Практикум на ЭВМ
Строки. Указатели. Динамическое выделение памяти. Структуры.

Баев А.Ж.

Казахстанский филиал МГУ

03 октября 2018

Строки

Строка в языке C — массив из ascii-символов, который заканчивается нулевым символом '\0':

```
char s1[] = "math";
char s2[] = {'m', 'a', 't', 'h', '\0'};
char s3[] = {109, 97, 116, 104, 0};
```

Константные строки оператором присваивать можно только при инициализации, то есть такая запись будет ошибочной:

```
char s[7];
s = "math";
```

Работа со строками ничем не отличается от массива с нулевым символов в конце.

Указатели. Взятие адреса.

& — взятие адреса

```
int a = 5;
printf("%p\n", &a); //0x10002000
```

Указатели. Переменная типа адрес.

Размер указателя зависит от ОС: для 32-битной системы будет равен 4 байта, для 64-битной — 8 байт.

```
int *int_pointer;
char *string;
void *ptr;
```

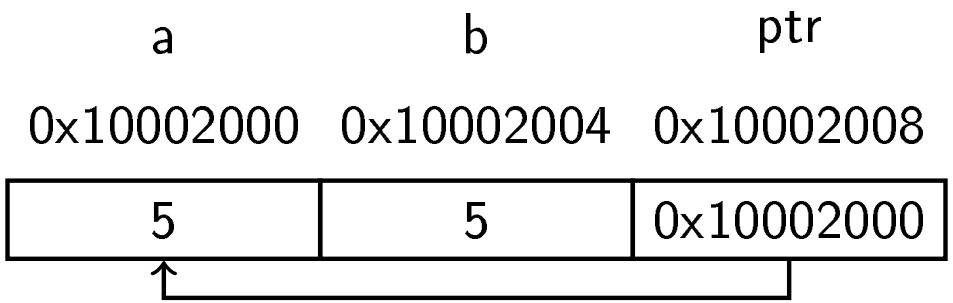
Указатель

```
int a = 5;
int *ptr = &a;
```

имя	a	ptr
адрес	0x10002000	0x100020004
значение	5	0x10002000

Указатели. «Косвенное» обращение к переменной

```
int a = 5, b;
int *ptr = &a;   //ptr = 0x10002000
b = *ptr;        //b = 5
```

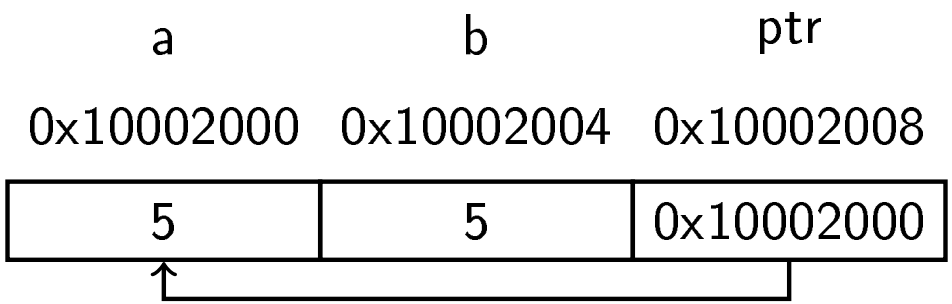


Указатели. Зачем нужно указывать тип указателя №1

```
char a = '+';
char *ptr = &a;
size_t n = sizeof(*ptr);
size_t k = sizeof(ptr);
```

Указатели. «Косвенное» изменение переменной

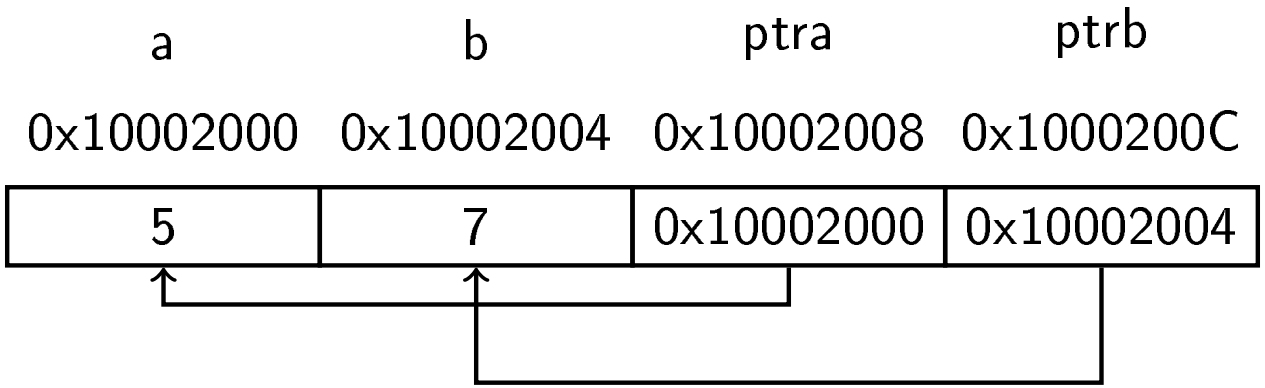
```
int a = 5;
int *ptr = &a;   //ptr = 0x10002000
*ptr = b;        //a = 5
```



Указатели. Приоритет

Приоритет выше арифметических операторов:

```
int a = 5;
int b = 7;
int *ptra = &a, *ptrb = &b;
*ptra = *ptra + *ptrb;          // a = a + b
printf("%d␣%d\n", a, b);        // 12 7
```



Указатели. Приоритет

Приоритет ниже инкремента.

```
int a = 5;
int *ptr = &a;
```

Что будет?

```
(*ptr)++;
```

```
*(ptr++);
```

```
*ptr++;
```

Указатели. Самая любимая ошибка

Разыменовывать можно только указатели, в которых хранятся адреса существующей памяти (переменных).

```
int *ptr;
*ptr = 5;
```

Что будет?

Указатели. Два золотых правила работы с указателями

1) Инициализировать указатели сразу при создании. Если необходимо объявить указатель, который пока никуда не указывает, его следует присвоить нулевому указателю с помощью специальной константы NULL.

```
int *ptr = NULL;
```

2) перед попыткой его разыменовать, проверять можно ли так делать.

```
int a = 5;
int *ptr = NULL;
if (ptr != NULL)                    // False
    printf("%d\n", *ptr);
ptr = &a;
if (ptr != NULL)                    // True
    printf("%d\n", *ptr);
```

Указатели. Два почти тождества

- `*(&a) == a` — верно всегда;
- `&(*a) == a` — верно, только если a — указатель на существующую переменную.

Указатели. Два почти тождества

Указатели. Зачем нужно указывать тип указателя №2

Смещение на соседние ячейки:

```
int a = 10;
int *ptr = &a;
ptr++;
--ptr;
ptr += 4;
ptr = ptr - 4;
```

Величина сдвига определяется из типа указателя (т.е. на `sizeof(тип)`)

Указатели на массивы

Пример на массивах.

```
char s[] = "math";
char *ptr = s;
```

Сколько здесь использовано памяти? Где она находится?

Указатели на массивах

```
*ptr = 'p';          // 'p', 'a', 't', 'h', '\0'
```

Чтобы изменить

```
*(ptr + 2) = 'l'; // 'p', 'a', 'l', 'h', '\0'
*(ptr + 3) = 'm'; // 'p', 'a', 'l', 'm', '\0'
```

Указатели. Еще одно тождество

В общем виде для массива a[] верно следующее соотношение:

```
a[i] == *(a + i);
```

Данное соотношение приводит еще к одному интересному соотношению:

```
&a[i] == &(*(a + i)) == a + i;
```

Указатели. Полезное применение

Для считывания массива:

```
int a[5], i;
for (i = 0; i < 5; i++) {
    scanf("%d", a + i);
}
```

С помощью указателя можно распечатать строку без индексирования строки как массива, передвигая указатель:

```
char s[5] = "math";
char *ptr;
for (ptr = s; *ptr; ptr++) {
    putchar(*ptr);
}
```

Указатели. Бесполезное применение

```
a[i] == i[a];
```

Как следствие

```
char a[] = "math";
2[a] = 's';
puts(a);
```

Динамическая память.

Статическая память — стек.
Динамическая память — куча.

Динамическая память.

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

Выделение непрерывной области памяти заданного размера n байт:

```
void *malloc(size_t n)
```

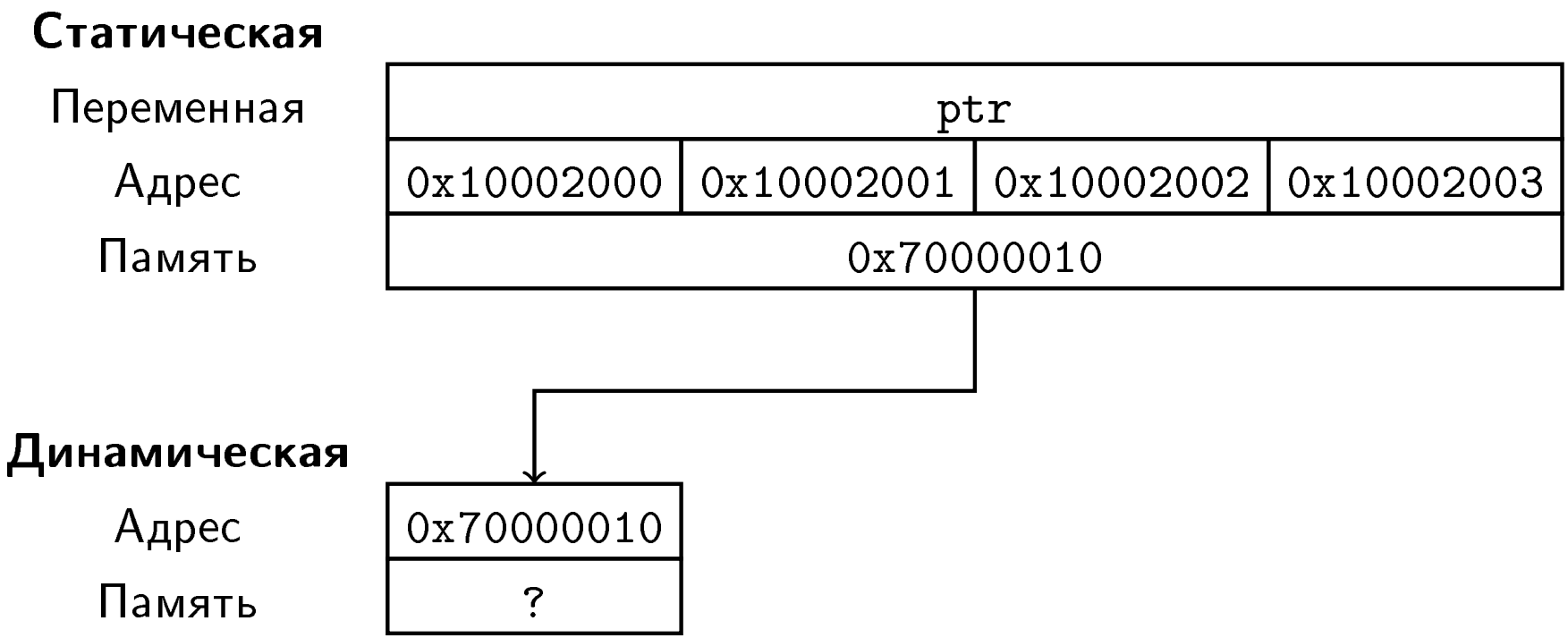
Освобождение памяти:

```
void free(void *ptr);
```

Золотое правило — кто создает, тот и чистит.

Динамическая память. Один байт.

```
char *ptr;
ptr = malloc(1);
```



Динамическая память. Много байт.

```
int *ptr = malloc(10 * sizeof(int));
ptr[8] = 8;
*(ptr + 9) = 9;
```

Динамическая память. Плоский код

Сгенерировать массив из 10 целых чисел от 1 до 10.

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

int main() {
    int n = 10, i;
    int *ptr = malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
        ptr[i] = i + 1;
    for (i = 0; i < n; i++)
        printf("%d␣", ptr[i]);
    free(ptr)
    return 0;
}
```

Динамическая память. Процедурный код

```
int *generate(int n) {
    int i, *ptr = malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
        ptr[i] = i;
    return ptr;
}
void print(int n, int *ptr) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d␣", ptr[i]);
}
int main() {
    int *ptr = generate(5);
    print(5, ptr);
    free(ptr);
    return 0;
}
```

Динамическая память. Нужно больше функций

```
void *calloc(size_t n, size_t size);
```

Выделяет массив из n элементов, каждый из которых имеет размер size байт (всего n*size байт).

```
void* realloc(void* ptr, size_t size);
```

1. если можно, то расширить массив, иначе выделить новую память размером size;
2. при необходимости скопировать элементы из исходного массива во второй (из памяти под указателем ptr в память под указателем new_ptr);
3. очистить исходную память (под указателем ptr).

Динамическая память. Скорость работы.

Работает быстрее чем вы думаете! Почему?

```
char *ptr = NULL;
for (size_t n = 1; n <= 1000000; n++) {
    ptr = realloc(ptr, n);
}
```

Динамическая память vs статическая память.

1. именованные переменные.
2. время жизни.
3. место для памяти.
4. возможность узнать размера массива.
5. размер массива: `n * sizeof(type)` и `n * sizeof(type) + sizeof(type*)`.
6. представление матриц.

Пример. Простые числа.

Дано целое число n от 1 до 100. В динамический массив записать все простые числа, которые не превосходят n. Вывести количество простых чисел и сами числа по возрастанию.

Ввод	8	2
Вывод	4 2 3 5 7	0

Пример. Простые числа.

```
int main() {
    int* ans = NULL;
    int m = 0, i, n;
    scanf("%d", &n);
    for (i = 2; i <= n; i++)
        if (isprime(i)) {
            ans = realloc(ans, (m + 1) * sizeof(int));
            ans[m] = i;
            m++;
        }
    printf("%d\n", m);
    for (i = 0; i < m; i++)
        printf("%d␣", ans[i]);
    putchar('\n');

    if (ans != NULL)
        free(ans);
    return 0;
}
```

Пример. Фильтрация массива.

Дано целое положительное n. Далее n целых чисел. Вывести только четные числа.

Ввод	3 14 11 12
Вывод	14 12

Описать функции без возвращаемого значения.

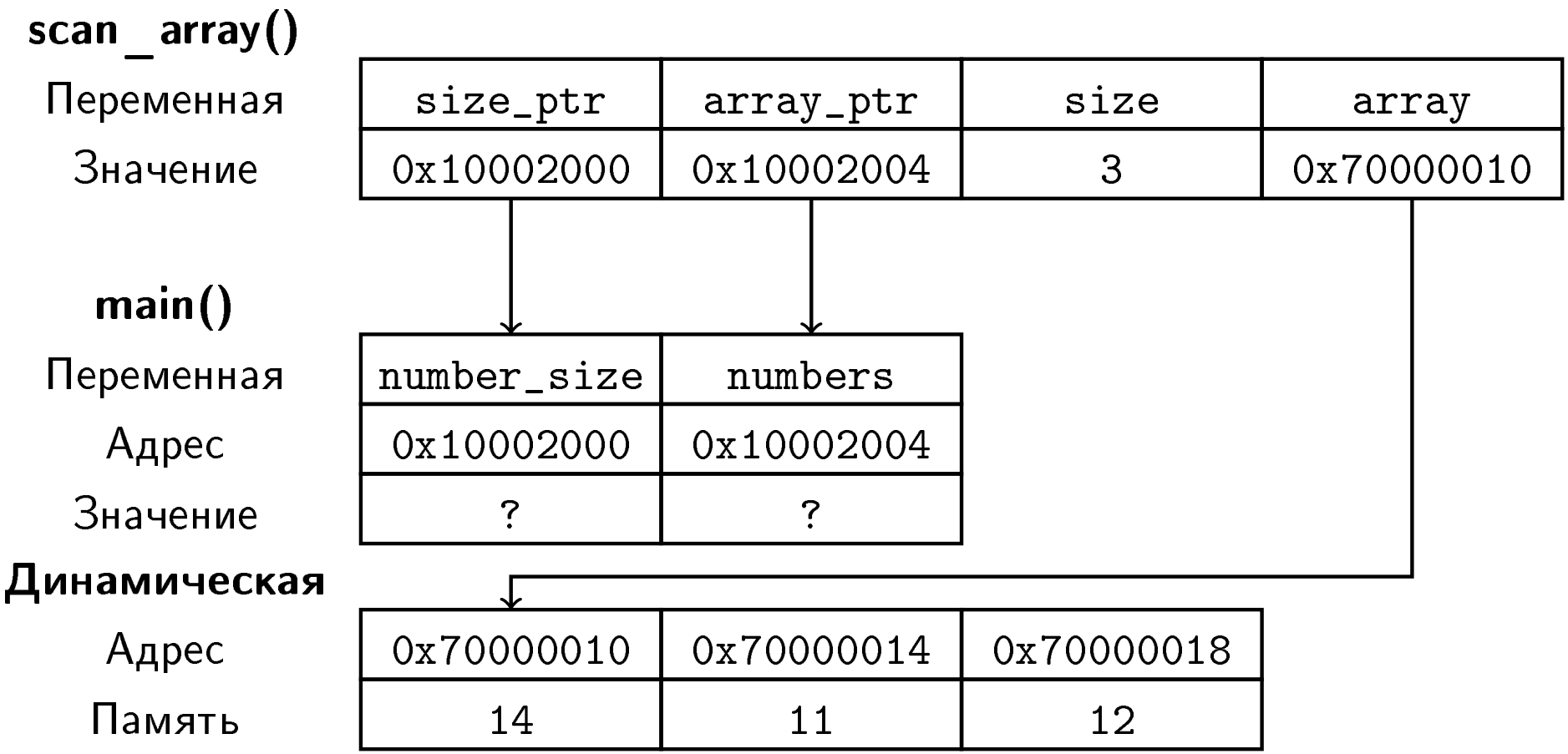
```
int main() {
    int *numbers, *even_numbers;
    int number_size, even_size;
    scan_array(&number_size, &numbers);
    even_filter(number_size, numbers,
                &even_size, &even_numbers);
    free(numbers);
    print_array(even_size, even_numbers);
    if (even_numbers != NULL)
        free(even_numbers);
    return 0;
}
```

Пример. Считать массив.

```
void scan_array(int *size_ptr, int **array_ptr) {
    int *array = NULL, size, i;
    scanf("%d", &size);
    array = malloc(size * sizeof(int));
    for (i = 0; i < size; i++)
        scanf("%d", &array[i]);

    *size_ptr = size;
    *array_ptr = array;
}
```

Пример. Считать массив.



Пример. Отфильтровать массив.

```
void even_filter(int src_size, int * src_array,
                 int * dst_size_ptr, int ** dst_array_pt
    int i;
    int *array = NULL;
    int size = 0;
    for (i = 0; i < src_size; i++)
        if (src_array[i] % 2 == 0) {
            array = (int*) realloc(array, (size + 1) * s
            array[size] = src_array[i];
            size++;
        }
    *dst_size_ptr = size;
    *dst_array_ptr = array;
    return;
}
```

Пример. Полезный

Дана строка из слов. Слова разделены пробелами, каждое слово состоит из печатных символов, отличных от пробела, табуляции и переноса строки. Считать каждое слово в динамический массив. Вывести слова.

```
char *get_word(char *last_char_ptr);
```

Пример. Полезный

```
char *get_word(char *last_char_ptr) {
    char delimiter = ' ', final = '\n';
    char *word = NULL, ch;
    int n = 0;

    ch = getchar();
    while (ch != delimiter && ch != final) {
        word = realloc(word, (n + 1) * sizeof(char));
        word[n] = ch;
        n++;
        ch = getchar();
    }
    word = realloc(word, (n + 1) * sizeof(char));
    word[n] = '\0';
    *last_char_ptr = ch;
    return word;
}
```

Пример. Полезный

```
int main() {
    char last_char = '\0';
    char *word = NULL;
    word = get_word(&last_char);
    while (last_char != '\n') {
        puts(word);
        free(word);
        word = get_word(&last_char);
    }
    return 0;
}
```

Статические матрицы

Массив `int a[2]` — указатель типа `int *`.
Матрица типа `int [2][3][4]` — указатель типа `int (*)[2][3]`.
Почему именно первая размерность?

Следующие прототипы одинаковы:

```
void f(int array[2][3][4]);
void f(int (*array)[3][4]);
```

Статические матрицы

Обратите внимание на круглые скобки.

```
int (*array)[3][4]
```

```
int *array[3][4]
```

Статические матрицы — плотно упакованные

```
void *memset(void *s, int c, size_t n);
void *memcpy(void *dest, void *src, size_t n);
```

Инициализация всей матрицы.

```
#include <string.h>

const int n = 10, m = 20;

int A[n][m];
memset(A, 0, n * m * sizeof(int));

int B[n][m];
memcpy(B, A, n * m * sizeof(int));

int C[n * m];
memcpy(C, A, n * m * sizeof(int));
```

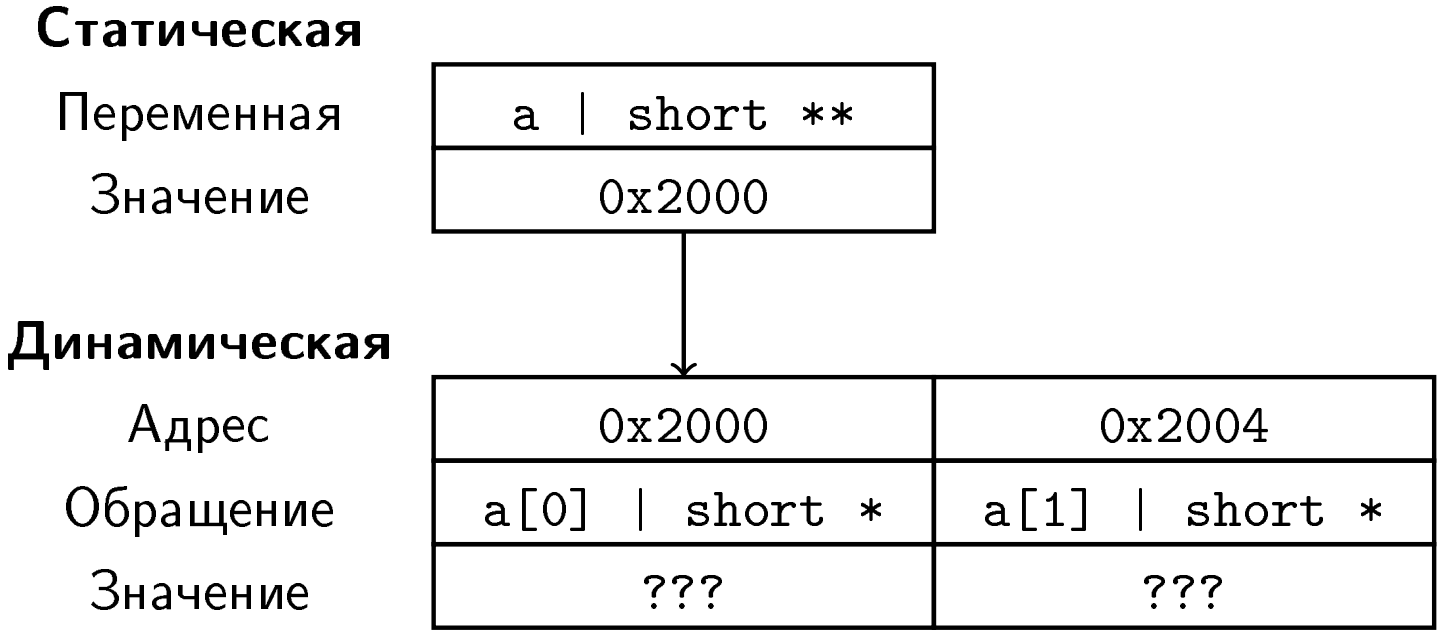
Динамические матрицы

1) уложить всю матрицу построчно в виде одного динамического массива (плотная упаковка);
2) создать дополнительный массив указателей на динамические массивы (неплотная упаковка).
Смотрим второе. Почему?

Динамические матрицы. Матрица 2x3

Каждая строка матрицы — динамический массив. Где хранить все эти указатели? Еще в одном динамическом массиве!

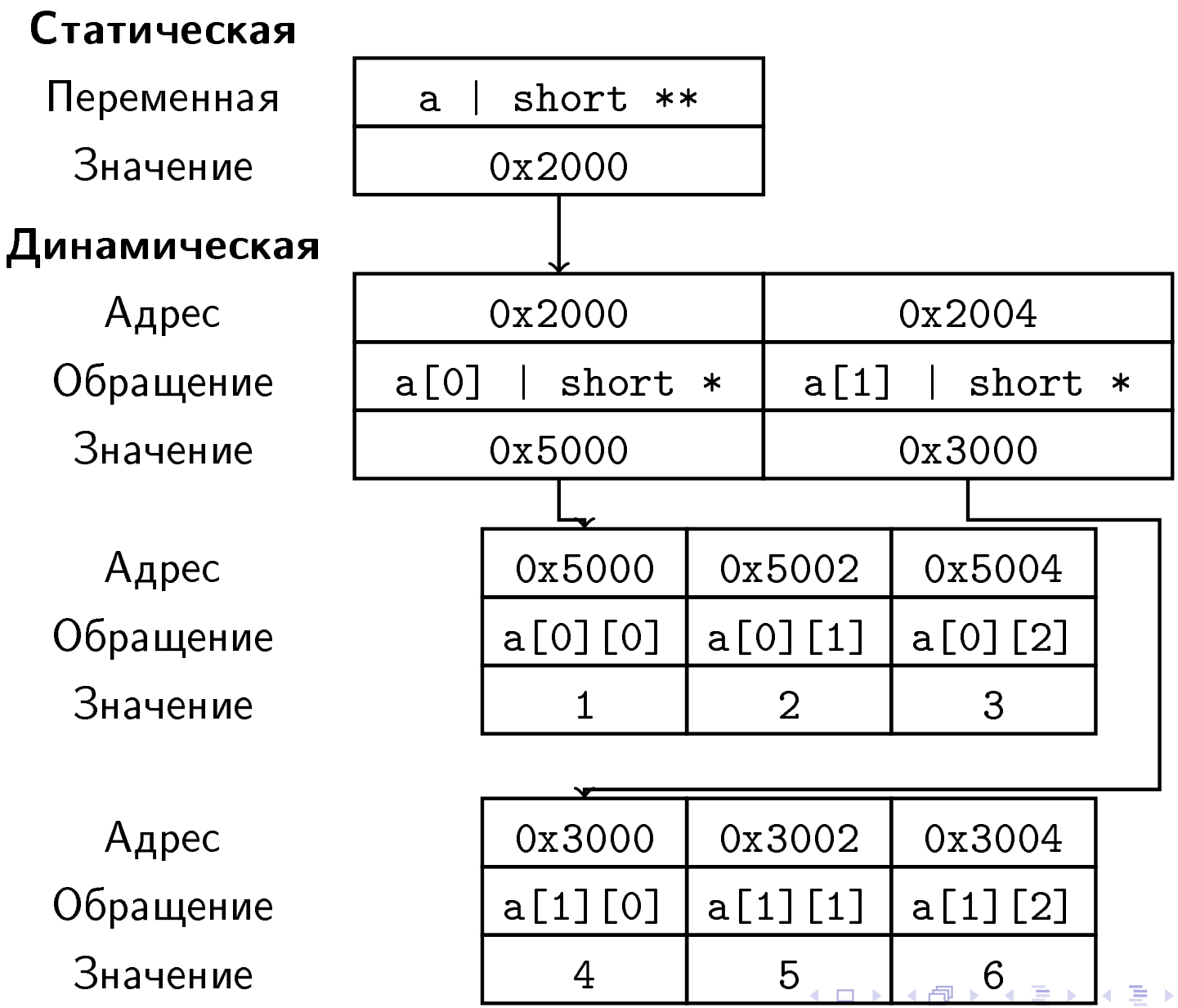
```
int **a = malloc(2 * sizeof(int *));
```



Динамические матрицы. Матрица 2x3

```
    a[0] = malloc(3 * sizeof(int));
    a[1] = malloc(3 * sizeof(int));
```

Динамические матрицы. Матрица 2x3



Динамические матрицы. Матрица 2x3

Очистка

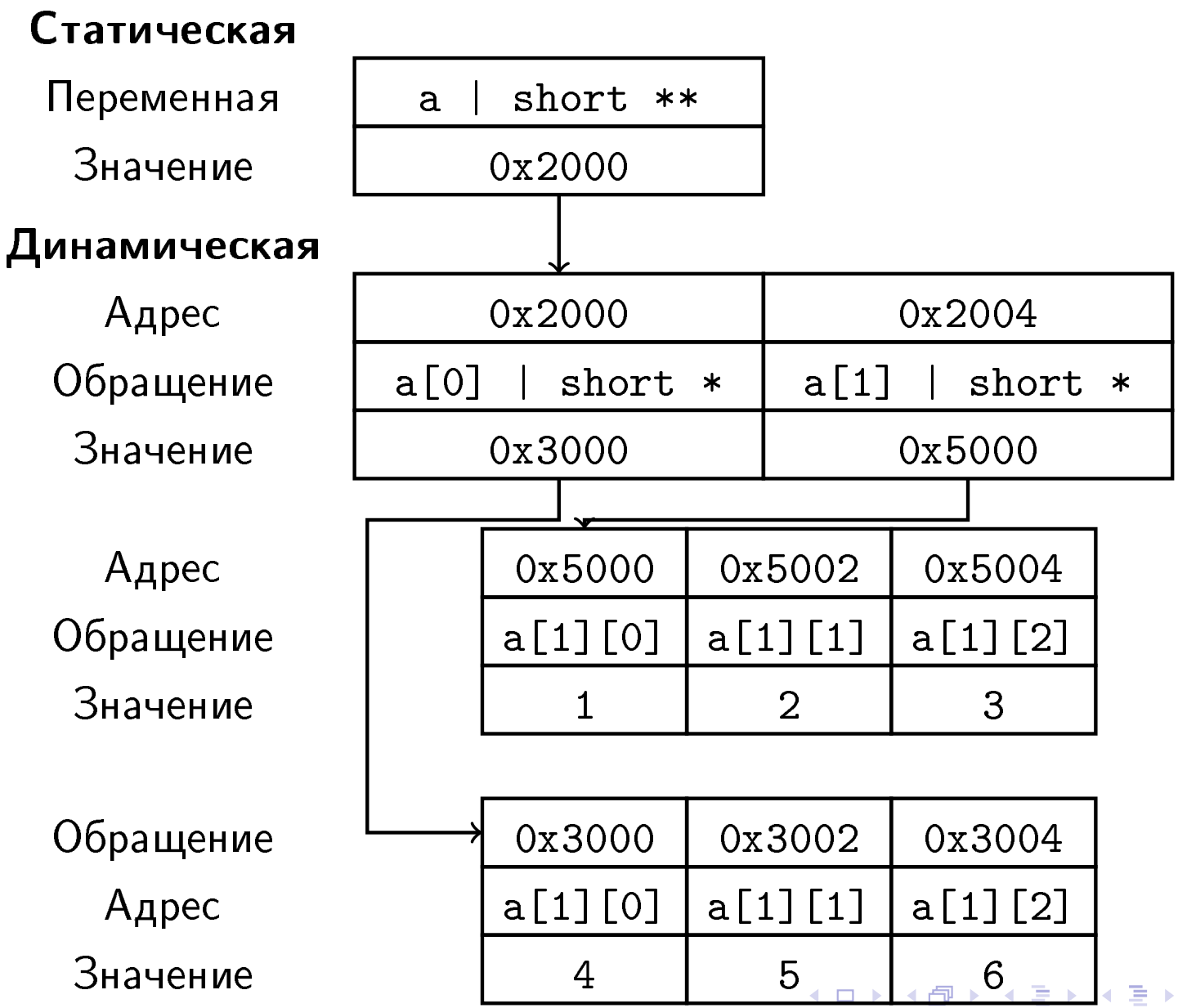
```
free(a[0]);
free(a[1]);
free(a);
```

Динамические матрицы. Любимая плюшка линала

Перестановка строк!

```
short *tmp = a[0];
a[0] = a[1];
a[1] = tmp;
```

Динамические матрицы. Матрица 2x3



Отличия динамической и статической матрицы.

1. матрица размера $n \times m$, где каждый элемент имеет размер s, а указатель — размер p. В случае со статической памятью: $n \cdot m \cdot s$. В случае с динамической памятью: $(1 + n) \cdot p$ для указателей и $n \cdot m \cdot s$ для самой матрицы.
2. динамические матрицы хранятся кусками по строкам, а статические — цельном блоком. Перестановка строк O(1).
3. Размеры разных строк могут быть разными.

Аргументы командной строки.

```
$ gcc prog.c -o prog -Wall
```

Здесь 4 аргумента:

```
int main(int argc, char** argv);
```

При запуске вашей программы, параметр argc будет содержать количество параметров командной строки, включая название программы (5).
Аргумент argv — это массив строк, которые сформированы из аргументов, включая название программы:

```
argv[0] == "gcc"
argv[1] == "prog.c"
argv[2] == "-o"
argv[3] == "prog"
argv[4] == "Wall"
```

Голомоломка.

0. Какие присваивания допустимы между указанными переменными (среди всех возможных пар)?

```
int a[2][3];
int (*b)[3];
int *c[3];
int **d;
```

1. Динамическая матрица из 3×4. Строки и столбы — динамические.
2. Динамическая матрица из 3×4. Строки — статические, столбы — динамические.
3. Динамическая матрица из 3×4. Строки — динамические, столбы — статические.
4. Написать программу, которая печатает свой исходный код, при условии, что он лежит рядом и имеет такое же название (prog и prog.c).
5. Написать программу, которая печатает свой бинарный код.

Динамическая память. Еще раз об ошибке сегментации

```
int *ptr = malloc(10);
ptr[9] = 0;
```

Может и не произойти. Массивы выделяются со смещением по степеням двойки.

Динамическая память. Еще раз об ошибке сегментации

```
int *ptr = malloc(10);
ptr[9] = 0;
```

Может и не произойти. Массивы выделяются со смещением по степеням двойки.

Отладка в gdb

Компилируем

```
gcc prog.c -o prog -Wall -Werror -lm -g
```

Запускаем отладчик

```
gdb prog
```

Посмотрим код

```
list 1
```

Ставим точку останова (до которой программа будет выполняться в обычном режиме). Лучше ставить сразу после ввода.

```
break 6
```

Запускаем

```
run
```

Добавляем переменную наблюдения (можно несколько переменных)

```
display n
display ans
```

Делаем построчное выполнение (первый раз надо набрать команду целиком, потом просто Enter).

```
next
```

Выход

```
quit
```