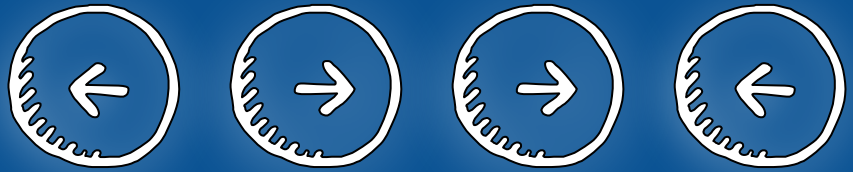
Алгоритмы и Алгоритмические Языки

Семинар #5:

- Память программы и указатели;
- Передача возвращаемых значений из функций;
- Массивы;
- Арифметика указателей и операция `sizeof`;

26.09.2023

Память программы и указатели



Адресация памяти компьютера

Память компьютера – адресуемый набор ячеек.

Размер одной ячейки – это размер **char** (часто – 8 бит).

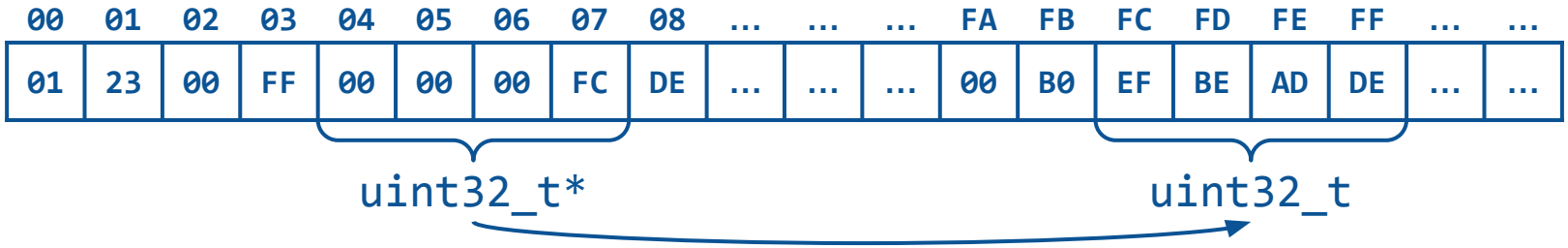
00	01	02	03	04	05	06	07	08	...	...	...	FA	FB	FC	FD	FE	FF	...	...
01	23	00	FF	DE	AD	BE	EF	DE	...	...	...	00	B0	1D	FA	CE	CA	...	...

В программе память выделена под:

- Исполняемый код вашей программы;
- Исполняемый код стандартной библиотеки;
- Локальные переменные;
- Глобальные переменные;
- Константные строки;

Указатели в языке Си

Тип данных, хранящий адрес ячейки памяти – указатель.
Обозначается добавление звёздочки к типу.




```
uint32_t  val = 0xDEADBEEF;
uint32_t* ptr = &val;
```

Взятие адреса переменной

```
*ptr = 0xB01DFACE;
*ptr += 0xBA1BEC;
```

Разыменование указателя

Вопрос: что будет лежать по адресу `ptr` после первой записи?

Ключевое слово **const** в указателях

В Си есть константные указатели и указатели на константу:

```
const char* const_str = "I'm in READ-ONLY memory UwU";
*const_str            =  '\0'; // Compilation error
const_str             = "OwO"; // OK

char* const str = "I'm in READ-ONLY memory UwU";
*str            =  '\0'; // Runtime error
str             = "OwO"; // Compilation error
```

Обычные переменные также бывают **const**.

Призвание **const** – перенос потенциальных ошибок на более ранний этап жизненного цикла разработки ПО.

Указатель на void

Указатель на void (void*) – тип указателя, к которому может быть неявно приведён любой другой тип указателя:

```
float value = 0.0;
char* error_ptr = &value; // Compilation error

char* float_ptr = (char*) &value; // OK
void* weird_ptr = &value;         // OK
```

Все остальные неявные приведения указателей разных типов – запрещены.

Операции с указателями: задачи

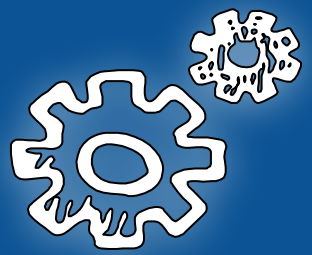
В программе объявлены следующие переменные:

```
int x, *p, **q;
// Или int x; int* p; int** q;
```

Определить, какие из выражений являются корректными:

```
1) *p
2) *x
3) p = &x
4) &p
5) &(&x)
6) x = p
7) p = x
8) q = &x
9) q = &p
10) **p
11) *q
12) **q
13) ***q
14) &*p
```

Передача возвращаемых значений из функций



Изменяемые аргументы функций

Передача аргументов функции по указателю (см. 05_pointers):

```
double x1 = 0, y1 = 0;
int ret = read_vector(&x1, &y1);
if (ret != 0)
{
    printf("Unable to parse ...\n");
    return EXIT_FAILURE;
}
```

Изменяемые аргументы функций

Функция, меняющая значение по принятому указателю:

```
int read_vector(double* x_ptr, double* y_ptr)
{
    *x_ptr = expr1;
    *y_ptr = expr2;
}
```

Вопросы:

- Может ли указатель x_ptr или y_ptr быть некорректным?
- Стоит ли проверять указатели на корректность?
- Как это сделать?

Обращение по нулевому указателю

Нулевой указатель обозначается как NULL:

```
int ret = read_vector(NULL, &y1);
```

Обращение по нулевому указателю – Undefined Behavior (зло).

```
❯ cd ~/path/to/repository/examples/05_pointers
❯ gcc dot.c -o dot -lm
❯ ./dot
1.0 0
[1] 15463 segmentation fault (core dumped)  ./dot
```

Проверка указателей

Функция assert из assert.h проверяет указатель на NULL:

```
int read_vector(double* x_ptr, double* y_ptr)
{
    assert(x_ptr);
    assert(y_ptr);
```

```
❯ ./dot
dot: dot.c:8: read_vector: Assertion `x_ptr' failed.
[1] 15789 abort (core dumped)  ./dot
```

Вопрос: почему проверка на NULL не выполняется с printf-ом?

Массивы

Массивы в языке Си

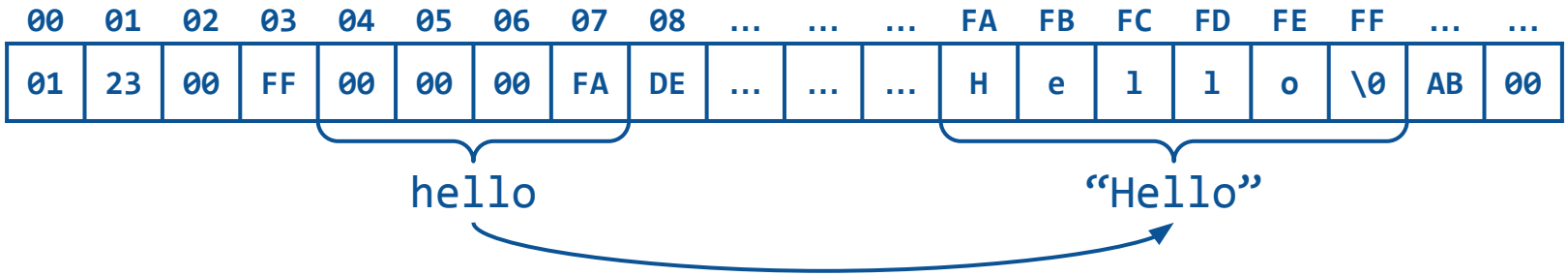
Декларация массива

```
char hello[6] = "Hello";
```

Доступ к элементам

```
hello[0] = 'Y';
hello[1] += 10;
```

Нижележащее представление в памяти программы:



Способы инициализации массивов

Способы инициализации массивов

```cpp
// Uninitialized:
int a[5];
```

```cpp
// Zero-initialized:
int d[5] = {0};
```

```cpp
// List-initialized:
int b[5] = {1, 2, 3, 4, 5};
```

```cpp
// Element-wise initialized:
int c[] = {
    [0] = 0,
    [4] = 4
};
```

```cpp
// 2D-array:
int matrix[4][4] = {
    {0, 0, 0, 0},
    {0, 1, 1, 1},
    {0, 1, 2, 2},
    {0, 1, 2, 3}
};
```

Передача массива в функцию

При передаче массива в функцию необходимо передавать длину:

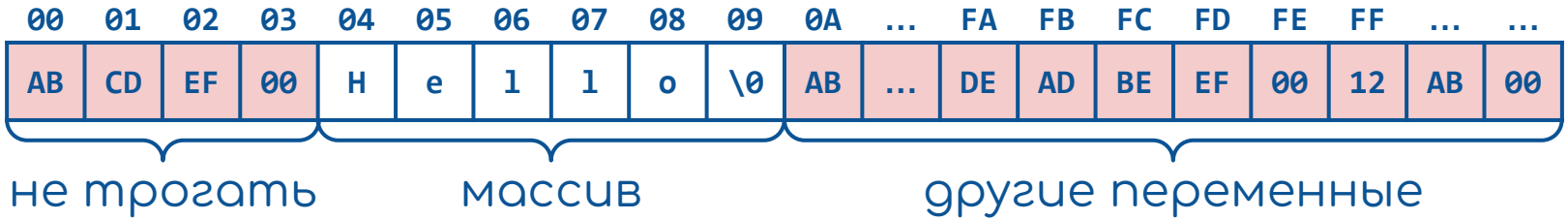
```
long sum(const int* array, unsigned length) {
    long sum = 0;

    for (int i = 0; i < length; i++) {
        sum += array[i];
    }

    return sum;
}
```

Выход за границы массива

Доступ за границы массива – Undefined Behavior (зло).



Способ поиска выхода за границы массива:

```
for (int i = start_i; continue_cycle(i); i = step(i))
{
    assert(0 <= i && i < array_size);
    ...
}
```

HSE University

Арифметика указателей и операция sizeof

Операция sizeof

Операция sizeof возвращает кол-во байт, которое тип или результат выражения будет занимать в памяти:

```
printf("sizeof(char)   = %lu\n", sizeof(char));   --> 1
printf("sizeof(int)    = %lu\n", sizeof(int));    --> 4
printf("sizeof(long)   = %lu\n", sizeof(long));   --> 8
printf("sizeof(float)  = %lu\n", sizeof(float));  --> 4
printf("sizeof(double) = %lu\n", sizeof(double)); --> 8

printf("sizeof(int*)   = %lu\n", sizeof(int*));   --> 8
printf("sizeof(void*)  = %lu\n", sizeof(void*));  --> 8

int array[5];
printf("sizeof(array)  = %lu\n", sizeof(array));  --> 20
```

Арифметика указателей

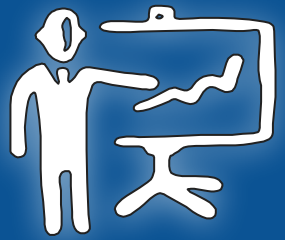
Основное правило арифметики указателей:

- При увеличении указателя типа `type*` на 1 происходит увеличение адреса на `sizeof(type)`.

Следствия:

- Конструкции `*(array + 4)` и `array[4]` эквивалентны;
- При вычитании адресов элементов одного массива получается разность индексов (разность – типа `ptrdiff_t`);
- Конструкции типа `ptr++` могут быть опасными!

Вопросы?



Красивые иконки взяты с сайта handdrawngoods.com