Семинар #12: Двумерный динамический массив. Связный список.

Часть 1: Изменение указателя внутри функции

Передача обычных переменных из функций

Чтобы лучше понять как передавать из функции указатель на данные, вспомним две возможности передачи из функции обычной переменной. Это можно сделать через возвращаемое значение или через аргумент-указатель.

Способ 1:
Просто возвращаем

```c
#include <stdio.h>
int f() {
    return 123;
}

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

Способ 2:
Передаём указатель на нашу переменную.
Изменяем переменную, используя указатель

```c
#include <stdio.h>
void f(int* p) {
    *p = 123;
}
int main() {
    int a;
    f(&a);
    printf("%i\n", a);
}
```

Передача указателя на данные в Куче из функции

Оба этих способа работают и для передачи из функции указателя. Только в этом случае тип возвращаемой переменной будет `int*`, а не `int`. Допустим, мы захотели написать функцию, которая создаёт в Куче массив из трёх элементов (`10, 20` и `30`) и передаёт указатель на этот массив.

Способ 1:

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

int* f() {
    int* p = malloc(sizeof(int) * 3);
    p[0] = 10;
    p[1] = 20;
    p[2] = 30;
    return p;
}
int main() {
    int* a = f();
    for (int i = 0; i < 3; ++i)
        printf("%i\n", a[i]);
    free(a);
}
```

Способ 2:

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

void f(int** pp) {
    *pp = malloc(sizeof(int) * 3);
    (*pp)[0] = 10;
    (*pp)[1] = 20;
    (*pp)[2] = 30;
}
int main() {
    int* a;
    f(&a);
    for (int i = 0; i < 3; ++i)
        printf("%i\n", a[i]);
    free(a);
}
```

Задача:

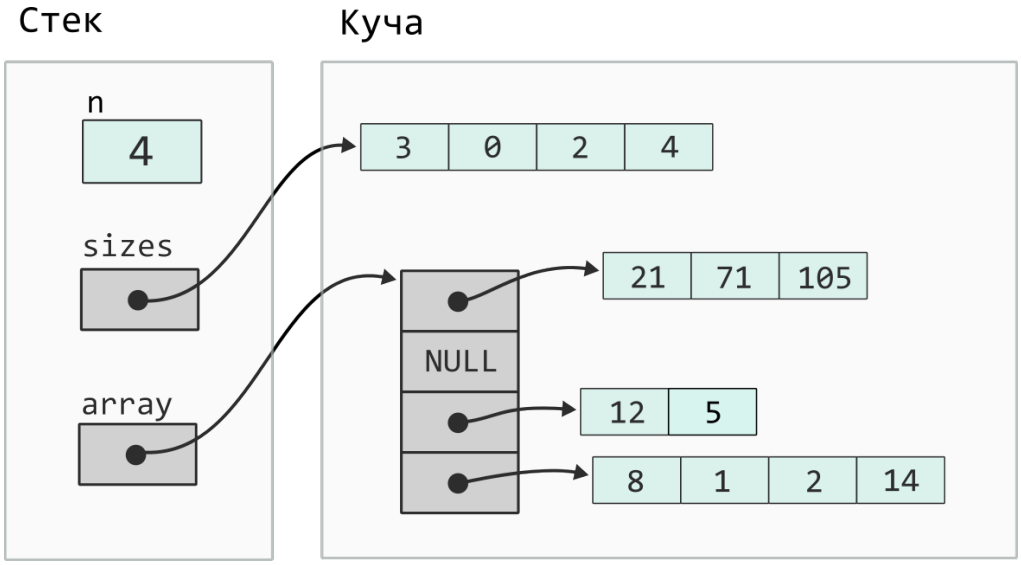
- Напишите 2 функции, которые будут выделять память и передавать указатель на массив в Куче, содержащий первые 10 квадратов натуральных чисел. Одна функция должна просто возвращать указатель, а вторая принимать адрес указателя и задавать его нужным значением.

Часть 2: Двумерный динамический массив

В стандартной библиотеке нет специальных средств по созданию двумерных массивов в Куче. Но есть 2 варианта для создания такого массива с использованием динамического выделения одномерных массивов:

1. Вместо создания двумерного динамического массива размером `n` на `m` можно создать одномерный динамический массив размера `n * m` и работать с ним. Это хороший вариант, когда длины всех строк массива равны или примерно равны и не меняются.

2. Создать динамический массив из указателей, каждый указатель будет соответствовать строке. Затем, для каждой строки динамически выделить столько памяти, сколько нужно. При этом нам нужно будет создать отдельный динамический массив(`sizes`), который будет хранить размеры каждой строки. Схема такого массива представлена на рисунке:



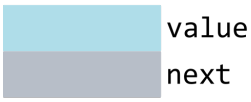
Задачи

- Напишите код, который будет выделять память в Куче и инициализировать её в соответствии со схемой на рисунке. Весь код должен быть в функции `main`.
- Напишите функцию `void print_two_dim_array(int n, int* sizes, int** array)` - печать.
- Напишите функцию `void inc_two_dim_array(int n, int* sizes, int** array)` - прибавить `1` ко всем элементам двумерного динамического массива.
- Напишите `void delete_two_dim_array(int n, int* sizes, int** array)` - освобождение памяти.
- Напишите функцию `void create_my_two_dim_array(int* p_n, int** p_sizes, int*** p_array)`, которая будет создавать в Куче двумерный динамический массив, инициализировать его в соответствии с рисунком и записывать значения `n`, `sizes` и `array` по передаваемым адресам `p_n`, `p_sizes` и `p_array` соответственно. Вызовите эту функцию из функции `main`.
- В файле `numbers.txt` содержится представление двумерного массива с разной длиной строк. В первой строке содержится число `n`, в следующей строке содержится массив `sizes` и в следующих `n` строках содежится сам массив. Напишите функцию
  `void load_two_dim_array(char filename[], int* p_n, int** p_sizes, int*** p_array)`,
  которая будет читать эти числа из файла под названием `filename` и записывать их в двуменый динамический массив. Эта функция сама должна выделять необходимую память.

Часть 3: Связный список

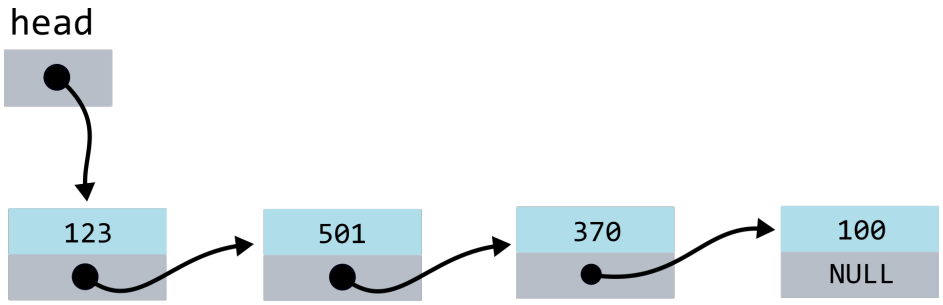
Создадим структуру `Node`, которая будет содержать:

- **Данные**: одно или несколько полей каких-угодно типов. В данном случае это 1 переменная `value`.
- **Связь**: указатель `next` на структуру того же типа `Node`.




```c
struct node {
    int value;
    struct node* next;
};
typedef struct node Node;
```

Используя такую структуру, можно создать Связный список:



Примечание: `NULL` - это просто константа равная 0. Её используют вместо нуля для указателей, чтобы различать числовые переменные и указатели.

В нашем случае указатель `head` будет зраниться в сегменте Стек, а все элементы будут выделяться динамически и, соответственно, храниться в сегменте Куча. Пустой связный список будет представлять собой просто один указатель `head`, равный `NULL`.

Вычислительные сложности операций со списком:

Операция	Массив	Односвязный список
Доступ по номеру	$O(1)$	$O(N)$
Поиск	$O(N)$	$O(N)$
Вставка в начало	$O(N)$	$O(1)$
Вставка в конец	$O(1)$	$O(N)$
Вставка в конец если известен указатель на последний элемент	$O(1)$	$O(1)$
Вставка в середину	$O(N)$	$O(N)$
Вставка в середину если известен указатель на предыдущий элемент	$O(N)$	$O(1)$

Чему равны вычислительные сложности следующих операций:

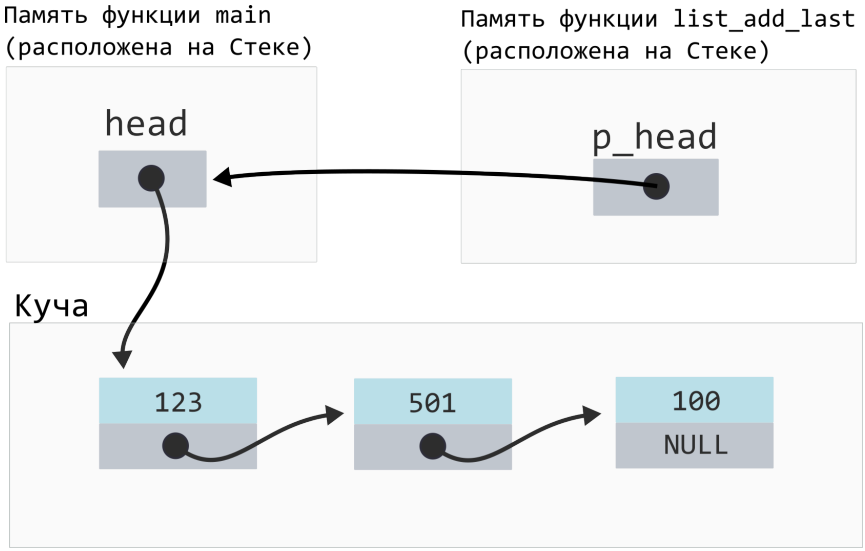
- Нахождение размера списка. Что можно сделать, чтобы нахождение размера выполнялось быстрее?
- Удаление элемента из начала списка.
- Удаление элемента из конца списка. Что нужно знать, чтобы эта операция выполнялась быстрее?
- Удаление элемента из середины списка. Что нужно знать, чтобы эта операция выполнялась быстрее?

Задание

Начальный код в файле `list.c`. Там уже написано 3 функции:

- `Node* list_create()` – инициализирует список (создаёт и возвращает список нулевого размера). Эта функция просто возвращает `NULL`. Зачем нужна такая простая функция? Она нужна для согласованности с реализациями других структур данных. Например, при реализации хеш-таблицы у нас будет более сложная функция `hashtable_create`, которая будет создавать хеш-таблицу.
- `void list_add_last(Node** p_head, int x)` – добавляет элемент x в конец списка. Также эта функция подробно разобрана в презентации.
- `void list_print(const Node* head, char separator[])` – распечатывает все элементы списка, разделённые строкой `separator`.

Можно заметить, что в одни функции передаётся сам указатель `head`, а в другие указатель на `head`, который называется `p_head`. Это связано с тем, что некоторые функции должны менять само значение `head` и, чтобы это сделать, нужно передать в функцию указатель на `head`. Например, если список пуст и `head == NULL`, то после вызова функции `list_add_last` указатель `head` должен измениться и указывать на новый элемент. Схематическое изображение выделенной памяти в случае передачи в функцию указателя на `head`:



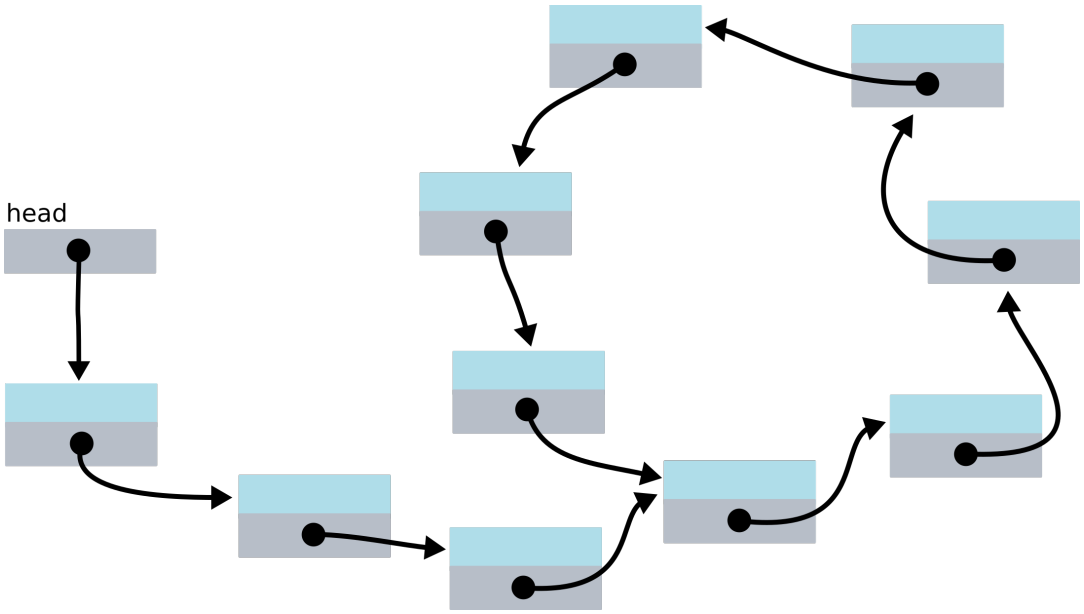
Напишите следующие функции для работы со связным списком:

- `void list_add_first(Node** p_head, int x)` – добавляет элемент x в начало списка. Чтобы добавить элемент, нужно для начала выделить необходимое количество памяти под этот элемент, затем задать поля нового элемента таким образом, чтобы он указывал на начало списка. В конце нужно поменять значение указателя `head`, используя `p_head`.
- `int list_remove_first(Node** p_head)` – удаляет элемент из начала списка и возвращает его значение. Не забудьте изменить `*p_head`.
- `int list_remove_last(Node** p_head)` – удаляет элемент из конца списка и возвращает его значение.
- `size_t list_size(const Node* head)` – возвращает количество элементов списка.
- `void list_destroy(Node* head)` – освобождает всю память, выделенную под список. Так как память выделялась под каждый элемент отдельно, то освобождать нужно также каждый элемент по отдельности.
- `void list_reverse(Node** p_head)` – переворачивает связный список. Первый элемент становится последним, а последний первым. В данной задаче вам не нужно перемещать элементы `value` или сами структуры. Нужно просто изменить указатели. Эта задача разобрана в презентации.
- `void list_concatenate(Node** p_head1, const Node* head2)`, которая добавляет второй связный список в конец первого. Особенно рассмотрите случай когда первый список – пуст.

- Добавить `typedef`-синоним для элементов связного списка. (функция для печати списка при этом может стать недействительной, так как там используется спецификатор `%i`).
- Вынести всю реализацию связного списка в отдельный файл `list.h`. Подключить этот файл к вашему `.c` файлу.
- Реализовать абстрактный тип данных стек(Stack) на основе связного списка. Реализация должна находится в файле `stack.h`.

Цикл в связном списке:

Если, вдруг, последний элемент связного списка имеет поле `next`, которое не равно нулю, а содержит указатель на один из элементов списка, то в списке образуется цикл. На изображении показано как это выглядит.



- Написать функцию `Node* create_my_looped_list()`, которая будет создавать в куче связный список, состоящий из трёх элементов (`10`, `20` и `30`). При этом указатель `next` последнего элемента должен указывать на первый элемент. Что будет если передать такой связный список в функцию `list_print`.
- Написать функцию `int list_is_loop(Node* head)`, которая проверяет, если в связном списке цикл. Функция должна вернуть `1` если в списке есть цикл и `0`, если его нет.
- Написать функцию `void list_fix_loop(Node* head)`, которая проверяет, если в связном списке цикл. И если цикл есть, то она размыкает его. То есть устанавливает поле `next` последнего элемента значением `NULL`.