Семинар #8: 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);
}
```

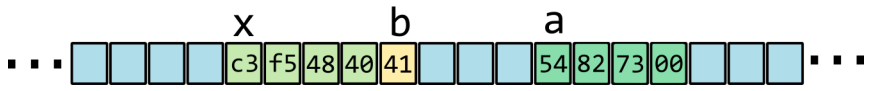
Также, можно печатать и считывать числа в этих системах счисления с помощью спецификаторов `%o` (для восьмеричной системы – **o**ctal) и `%x` (для шестнадцатеричной – he**x**adecimal). Спецификатор `%d` можно использовать для десятичной системы – **d**ecimal. Пример программы, которая считывает число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

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

Часть 2: Переменные в памяти. Little и Big Endian

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:

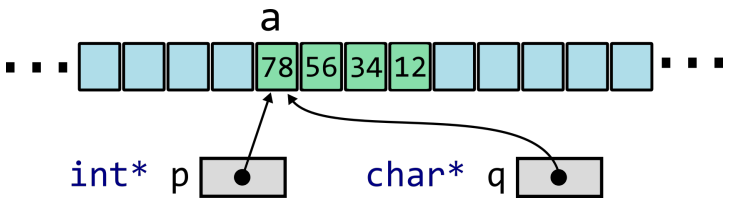
- Значение одного байта памяти удобно представлять двузначным шестнадцатиричным числом.
- Каждая переменная заняла столько байт, чему равен её размер.
- Переменные в памяти могут хранится не в том порядке, в котором вы их объявляете.
- Переменные в памяти хранятся не обязательно вплотную друг к другу.
- Байты переменных `a` и `b` 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 $65 = 41_{16}$.

Часть 3: 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.

```
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?

```
#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:

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

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

Часть 4: Просмотр байт

Просмотр байт переменной

Просмотреть, что содержится в байтах какого-либо объекта можно с помощью указателя на `unsigned char`.

```
#include <stdio.h>
int main()
{
    int a = 0x11223344;

    unsigned char* p = (unsigned char*)&a;
    for (size_t i = 0; i < sizeof(a); ++i)
        printf("%x ", *(p + i));
    printf("\n");
}
```

Задача:

- Напечатайте байты объекта `a` типа `double`.

  ```
  double a = 123.456;
  ```

- Напечатайте байты объекта `b` типа `int`.

  ```
  int b = -1;
  ```

- Напечатайте байты объекта `c` типа `struct cat`.

  ```
  struct cat
  {
      char first;
      int second;
  };

  int main()
  {
      struct cat c = {0x50, 0x12345678}
  }
  ```

Просмотр байтов файла с помощью программы `xxd`

`xxd` - это простая программа, которая выводит на экран всё содержимое файла побайтово. Если, например, запустить программму следующим образом: `xxd a.out`, то она выведет на экран всё содержимое этого исполняемого файла. Часто используемые опции командной строки: `-h` (сокращение от help) и `-v` (сокращение от version).

- Запустите `xxd` с аргументом – именем файла `hello.txt`. Этот файл содержит лишь строку `Hello`. `xxd` покажет вам содержимое этого файла в шестнадцатеричном виде и в виде `ASCII`.
- Запустите `xxd` с опцией `-h`.
- Запустите `xxd` с нужной опцией, чтобы вывод файла `hello.txt` был представлен в двоичном виде.
- * Если файл большой, то весь вывод `xxd` не поместится на экран. Перенаправить вывод в нужный файл можно следующим образом: `xxd a.out > temp.txt`. После этого в файле `temp.txt` будет хранится всё, что было бы напечатано на экран.
- * Создайте программу Hello World и скомпилируйте её в файл `a.out`. Сохраните вывод `xxd ./a.out` в отдельном файле `hw.txt`. Измените файл `hw.txt`, так чтобы программа печатала Hello MIPT. Создайте исполняемый файл из файла `hw.txt`, используя `xxd` с опцией `-r`.

Часть 5: Работы с бинарными файлами `fread` и `fwrite`

`fwrite` записывает некоторый участок памяти в файл без обработки.
`fread` считывает данные из файла в память без обработки.

Пример. Записываем 4 байта памяти переменной `a` в файл `binary.dat`:

```
#include <stdio.h>
int main()
{
    int a = 0x11223344;
    FILE* fb = fopen("binary.dat", "wb");
    fwrite(&a, sizeof(int), 1, fb);
    fclose(fb);
}
```

- **Печать в текстовом и бинарном виде:**
  В файле `text_and_binary.c` содержится пример записи числа в текстовом и бинарном виде. Скомпилируйте эту программу и запустите. Должно появиться 2 файла (`number.txt` и `number.bin`). Изучите оба эти файла, открывая их в текстовом редакторе, а также с помощью утилиты `xxd`. Объясните результат.

- **Печать массива в бинарном виде:**
  Пусть есть массив из чисел типа `int`: `int array[5] = {111, 222, 333, 444, 555};`
  Запишите эти числа в текстовый файл `array.txt`, используя `fprintf`. Изучите содержимое этого файла побайтово с помощью `xxd`.
  Запишите эти числа в бинарный файл `array.bin`, используя `fwrite`. Изучите содержимое этого файла побайтово с помощью `xxd`.

Часть 4: Стандартные функции `memcpy` и `memset`

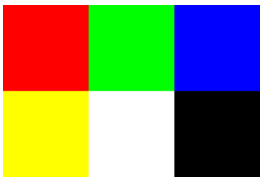
Часть 6: Работа с изображениями формата `.ppm`

Простейший формат для изображение имеет следующую структуру

```
P3
3 2
255
255 0 0
0 255 0
0 0 255
255 255 0
255 255 255
0 0 0
```

- В первой строке задаётся тип файла `P3` - означает, что в этом файле будет храниться цветное изображение, причём значения пикселей будет задаваться в текстовом формате.
- Во второй строке задаются размеры картинки - 3 на 2 пикселя.
- Во третьей строке задаётся максимальное значение RGB компоненты цвета.
- Дальше идут RGB компоненты цветов каждого пикселя в текстовом формате.

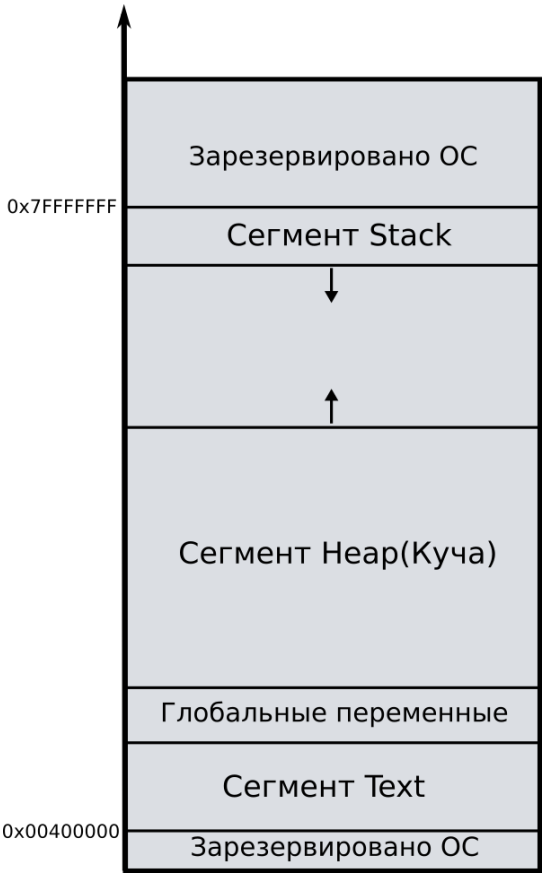
Картинка имеет следующий вид:



Задачи

- Написать программу, которая генерирует одноцветную картинку (500 на 500) в формате `.ppm`. Цвет должен передаваться через аргументы командной строки.
- **Белый шум:** Написать программу, которая случайное изображение в формате `.ppm`. Цвет каждого пикселя задаётся случайно.
- **Градиент:** Написать программу, которая генерирует градиентную картинку в формате `.ppm`. Два цвета должны передаваться через аргументы командной строки.
- **Черно-белая картинка:** Написать программу, которая считывает изображение в формате `.ppm` и сохраняет его в черно-белом виде. Файл изображения должен передаваться через аргументы командной строки. Считайте файл `russian_peasants_1909.ppm` и сделайте его черно-белым.

Сегменты памяти. Указатели на функцию.



1. **Сегмент памяти Стек (Stack)**

   - При обычном объявлении переменных и массивов все они создаются в стеке: `int a;` или `int array[10];`
   - Память на эти переменные выделяется в начале функции и освобождается в конце функции.
   - Маленький размер (несколько мегабайт)
   - Выделение памяти происходит быстрее чем в куче

2. **Сегмент памяти Куча (Heap)**

   - `malloc` выделяет память в Куче.
     `int* p = (int*)malloc(10 * sizeof(int));`
   - Память выделяется при вызове `malloc` и освобождается при вызове `free`.
   - Размер ограничен свободной оперативной памятью - гигабайты.
   - Выделение памяти происходит медленней чем в стеке

3. **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.

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

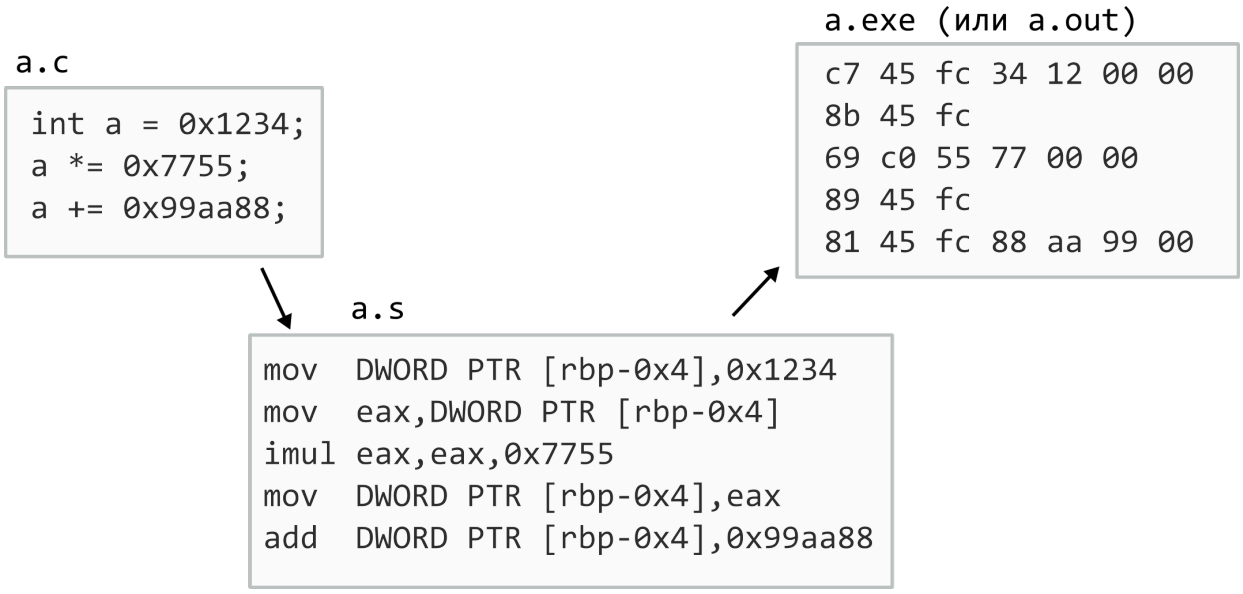
В библиотеке `stdlib.h` уже реализована функция `qsort`, которая сортирует произвольные элементы, используя быструю сортировку. Пример использования этой функции:

```
#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`.

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.