Семинар #3: Функции.

Функции

Функция - это сущность, которая ассоциирует с собой набор инструкций и входящие в неё параметры. Можно использовать функции, написанные другими программистами. Например, вы уже использовали функции `printf` и `scanf` из стандартной библиотеки `stdio.h`.

Можно писать и свои функции. В примере ниже была написана функция `square`, которая принимает одно целое чиисло и возвращает квадрат этого числа. Функция `square` используется в функции `main`, чтобы вычислить квадрат числа 5. Чтобы вызвать функцию `square` нужно написать её имя и сразу после имени в скобках передать одно целое число. Как только программа "увидет"выражение `square(5)` она зайдёт в функцию `square`, при этом будет считать что параметр `x` равен 5. После того как функция выполнится и вернёт число 25, программа подставит за место выражения `square(5)` число 25.

```c
#include <stdio.h>

int square(int x)
{
    return x * x;
}

int main()
{
    printf("%i\n", square(5));
}
```

Помимо функций с возвращаемым значением можно написать и функцию, которая ничего не возвращает. Например, `print_n_times` – печатает число `n` раз и ничего не возвращает. У функций, которые ничего не возвращают на месте возвращаемого типа стоит ключевое слово `void`.

```c
#include <stdio.h>

void print_n_times(int a, int n)
{
    for (int i = 0; i < n; ++i)
        printf("%i ", a);
}

int main()
{
    print_n_times(7, 3);
}
```

Задачи:

Решите задачи в папке `code/0function`.

Копирование при передаче в функцию

Одна из самых важных вещей, которые нужно знать о функциях: При передаче в функцию объекты копируются. Соответственно, функция работает с копией передаваемого объекта.

Рассмотрим, например, пример ниже. В функции `main` мы создаём переменную `a`, равную 10 и передаём её в функцию `inc`. Но при передаче в функцию `inc` переменная `a` копируется и функция `inc` работает с копией переменной `a`. Соответственно, если мы увеличим на 1 переменную `a` в функции `inc`, то переменная `a` в функции `main` не поменяется. В результате на экран будет напечатано число 10.

```c
#include <stdio.h>

void inc(int a)
{
    a += 1;
}

int main()
{
    int a = 10;
    inc(a);
    printf("%i\n", a);
}
```

Единственные объекты, которые не копируются в функции при передаче это массивы. Подробнее о передаче массивов в функции будет чуть позже.

Задачи:

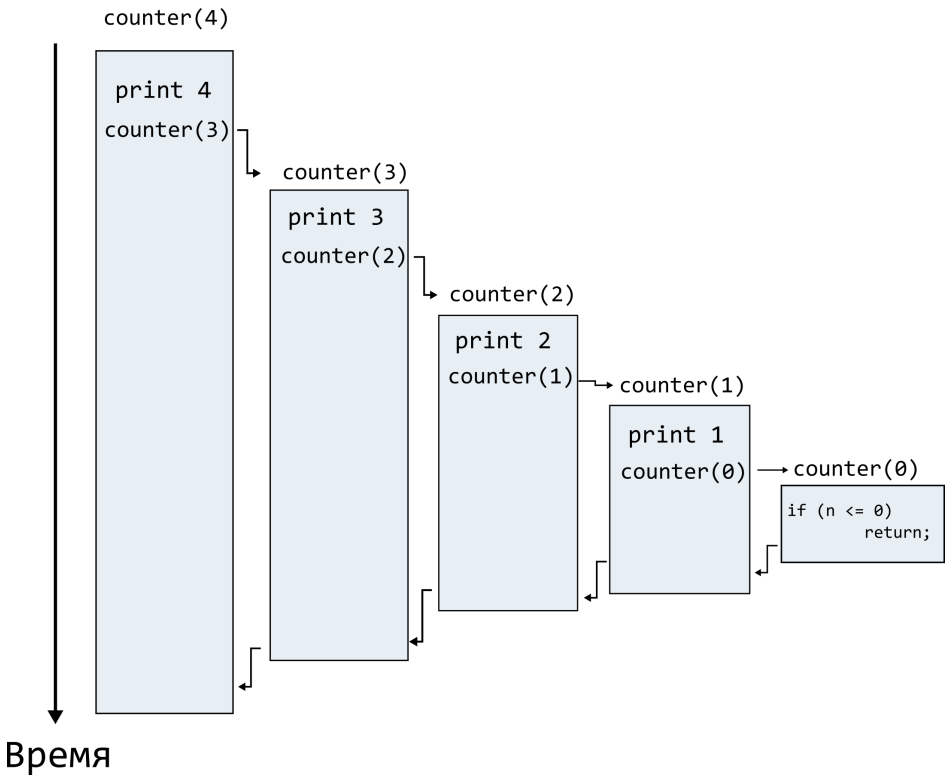
Решите задачи в папке `code/1scope`.

Рекурсия

Рекурсивная функция – это функция, которая вызывает саму себя. В примере ниже функция `counter` – рекурсивная. Если этой функции передать, скажем, число 4, то она напечатает это число и вызовет функцию `counter`, передав ей число 3. Новый вызов функции `counter` с аргументом 3 напечатает 3 и вызовет функцию `counter`, передав ей число 2. Так будет продолжаться пока число не дойдёт до 0 и сработает проверка `n <= 0`.

```c
#include <stdio.h>

void counter(int n)
{
    if (n <= 0)
        return;

    printf("%i ", n);
    counter(n - 1);
}

int main()
{
    counter(4);
}
```

Для лучшего понятия, что происходит в этой функциях `counter` рассмотрим на схемы вызовов рекурсивной функции. Для изначальной функции `counter` она выглядит следующим образом (предположим, что вызываем `counter(4)`):



Задачи:

Решите задачи в папке `code/2recursion`.

Передача массива в функцию

Массивы можно передавать в функции. Однако, передача массива в функцию в языке `C` устроена таким образом, что узнать размер массива внутри функции нельзя. Поэтому размер массива нужно тоже передавать в функцию вместе с массивом.

Ещё одной особенностью передачи массива в функцию является то, что массив это единственный объект, который НЕ копируется при передаче в функцию. Вместо этого функция работает с оригинальным массивом. Благодаря этой особенности при изменении передаваемого массива внутри функции он изменится и вне функции. Это можно увидеть на примере функции `inc` в примере кода ниже.

Пример двух функций: одна печатает массив на экран, другая прибавляет ко всем элементам массива `1`.

```c
#include <stdio.h>

void print_array(int array[], int size)
{
    for (int i = 0; i < size; ++i)
        printf("%i ", array[i]);

    printf("\n");
}

void inc(int array[], int size)
{
    for (int i = 0; i < size; ++i)
        array[i] += 1;
}

int main()
{
    int a[10] = {4, 8, 15, 16, 23, 42};
    print_array(a, 6);
    inc(a, 6);
    print_array(a, 6);
}
```

Задачи:

Решите задачи в папках `code/3array_to_function` и `code/4function_prototypes`.

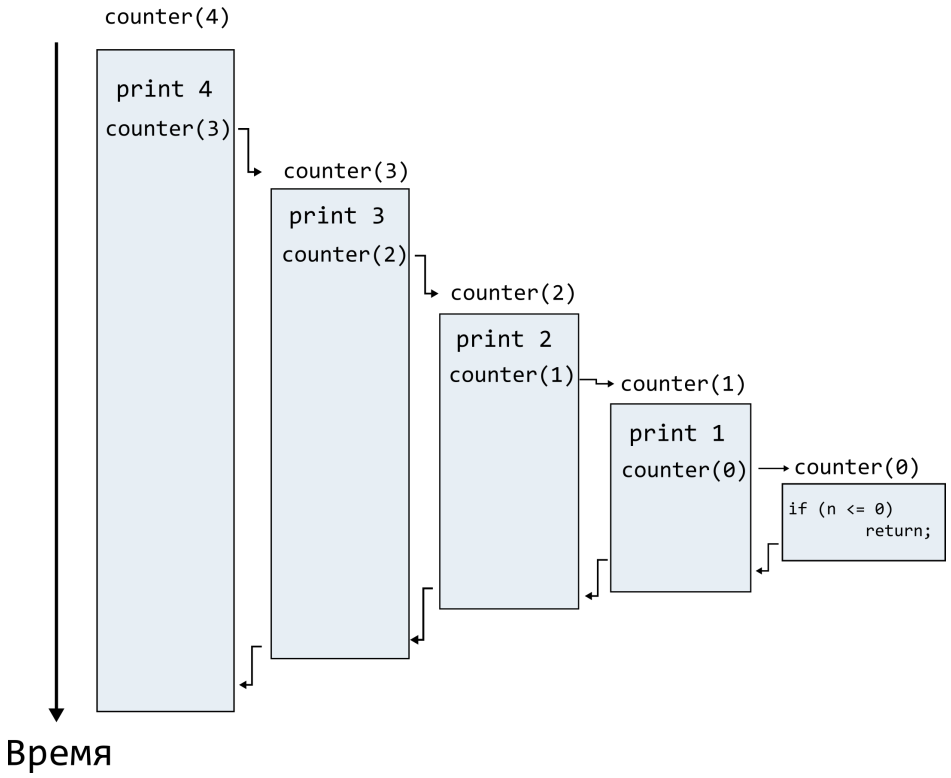
Решения задач на рекурсию

Эту главу следует читать только после того как вы прорешаете задачи из папки `code/2recursion`.

- Функция `counter`, которая рекурсивно печатает числа от `n` до `1`:

```c
void counter(int n)
{
    if (n <= 0)
        return;

    counter(n - 1);
    printf("%i ", n);
}
```

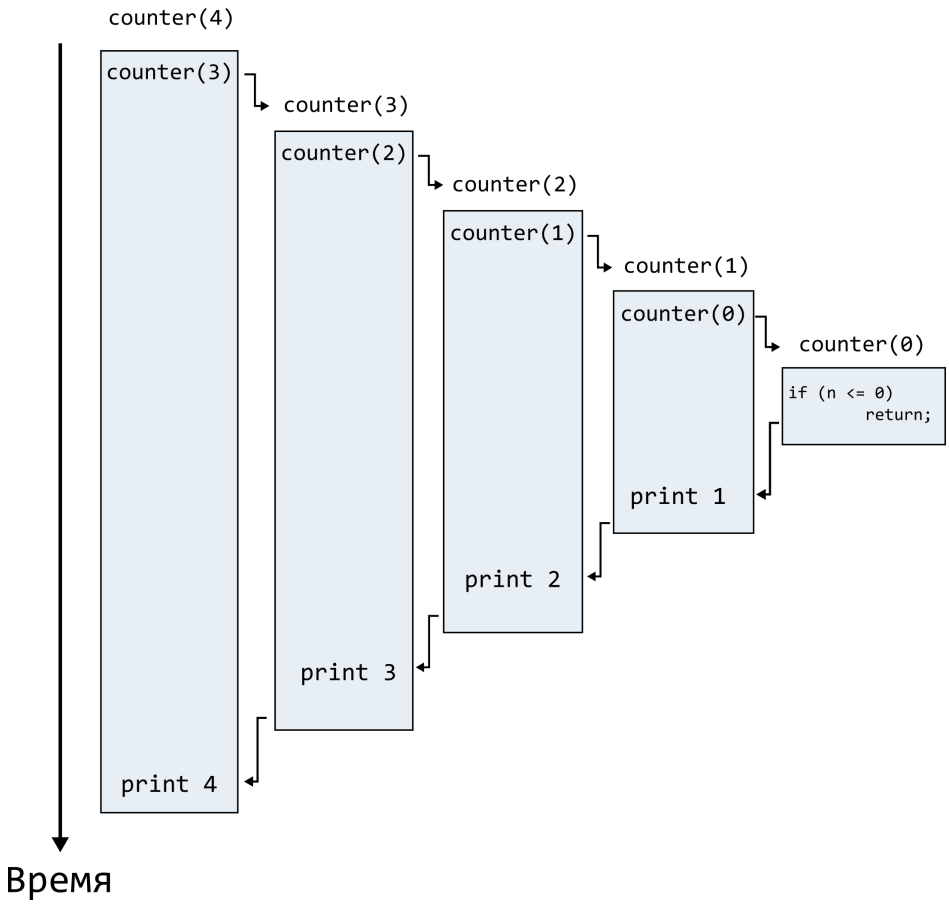
Для лучшего понятия, что происходит в этой функциях `counter` рассмотрим на схемы вызовов рекурсивной функции. Для изначальной функции `counter` она выглядит следующим образом (предположим, что вызываем `counter(4)`):



- Немного изменим изначальную функцию `counter` чтобы она печатала числа по возрастанию от 1 до `n`:

```
void counter(int n)
{
    if (n <= 0)
        return;

    printf("%i ", n);
    counter(n - 1);
}
```

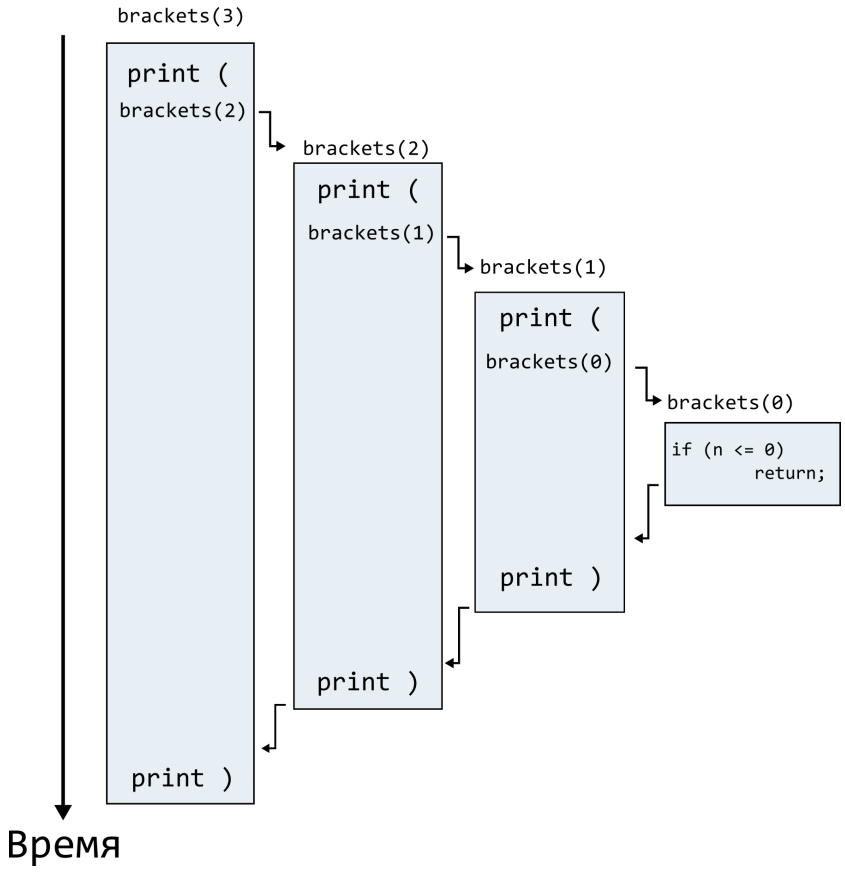
Схема для функции `counter`, которая печатает числа по возрастанию выглядит так:



- **Скобочки:** Напишите рекурсивную функцию `brackets`, которая будет печатать некоторую скобочную последовательность. `brackets(n)` должна сначала печатать `n` открывающихся скобочек, а затем `n` закрывающихся. Например, вызов `bracket(4)` должен напечатать `(((())))`.
  Чтобы это сделать рекурсивно нужно сделать следующее:
  - Напечатать открывающуюся скобку
  - Напечатать `n-1` открывающихся и `n-1` закрывающихся скобок вызовом рекурсивной функции
  - Напечатать закрывающуюся скобку

```c
#include <stdio.h>

void brackets(int n)
{
    if (n == 0)
        return;

    printf("(");
    brackets(n - 1);
    printf(")");
}

int main()
{
    brackets(5);
}
```

Схема вызовов для этой функции(вызываем `brackets(3)`):



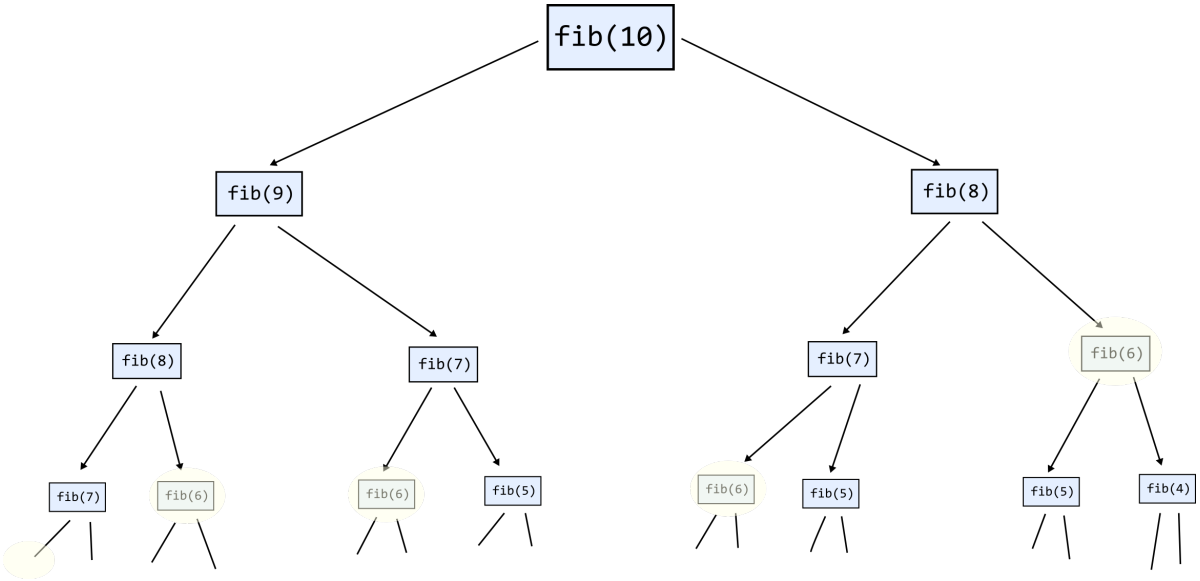
- **Фибоначчи:** Для вычисления числа Фибоначчи было написано 2 функции. Функция `fib` вычисляет число Фибоначчи итеративно(то есть с помощью цикла), а функция `fibrec` вычисляет то же самое рекурсивно.

```c
#include <stdio.h>

int fib(int n)
{
    int a = 0, b = 1;
    for (int i = 1; i < n; ++i)
    {
        int temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}

int fibrec(int n)
{
    if (n < 2)
        return n;

    return fibrec(n - 1) + fibrec(n - 2);
}
int main()
{
    printf("%i\n", fib(45));
    printf("%i\n", fibrec(45));
}
```

Кажется, что обе функции работают, но почему-то рекурсивная функция работает намного медленней. Почему это происходит?

Замечание: Это не связано с переполнением числа. 45-е число Фибоначчи может храниться в переменной типа `int`.

Рекурсивная функция вычисления чисел Фибоначчи работает так медленно потому что она по многу раз вычисляет одни и те же значения. На изображении ниже предсталена схема вызовов этой функции для вычисления 10-го числа Фибоначчи.



Видно что промежуточные числа Фибоначчи вычисляются заново по много раз. Например, `fib(6)` вызывается 5 раз в процесе вычисления `fib(10)`, `fib(5)` вызывается 8 раз, а меньшие числа Фибоначчи вычисляются ещё чаще. Получается, что количество операций для вычисления одного числа Фибоначчи рекурсивным способом экспоненциально большое.

Рекурсивную функцию можно исправить если в процессе выполнения запоминать все промежуточные вычисления и не делать их заново. Для этого заведём глобальный массив `data` в котором будем хранить все промежуточные числа Фибоначчи. В начале все элементы этого массива равны нулю. Теперь мы продолжаем рекурсию только если мы не вычисляли данное число ранее и `data[n] == 0`.

```c
#include <stdio.h>

int data[1000];

int fibrec(int n)
{
    if (n < 2)
        return n;

    if (data[n] == 0)
        data[n] = fibrec(n - 1) + fibrec(n - 2);

    return data[n];
}

int main()
{
    printf("%i\n", fibrec(45));
}
```