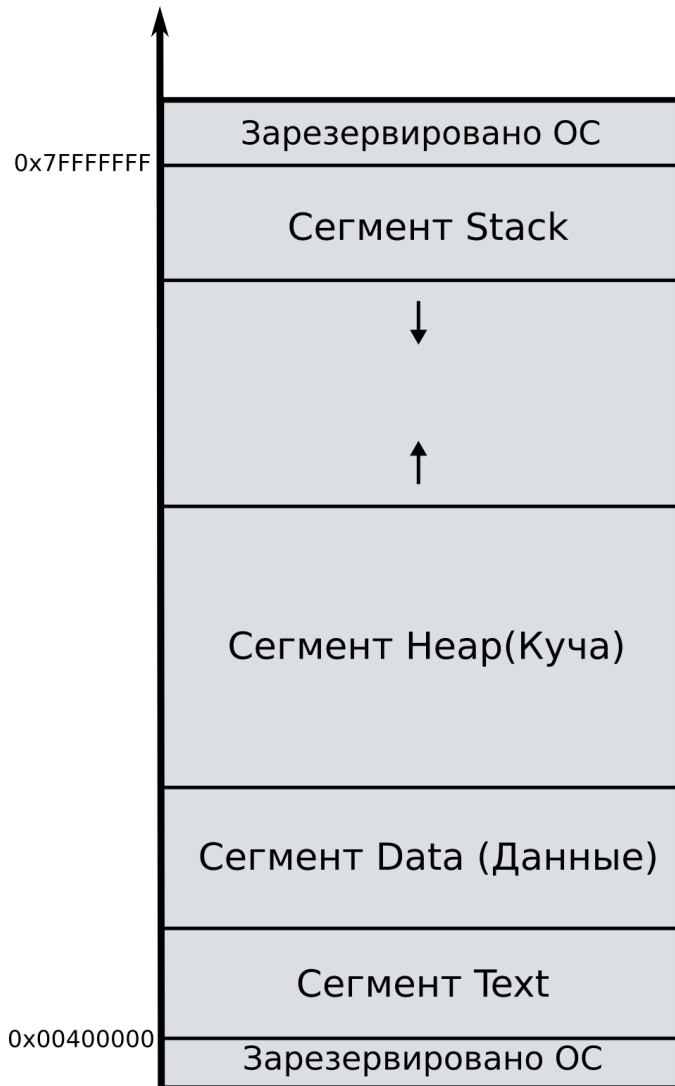


Семинар #5: Сегменты памяти.

Обзор основных сегментов 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);
```

- Можно выделить намного больше памяти, чем в стеке. Как правило, можно использовать всю свободную память системы.
- Выделение памяти происходит медленнее чем в стеке.

3. Сегмент памяти Данные (Data)

- В этом сегменте хранятся глобальные и статические переменные а также строковые литералы.
- Можно выделить намного больше памяти, чем в стеке. Как правило, можно использовать всю свободную память системы.
- Выделение памяти происходит при запуске программы, а освобождение при завершении.
- Также часто отдельно выделяют сегменты `BSS` для не инициализированных данных и `Rodata` для константных данных и строковых литералов.

4. Сегмент памяти Текст (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.

Сегмент памяти Куча

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

- `void* malloc(size_t n)` – выделяет `n` байт в сегменте памяти куча и возвращает указатель типа `void*` на начало этой памяти. Если память выделить не получилось (например памяти не хватает), то функция вернёт значение `NULL`.
- `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. Правило при работе с `free`: число вызовов `free` должно быть равно числу вызовов `malloc`.
- `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.

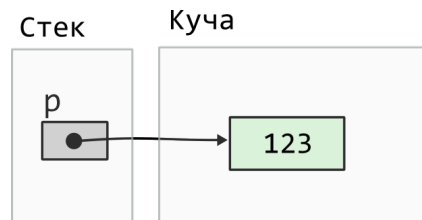
Давайте рассмотрим как работать с этими функциями. Предположим, что мы хотим создать в куче одну переменную типа `int`. Так как мы знаем, что `int` занимает 4 байта, то мы можем написать следующее:

```
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`.

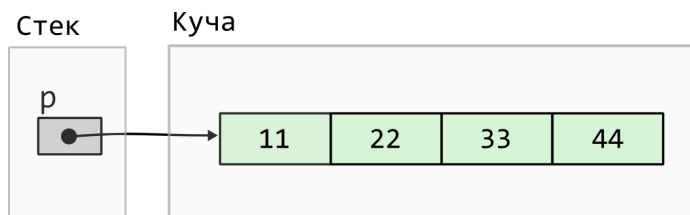
Схематически выделение одного `int`-а в куче можно изобразить следующим образом:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int* p = (int*)malloc(sizeof(int));
    *p = 123;
    printf("%i\n", *p);
    free(p);
}
```



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:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int* p = (int*)malloc(4 * sizeof(int));
    p[0] = 11;
    p[1] = 22;
    p[2] = 33;
    p[3] = 44;
    printf("%i\n", p[2]);
    free(p);
}
```



Благодаря тому, что к указателям можно применять квадратные скобки, работа с указателем `p` ничем не отличается от работы с массивом размером в 4 элемента.

Обработка ошибок 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`. 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);
}
```

В учебных примерах мы это делать не будем, чтобы код оставался коротким и более понятным, однако при разработке на языках C и C++ крайне важно делать эту проверку.

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 написать в терминале:

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

Повторное освобождение той же памяти

Если вы попытаетесь освободить уже освобождённую память, это приведёт к неопределённому поведению:

```
int* p = (int*)malloc(1000 * sizeof(int));
free(p);
// ...
free(p); // Забыли, что p уже освободили и освобождаем снова, это UB
```

Для минимизации шанса возникновения этой ошибки, можно занулять указатель после освобождения памяти:

```
int* p = (int*)malloc(1000 * sizeof(int));
free(p);
p = NULL;
// ...
free(p); // Функция free ничего не будет делать, если ей передать NULL. Тут нет ошибки.
```

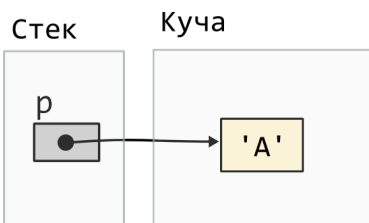
Примеры создания объектов в куче

Напишем код, который будет создавать в куче различные объекты. В каждом примере напечатаем созданные в куче объекты и освободим всю память, которую вы выделили. Исходный код всех примеров из этого раздела можно найти в папке `examples`.

1. Один символ:

```
char* p = (char*)malloc(sizeof(char));
*p = 'A';

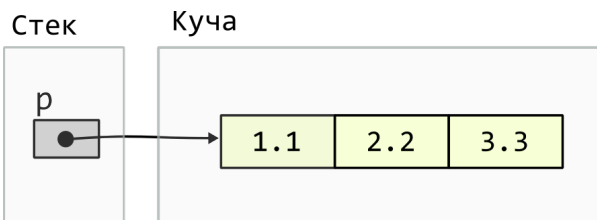
printf("%c\n", *p);
free(p);
```



2. Массив из трёх элементов типа float:

```
float* p = (float*)malloc(sizeof(float)*3);
p[0] = 1.1;
p[1] = 2.2;
p[2] = 3.3;

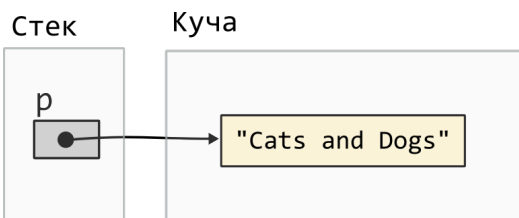
for (size_t i = 0; i < 3; ++i)
    printf("%f ", p[i]);
printf("\n");
free(p);
```



3. Строку (массив char) "Cats and Dogs":

```
char* p = (char*)malloc(sizeof(char) * 14);
strcpy(p, "Cats and Dogs");

printf("%s\n", p);
free(p);
```

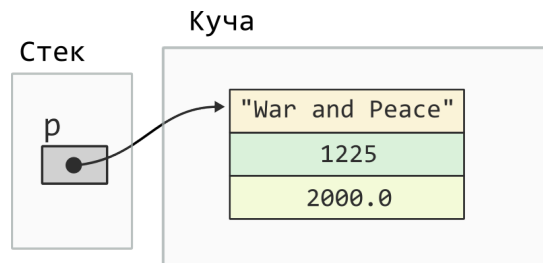


4. Структуру Book из семинара на структуры:

```
Book* p = (Book*)malloc(sizeof(Book));
strcpy(p->title, "War and Peace");
p->pages = 1225;
p->price = 2000.0;

printf("Book: %s, Pages: %i, Price: %f\n",
      pb->title, pb->pages, pb->price);

free(p);
```



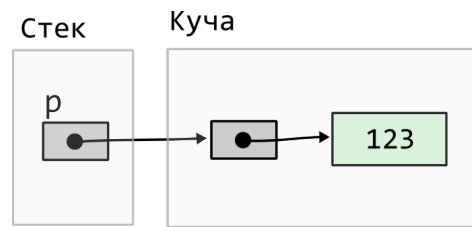
5. Указатель, который указывает на число `int`:

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

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

**p = 123;

printf("%i\n", **p);
free(*p);
free(p);
```

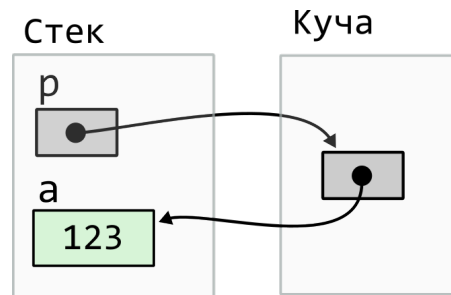


6. Указатель, который указывает на число `int`, которое находится в стеке:

```
int a = 123;

int** p = (int**)malloc(sizeof(int*));
*p = &a;

printf("%i\n", **p);
free(p);
```



7. Массив из указателей на `int`:

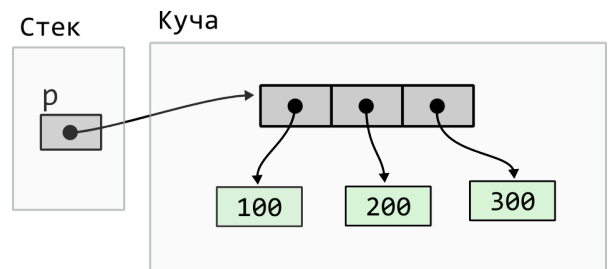
```
int** p = (int**)malloc(3 * sizeof(int*));

for (size_t i = 0; i < 3; ++i)
    p[i] = (int*)malloc(sizeof(int));

*p[0] = 100;
*p[1] = 200;
*p[2] = 300;

for (size_t i = 0; i < 3; ++i)
    printf("%i ", *p[i]);
printf("\n");

for (size_t i = 0; i < 3; ++i)
    free(p[i]);
free(p);
```

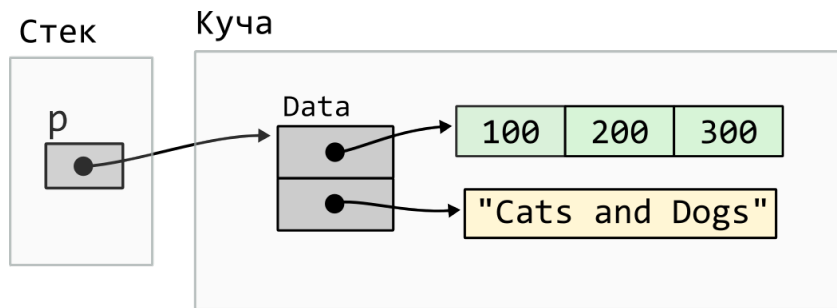


8. Пусть есть структура, которая хранит 2 указателя: `numbers` и `symbols`:

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

typedef struct data Data;
```

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



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

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

int main()
{
    Data* p = (Data*)malloc(sizeof(Data));
    p->numbers = (int*)malloc(sizeof(int) * 3);
    p->symbols = (char*)malloc(sizeof(char) * 14);

    p->numbers[0] = 100;
    p->numbers[1] = 200;
    p->numbers[2] = 300;

    strcpy(p->symbols, "Cats and Dogs");

    for (size_t i = 0; i < 3; ++i)
        printf("%i ", p->numbers[i]);
    printf("\n");

    printf("%s\n", p->symbols);

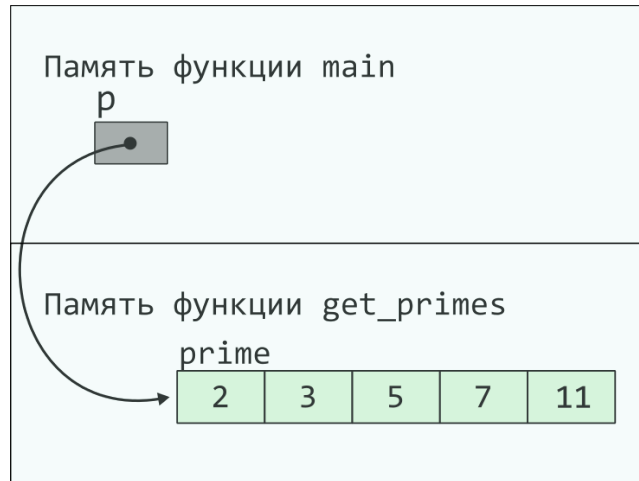
    free(p->numbers);
    free(p->symbols);
    free(p);
};
```

Возврат массива из функции

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

Допустим мы хотим создать функцию, которая должна будет возвращать массив. Можно попытаться создать 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 при выходе из функции. Таким образом, возвращаемый указатель будет ссылаться на уже освобождённую память. Ошибка!

```
#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]);
}
```

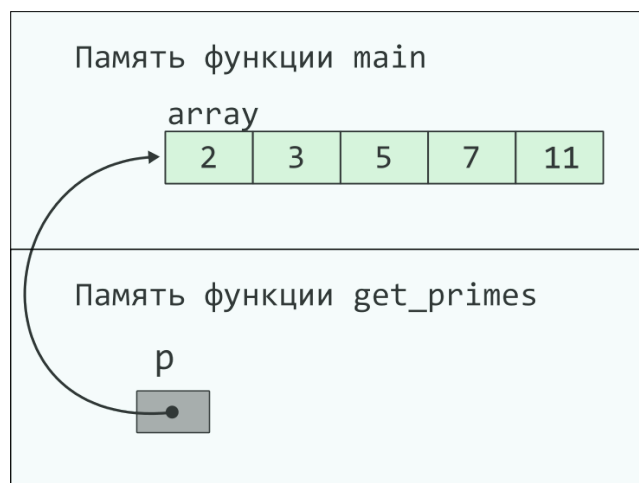


Что напечатает программа, если раскомментировать строку?

Возврат массива, созданного 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.

```
#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, созданного 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.

```
#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;
}

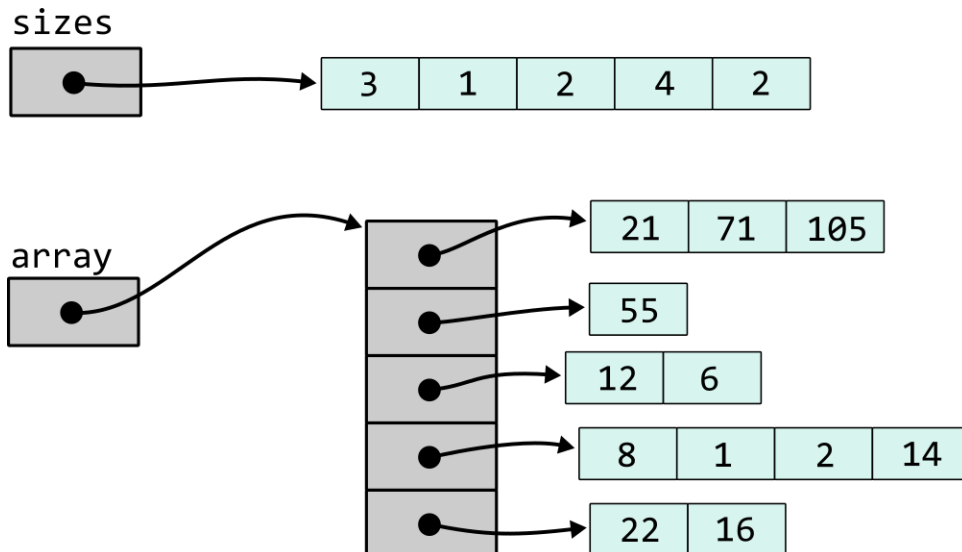
int main()
{
    int* p = get_primes();
    for (int i = 0; i < 5; ++i)
        printf("%i ", p[i]);
    free(p);
}
```



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.



Сегмент памяти Данные (Data)

Глобальные переменные

Глобальные переменные это просто переменные, которые определяются вне всех функций.

```
#include <stdio.h>
int a = 10; // Глобальная переменная a находится сегменте data

int main()
{
    int b = 20; // Переменная b находится сегменте stack
    printf("%i %i\n", a, b);
}
```

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 строка `static int n = 0;` выполняется только один раз и переменная `n` будет оставаться в памяти до завершения программы.

Сегмент памяти Текст. Указатели на функции.

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:

```
#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);
}
```

Подробнее в файле funcpointers/0funcpointer.c.

Стандартная функция qsort

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

```
#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.

Как код превращается в последовательность байт.

a.c

```
int a = 0x1234;  
a *= 0x7755;  
a += 0x99aa88;
```

a.s

```
mov    DWORD PTR [rbp-0x4],0x1234  
mov    eax,DWORD PTR [rbp-0x4]  
imul   eax,eax,0x7755  
mov    DWORD PTR [rbp-0x4],eax  
add    DWORD PTR [rbp-0x4],0x99aa88
```

a.exe (или a.out)

```
c7 45 fc 34 12 00 00  
8b 45 fc  
69 c0 55 77 00 00  
89 45 fc  
81 45 fc 88 aa 99 00
```

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 достигнуть 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 (`imul`) 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 разыменовывание. 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`.
- `imul 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 отличие от кода на языке C, код на языке ассемблера различается на разных процессорах. Код с вычислительной системы одной архитектуры скорее 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.