Семинар #7: Память. Классные задания.

Часть 1: Системы счисления

Мы привыкли пользоваться десятичной системой счисления и не задумываемся, что под числом в десятичной записи подразумевается следующее:

$$123.45_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

Конечно, в числе 10 нет ничего сильно особенного с математической точки зрения. Оно было выбрано исторически, скорей всего по той причине, что у человека 10 пальцев. Компьютеры же работают с двоичными числами, потому что оказалось что процессоры на основе двоичной логики сделать проще. В двоичной системе счисления есть всего 2 цифры: `0` и `1`. Под записью числа в двоичной системе подразумевается примерно то же самое, что и в десятичной:

$$101.01_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 5.25_{10}$$

При работе с компьютером на низком уровне имеет смысл использовать двоичную систему за место десятичной. Но человеку очень сложно воспринимать числа в двоичной записи, так как они получаются слишком длинными. Поэтому популярность приобрели восьмеричная и шестнадцатиричная системы счисления. В шестнадцатиричной системе счисления есть 16 цифр: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f`.

$$1a.8_{16} = 1 \cdot 16^1 + 10 \cdot 16^0 + 8 \cdot 16^{-1} = 26.5_{10}$$

Задача. Переводите следующие числа в десятичную систему:

- 11011_2
- $2b_{16}$
- 40_8
- 1.1_2
- $a.c_{16}$
- 10_{123}

Шестнадцатиричная и восьмеричная системы в языке `C`:

Язык `C` поддерживает шестнадцатиричные и восьмеричные числа. Чтобы получить восьмеричное число нужно написать `0` перед числом. Чтобы получить шестнадцатиричное число нужно написать `0x` перед числом.

```c
#include <stdio.h>
int main() {
    int a = 123;    // Десятичная система
    int b = 0123;   // Восьмеричная система
    int c = 0x123;  // Шестнадцатиричная система
    printf("%i %i %i\n", a, b, c);
}
```

Также, можно печатать и считывать числа в этих системах счисления с помощью спецификаторов `%o` (для восьмеричной системы – **o**ctal) и `%x` (для шестнадцатеричной – he**x**adecimal). Спецификатор `%d` можно использовать для десятичной системы – **d**ecimal. Пример программы, которая считывает число в шестнадцатеричной системе и печатает в десятичной:

```c
#include <stdio.h>
int main() {
    int a;
    scanf("%x", &a);
    printf("%d\n", a);
}
```

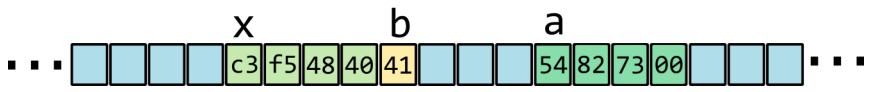
Задача:

Напишите программу, которая будет считывать число в десятичной системе и печатать в шестнадцатеричной. Переведите с помощью этой программы в шестнадцатеричную систему числа: `14`, `255`, `256`, `65535`, `14598366`.

Часть 2: Переменные в памяти

Положение любой переменной в памяти характеризуется двумя числами: её адресом(номером первого байта этой переменной) и её размером. Рассмотрим ситуацию, когда были созданы 3 переменные типов `int` (размер 4 байта), `char` (размер 1 байт) и `float` (размер 4 байта). На рисунке представлено схематическое расположение этих переменных в памяти (одному квадратику соответствует 1 байт):

```
int a = 0x738254;
char b = 'A';
float x = 3.14;
```



Какие выводы можно сделать из этого изображения:

- Каждая переменная заняла столько байт, чему равен её размер.
- Переменные в памяти могут хранится не в том порядке, в котором вы их объявляете.
- Переменные в памяти хранятся не обязательно вплотную друг к другу.
- Каждый байт памяти представляется двузначным шестнадцатиричным числом (это удобно).
- Байты переменной `a` хранятся в обратном порядке. Такой порядок байт называется `Little Endian`. Обратите внимание, что обращается только порядок байт, а не бит. Большинство компьютеров применяют именно такой порядок байт. Но в некоторых системах может использоваться обычный порядок байт – `Big Endian`. Обратный порядок байт применяется не только к типу `int`, но и ко всем базовым типам.
- Переменная `b` хранит ASCII-код символа `A`. Он который равен $65 = 41_{16}$.

Каждый байт памяти занумерован и номер первого байта переменной называется её адресом. В 64-х битных системах адрес – это 64-битное число и может принимать значения от `0` и до `0xffffffffffffffff`. Это адресное пространство предоставляется операционной системой и не соответствует адресному пространству физической памяти. Поэтому, например, адрес переменной может быть больше чем общее количество оперативной памяти.

Операторы получения адреса и размера переменных

Чтобы найти адрес переменной, нужно перед ней поставить знак амперсанда `&`. Для печати адреса используется спецификатор `%p`, хотя можно использовать и `%llx` или `%llu`. Чтобы найти размер переменной нужно использовать оператор `sizeof`. Размер это обычно тоже 8-ми байтовое беззнаковое число (потому что размер какой-нибудь структуры может быть очень большим), поэтому нужно использвать `%llu`.

```c
#include <stdio.h>
int main() {
    int a = 123;
    printf("Address = %p\n", &a);
    printf("Size = %llu\n", sizeof(a));
}
```

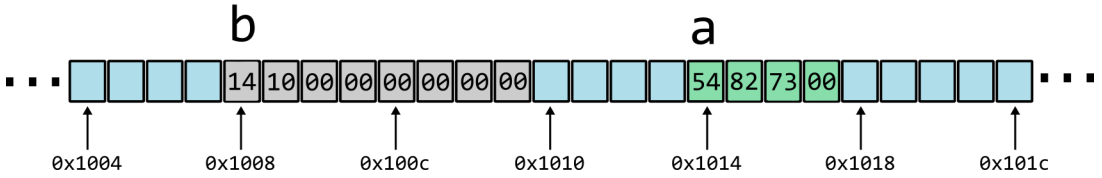
Задачи:

- Найдите чему равны размеры следующих типов на вашей системе: `char`, `short`, `long`, `long long`, `float`, `double`, массив чисел `int` размером 100 элементов.
- Объявите 2 переменные `int` и напечатайте их адреса в шестнадцатеричном и десятичном виде. Чему равна разница между этими числами? Убедитесь, что адреса переменных разные при каждом запуске программы.

Часть 3: Указатели

Для хранения адресов в языке C введены специальные переменные, которые называются указатели. Тип переменной указателя = тип той переменной, чей адрес он хранит + звёздочка на конце. Например, указатель, который будет хранить адреса переменных типа `int` должен иметь тип `int*`.

```
int a = 0x738254;
int* b = &a
```



Пояснения по рисунку:

- В данном примере для простоты выбраны очень маленькие адреса. В действительности же адрес скорей всего будет очень большим числом.
- Указатель тоже является переменной и хранится в памяти.
- Указатель хранит номер одной из ячеек памяти (в данном случае – первый байт `a`).
- Для указателя применяется тот же порядок байт, что и для других переменных базовых типов. В данном случае – обратный.

Задачи:

- Для примера выше напечатайте следующие величины:
  - Значение, адрес и размер переменной `a`
  - Значение, адрес и размер переменной `b` (указатель, хранящий адрес `a`)
- Напечатайте размеры следующих типов: `char*`, `short*`, `int*`, `long long*`, `float*`, `double*`, `int**`.

Операция разыменования:

Разыменования – это получение самой переменной по указателю на неё. Чтобы разыменовать указатель нужно перед ним поставить звёздочку. Не следует путать эту звёздочку со звёздочкой, используемой при объявлении указателя. То есть, если `b` это указатель, хранящий адрес `a`, то `*b` означает следующее:

Пройди по адресу, хранящемуся в `b`. Возьми соответствующее количество байт, начиная с этого адреса. Воспринимай эти байты как переменную соответствующего типа.

```c
#include <stdio.h>
int main() {
    int a = 123;
    int* b = &a;
    *b += 1;
    printf("%d\n", a);
}
```

Задачи:

- Умножьте значение переменной `a` на 10 используя только указатель на неё.
- Возведите значение переменной `a` в квадрат используя только указатель на неё.

Схематическое изображение указателей в памяти:

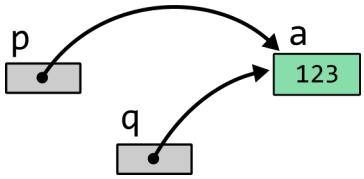
Так как постоянно рисовать переменные в памяти слишком громоздко и затруднительно, будем изображать из схематически. Стрелочкой будем указывать на переменную, адрес которой хранит указатель. Размеры прямоугольников не соответствуют размерам переменных. Пример выше тогда будет выглядеть так:

Задачи:

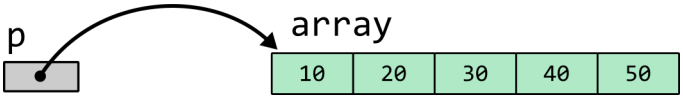
Напишите код, который будет соответствовать следующим рисункам. В каждой задаче разыменуйте указатели и напечатайте то, на что они указывают.

- Указатель на переменную типа `char`.

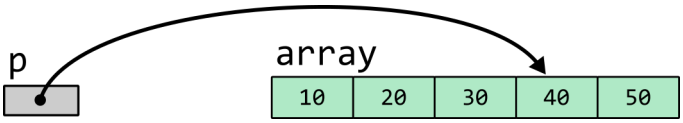
- Два указателя, которые указывают на одну переменную типа `int`



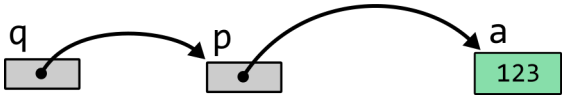
- Указатель типа `int*`, указывает на первый элемент массива `int`-ов под названием `array`



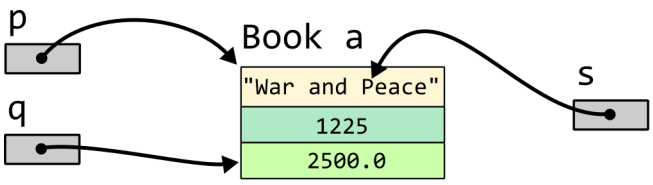
- Указатель типа `int*`, указывает на четвёртый элемент массива `int`-ов под названием `array`



- Указатель типа `int**`, указывает на указатель `int*`, который указывает на переменную типа `int`.



- Пусть есть структура `Book` из задания на структуры. Она содержит поля `title` (массив `char`), `pages` (тип `int`) и `price` (тип `float`). Создайте переменную структуры и 3 указателя `p`, `q` и `s`. Указатель `p` должен указывать на саму структуру. Указатель `q` должен указывать на поле `price`. Указатель `s` должен указывать на символ `'P'` строки `"War and Peace"`.



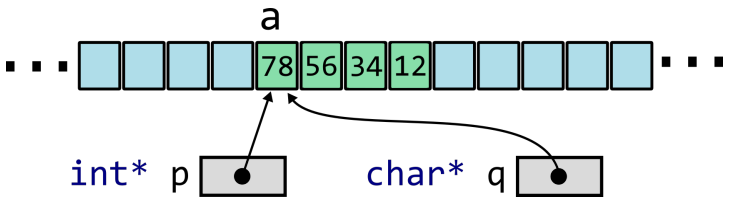
Часть 4: Указатели разных типов

Как вы могли заметить тип указателя зависит от типа элемента на который он указывает. Но все указатели, независимо от типа, по сути хранят одно и то же (адрес первого байта переменной). Чем же они различаются друг от друга? Разница проявляется как раз при их разыменовывании . Например, при разыменовывании указатель `int*` берёт 4 байта и воспринимает их как переменную типа `int`, а указатель `char*` берёт 1 байт и воспринимает его как переменную типа `char`.

Рассмотрим следующий пример. На переменную `a` указывают две переменные разных типов: `int*` и `char*`. Оба указателя хранят одно и то же значение, но работают по разному при разыменовании.

```c
#include <stdio.h>
int main() {
    int a = 0x12345678;
    int* p = &a;
    char* q = &a;

    printf("%p %p\n", p, q);
    printf("%x\n", *p);
    printf("%x\n", *q);
}
```



Преобразование типов указателя

В предыдущем примере есть такая строка `char* p = &a;` Необычность этой строки в том, что слева и справа от знака `=` находятся объекты разных типов. Слева – `char*`, а справа – `int*`. В этот момент происходит неявное преобразование типов один тип указателя преобразуется в другой. Это всё похоже на преобразование типов обычных переменных.

```c
int a = 4.1;          // Неявное преобразование из double в int
int b = (int)4.1;     // Явное преобразование из double в int

char* p = &a;         // Неявное преобразование из int* в char*  ( не работает в C++ )
char* p = (char*)&a; // Явное преобразование  из int* в char*
```

Надо отметить, что язык C++ строже относится к соблюдению типов, чем язык C, и не позволит вам неявно преобразовать указатель одного типа в указатель другого типа.

Задача: Что напечатает следующая программа и почему она это напечатает?

```c
#include <stdio.h>
int main() {
    int a = 7627075;
    char* p = (char*)&a;
    printf("%s\n", p);
}
```

Указатель `void*`

Помимо обычных указателей в языке есть специальный указатель `void*`. Этот указатель не ассоциирован не с каким типом, а просто хранит некоторый адрес. При попытке его разыменования произойдёт ошибка.

Задача:

```c
int a = 123;
void* p = (void*)&a;
```

Увеличьте переменную `a` в 2 раза и напечатайте её используя только указатель `p`.

Часть 5: Арифметика указателей

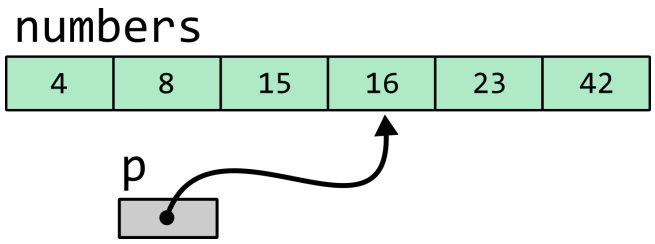
С указателями можно производить следующие операции:

- Разыменование `*p`
- Инкремент `p++`. В этом случае указатель не увеличивается на `1`, как было можно подумать. Он увеличивается на размер типа, на который он указывает. Благодаря этой особенности указателей с их помощью удобно проходить по массиву.
- Декремент `p++`. Уменьшается на размер типа, на который он указывает.
- Прибавить или отнять число `p + k`. В этом случае указатель не увеличивается на `k`, как было можно подумать. Он увеличивается на `k * sizeof(*p)`. Благодаря этой особенности указателей с их помощью удобно проходить по массиву. Если `p` указывает на `i`-ый элемент массива, то `p + 1` будет указывать на `i + 1` элемент массива.
- Вычитать 2 указателя `p - q`. Вернётся разница между указателями делённая на размер типа указателя.
- Квадратные скобки (прибавить число + разыменование): `p[i] == *(p+i)`

Задачи:

- Пусть есть одномерный статический массив и указатель на 4-й элемент этого массива:

```
int numbers[6] = {4, 8, 15, 16, 23, 42};
int* p = &numbers[3];
```



Чему равны следующие выражения:

1. `numbers[5]`
2. `*p`
3. `*(p+1)`
4. `*(p-2)`
5. `p[0]`
6. `p[1]`
7. `p[-2]`
8. `*numbers`
9. `*(numbers+5)`
10. `p - numbers`
11. `(short*)p - (short*)numbers`
12. `(char*)p - (char*)numbers`

Подсказка: имя массива во многих случаях ведёт себя как указатель на первый элемент массива.

- Обход массива с помощью указателя:

```
#include <stdio.h>
int main() {
    int numbers[6] = {4, 8, 15, 16, 23, 42};
    for (int* p = &numbers[0]; p != &numbers[6]; ++p) {
        printf("%i\n", *p);
    }
}
```

Используйте такой обход, но с указателем `char*`, чтобы напечатать каждый байт массива `numbers` в шестнадцатеричном виде.

Часть 6: Динамическое выделение памяти в Куче:

Основные функции для динамического выделения памяти:

- `void* malloc(size_t n)` – выделяет `n` байт в сегменте памяти Куча и возвращает указатель `void*` на начало этой памяти. Если память выделить не получилось (например памяти не хватает), то функция вернёт значение `NULL`. (`NULL` – это просто константа равная нулю).

- `void free(void* p)` – освобождает выделенную память. Если ненужную память вовремя не освободить, то она останется помеченной, как занятая до момента завершения программы. Произойдёт так называемая утечка памяти.

- `void* realloc(void* p, size_t new_n)` – перевыделяет выделенную память. Указатель `p` должен указывать на ранее выделенную память. Память, на которую ранее указывал `p`, освободится.
  Если память перевыделить не получилось (например памяти не хватает), то функция вернёт значение `NULL`. При этом указатель `p` будет продолжать указывать на старую память, она не освободится.

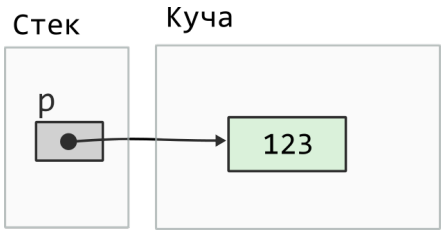
Давайте рассмотрим как работать с этими функциями. Предположим, что мы хотим создать на куче одну переменную типа `int`. Так как мы знаем, что `int` занимает 4 байта, то мы можем написать следующее.

```
int* p = (int*)malloc(4);
```

Обратите внимание на то что мы привели указатель `void*`, который возвращает `malloc`, к указателю `int*`.
В языке `C` такое приведение можно не писать, а в языке `C++` это обязательно. Однако, такое использование `malloc` не совсем верно, так как тип `int` не всегда имеет размер 4 байта. На старых системах он может иметь размер 2 байта, а на очень новых – даже 8 байт. Поэтому лучше использовать оператор `sizeof`.

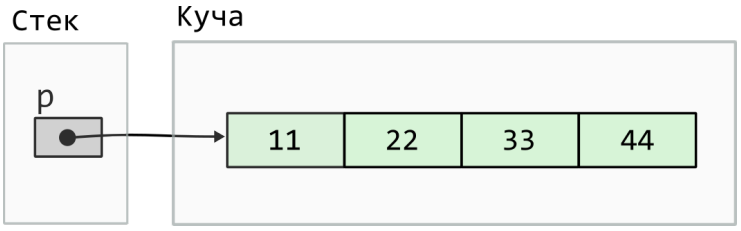
Схематически выделение одного `int`-а в куче можно изобразить следующим образом:

```
int* p = (int*)malloc(sizeof(int));
*p = 123;
```



Конечно, основное преемущество кучи это её размер, который ограничен только доступной физической памятью. Поэтому на куче обычно выделяют не одиночные переменные, а массивы. Вот схематическое изображение выделения массива из 4-х элементов на куче:

```
int* p = (int*)malloc(4 * sizeof(int));
p[0] = 11;
p[1] = 22;
p[2] = 33;
p[3] = 44;
```



Благодаря тому, что к указателям можно применять квадратные скобки, работа с указателем `p` ничем не отличается от работы с массивом размером в 4 элемента.
После того как вы поработали с памятью в куче и она стала вам не нужна, память нужно освободить так:

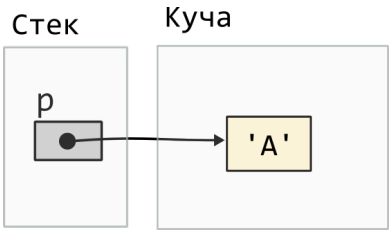
```
free(p);
```

Если это не сделать, то выделенные в куче объекты будут занимать память даже когда они уже перестали быть нужны. Эта память освободится только при завершении программы. Правило при работе с `free`: число вызовов `free` должно быть равно числу вызовов `malloc`.

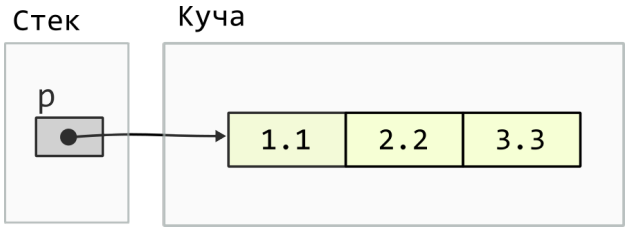
Задачи:

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.

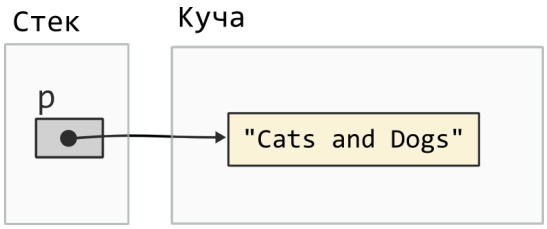
- Один символ.



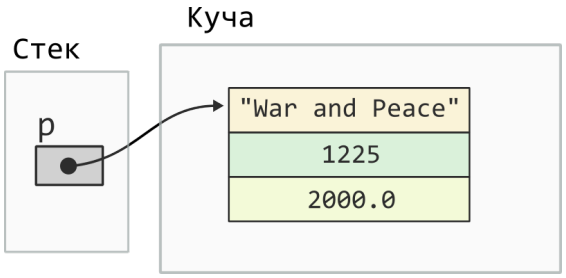
- Массив из трёх элементов типа `double`.



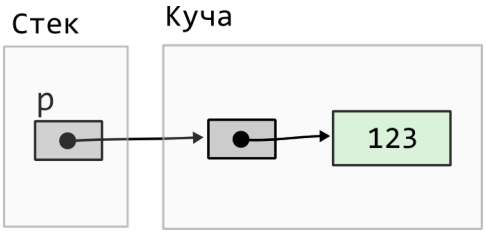
- Строку (массив `char`) `"Cats and Dogs"`. Чему должен быть равен размер массива символов? Для присваивания значения строке используйте функцию `strcpy`.



- Структуру `Book` из семинара на структуры. Для присваивания значения строке используйте `strcpy`.

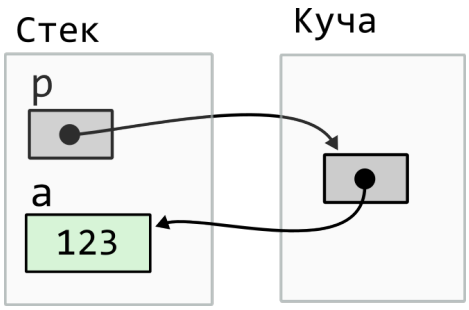


- Указатель, который указывает на число `int`.

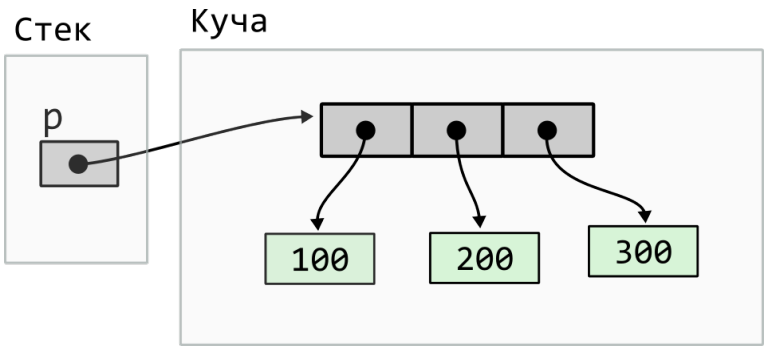


  В этом случае нужно использовать 2 вызова `malloc` и 2 вызова `free`.

- Указатель, который указывает на число `int`, которое находится в стеке.



- Массив из указателей на `int`.

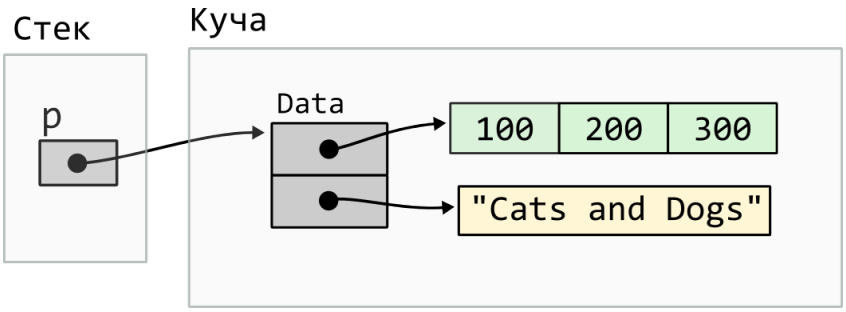


  В этой задаче нужно использовать 4 вызова `malloc` и 4 вызова `free`.

- Пусть есть структура, которая хранит 2 указателя: `numbers` и `symbols`:

```
struct data {
    int* numbers;
    char* symbols;
};
typedef struct data Data;
```

  Используйте эту структуру, чтобы выделить память в куче следующим образом:



  В этой задаче нужно использовать 3 вызова `malloc` и 3 вызова `free`.

Часть 7: Ошибки при использовании динамического выделения памяти

Утечки памяти

Если вы забудите освободить память с помощью `free`, когда она перестанет быть нужна, то программа будет использовать больше памяти чем нужно. Произойдёт так называемая утечка памяти. Если в программе есть утечки памяти, то с течением времени она будет потреблять всё больше и больше памяти. При завершении программы всё память, конечно, освобождается.

```
#include <stdlib.h>
void func(int n) {
    int* p = (int*)malloc(n * sizeof(int));
    // ...
}

int main() {
    func(10000);
    // После выполнения функции func мы не сможем освободить память, даже если захотим
    // так как не знаем указатель на начало этой памяти

    // При каждом вызове функции будет тратиться память
    for (int i = 0; i < 100; ++i) {
        func(10000);
    }
}
```

Существуют специальные программы, которые проверяют нет ли у вас в программе утечек памяти. Одна из таких программ – `valgrind` на ОС семейства `Linux`. Чтобы её использовать, нужно просто написать в терминале:

```
valgrind ./a.out
```

- Протестируйте программу в файле `code/memory_leak.cpp` с помощью `valgrind`.
- Исправьте утечку памяти в той программе и снова протестируйте её с помощью `valgrind`.

Ошибка при выделении памяти

Если при вызове `malloc` произошла какая-либо ошибка, например, вы просите больше памяти, чем осталось, то `malloc` вернёт нулевой указатель равный `NULL` (то есть `0`). Поэтому при каждом вызове `malloc` желательно проверять, сработал ли он корректно:

```
int* p = (int*)malloc(1000 * sizeof(int));
if (p == NULL) {
    printf("Error! Out of memory.\n");
    exit(1);
}
```

Второе освобождение той же памяти

```
int* p = (int*)malloc(1000 * sizeof(int));
free(p);
free(p);
```