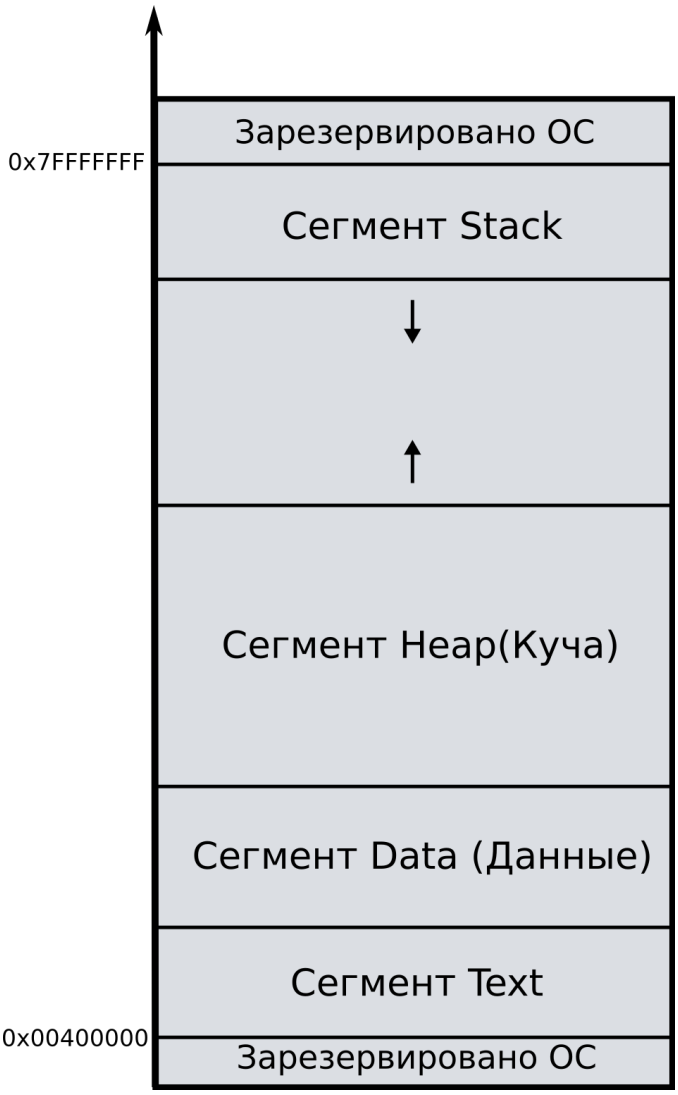
Семинар #8: Сегменты памяти. Классные задания.

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
   - Увеличивается в сторону меньших адресов.

2. **Сегмент памяти Куча (Heap)**

   - Выделеть 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.
   - Можно выделить намного больше памяти, чем в стеке.
   - Выделение памяти происходит медленней чем в стеке

3. **Сегмент памяти Data**

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

4. **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.).
   - Адрес функции - адрес первого байта инструкций в этом сегменте.

Часть 2: Динамическое выделение памяти в Куче:

Основные функции для динамического выделения памяти содержатся в библиотеке `stdlib.h`:

- `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.

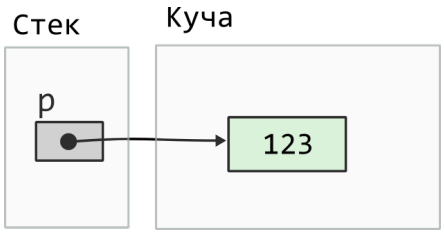
Давайте рассмотрим как работать с этими функциями. Предположим, что мы хотим создать в куче одну переменную типа `int`. Так как мы знаем, что `int` занимает 4 байта, то мы можем написать следующее.

```
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`.

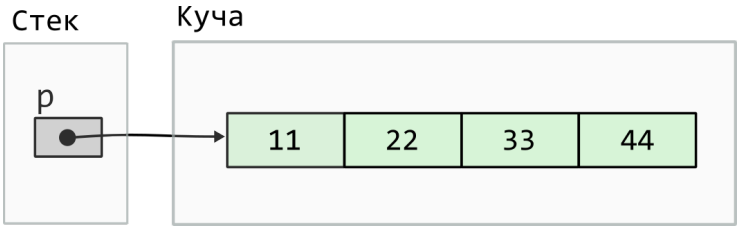
Схематически выделение одного `int`-а в куче можно изобразить следующим образом:

```
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:

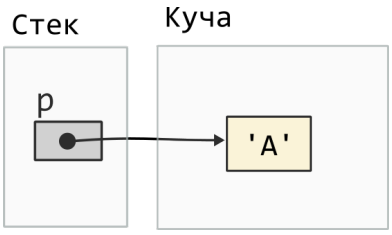
```
free(p);
```

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`.

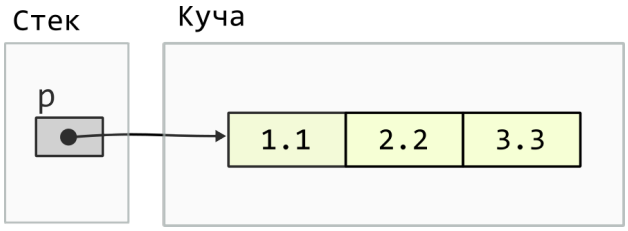
Задачи:

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.

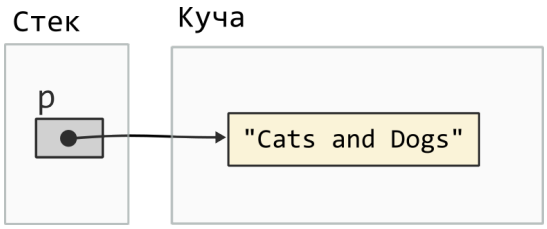
1. Один символ.



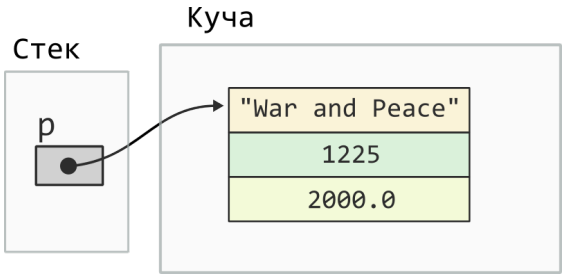
2. Массив из трёх элементов типа `double`.



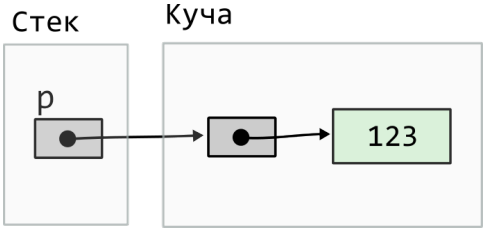
3. Строку (массив `char`) `"Cats and Dogs"`. Чему должен быть равен размер массива символов? Для присваивания значения строке используйте функцию `strcpy`.



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

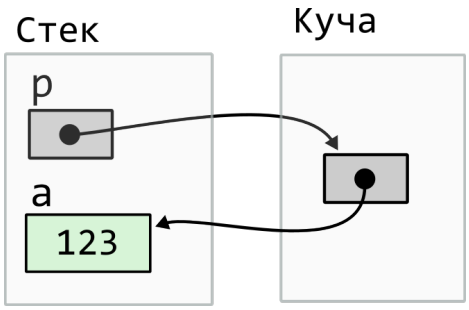


5. Указатель, который указывает на число `int`.

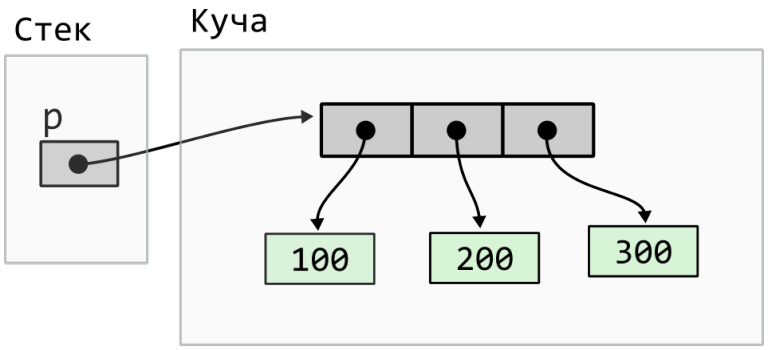


   В этом случае нужно использовать 2 вызова `malloc` и 2 вызова `free`.

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



7. Массив из указателей на `int`.

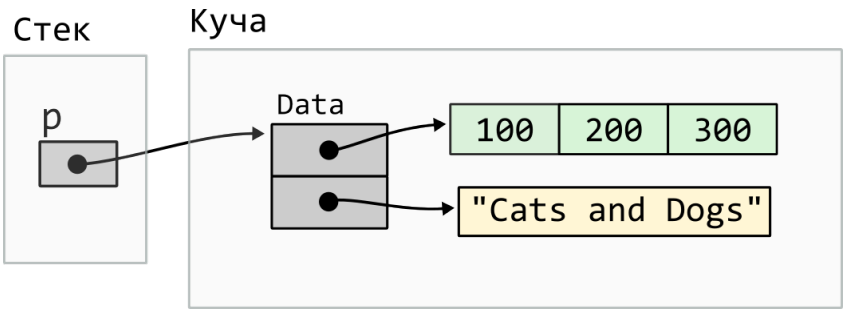


В этой задаче нужно использовать 4 вызова `malloc` и 4 вызова `free`.

8. 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. Но при завершении программы вся память, конечно, освобождается.

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

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
```

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 попробуете освободить уже освобождёнынй участок памяти

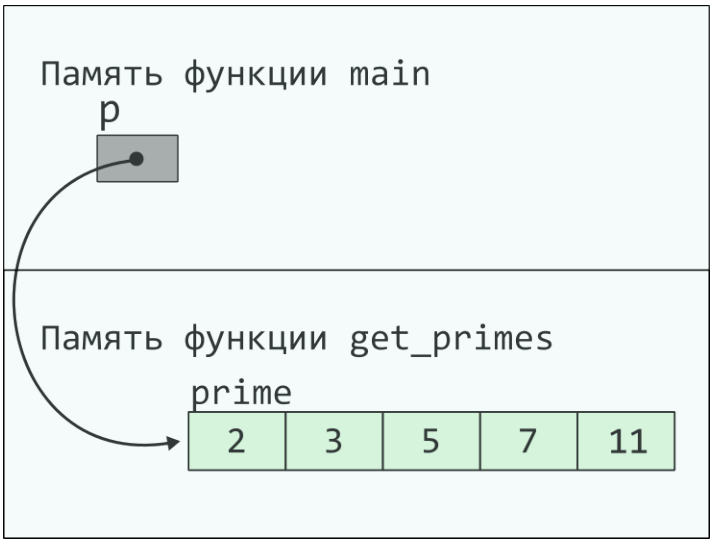
```
int* p = (int*)malloc(1000 * sizeof(int));
free(p);
free(p);
```

Часть 4: Возврат массива из функции

Возврат массива, созданого на стеке, из функции (ошибочный способ)

Допустим мы хотим создать функцию, которая должна будет возвращать массив. Известно, что при передаче массива в функцию он всегда передаётся по указателю. Можно попытаться создать статический массив внутри функции и вернуть указатель на него как это сделано в примере ниже. Однако это является грубой ошибкой. Дело в том, что память на стеке выделяется в начале выполнения функции и освобождается по выходу из функции. Таким образом, возвращаемый указатель будет ссылаться на уже освобождённую память. Ошибка!

```c
#include <stdio.h>
int* get_primes()
{
    int primes[5] = {2, 3, 5, 7, 11};
    int* result = &primes[0];
    return result;
}
int* other_function()
{
    int arr[5] = {55, 66, 77, 88, 99};
    int* result = &arr[0];
    return result;
}
int main()
{
    int* p = get_primes();
    // other_function();
    printf("%i\n", p[4]);
}
```

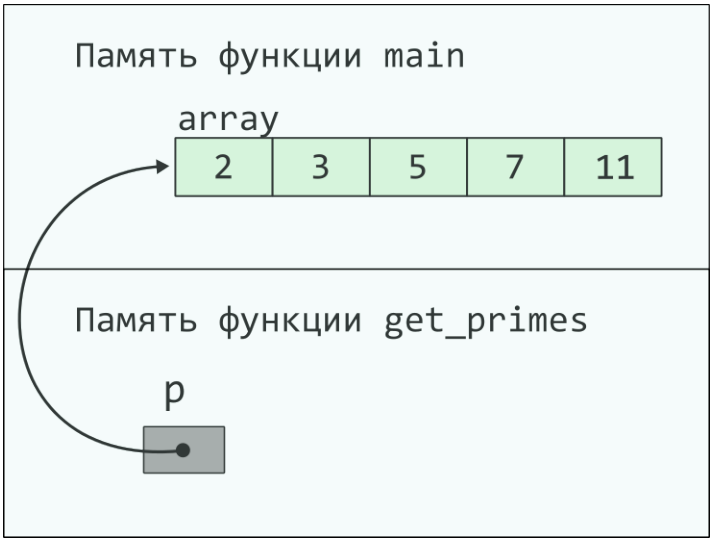


Что напечатает программа, если раскомментить строку?

Возврат массива, созданого на стеке, из функции (через аргумент)

Другой способ “возврата” массива: передадим функции указатель на уже созданный массив и попросим его заполнить. При этом нужно следить, чтобы функция не вышла за пределы массива. Так мы передавали массивы в функции в предыдущих семинарах.

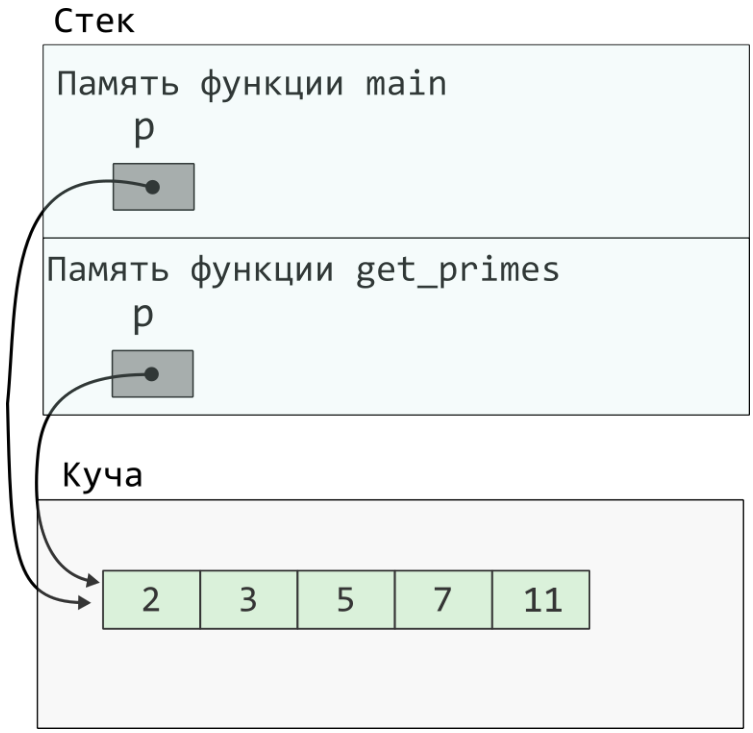
```c
#include <stdio.h>
void get_primes(int* p)
{
    p[0] = 2;
    p[1] = 3;
    p[2] = 5;
    p[3] = 7;
    p[4] = 11;
}
int main()
{
    int array[100];
    get_primes(array);
    for (int i = 0; i < 5; ++i)
        printf("%i ", array[i]);
}
```



Возврат массива, созданого в куче, из функции

Наконец, ещё один, новый способ возврата массива из функции – это создание массива в куче с помощью `malloc` и возвращение указателя на него. При завершении функции, выделенная в куче память не освобождается и этот массив можно использовать. Только нужно не забыть вызвать `free`, когда массив станет не нужен.

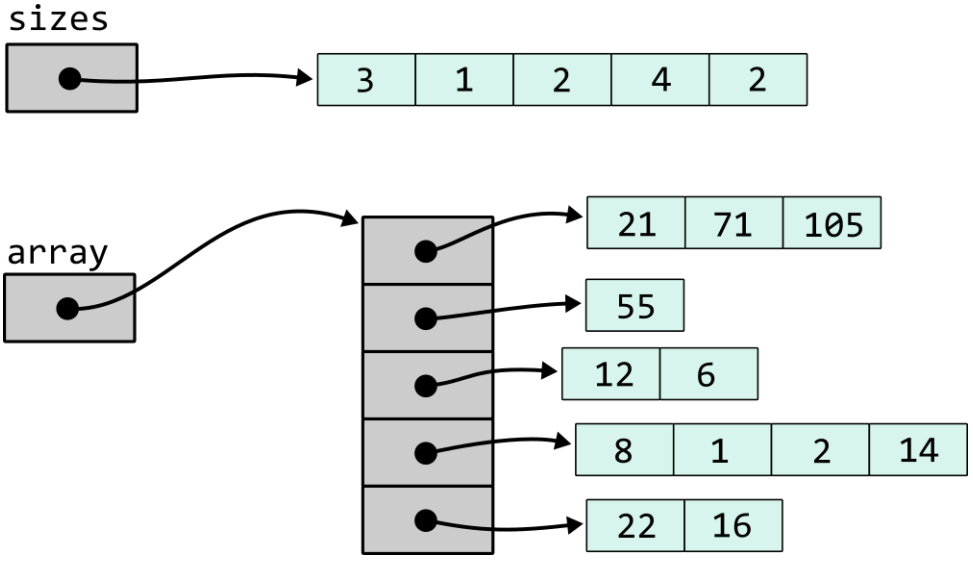
```c
#include <stdlib.h>
#include <stdio.h>

int* get_primes()
{
    int* p = (int*)malloc(5*sizeof(int));
    p[0] = 2;
    p[1] = 3;
    p[2] = 5;
    p[3] = 7;
    p[4] = 11;
    return p;
}
int main()
{
    int* p = get_primes();
    for (int i = 0; i < 5; ++i)
        printf("%i ", p[i]);
    free(p);
}
```



Часть 5: Двумерный динамический массив

В стандартной библиотеке нет специальных средств по созданию двумерных динамических массивов. Есть 2 варианта для создания такого массива:

1. 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 массив(`sizes`), который будет хранить размеры каждой строки.



Часть 6: Сегмент памяти Данные

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 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.

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 внмание, 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 7: Сегмент памяти Текст. Указатели на функцию.

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.

Указатели на функции

Пример работы с указателем на функцию:

```c
#include <stdio.h>

void print(int a)
{
    printf("%d\n", a);
}
int main ()
{
    // Создадим указатель на функцию ( вместо названия функции - *p )
    void (*p)(int a) = print;

    // Теперь с p можно работать также как и с print
    p(123);
}
```

Подробней в файле `funcpointers/0funcpointer.c`. **Задачи на указатели на функцию:**

- В файле `funcpointers/1foreach.c` лежит заготовка исходного кода. Вам нужно написать функцию `void foreach(int* array, int size, int (*f)(int))`, которая будет принимать на вход массив размера `size` и применять к каждому элементу функцию `f`.
- В файле `funcpointers/2foreach_second_argument.c` лежит заготовка исходного кода. Вам нужно написать функцию `void foreach(int* array, int size, int (*f)(int, int), int b)`, которая будет принимать на вход массив размера `size` и применять к каждому элементу функцию `g(x) = f(x, b)`.

Стандартная функция qsort

В библиотеке `stdlib.h` уже реализована функция `qsort`, которая сортирует произвольные элементы, используя быструю сортировку. Пример использования этой функции:

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

int cmp(const void* a, const void* b)
{
    // В этот компаратор передаются указатели на void,
    // Поэтому их нужно привести в нужный нам тип:
    int* pa = (int*)a;
    int* pb = (int*)b;
    return (*pa - *pb);
}

int main()
{
    int arr[] = {163, 624, 7345, 545, 41, 78, 5, 536, 962, 1579};
```

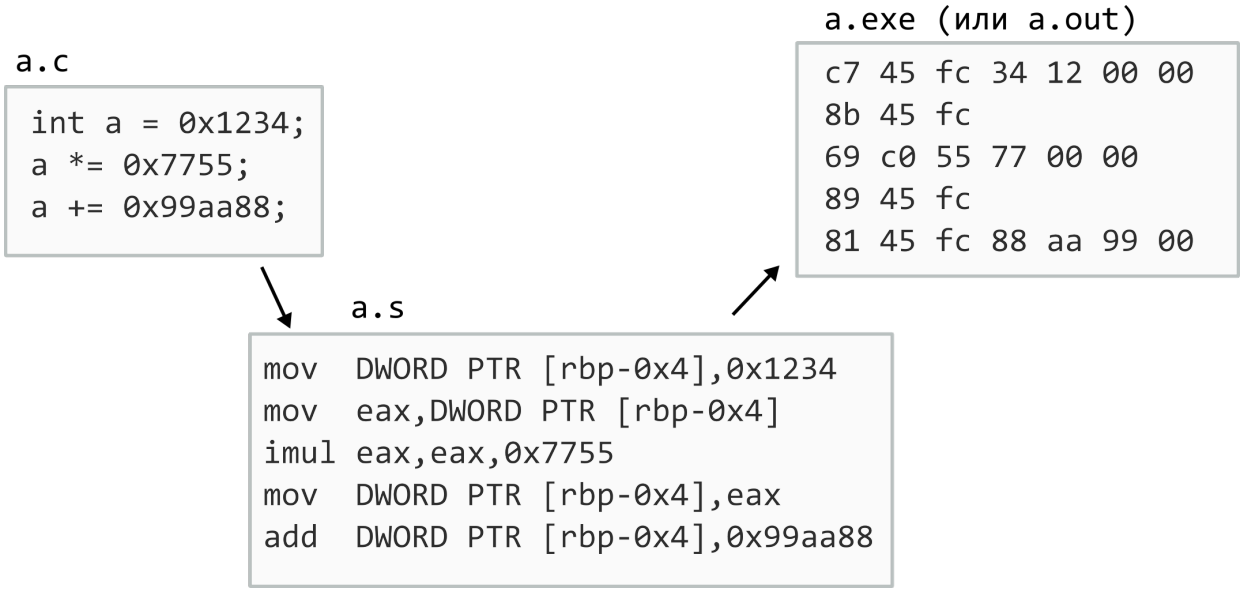
```
    qsort(arr, 10, sizeof(int), cmp);
    // qsort( массив, количество элементов, размер каждого элемента, компаратор )
    // Функция принимает на вход указатель на функцию cmp

    print_array(10, arr);
}
```

Функция-компаратор стандартной функции `qsort` отличается от той, что была написана нами для сортировки городов и звёзд только тем, что она принимает на вход указатели типа `void*`. Это сделано для того, чтобы эта функция была более общей. С помощью неё можно отсортировать как массив чисел, так и массив указателей или массив любых структур. В функции `cmp` нужно привести указатель `void*` к указателю нужного типа.

Задача на стандартную функцию `qsort`:

- Перепишите сортировку звёзд с использованием функции `qsort`.

Часть 8: Как код превращается в последовательность байт.



Из кода на C в код ассемблера:

- Код на языке `C` (`a.c`) переводится в код на языке ассемблера (`a.s`). Эту операцию можно сделать командой

  ```
  gcc -S -masm=intel ./a.c
  ```

- Регистры процессора – это сверхбыстрая память, которая находится внутри процессора. Её размер очень мал(десятки байт), но процессор может доступиться к ней очень быстро (за 1 такт). В примере выше используются 2 регистра: `rbp` и `eax` (`eax` это часть регистра `rax`).

- Процессор может делать множество различных операций. Например, он может переместить некоторое количество байт из одного места в другое. Такие операции называются `mov`. Он может прибавить число (`add`) или умножить на целое (`imull`) и многое другое. `DWORD PTR` просто означает, что операция будет работать с 4-мя байтами.

- В примере выше в регистре `rbp` содержится некоторый адрес. Квадратные скобочки означают разыменование. Поэтому строка

  ```
  mov DWORD PTR [rbp-0x4],0x1234
  ```

  означает, что нужно положить число `0x1234` в 4 байта по адресу `rbp-0x4`

- `mov eax,DWORD PTR [rbp-0x4]`

  означает, что нужно переместить 4 байта, которые хранятся по адресу `rbp-0x4` в регистр `eax`.

- `imull eax,eax,0x7755`

  означает, что нужно умножить содержимое `eax` на `0x7755` и сохранить результат в `eax`.

- `mov  DWORD PTR [rbp-0x4],eax`

  означает, что нужно переместить содержимое `eax` в память по адресу `rbp-0x4`.

- `add  DWORD PTR [rbp-0x4],0x99aa88`

  означает, что нужно добавить к числу по адресу `rbp-0x4` число `0x99aa88`.

- В отличии от кода на языке `C`, код на языке ассемблера различаться на разных процессорах. Код с вычислительной системы одной архитектуры скорей всего не будет работать на другой.

Из кода ассемблера в бинарный код (`.exe`):

- Код на языке ассемблера (`a.s`) переводится в исполняемый файл. Эту операцию можно сделать командой `gcc a.s`
- Каждая операция кодируется некоторым числом, называемым кодом операции (opcode).
- Код операции `mov` на процессорах архитектуры `x86-64` может равняться `c7` или `8b` или `89` или некоторым другим значениям(в зависимости от того куда и откуда мы копируем).
- Например в строке:

  `c7 45 fc 34 12 00 00`

  - `c7` означает, что это операция `mov` (присвоить число переменной в памяти)
  - `45` кодирует регистр `rbp`
  - `fc` кодирует смещение `-0x4`
  - `34 12 00 00` – это 4-х байтовое число `0x1234` (порядок байт – Little Endian)
- `8b 45 fc`

  - `8b` означает, что это операция `mov` (записать число, хранящееся в памяти, в `eax`)
  - `45` кодирует регистр `rbp`
  - `fc` кодирует смещение `-0x4`
- Все коды можно посмотреть тут ref.x86asm.net/coder64.html
- Получается, что в результате компиляции программы код превращается в последовательность байт (инструкций процессора). Эта последовательность байт и хранится в сегменте Текст.
- А указатель на функцию является просто номером первого байта, с которого начинается функция в этом сегменте.
- Менять сегмент Текст во время выполнения программы в большинстве современных операционных систем нельзя.