Лекция 16

Классы памяти. Рекурсивные функции

Преподаватель Палехова Ольга Александровна,
кафедра О7 БГТУ «Военмех»

Внутренние и внешние объекты

Объект программы является

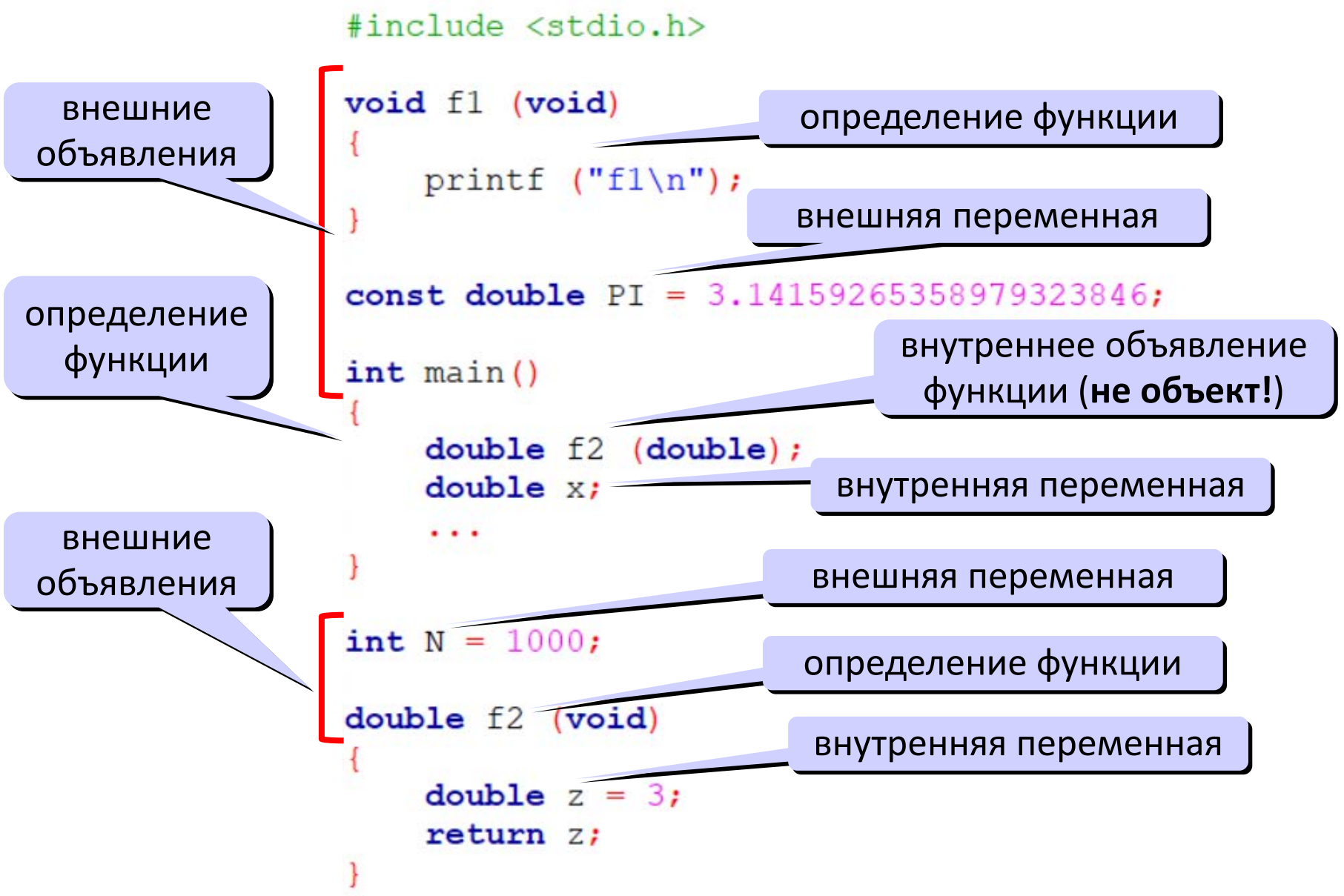
- **внешним**, если определяется **вне** функции;
- **внутренним**, если определяется **внутри** функции.

Функции – ВНЕШНИЕ объекты!

Особенности внешних объектов:

- внешние объекты можно хранить в разных файлах и компилировать независимо друг от друга;
- одинаковые внешние имена, используемые в разных файлах, относятся к одному и тому же внешнему объекту;
- внешние объекты доступны всем объектам программы.

Внутренние и внешние объекты



Внешние переменные

Внешняя переменная – это переменная, объявленная вне всех функций.

Достоинство:

- может использоваться всеми функциями, не надо передавать через параметры.

Недостатки:

- использование внешних переменных ухудшает структуру программы;
- изменение значения внешней переменной сложно контролировать.

Области видимости имен

Областью видимости имени называется часть программы, в которой это имя можно использовать.

Область видимости:

- ***внутренней*** переменной, объявленной внутри блока – ***до конца блока***;
- внутренней переменной, являющейся ***параметром*** функции – ***до конца функции***;
- ***внешней*** переменной или функции – от точки программы, где она объявлена, ***до конца файла***.

Область видимости идентификатора может быть «перекрыта» из функции или блока объявлением одноименной переменной

Области видимости имен

```
int x = 1000;

int main()
{
    double x = 5.5;
    ...
    {
        int x = 1;
        ...
    }
    ...
}
```

внешняя переменная, область видимости – весь файл

локальная переменная, область видимости – функция *main()*, внешняя переменная ***x*** невидима

локальная переменная, область видимости – до конца блока, другие переменные ***x*** невидимы

Классы памяти

На область видимости и время жизни переменной влияют **классы памяти.**

Классы памяти:

- **auto** – автоматическая (локальная);
- **register** – регистровая (локальная);
- **static** – статическая (локальная);
- **extern** – внешняя (глобальная)

По умолчанию память под внутренние переменные выделяется в стеке (автоматическая), а под внешние – в сегменте данных (статическая)

Классы памяти локальных переменных

auto

автоматическая (локальная) память, переменные **создаются** при входе в функцию или блок **и уничтожаются** при выходе из него, память выделяется в стеке;

register

регистровая (локальная) – рекомендация компилятору помещать часто используемую переменную в регистры процессора для ускорения программы, при невозможности такого размещения переменная остается автоматической, **адрес** регистровой переменной **получить нельзя**;

static

статическая память, время жизни переменных, в том числе внутренних, – **всё время работы программы**, значение переменной сохраняется между вызовами функции память выделяется в сегменте данных.

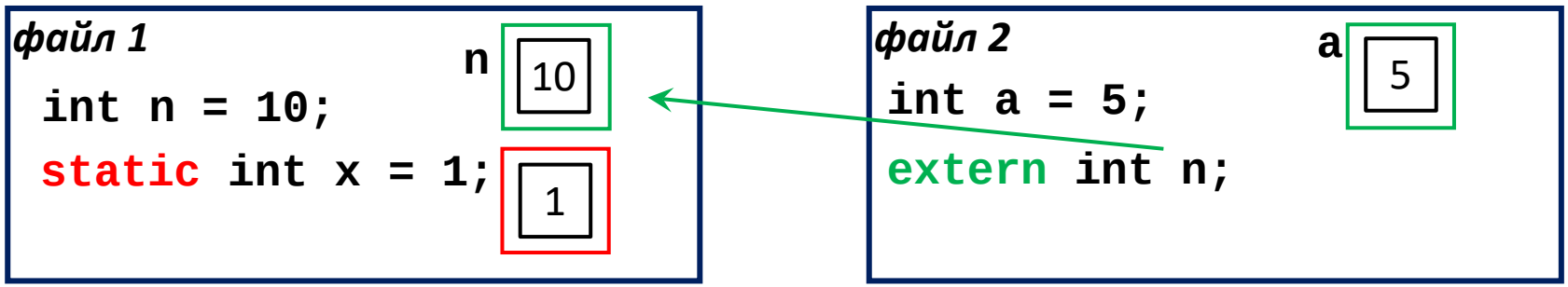
Классы памяти внешних переменных

static

для внешних переменных область видимости – текущий файл, сделать переменную доступной из другого файла проекта при этом нельзя;

extern

позволяет «расширить» область видимости внешней переменной на другой файл. При объявлении переменной с ключевым словом ***extern*** память не выделяется.



Классы памяти

Файл *main.c*

```c
#include <stdio.h>

int var = 5;

void func (void);
void f1 (void);
void f2 (void);
void f3 (void);

int main(int argc, char *argv[])
{
  int var = 1000;
  func();
  printf ("main - %d\n", var++);
  f1();
  f2();
  f3();
  {
     int var = 1000000;
     printf ("block - %d\n", var++);
  }
  func();
  printf ("main - %d\n", var++);
  f1();
  f2();
  f3();
  printf ("main - %d\n", var++);
  return 0;
}
```

Проект содержит 3 файла: *main.c*, *func.c* и *functions.c*

определение глобальной переменной

объявления функций, их определения находятся в других файлах

локальная переменная функции main() перекрывает видимость глобальной переменной

локальная переменная перекрывает видимость одноименных глобальной и локальной

блок завершился, переменной блока больше нет

вывод значения локальной переменной функции main()

Классы памяти

Файл *func.c*

```
#include <stdio.h>

void func(void)
{
     extern int var;
      printf ("func - %d\n", var++);
}
```

объявление переменной, гласящее, что используется переменная, определенная в другом файле

Файл *function.c*

```
#include <stdio.h>

extern int var;

void f1 (void)
{
    static int var = 5555;
    printf ("f1 - %d\n", var++);
}

void f2 (void)
{
    int var = 100;
    printf ("f2 - %d\n", var++);
}

void f3 (void)
{
    printf ("f3 - %d\n", var++);
}
```

локальная переменная функции f1() перекрывает видимость глобальной переменной, значение переменной сохраняется между вызовами функции

локальная переменная функции f2() перекрывает видимость глобальной переменной

функция использует значение глобальной переменной

Классы памяти

Результат работы программы:

```
func - 5          /* глобальная переменная */
main - 1000       /* локальная переменная функции main() */
f1 - 5555         /* локальная статическая переменная функции f1 */
f2 - 100          /* локальная автоматическая переменная функции f2 */
f3 - 6            /* глобальная переменная */
block - 1000000   /* локальная переменная блока внутри main() */
func - 7          /*глобальная переменная */
main - 1001       /* локальная переменная функции main() */
f1 - 5556         /* локальная статическая переменная функции f1 */
f2 - 100          /* локальная автоматическая переменная функции f2 */
f3 - 8            /* глобальная переменная */
main - 1002       /* локальная переменная функции main() */
```

Рекурсия

Рекурсия – определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса, то есть ситуация, когда объект является частью самого себя.

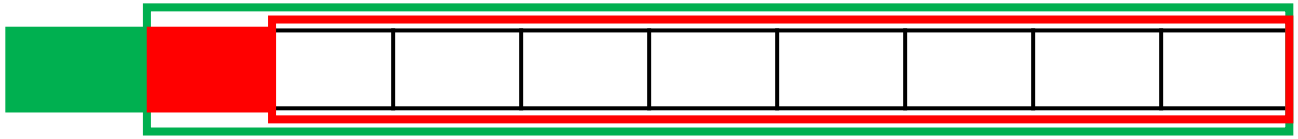
Рекурсивные данные и алгоритмы

Рекурсивными могут быть **структуры данных и алгоритмы**.

Структура данных является **рекурсивной**, если ее ***части*** выглядят так же, как вся структура в ***целом***.

Пример

массив состоит из первого элемента и массива, содержащего на один элемент меньше.



Алгоритм является рекурсивным, если прямо или опосредованно вызывает сам себя.

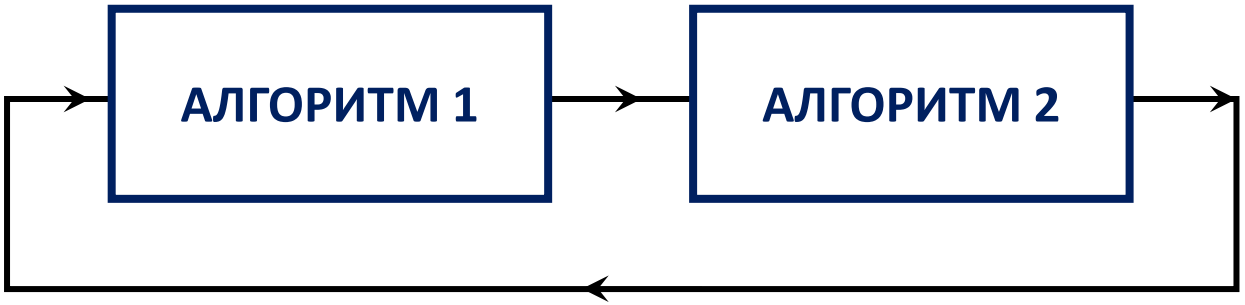
Пример

чтобы вычислить сумму *N* слагаемых, нужно вычислить сумму *N*-1 слагаемых и прибавить последнее слагаемое.

Прямая и косвенная рекурсия

Прямая рекурсия – способ организации работы алгоритма, при котором в нем содержится обращение к тому же алгоритму, но с другим набором входных данных.

Косвенная рекурсия – способ организации работы алгоритма, при котором в нем содержится обращение к другому алгоритму, в процессе работы которого требуется обращение к первому алгоритму.



Рекурсивный спуск

Рекурсивная функция – это функция, реализующая рекурсивный алгоритм.

Процесс рекурсивных вызовов функции называется **рекурсивным спуском.** При каждом вызове функции выделяется память под параметры и локальные переменные функции.

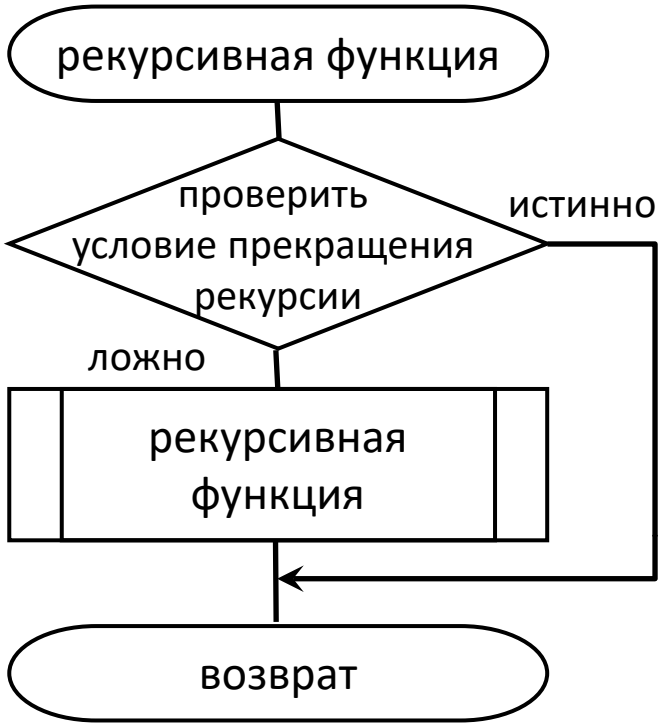
Наибольшее возможное количество рекурсивных обращений называется **глубиной рекурсии**. Максимальная глубина рекурсии определяется отношением размера программного стека к объему памяти, требуемому для хранения локальных переменных функции.

Завершенность рекурсивных вызовов обеспечивает **базис рекурсии** – случай, результат для которого очевиден и не требует проведения расчетов.

Структура рекурсивной функции

Структура рекурсивной функции – ***ветвление***:

- хотя бы одна ***рекурсивная*** ветвь,
- хотя бы одна ***терминальная*** ветвь.



Рекурсивная ветвь содержит хотя бы один рекурсивный вызов и выполняется, когда условие прекращения рекурсии ложно.

Терминальная ветвь выполняется, когда условие прекращения рекурсии истинно; функция возвращает значение, не выполняя рекурсивного вызова.

! В правильно написанной рекурсивной функции число рекурсивных вызовов **конечно**

Функция вычисления факториала

Факториал $N! = \begin{cases} 1, \text{если } N = 0 \text{ или } N = 1 \\ N \cdot (N-1)!, \text{если } N > 1 \end{cases}$

базис рекурсии

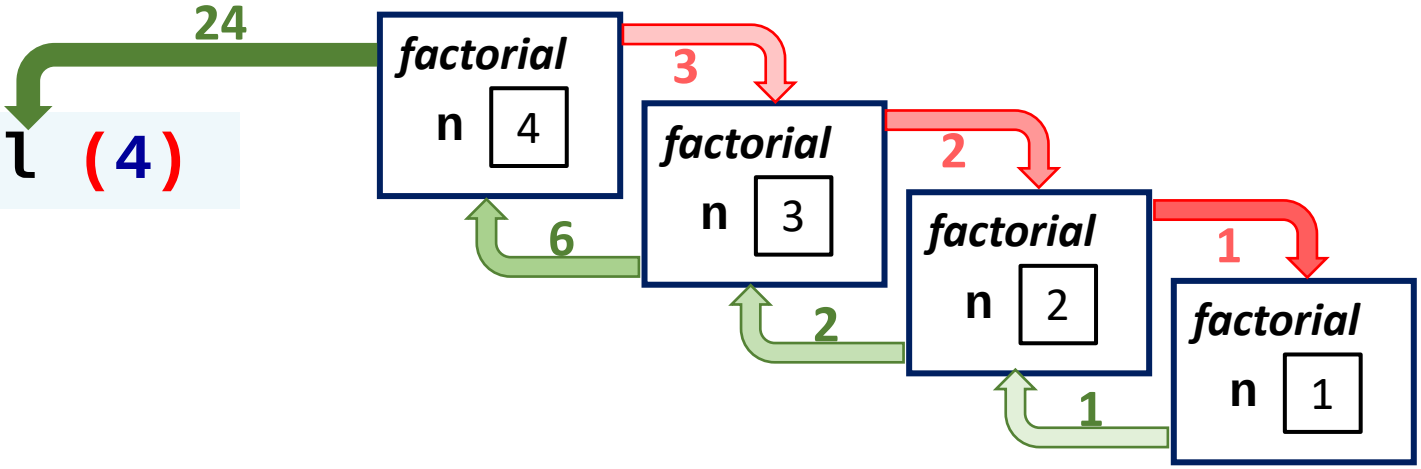
```
double factorial ( unsigned char n )
{
    if ( !n || n == 1 )
        return 1;
    return n * factorial ( n - 1 );
}
```

условие прекращения рекурсии

рекурсивный вызов

Вызов:

```
factorial (4)
```



Функция вывода массива

```
void output_array ( int a[], int n )
{
    if ( !n )
        return;
    printf ("%d ", *a);
    output_array ( a+1, n-1 );
}
```

условие прекращения рекурсии

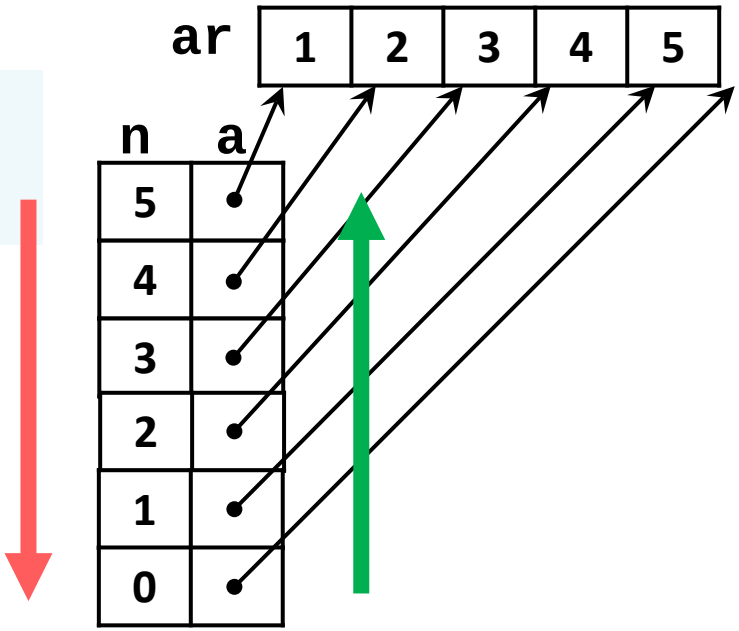
действие на рекурсивном спуске

рекурсивный вызов

хвостовая рекурсия

Вызов:

```
int ar[5] = {1,2,3,4,5};
output_array (ar, 5);
```



Результат:

```
1 2 3 4 5
```

! Хвостовая рекурсия заменяется циклом

Функция вывода массива

```
void output_array_back ( int a[], int n )
{
    if ( !n )
        return;
    output_array_back ( a+1, n-1 );
    printf ("%d ", *a);
}
```

условие прекращения рекурсии всегда на рекурсивном спуске

рекурсивный вызов

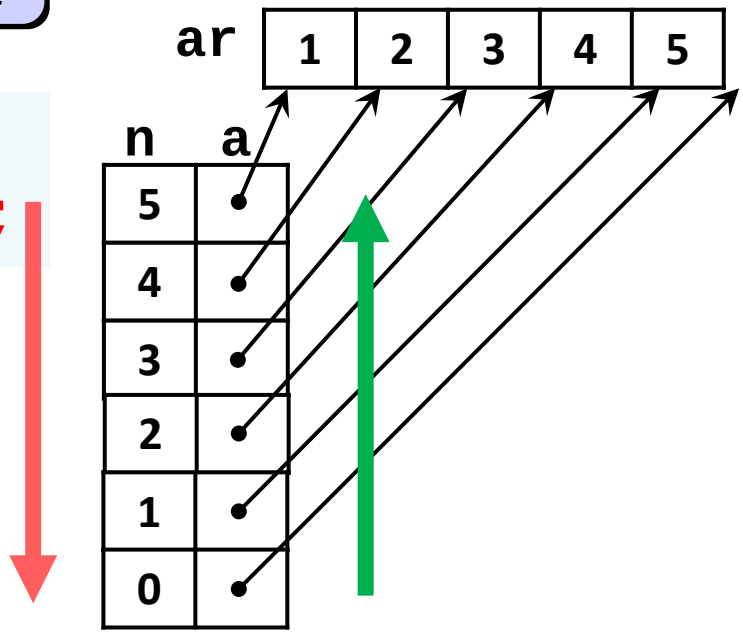
действие на рекурсивном возврате

Вызов:

```
int ar[5] = {1,2,3,4,5};
output_array_back (ar, 5);
```

Результат:

```
5 4 3 2 1
```



Задача о Ханойских башнях

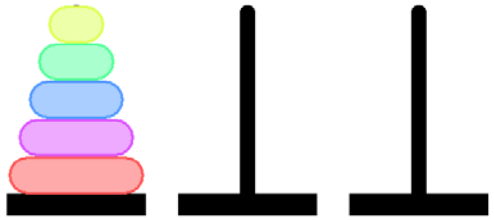
В Великом храме города Бенарес, под собором, отмечающим середину мира, находится бронзовый диск, на котором укреплены 3 алмазных стержня, высотой в один локоть и толщиной с пчелу. Давным-давно, в самом начале времён, монахи этого монастыря провинились перед богом Брахмой. Разгневанный Брахма воздвиг три высоких стержня и на один из них возложил 64 диска, сделанных из чистого золота, причём так, что каждый меньший диск лежит на большем.

Как только все 64 диска будут переложены со стержня, на который Брахма сложил их при создании мира, на другой стержень, башня вместе с храмом обратятся в пыль и под громовые раскаты погибнет мир.

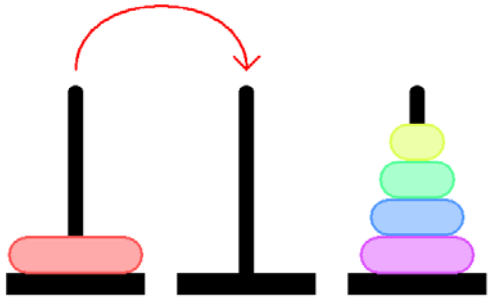
(автор - François Édouard Anatole Lucas)

Задача о Ханойских башнях

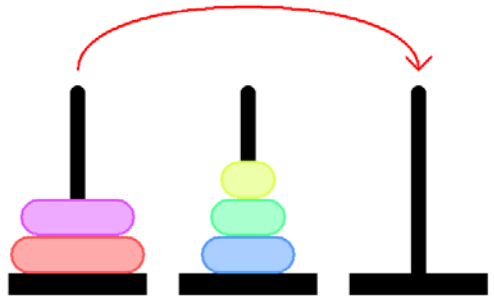
Задача. Есть 1 пирамидка с дисками разного размера, и еще 2 пустые пирамидки. Надо переместить диски с одной пирамидки на другую. Перекладывать можно только по одному диску за ход. Складывать диски можно только меньший на больший.



Чтобы переложить пирамидку на вторую ось, надо переложить самый нижний диск, а сделать это можно только тогда, когда 4 верхних диска будут на третьей оси:



Чтобы переложить 4 диска на третью ось, нужно решить ту же задачу, но для 4-х дисков. То есть на третью ось можно переложить 4-й диск только тогда, когда 3 диска на второй оси:



Задача о Ханойских башнях

Алгоритм

1. Переложить n-1 диск на вспомогательный стержень
2. Переложить 1 диск на целевой стержень
3. Переложить n-1 диск на целевой стержень

```
void hanoj ( int n, int a, int b, int c )
{
    if ( n == 1 )
    {
        printf ("%d -> %d\n", a, b);
        return;
    }
    hanoj (n-1, a, c, b);
    hanoj (1, a, b, c);
    hanoj (n-1, c, b, a);
}
```

условие прекращения рекурсии

рекурсивные вызовы

! Потребуется 2^n-1 операция

Подводим итоги

1. Многие задачи можно решить как итерационно, так и с помощью рекурсии
2. Рекурсия требует времени на вызов функции
3. Рекурсия требует дополнительной памяти для хранения локальных переменных

Достоинства рекурсии:

- в некоторых случаях рекурсия позволяет более **ясно** и **просто** сформулировать идею алгоритма;
- рекурсивный код читать **легче**, чем итерационный

Недостатки рекурсии:

- при больших объемах локальных данных и/или большой глубине рекурсии возможно **переполнение стека**;
- на рекурсивные вызовы требуется **больше времени**.