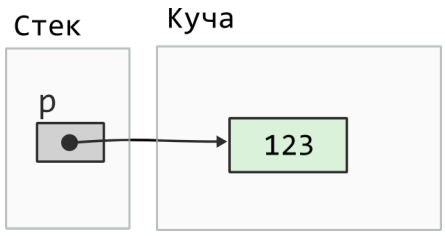
# Семинар #5: Сегменты памяти. Домашнее задание.

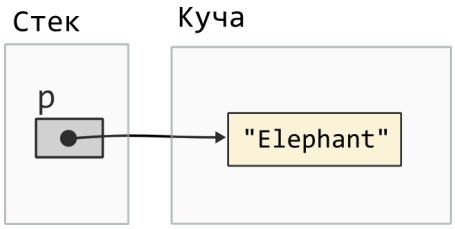
## Задача 1. Создание объектов в куче

Напишите код, который будет создавать в куче объекты, соответствующие следующим рисункам. В каждой задаче напечатайте созданные в куче объекты. В каждой задаче освободите всю память, которую вы выделили.

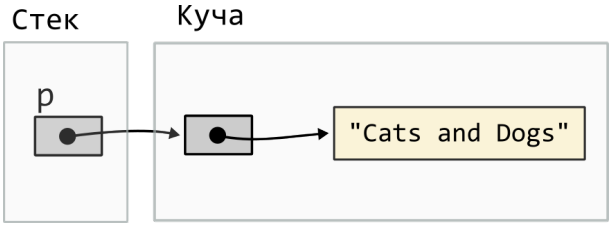
1. Одно число типа `size_t`.




2. Строка (массив из элементов типа `char`).

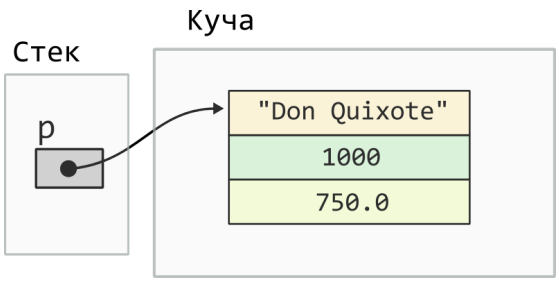



3. Указатель, указывающий на строку.

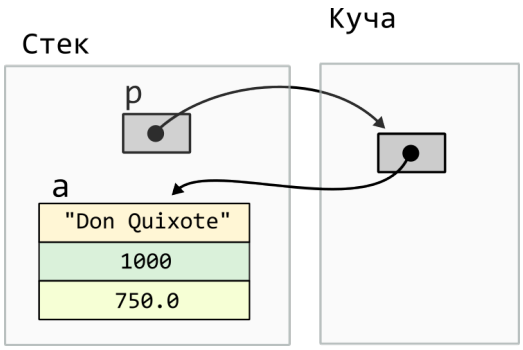



4. Структура типа `Book` из семинара про структуры. Для печати такой структуры можете использовать функцию `print_book` из семинара про структуры.

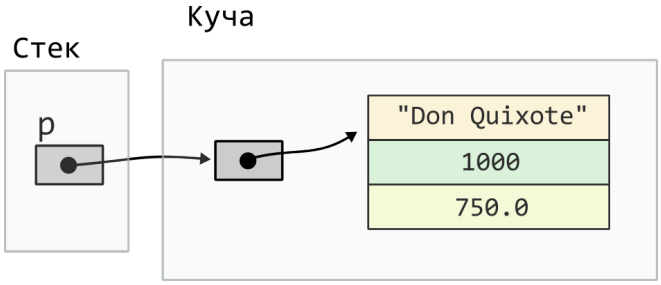
```c
struct book
{
    char title[50];
    int pages;
    float price;
};
typedef struct book Book;
```



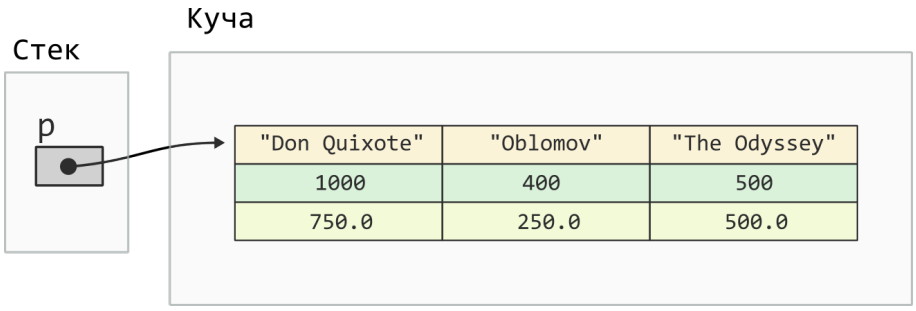
5. Указатель, указывающий на структуру на стеке.




6. Указатель, указывающий на структуру в куче.

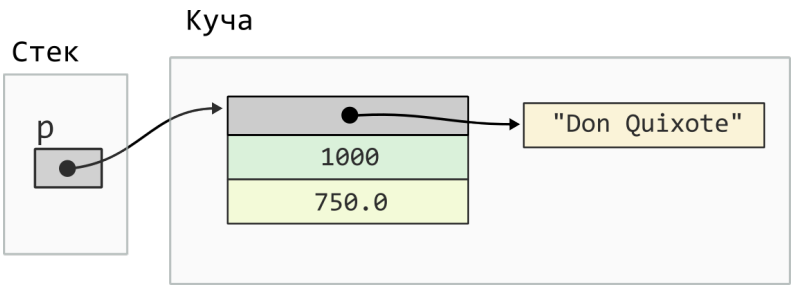



7. Массив структур.



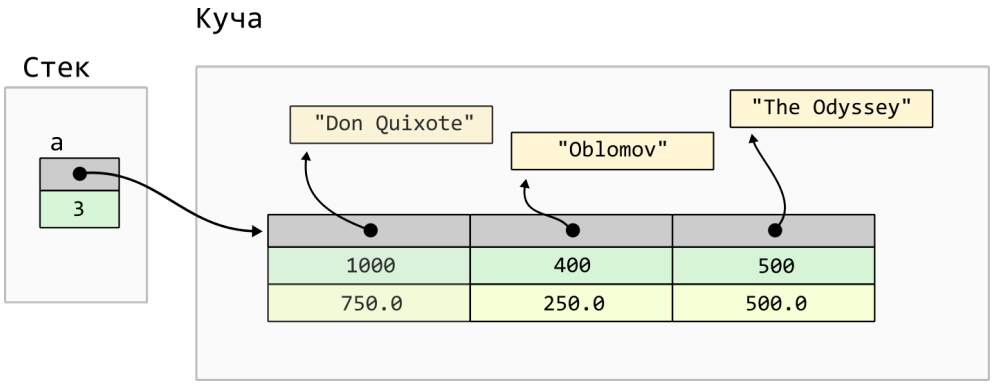

8. Видоизмените структуру `Book`, чтобы она, вместо строки, хранила указатель на строку в куче. Таким образом у нас не будет ограничения на длину названия. Создайте такую структуру в куче. Функцию `print_book` для такой структуры потребуется немного изменить.

```
struct book
{
    char* title;
    int pages;
    float price;
};
typedef struct book Book;
```



9. Создадим структуру `Library`, которая сможет хранить информацию о произвольном количестве книг.

```
struct library
{
    Book* books;
    int number_of_books;
};
typedef struct library Library;
```




Напишите функции для удобной работы с такой структурой:

- `library_create` - функция должна задавать поля `books` и `number_of_books` структуры `Library`. При этом данная функция должна выделять необходимое количество памяти.
- `library_set` - должна задавать значение `i`-й книги.
- `library_get` - принимает на вход индекс `i` и возвращает указатель на `i`-ю книгу.
- `library_print` - должна печатать все книги библиотеки на экран.
- `library_destroy` - должна освобождать всю память и устанавливать значения полей в `NULL` и `0` соответственно.

```
Library a;
library_create(&a, 3);
library_set(a, 0, "Don Quixote", 1000, 750.0);
library_set(a, 1, "Oblomov", 400, 250.0);
library_set(a, 2, "The Odyssey", 500, 500.0);
library_print(a);
print_book(library_get(a, 1));
library_destroy(&a);
```

Обратите внимание, что функции `library_create` и `library_destroy` должны принимать указатель на структуру `Library` так как они должны менять поля этой структуры.

## Задача 2. Геометрическая прогрессия

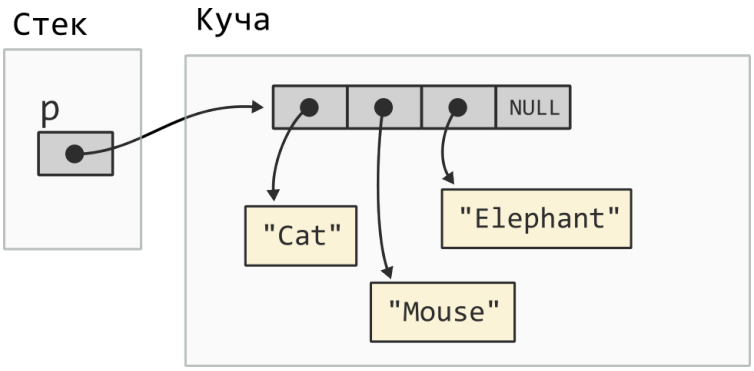
Напишите функцию `float* get_geometric_progression(float a, float r, int n)`, которая возвращает указатель на массив в куче, содержащий геометрическую прогрессию из $n$ чисел: $a, ar, ar^2, \ldots$ Память должна выделяться динамически. Вызовите эту функцию из `main` и напечатайте первые 10 степеней тройки.

## Задача 3. Конкатенация с выделением памяти

Напишите функцию `char* concat(const char* a, const char* b)`, которая принимает на вход две строки и возвращает новую строку, являющуюся конкатенацией первых двух. Память под новую строку функция должна выделять в куче.

## Задача 4. Сортировщик строк

Динамический массив строк – это двумерный динамический массив элементов типа `char`. Но с одной особенностью: размер строки задаётся не числом в отдельной переменной, а специальным нулевым символом на конце строки. Поэтому мы можем не хранить размер каждой строки.
Мы можем хранить количество строк в таком массиве в отдельной переменной, а можем просто выделить в массиве указателей на один элемент больше и хранить в этом элементе значение `NULL`. Таким образом мы можем найти количество строк в массиве.




1. Написать функцию `char** get_test_strings()` который будет создавать массив строк, представленный на рисунке выше, и возвращать его.
2. Написать функцию `void print_strings(const char** string_array)` который будет печатать переданный массив строк `string_array`.
3. Написать функцию `size_t* get_sizes(const char** string_array)` которая будет возвращать массив, содержащий размеры всех строк. Память под этот массив должна выделяться в куче.
4. Написать функцию `char** load_lines(const char* filename)`, которая будет создавать динамический массив строк, считывать из файла все строки в этот массив строк, и возвращать его. Один из способов как это можно сделать:
   - Пройдите по файлу с помощью `fgetc` и посчитайте количество строк в файле (количество `\n` + 1).
   - Выделите необходимую память в куче под массив указателей (учтите нулевой указатель в конце).
   - Вернитесь в начало файла с помощью `fseek`.
   - Пройдите файл заново, подсчитайте количество символов в каждой строке и сохраните эти данные во временном массиве в куче.
   - Выделите необходимую память для каждой строки (учтите нулевой символ в конце строк).
   - Вернитесь в начало файла с помощью `fseek`.
   - Пройдите файл заново, считайте и запишите каждую строку в свой элемент массива.
   - Освободите память под временный массив, закройте файл и верните результат.

   Это не совсем оптимальный способ, так как приходится 3 раза проходить по файлу. Если есть желание, можно написать более оптимальный алгоритм с использованием `realloc` и `fgets`.
5. Написать функцию `void destroy_strings(char*** p_string_array)`, которая будет уничтожать динамический массив строк. Эта функция должна освобождать всю память (память под каждую строку и память под массив указателей). Также эта функция должна присваивать указателю `p` значение `NULL`.

   ```
   char** p = load_lines("three_little_pigs.txt");
   destroy_strings(&p);
   ```
6. Написать функцию `void sort_strings(char** words)`, которая будет сортировать все строки по алфавиту. Используйте функцию `strcmp`.
7. Напишите программу `line_sorter`, которая будет считывать текстовый файл, сортировать строки этого файла по алфавиту и записывать результат в другой файл. Названия файлов функция должна принимать через аргументы командной строки. Пример использования такой программы:

   ```
   $ ./line_sorter invisible_man.txt result.txt
   ```

## Задача 5. Разные сегменты

В следующей программе на экран печатаются адреса различных объектов, созданных в программе.

```
#include <stdio.h>
#include <stdlib.h>

int a;
int b = 10;
const int c = 20;
void f()
{
    printf("hello f\n");
}
int main()
{
    int d;
    static int e;

    int* pi = (int*)malloc(sizeof(int) * 10);

    const char s[] = "cat";
    const char* ps = "dog";

    printf("&a   = %p\n", &a);
    printf("&b   = %p\n", &b);
    printf("&c   = %p\n", &c);
    printf("&d   = %p\n", &d);
    printf("&e   = %p\n", &e);
    printf("&pi  = %p\n", &pi);
    printf("pi   = %p\n", pi);
    printf("pi+5 = %p\n", pi + 5);
    printf("&s   = %p\n", &s);
    printf("s    = %p\n", s);
    printf("&ps  = %p\n", &ps);
    printf("ps   = %p\n", ps);
    printf("f    = %p\n", f);
    printf("&f   = %p\n", &f);
    printf("main   = %p\n", main);
    printf("\"fox\"  = %p\n", "fox");

    free(pi);
}
```

Нужно определить к какому сегменту соответствует каждый из печатаемых адресов. Варианты следующие: `stack`, `heap`, `text`, `data`, `rodata` и `bss`. Чтобы сдать эту задачу нужно создать файл в формате `.txt` и, используя любой текстовый редактор, записать в него ответы в следующем формате (ответы ниже неверны):

```
&a - stack
&b - heap
&c - text
&d - data
&pi - rodata
pi - bss
...
```

После этого, файл нужно поместить в ваш репозиторий на github.

## Задача 6. Сортировка с компараторами

Простой алгоритм сортировки, называемый сортировка выбором, можно написать следующим образом:

```
void sort(int* a, size_t n)
{
    for (size_t j = 0; j < n; ++j)
    {
        size_t min_index = j;
        for (size_t i = j + 1; i < n; ++i)
        {
            if (a[i] < a[min_index])
                min_index = i;
        }

        int temp = a[j];
        a[j] = a[min_index];
        a[min_index] = temp;
    }
}
```

Но эта функция сортирует числа только по возрастанию. Нужно изменить эту функцию так, чтобы она принимала на вход третий аргумент – компаратор. Компаратор должен представлять собой указатель на функцию типа:

```
int (*)(int a, int b)
```

Использование новой функции с компаратором должно выглядеть так:

```
#include <stdio.h>
#include <stdlib.h>

void print(int* a, size_t n)
{
    for (size_t i = 0; i < 6; ++i)
        printf("%i ", a[i]);
    printf("\n");
}
// Тут нужно написать новую функцию sort и функции - компараторы:
// less - сортировка по возрастанию
// greater - по убыванию
// last_digit_less - по возрастанию последней цифры

int main()
{
    int a[] = {32, 63, 29, 54, 81};

    sort(a, 5, less);
    print(a, 5); // Должен напечатать 29 32 54 63 81

    sort(a, 5, greater);
    print(a, 5); // Должен напечатать 81 63 54 32 29

    sort(a, 5, last_digit_less);
    print(a, 5); // Должен напечатать 81 32 63 54 29
}
```

## Задача 7. Подсчёт с помощью предиката

Напишите функцию `count_if`, которая будет подсчитывать сколько элементов в массиве удовлетворяют какому-либо условию. Условие должно задаваться путём передачи в функцию `count_if` указателя на другую функцию – так называемую функцию-предикат.

```
#include <stdio.h>
// Тут нужно написать функцию count_if и функции - предикаты:
// is_negative - проверяет, является ли число отрицательным
// is_even - проверяет, является ли число чётным
// is_square - проверяет, является ли число квадратом целого числа

int main()
{
    int a[] = {89, 81, 28, 52, 44, 16, -64, 49, 52, -79};

    printf("%zu\n", count_if(a, 10, is_negative));  // Должен напечатать 2
    printf("%zu\n", count_if(a, 10, is_even));      // Должен напечатать 6
    printf("%zu\n", count_if(a, 10, is_square));    // Должен напечатать 3
}
```

## Задача 8. Сумматор

Создайте функцию `adder`, которая будет принимать на вход число и возвращать сумму всех чисел, которые приходили на вход этой функции за время выполнения программы.

```
#include <stdio.h>
// Тут нужно написать функцию adder

int main()
{
    printf("%i\n", adder(10));  // Должен напечатать 10
    printf("%i\n", adder(15));  // Должен напечатать 25
    printf("%i\n", adder(70));  // Должен напечатать 95
}
```

## Задача 9. Когда выделяется память

В следующей программе выделяется $10^9$ байт в сегменте памяти BSS. Программа "видит" весь массив как доступный с начала выполнения. Но операционная система может сэкономить ресурсы и не выделять память сразу. Рассмотрим следующую программу:

```
#include <stdio.h>

#define SIZE 1'000'000'000
char data[SIZE];

int main()
{
    getchar();
    printf("1. Setting first char to 'A'\n");
    data[0] = 'A';

    getchar();
    printf("Setting last char to 'B'\n");
    data[SIZE - 1] = 'B';
```

```
    getchar();
    printf("Printing first char = %c\n", data[0]);

    getchar();
    printf("Printing last char = %c\n", data[SIZE - 1]);

    getchar();
    printf("Setting many chars to X\n");
    for (size_t i = 0; i < SIZE; i += 1000)
        data[i] = 'X';

    getchar();
    printf("Printing all set chars\n");
    for (size_t i = 0; i < SIZE; i += 1000)
        printf("%c ", data[i]);

    getchar();
}
```

a. Определить на каком шаге операционная система действительно выделяет ресурсы. Для того, чтобы сдать эту подзадачу, создайте файл `09a.txt` и опишите в нём на каком шаге происходит выделение памяти.

b. Перепишите программу так, чтобы память выделялась в куче и проверьте когда память будет выделяться в этом случае.Для того, чтобы сдать эту подзадачу, создайте файлы `09b.c`, в котором будет переписанная программа и файл `09b.txt` в котором будет описана на каком этапе будет на самом деле выделяться память в этом случае.

c. Перепишите программу так, чтобы память выделялась на стеке и проверьте что будет происходить в этом случае. Объясните поведение программы в файле `09c.txt`.

После этого, файлы нужно поместить в ваш репозиторий на github.