

Семинар #7: Память и бинарные файлы.

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, скорее 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 шестнадцатеричная системы счисления. В шестнадцатеричной системе счисления есть 16 цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f.

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

$$1ab_{16} = 1 \cdot 16^2 + 10 \cdot 16^1 + 11 = 427$$

$$ff.c_{16} = 15 \cdot 16^1 + 15 \cdot 16^0 + 12 \cdot 16^{-1} = 255.75$$

Шестнадцатеричная, восьмеричная и бинарная системы счисления в языке C

Язык C поддерживает шестнадцатеричные, восьмеричные и бинарные литералы. Чтобы получить шестнадцатеричное число нужно написать 0x перед числом. Чтобы получить восьмеричное число нужно написать 0 перед числом. Чтобы получить бинарное число нужно написать 0b перед числом.

```
#include <stdio.h>
int main()
{
    int a = 123;      // Десятичная система
    int b = 0x123;     // Шестнадцатеричная система
    int c = 0123;      // Восьмеричная система
    int d = 0b10101;   // Бинарная система
    printf("%i %i %i %i\n", a, b, c, d);
}
```

Также, можно печатать и считывать числа в других системах счисления с помощью спецификаторов %x (для шестнадцатеричной – hexadecimal) и %o (для восьмеричной системы – octal). Спецификатор %d можно использовать для десятичной системы – decimal (получается, что спецификатор %d это то же самое, что и %i). Для отображения адресов при печати с помощью спецификатора %p используется шестнадцатеричная система счисления.

```
#include <stdio.h>
int main()
{
    int a;
    scanf("%d", &a);    // Считаем число в десятичной системе
    printf("%x\n", a);  // Напечатает это же число в шестнадцатеричной системе

    printf("%p\n", &a); // Для адресов используется шестнадцатеричная система
}
```

Представление чисел в памяти

Представление целых чисел в памяти

Положительные числа представляются в памяти в соответствии с их записью в бинарной системе счисления. Для представления отрицательных чисел в памяти используется способ, который называется дополнительный код. Однобайтовые числа представляются в памяти следующим образом:

unsigned char		signed char	
0	00000000	0	00000000
1	00000001	1	00000001
2	00000010	2	00000010
3	00000011	...	
4	00000100	126	01111110
5	00000101	127	01111111
6	00000110	-128	10000000
...		-127	10000001
253	11111101	...	
254	11111110	-2	11111110
255	11111111	-1	11111111

Целые числа большего размера представляются в памяти аналогичным образом.

Представление чисел с плавающей точкой в памяти

Разберём как числа с плавающей точкой хранятся в памяти на примере. Пусть у нас есть число 123.456 типа `float`. Как это число хранится в двоичном коде? Для начала переведём это число из десятичной системы счисления в двоичную:

$$123.456_{10} = 1111011.01110100101111000111_2$$

Затем представим это число в научной записи:

$$1111011.01110100101111000111_2 = 1.11101101110100101111000111_2 \cdot 2^6$$

Получилось число:

$$+1.11101101110100101111000111_2 \cdot 2^6$$

Части этой записи числа и хранятся в памяти. Число типа `float` имеет размер 4 байта или 32 бита. Из них:

- **1 бит** приходится на знак числа. 0 – для положительных и 1 для отрицательных.
- **8 бит** приходится на степень двойки. К степени двойки в двоичной научной записи прибавляется число 127, а затем это число хранится в этих битах. В нашем примере будет храниться число $6 + 127 = 133 = 10000101_2$.
- **23 бита** приходится на мантиссу. В двоичной научной записи это просто 23 цифры после точки. В нашем примере это 11101101110100101111001 (округляем последнюю цифру).

Таким образом число 123.456 типа `float` будет храниться в памяти как:

$$0 \quad 10000101 \quad 11101101110100101111001$$

Разобьём эту запись на кусочки по 8 бит:

$$01000010 \quad 11110110 \quad 11101001 \quad 01111001$$

Переведём каждый из кусочков в шестнадцатеричную систему:

$$42 \quad F6 \quad E9 \quad 79$$

Это значения которые будут иметь байты числа типа `float` при записи в него числа 123.456. Числа типа `double` хранятся аналогичным образом, но для хранения степени и мантиссы используется 11 и 52 бита соответственно.

Побитовые операторы

Побитовое И

Побитовое И применяет операцию логического И для каждого бита. Обозначается символом `&`.

```
x      = 00010100 = 20
y      = 01100100 = 100
x & y  = 00000100 = 4

printf("%i\n", 20 & 100); // Напечатает 4
```

Побитовое ИЛИ

Побитовое ИЛИ применяет операцию логического ИЛИ для каждого бита. Обозначается символом `|`.

```
x      = 00010100 = 20
y      = 01100100 = 100
x | y  = 01110100 = 116

printf("%i\n", 20 | 100); // Напечатает 116
```

Побитовое исключающее ИЛИ

Побитовое исключающее ИЛИ применяет операцию логического исключающее ИЛИ (также известную как *exclusive OR* или просто *XOR*) для каждого бита. Операция логического исключающего ИЛИ задаётся следующими выражениями:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

Побитовое исключающее ИЛИ в коде обозначается символом `^`.

```
x      = 00010100 = 20
y      = 01100100 = 100
x ^ y  = 01110000 = 112

printf("%i\n", 20 ^ 100); // Напечатает 112
```

Побитовое НЕ

Побитовое НЕ применяет операцию логического НЕ для каждого бита. То есть просто обращает каждый бит числа. Обозначается символом `~`.

```
x      = 00010100 = 20
~x     = 11101011 = -21

printf("%i\n", ~20); // Напечатает -21
```

Побитовый сдвиг влево

Побитовый сдвиг влево сдвигает все биты числа влево на заданное число. Обозначается как `<<`.

```
x      = 00010100 = 20
x << 2 = 01010000 = 80

printf("%i\n", 20 << 2); // Напечатает 80
```

Побитовый сдвиг вправо

Побитовый сдвиг вправо сдвигает все биты числа вправо на заданное число. Обозначается как `>>`.

```
x      = 00010100 = 20
x >> 2 = 00000101 = 5

printf("%i\n", 20 >> 2); // Напечатает 5
```

Если побитовый сдвиг вправо применяется к отрицательному числу, то, на большинстве системах, левые биты заполняются не нулями а единицами:

```
x      = 11101011 = -21
x >> 2 = 11111010 = -6

printf("%i\n", -21 >> 2); // Напечатает -6
```

Сдвиги и неопределённое поведение

Неопределённое поведение возникает при сдвигах отрицательного числа влево, а также если расстояние сдвига (правый операнд) отрицательно или больше или равно ширине типа данных.

Печать битового представления числа

Данная функция печатает все биты числа типа `int`:

```
#include <stdio.h>
void print_binary(const char* prefix, int x)
{
    printf("%s", prefix);
    for (int i = 8 * sizeof(x) - 1; i >= 0; --i)
    {
        printf("%u", (x >> i) & 1);
        if (i % 8 == 0)
            printf(" ");
    }
    printf("\n");
}
int main()
{
    int x = 0b00010100;
    int y = 0b01100100;
    print_binary("x      = ", x);
    print_binary("y      = ", y);
    print_binary("x & y   = ", x & y);
    print_binary("x | y   = ", x | y);
    print_binary("x ^ y   = ", x ^ y);
    print_binary("~x     = ", ~x);
    print_binary("x << 2 = ", x << 2);
    print_binary("x >> 2 = ", x >> 2);
}
```

Получение значения i-го бита числа

Для получения *i*-го бита числа (нумерировать биты будем, начиная с нуля) нужно сдвинуть число на *i* бит вправо и применить побитовое и с единицей.

x	=	00010100	
x >> 2	=	00000101	<code>int x = 0b00010100;</code>
(x >> 2) & 1	=	00000001	<code>printf("%i\n", (x >> 2) & 1);</code>

Установка i-го бита в значение 1

Для того, чтобы установить *i*-й бит числа в значение 1 нужно применить побитовое или с числом $(1 << i)$.

x	=	00010100	
1	=	00000001	<code>int x = 0b00010100;</code>
1 << 3	=	00001000	<code>printf("%i\n", x (1 << 3));</code>
x (1 << 3)	=	00011100	

Установка i-го бита в значение 0

Для того, чтобы установить *i*-й бит числа в значение 0 нужно применить побитовое и с числом $\sim(1 << i)$.

x	=	00010100	
1 << 2	=	00000100	<code>int x = 0b00010100;</code>
$\sim(1 << 2)$	=	11111011	<code>printf("%i\n", x & $\sim(1 << 2)$);</code>
x & $\sim(1 << 2)$	=	00010000	

Работа с участками памяти в C

Массивы из unsigned char как участки памяти

TODO

Просмотр байт переменной

Просмотреть, что содержится в байтах какого-либо объекта можно с помощью указателя `unsigned char*`. Используя такой указатели и арифметику указателей можно удобно просматривать значения каждого байта любого объекта.

```
#include <stdio.h>
int main()
{
    int a = 0x42f6e979;

    unsigned char* p = (unsigned char*)&a;
    for (size_t i = 0; i < sizeof(a); ++i)
        printf("%02x ", *(p + i));
    printf("\n");
}
```

Спецификатор `%02x` означает, что нужно распечатать число в шестнадцатеричной системе счисления, при этом число будет печататься двумя символами, а если число состоит меньше, чем из двух цифр, то дополнить его нулями. То есть, если бы мы использовали спецификатор `%x`, то однозначные числа печатались бы одной цифрой, а со спецификатором `%02x` у однозначных чисел будет ставиться 0 в начале. Таким образом, значение каждого байта будет отпечатано двузначным шестнадцатеричным числом.

Стандартные функции `memset`, `memcpy` и `memmove`

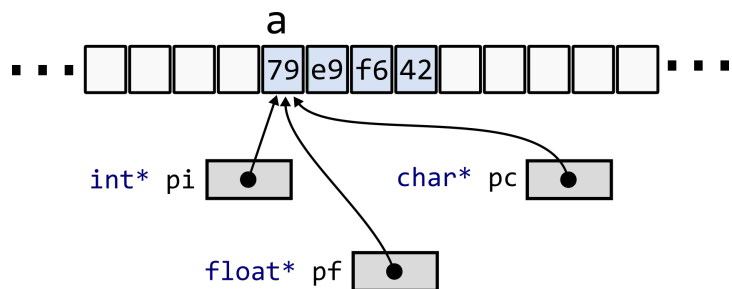
TODO

Указатели разных типов, указывающие на одно и то же место в памяти

Рассмотрим следующий пример. На переменную `a` указывают три указателя разных типов: `int*`, `float*` и `char*`. Какие значения мы получим, если разыменуем каждый из указателей?

```
#include <stdio.h>
int main()
{
    int a = 0x42f6e979;
    int* pi = &a;
    float* pf = (float*)&a;
    char* pc = (char*)&a;

    printf("%x\n", *pi);
    printf("%f\n", *pf);
    printf("%c\n", *pc);
}
```



- При разыменовывании указателя `int*` мы пройдем по адресу, возьмем `sizeof(int) = 4` байта памяти и будем воспринимать их как объект типа `int`. В итоге напечатается `42f6e979`.
- Доступ до объекта типа `int` с помощью указателя `float*` запрещен строгими правилами алиасинга. Поэтому данный код ошибочен – он содержит UB.

Но если отключить строгие правила алиасинга, то произойдет следующее. При разыменовывании указателя `float*` мы пройдем по адресу, возьмем `sizeof(float) = 4` байта памяти и будем воспринимать их как объект типа `float`. В итоге напечатается число `123.456`.

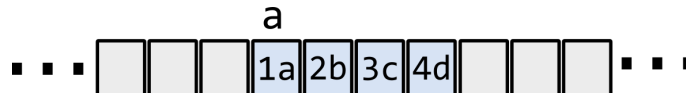
- Доступ до объекта типа `int` с помощью указателя `char*` разрешен строгими правилами алиасинга. Тут никакого UB нет.

При разыменовывании указателя `char*` мы пройдем по адресу, возьмем `sizeof(char) = 1` байт памяти и будем воспринимать его как объект типа `char`. В результате напечатает символ с кодом `0x79 = 121`, то есть символ `y`.

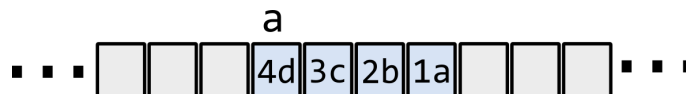
Порядок байт. Little и Big Endian

То в каком порядке лежат байты многобайтового числа в памяти может различаться на разных системах. Различают два основных порядка байт. Разберем их на примере числа: `int a = 0x1a2b3c4d`;

- Прямой порядок байт или *Big Endian*
При таком порядке записи, число записывается в памяти от старшего байта к младшему.

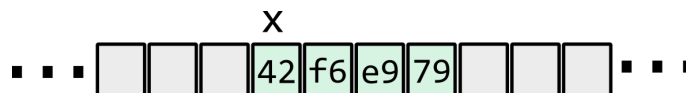


- Обратный порядок байт или *Little Endian*
При таком порядке записи, число записывается в памяти от младшего байта к старшему.

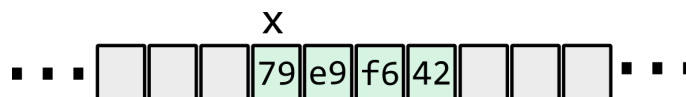


То же самое работает и для других скалярных типов данных, таких как числа с плавающей точкой и указатели. Рассмотрим число с плавающей точкой `float x = 123.456`. В памяти оно будет выглядеть следующим образом:

- При использовании порядка байт *Big Endian*:



- При использовании порядка байт *Little Endian*:



На большинстве систем используется порядок байт Little Endian. В дальнейших примерах по умолчанию будет использоваться этот порядок.

Объединения

Объединение (англ. *union*)

Битовые поля

Битовые поля (англ. *bit fields*)

Каламбур типизации (Type punning)

Уровни оптимизации

Рассмотрим следующую программу, в которой к целому числу добавляется единица 2 миллиарда раз.

```
#include <stdio.h>
int main()
{
    int a = 0;
    while (a < 2000000000)
        a += 1;
    printf("%i\n", a);
}
```

Как долго будет исполняться данная программа? Если скомпилировать эту программу так:

```
gcc prog.c
```

и запустить, то программа действительно может исполняться достаточно долго для такой задачи (несколько секунд). Если же скомпилировать программу с включенными оптимизациями:

```
gcc -O2 prog.c
```

то программа исполнится почти мгновенно.

Всё дело в том, что компилятор оптимизировал код программы. Он догадался, что цикл в программе можно не делать, а просто сразу напечатать итоговое число. У компилятора есть множество оптимизаций, которые позволяют сильно ускорить вашу программу. За включение этих оптимизаций отвечают флаги:

- `-O0` - отключает (почти) все оптимизации, этот уровень используется по умолчанию
- `-O1` - включает основные оптимизации
- `-O2` - включает большинство оптимизаций
- `-O3` - в дополнении к `-O2` включает агрессивные оптимизации, может как ускорить, так и замедлить программу по сравнению с `-O2`

Правило строгого алиасинга (англ. *strict aliasing rule*)

Алиасинг - это ситуация, когда в программе к одним и тем же данным можно получить доступ, используя разные имена. Например:

```
int a = 10;
int* p = &a;
char* q = (char*)&a;
```

К данным, которые соответствуют переменной `a`, можно обратиться, используя разные имена `a`, `*p` или `*q`. Строгий алиасинг – это одна из оптимизация компилятора, которая может изменить поведение программы. Идея этой оптимизации заключается в том, что компилятор предполагает, что к одному объекту нельзя обращаться через указатель на тип, отличный от типа данного объекта. Рассмотрим как проявляется эта оптимизация на следующем примере:

```

#include <stdio.h>
int func(int* p, float* q)
{
    *p = 10;
    *q = 0.0;
    return *p;
}

int main()
{
    int a = 20;
    printf("%i\n", func(&a, (float*)&a));
}

```

Что напечатает данная программа? Если скомпилировать эту программу без оптимизаций, то напечатается число 0, так как внутри функции в строке `*q = 0.0;` мы занулили все байты объекта `a`. Если же скомпилировать эту программу с флагом `-O2` или `-O3`, то будет напечатано 10. Почему так происходит? Дело в том, что при таких уровнях оптимизации подключается оптимизация строгого алиасинга. Поэтому компилятор предполагает, что два указателя разных типов не могут указывать на один объект, следовательно указатели `p` и `q` в функции `func` не указывают на один объект, поэтому эту функцию можно оптимизировать так, чтобы она всегда возвращала число 10.

Нарушение правила строгого алиасинга приведёт к тому, что программа будет содержать UB.

Исключения из правила строгого алиасинга

Из правила строгого алиасинга есть исключения, которые говорят, что в некоторых ситуациях всё-таки можно использовать объект, через указатель другого типа:

- Если типы отличаются на `const`. Например, объект типа `int` можно читать с помощью указателя типа `const int*`.
- Если типы отличаются на `unsigned` или на `const unsigned`. Например, с объектом типа `int` можно работать через указатели типа `unsigned int*` или `const unsigned int*`.
- Если один из типов является структурой, которая содержит поле, чей тип совпадает с другим типом (с учётом первых двух пунктов). Например, если есть структура:

```

struct point
{
    int x;
    int y;
};

```

То с ней можно работать через указатель `int*` или `const unsigned int*`.

- Если один из типов является типом `char` или `unsigned char`. Благодаря этому исключению, мы можем безопасно просматривать байты любого объекта.

Работа с бинарными файлами

`fputc` .

`fgetc` .

`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`.

Функция fgetc.

Функция `fgetc` считывает 1 символ и возвращает код ASCII символа или EOF если дошли до конца файла (EOF это просто константа равная -1). Пример считывания:

```
#include <stdio.h>
int main()
{
    FILE* f = fopen("test.txt", "r");
    while (1)
    {
        // Считываем 1 символ
        int c = fgetc(f);

        // Если он равен EOF, то выходим из цикла
        if (c == EOF)
            break;

        printf("%c\n", c);
    }
    fclose(f);
}
```

- Напишите программу, которая печатает количество строк в файле.
- Напишите программу, которая печатает размер самой длинной строки файла.

Функции ftell и fseek.

Процесс считывания файла можно представить как перемещение по набору байт. При открытии файла указатель положения равен нулю. При считывании он увеличивается на количество считанных байт.

указатель
положения
(равен 9)
↓

W	e	s	t	h	o	u	s	e	4	3	1	2	\n	E	a	s	t	w
o	r	l	d	5	5	9	1	\n										

Однако, положение в файле можно менять и без считывания при помощи функции `fseek`:

`fseek(<файловый указатель>, <смещение>, <начало отсчёта>)`

Начало отсчёта в этой функции может принимать 3 значения:

1. `SEEK_SET` – отсчитывать от начала файла
2. `SEEK_CUR` – отсчитывать от текущего положения
3. `SEEK_END` – отсчитывать от конца файла

Например:

```
#include <stdio.h>
int main()
{
    FILE* f = fopen("test.txt", "r");
```

```
fseek(f, 10, SEEK_SET); // Перемещаемся на 11 - й символ
fseek(f, -1, SEEK_END); // Перемещаемся к последнему символу

fseek(f, -1, SEEK_CUR); // Перемещаемся на 1 символ назад
fseek(f, 0, SEEK_SET); // Возвращаемся к началу
fclose(f);
}
```

Функция `ftell(<файловый указатель>)` возвращает целое число – текущее положение в файле.

Работа с изображениями формата .ppm

Простейший формат для изображения 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.
- Далее 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.

Работа с изображениями формата .jpeg