позиционируются 😊

[Ответить](#)



1. *Andrew Gulenko:*

[7 сентября 2020 в 22:38](#)

Спасибо. Так точно понятнее.

[Ответить](#)

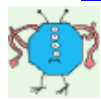


2. *Антон:*

[19 октября 2020 в 13:09](#)

ну не знаю, если человек дошел до сюда, и хорошо понимает что void не возвращает ничего, то, как по мне, можно догадаться, что имелось в виду автором, да и те кто читают уроки про рекурсию, уже явно не те новички, которыми были перед тем как прочитали эти 100 уроков

[Ответить](#)



8. *kmish:*

[15 февраля 2019 в 12:02](#)

2е задание:

```
1 int sumDigits(char *digits, int length)
2 {
3     std::cout << "Symbol " << *digits << ", sym_number is " << static_cast<int>(*digits) <
4     if (length < 1)
5         return 0;
6     return sumDigits(digits + 1, length - 1) + static_cast<int>(*digits) - 48;
7 }
```

```
8 |
9 | int main()
10 | {
11 |     std::cout << "Enter a number: ";
12 |     char digits[255];
13 |     std::cin.getline(digits, 255);
14 |     std::cout << "You've entered number: " << digits << ", strlen(digits) " << strlen(digi
15 |     int sum = sumDigits(digits, strlen(digits));
16 |     std::cout << "The sum of your digits is: " << sum;
17 |
18 |     return 0;
19 | }
```

[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