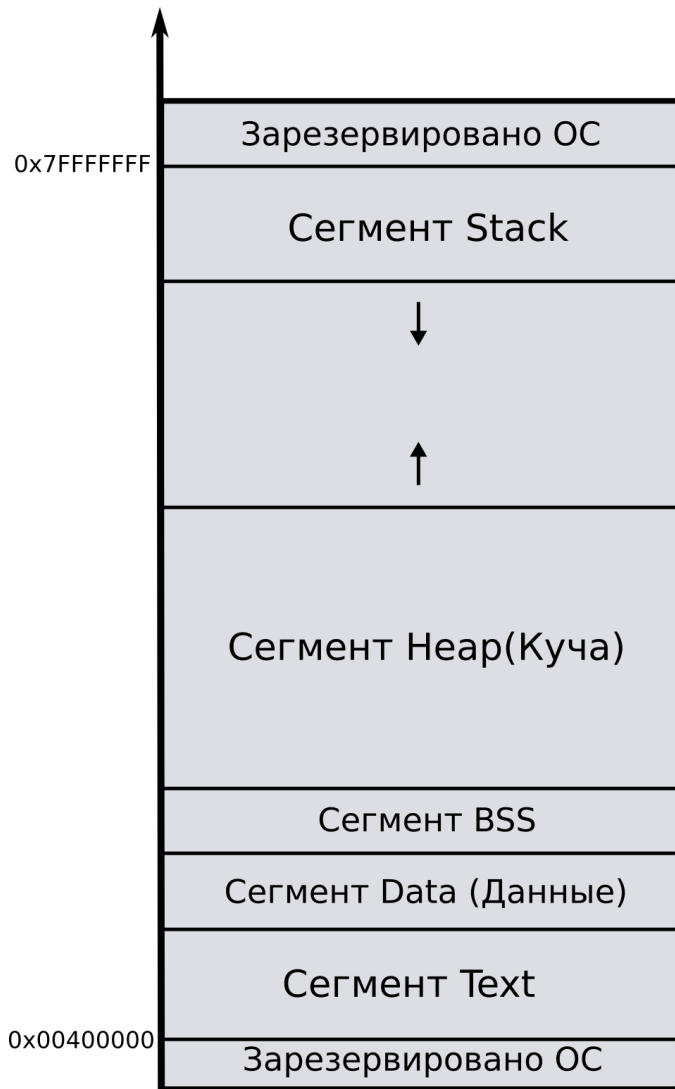


Семинар #10: Сегменты памяти. Классные задания.

Часть 1: Сегменты памяти



1. Сегмент памяти Стек (Stack)

- При обычном объявлении переменных (в том числе массивов) внутри функций все они создаются в стеке:

```
int a;  
int array[10];
```

- Память на локальные переменные функции выделяется при вызове этой функции и освобождается при завершении функции.
- Маленький размер (несколько мегабайт, зависит от настроек операционной системы).
- Выделение памяти происходит быстрее чем в куче

2. Сегмент памяти Куча (Heap)

- Выделить память в куче можно с помощью стандартной функции `malloc`.

```
int* p = malloc(10 * sizeof(int));
```

- Освободить память в куче можно с помощью стандартной функции `free`

```
free(p);
```

- Память можно выделяется/освобождать в любом месте.
- Размер ограничен свободной оперативной памятью.
- Выделение памяти происходит медленней чем в стеке

3. Сегмент памяти Data

- В этом сегменте хранятся инициализированные глобальные и статические переменные а также строковые литералы

4. Сегмент памяти BSS

- В этом сегменте хранятся неинициализированные глобальные и статические переменные
- В большинстве систем все эти данные автоматически инициализируются нулями

5. Сегмент памяти Text

- В этом сегменте хранится машинный код программы (Код на языке C, сначала, переводится в код на языке Ассемблера, а потом в машинный код. Как это происходит смотрите ниже.).
- Адрес функции - адрес первого байта инструкций в этом сегменте.

Создание массива в разных сегментах памяти

Ниже представлен пример программы в которой создаются 4 массива в разных сегментах памяти.

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

int array_data[5] = {1, 2, 3, 4, 5};
int array_bss[5];

int main()
{
    int array_stack[5];
    int* array_heap = (int*)malloc(5 * sizeof(int));
}
```

- Напечатайте адрес начала каждого из массивов. Помните, что для печати адресов используется спецификатор %p.

Переполнение стека – Stackoverflow

- Определите размер стека на вашей системе экспериментальным путём. Создайте массив такого большого размера на стеке, чтобы перестала работать. Минимальный размер массива, при котором падает программа будет примерно равен размеру стека.
- При каждом вызове функции в стеке хранятся локальные переменные функции, аргументы функции а также адрес возврата функции. Даже функция без локальных переменных и аргументов будет хранить на стеке как минимум адрес возврата (8 байт). Определите размер стека на вашей системе экспериментальным путём с помощью рекурсии.

Статические переменные

Помимо глобальных переменных, в сегменте Data хранятся статические переменные. Такие переменные объявляются внутри функций, но создаются в сегменте Data и не удаляются при завершении функции. Вот пример функции со статической переменной.

```
#include <stdio.h>
void counter()
{
    static int n = 0;
    n++;
    printf("%i\n", n);
}
int main()
{
    counter();
    counter();
    counter();
}
```

Обратите внимание, что в этой функции строка `static int n = 0;` не исполняется при заходе в функцию. Эта строка просто объявляет инициализирует статическую переменную, причём инициализация происходит в самом начале исполнения программы (даже до функции `main`).

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

```
printf("%i\n", adder(10)); // Напечатает 10
printf("%i\n", adder(15)); // Напечатает 25
printf("%i\n", adder(70)); // Напечатает 95
```

Часть 2: 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.

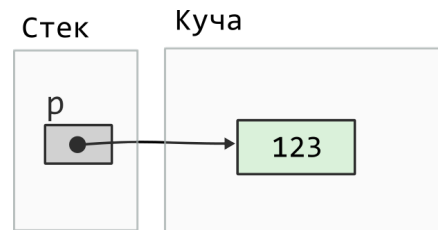
Давайте рассмотрим как работать с этими функциями. Предположим, что мы хотим создать на куче одну переменную типа `int`. Так как мы знаем, что `int` занимает 4 байта, то мы можем написать следующее.

```
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`.

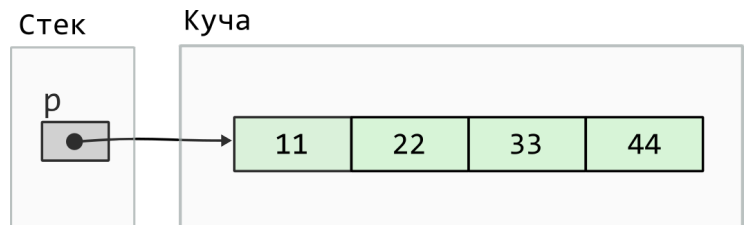
Схематически выделение одного `int`-а в куче можно изобразить следующим образом:

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



Конечно, основное преимущество 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:

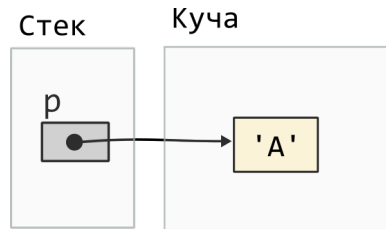
```
free(p);
```

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`.

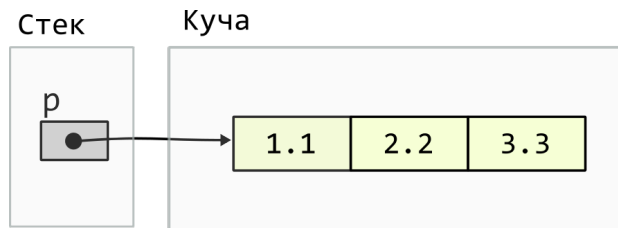
Задачи:

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.

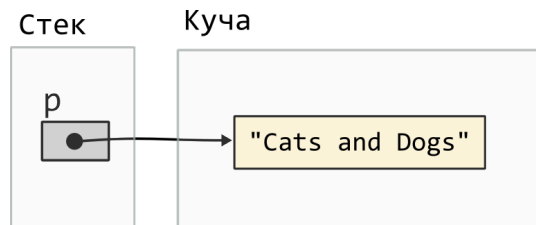
- Один символ.



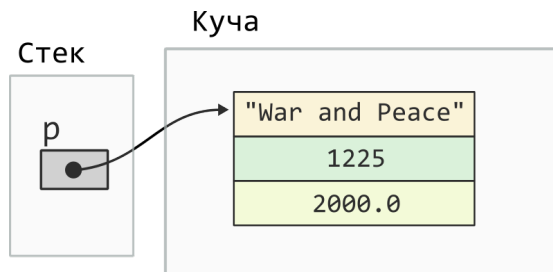
- Массив из трёх элементов типа double.



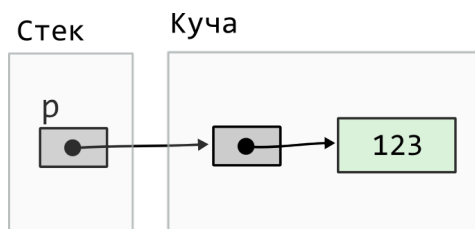
- Строку (массив char) "Cats and Dogs". Чему должен быть равен размер массива символов? Для присваивания значения строке используйте функцию `strcpy`.



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

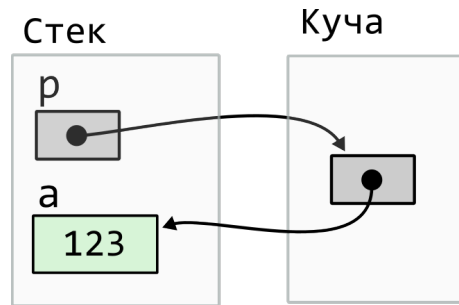


- Указатель, который указывает на число `int`.

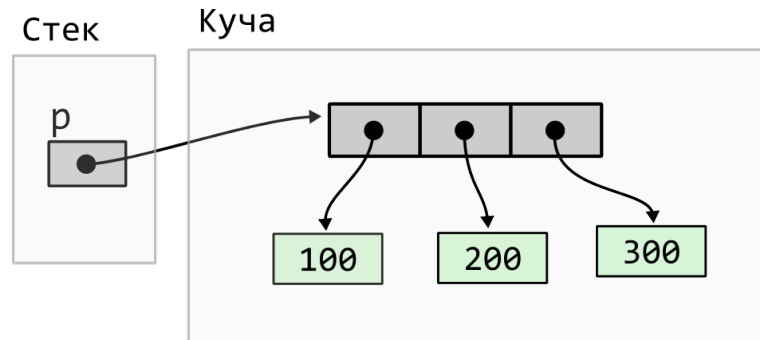


В этом случае нужно использовать 2 вызова `malloc` и 2 вызова `free`.

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



- Массив из указателей на `int`.

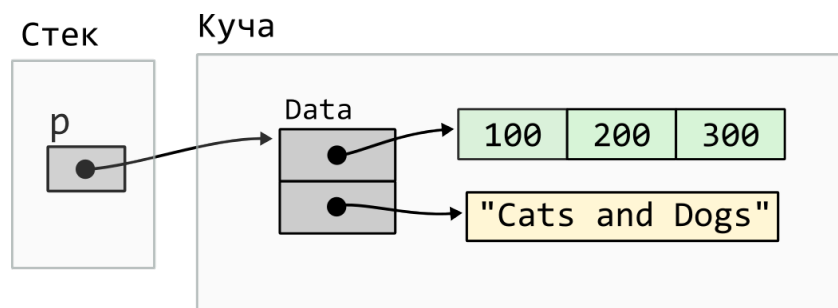


В этой задаче нужно использовать 4 вызова `malloc` и 4 вызова `free`.

- 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 3: Ошибки при использовании динамического выделения памяти

Утечки памяти

Если вы забудете 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);
}
```

Повторное освобождение той же памяти

Если вы попытаете освободить уже освобождённый участок памяти

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