

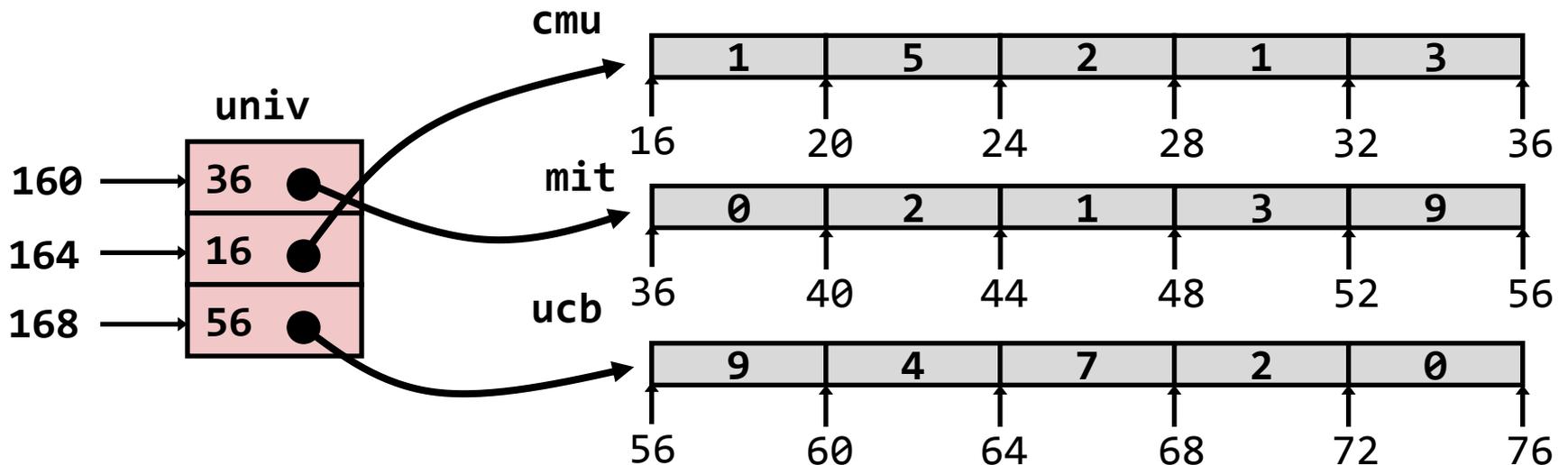
# Лекция ОхА

14 марта

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Переменная **univ** представляет собой массив из 3 элементов
- Каждый элемент – указатель (размером 4 байта)
- Каждый указатель ссылается на массив из **int**'ов



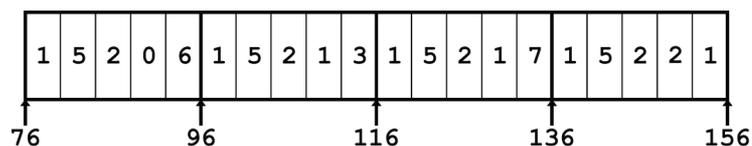
```
int get_univ_digit (int index, int dig) {  
    return univ[index][dig];  
}
```

```
mov    eax, dword [ebp + 8]           ; index  
mov    edx, dword [univ + 4 * eax]    ; p = univ[index]  
mov    eax, dword [ebp + 12]         ; dig  
mov    eax, dword [edx + 4 * eax]     ; p[dig]
```

- Доступ к элементу Mem[Mem[univ+4\*index]+4\*dig]
- Необходимо выполнить два чтения из памяти
  - Первое чтение получает указатель на одномерный массив
  - Затем второе чтение выполняет выборку требуемого элемента этого одномерного массива

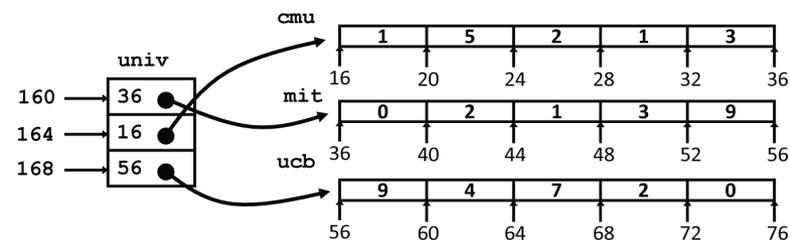
## Многомерный массив

```
int get_pgh_digit
(int index, int dig)
{
    return pgh[index][dig];
}
```



## Многоуровневый массив

```
int get_univ_digit
(int index, int dig)
{
    return univ[index][dig];
}
```



- Значительное внешнее сходство в Си
- Существенное различие в ассемблере

$\text{Mem}[\text{pgh} + 20 * \text{index} + 4 * \text{dig}]$

$\text{Mem}[\text{Mem}[\text{univ} + 4 * \text{index}] + 4 * \text{dig}]$

# Матрица N X N

- Фиксированные размерности
  - Значение N известно во время компиляции
- Динамически задаваемая размерность. Требуется явное преобразование индексов
  - Традиционный способ реализации динамических массивов
- Динамически задаваемая размерность с неявной индексацией.
  - Поддерживается последними версиями gcc

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele
  (fix_matrix a, int i, int j)
{
  return a[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele
  (int n, int *a, int i, int j)
{
  return a[IDX(n,i,j)];
}
```

```
/* Get element a[i][j] */
int var_ele
  (int n, int a[n][n], int i, int j) {
  return a[i][j];
}
```

# Матрица 16 X 16

## ■ Доступ к элементу матрицы

- Адрес  $A + i * (C * K) + j * K$
- $C = 16, K = 4$

```
/* Получение элемента a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
    return a[i][j];
}
```

```
mov    edx, dword [ebp + 12]    ; i
sal    edx, 6                  ; i*64
mov    eax, dword [ebp + 16]    ; j
sal    eax, 2                  ; j*4
add    eax, dword [ebp + 8]     ; a + j*4
mov    eax, dword [eax + edx]   ; *(a + j*4 + i*64)
```

# Матрица $n \times n$

## ■ Доступ к элементу матрицы

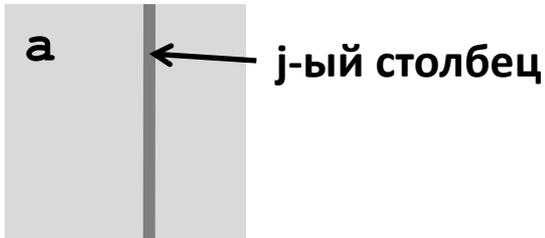
- Адрес  $A + i * (C * K) + j * K$
- $C = n, K = 4$

```
sizeof(a) = ?
sizeof(a[i]) = ?
```

```
/* Получение элемента a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
    return a[i][j];
}
```

```
mov    edx, dword [ebp + 8]      ; n
sal    edx, 2                   ; n*4
imul   edx, dword [ebp + 16]    ; i*n*4
mov    eax, dword [ebp + 20]    ; j
sal    eax, 2                   ; j*4
add    eax, dword [ebp + 12]    ; a + j*4
mov    eax, dword [eax + edx]   ; *(a + j*4 + i*n*4)
```

# Оптимизация доступа к элементам массива



```
#define N 16  
typedef int fix_matrix[N][N];
```

- Вычисления
  - Проход по всем элементам в столбце  $j$
- Оптимизация
  - Выборка последовательных элементов из отдельного столбца

```
/* Выборка столбца j из массива */  
void fix_column  
  (fix_matrix a, int j, int *dest)  
{  
  int i;  
  for (i = 0; i < N; i++)  
    dest[i] = a[i][j];  
}
```

# Оптимизация доступа к элементам массива

## • Оптимизация

– Вычисляем  $ajp = \&a[i][j]$

- Начальное значение  $a + 4*j$
- Шаг  $4*N$

Регистр	Значение
ecx	ajp
ebx	dest
edx	i

```
/* Выборка столбца j из массива */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

```
.L8:                                     ; loop:
  mov    eax, dword [ecx]                 ; считываем *ajp
  mov    dword [ebx + 4 * edx], eax       ; сохраняем в dest[i]
  add    edx, 1                            ; i++
  add    ecx, 64                           ; ajp += 4*N
  cmp    edx, 16                           ; i vs. N
  jne    .L8                               ; if !=, goto loop
```

# Оптимизация доступа к элементам массива

– Вычисляем  $ajp = \&a[i][j]$

- Начальное значение  $a + 4*j$
- Шаг  $4*n$

Регистр	Значение
ecx	ajp
edi	dest
edx	i
ebx	4*n
esi	n

```
/* Выборка столбца j из массива */
void var_column
(int n, int a[n][n],
 int j, int *dest)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = a[i][j];
}
```

```
.L18:                                     ; loop:
    mov     eax, dword [ecx]              ; считываем *ajp
    mov     dword [edi + 4 * edx], eax    ; сохраняем в dest[i]
    add     edx, 1                        ; i++
    add     ecx, ebx                      ; ajp += 4*n
    cmp     esi, edx                      ; n vs. i
    jg     .L18                          ; if (>) goto loop
```

# Оптимизация доступа к элементам массива

## – Изменение направления прохода по циклу

- Выход из цикла по нулевому счетчику
- Шаг отрицательный
- Меняются начальные значения указателей
- Достаточно вывести к нулю один из индексов

```
/* Выборка столбца j из массива */
void var_column
(int n, int a[n][n],
 int j, int *dest) {

    int i;
    for (i = n-1; i >=0; i--) {
        dest[i] = a[i][j];
    }
}
```

.L18:

```
mov    eax, dword [ecx]
mov    dword [edi + 4 * edx], eax
add    edx, 1
add    ecx, ebx
cmp    esi, edx
jg     .L18
```

; loop:

```
; считываем *ajp
; сохраняем в dest[i]
; i++
; ajp += 4*n
; n vs. i
; if (>) goto loop
```

# Оптимизация доступа к элементам массива

Регистр	Начальное значение
ecx	$a + 4 * n * (n - 1) + 4 * j$
edi	dest - 4
edx	n
ebx	$4 * n$
esi	<b>освободился</b>

```

/* Выборка столбца j из массива */
void var_column
(int n, int a[n][n],
 int j, int *dest) {

    int i;
    dest--;
    for (i = n; i != 0; i--)
        dest[i] = a[i-1][j];
}

```

```

.L18:
    mov     eax, dword [ecx]
    mov     dword [edi + 4 * edx], eax
    sub     ecx, ebx
    sub     edx, 1
    jnz    .L18
; loop:
; считываем *(ajp+...)
; сохраняем в dest[i]
; ajp -= 4*n
; i--
; if (!=) goto loop

```

# Обратная задача

M = ?, N = ?

```

; пролог функции пропущен
mov  ecx, dword [ebp + 8]      ; 1
mov  edx, dword [ebp + 12]    ; 2
lea  eax, [8 * ecx]           ; 3
sub  eax, ecx                  ; 4
add  eax, edx                  ; 5
lea  edx, [edx + 4 * edx]     ; 6
add  edx, ecx                  ; 7
mov  eax, dword [m1 + 4 * eax] ; 8
add  eax, dword [m2 + 4 * edx] ; 9
; эпилог функции пропущен

```

```

int m1[M][N];
int m2[N][M];

int sum_element(int i, int j) {
    return m1[i][j] + m2[j][i];
}

```

# Типы данных языка Си

## Далее

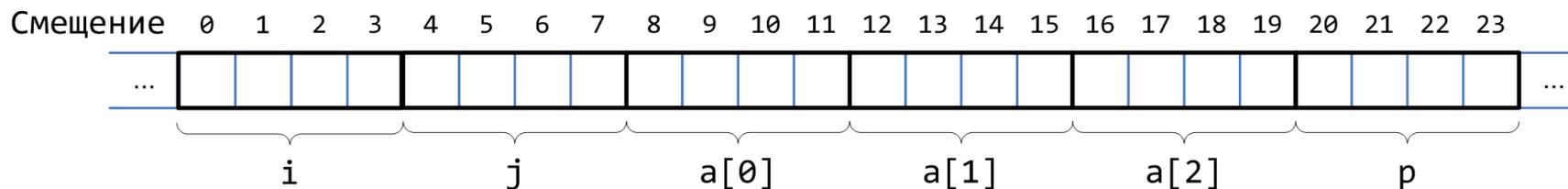
- char
- Стандартные знаковые целочисленные типы
  - signed char
  - short int
  - int
  - long int
  - long long int
- Стандартные беззнаковые целочисленные типы
  - `_Bool`
- Перечисление
- Типы чисел с плавающей точкой
  - float
  - double
  - long double
  - `_Complex`
- Производные типы
  - Массивы
  - **Структуры**
  - **Объединения**
  - Указатели
  - Указатели на функции

# Структуры

```
struct rec {  
    int i;  
    int j;  
    int a[3];  
    struct rec *p;  
}
```

- Непрерывный блок памяти
- Обращение к полям структуры осуществляется по их именам
- Поля могут быть разных типов

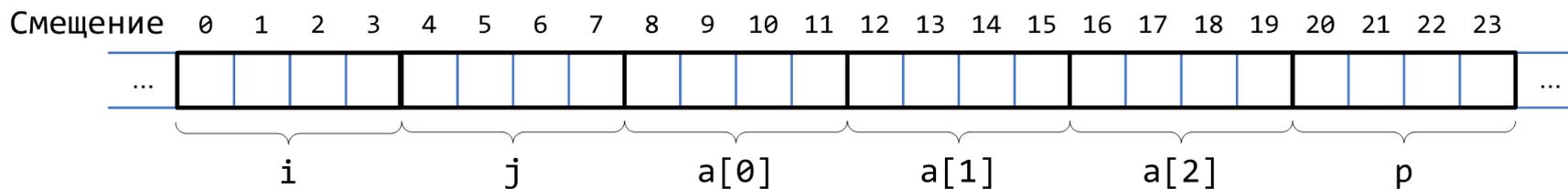
## Расположение в памяти



# Доступ к полям

```
struct rec {
    int i;
    int j;
    int a[3];
    struct rec *p;
}
```

- `struct rec *x` – указатель на первый байт структуры
- Каждое поле расположено на определенном смещении от начала структуры



```
static struct rec *x;
...
x->j = x->i;
```

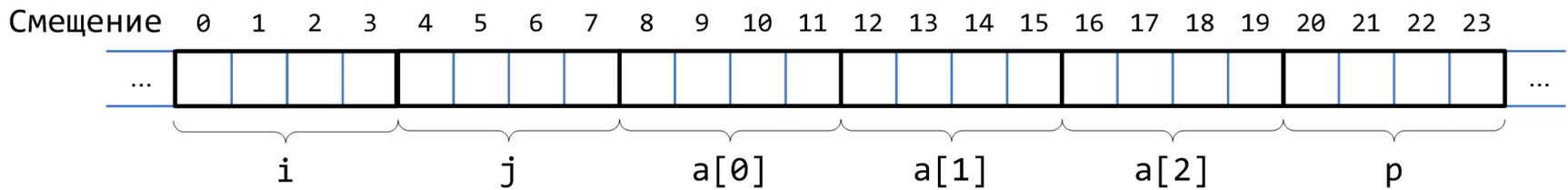
```
mov edx, dword [x] ; (1)
mov eax, dword [edx] ; (2)
mov dword [edx + 4], eax ; (3)
```

# Указатель на поле структуры

```
struct rec {
    int i;
    int j;
    int a[3];
    struct rec *p;
}
```

- Смещение каждого поля известно во время компиляции

```
static struct rec *x;
static int i;
...
&(x->a[i]);
```



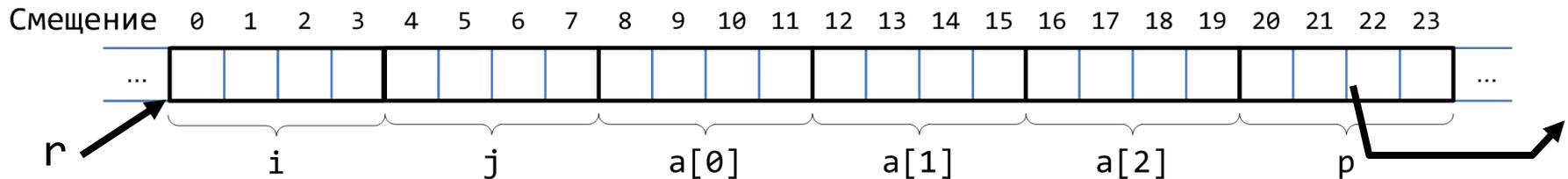
```
mov edx, dword [i] ; (1)
mov eax, dword [x] ; (2)
lea eax, [eax + 4 * edx + 8] ; (3)
```

## • Проход по связанному списку

```
void set_val (struct rec *r, int val) {
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->p;
    }
}
```

```
struct rec {
    int i;
    int j;
    int a[3];
    struct rec *p;
}
```

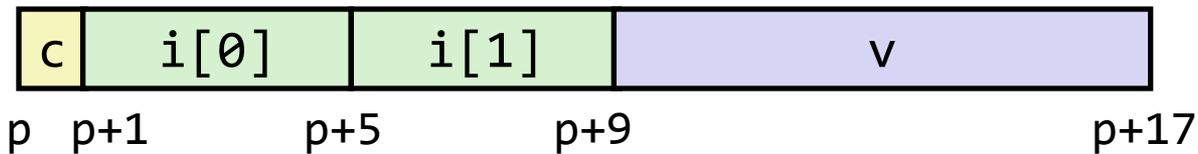
Регистр	Переменная
edx	r
ecx	val



```
.L17: ; цикл
mov    eax, [edx]          ; r->i
mov    [edx + 4 * eax + 8], ecx ; r->a[i] = val
mov    edx, [edx + 20]     ; r = r->p
test   edx, edx           ; r?
jne    .L17               ; If != 0 goto .L17
```

# Выравнивание полей в структурах

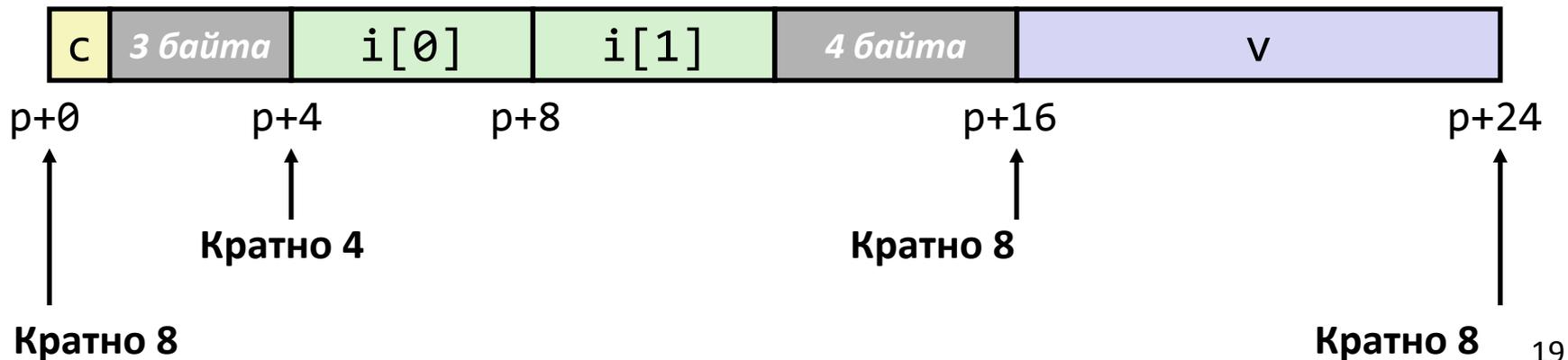
- Невыровненные данные



```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

- Выровненные данные

- Если примитивный тип данных требует  $K$  байт
- Адрес должен быть кратен  $K$



# Почему выравнивают данные

- Выровненные данные
  - Размер примитивного типа данных  $K$  байт
  - Адрес должен быть кратен  $K$
  - Для некоторых архитектур это требование обязательно должно выполняться
  - Для IA-32 требование к выравниванию имеет рекомендательный характер
    - Требования **различаются** для IA-32/x86-64, Linux/Windows
- Причины
  - Доступ к физической памяти осуществляется блоками (выровненными) по 4 или 8 байт (зависит от аппаратуры)
    - Эффективность теряется при обращении к данным, расположенным в двух блоках
    - Виртуальная память...
- Компилятор
  - Расставляет пропуски между полями для сохранения выравнивания

# Правила выравнивания (IA-32)

- 1 байт : `char`, ...
  - Ограничений нет
- 2 байта : `short`, ...
  - Младший бит адреса должен быть  $0_2$
- 4 байта : `int`, `long`, `float`, `char *`, ...
  - Два младших бита адреса должны быть  $00_2$
- 8 байт : `double`, ...
  - Windows ( и другие ...):
    - Младшие три бита адреса должны быть  $000_2$
  - Linux:
    - Два младших бита адреса должны быть  $00_2$
    - Т.е. рассматриваются как и 4-байтные примитивные типы данных
- 12 байт : `long double` (gcc)
  - Windows, Linux:
    - Два младших бита должны быть  $00_2$
    - Т.е. рассматриваются как и 4-байтные примитивные типы данных

# Правила выравнивания (x86-64)

- 1 байт : char, ...
  - Ограничений нет
- 2 байта : short, ...
  - Младший бит адреса должен быть  $0_2$
- 4 байта : int, float, ...
  - Два младших бита адреса должны быть  $00_2$
- 8 байт: double, long, char \*, ...
  - Windows & Linux:
    - Младшие три бита адреса должны быть  $000_2$
- 16 байт: long double
  - Linux:
    - Младшие три бита адреса должны быть  $000_2$
    - Т.е. рассматриваются как и 8-байтные примитивные типы данных

Тип long double в компиляторе MS VC

- 16-разрядная архитектура
  - `sizeof(long double) = 10 // 80 бит`
- 32-разрядная архитектура и далее ...
  - `long double ≡ double`

# Выполнение правил выравнивания для полей

- Внутри структуры
  - Выравнивание должно выполняться для каждого поля
- Размещение всей структуры
  - Для каждой структуры определяется требование по выравниванию в **К** байт
    - **К** = Наибольшее выравнивание среди всех полей
  - Начальный адрес структуры и ее длина должны быть кратны **К**
- Пример (для Windows или x86-64):
  - **К** = 8, из-за присутствия поля типа double

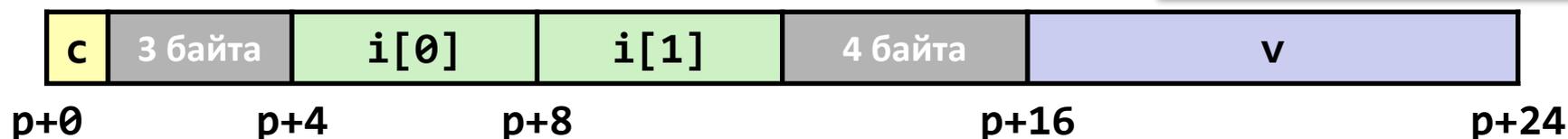
```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```



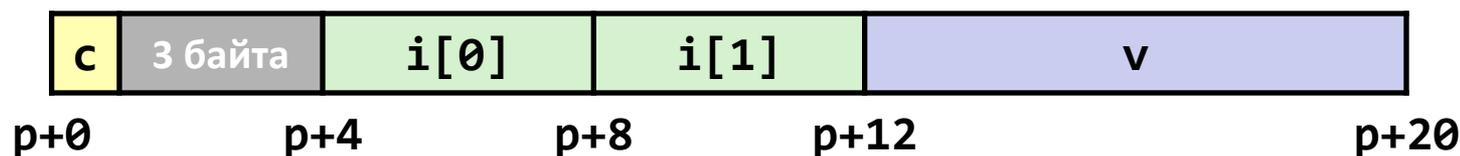
## Различные соглашения о выравнивании

- x86-64 или IA-32 Windows:
  - $K = 8$ , из-за наличия поля типа **double**

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```



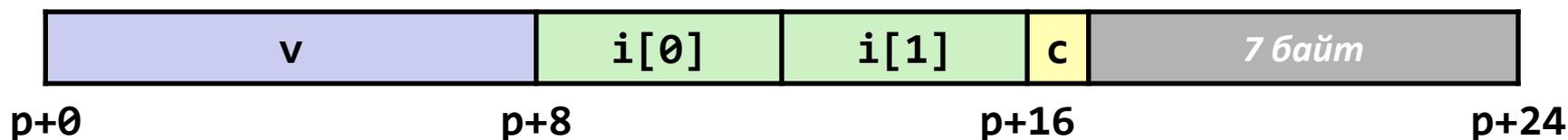
- IA-32 Linux
  - $K = 4$ ; **double** рассматривается аналогично 4-байтным типам данных



# Выравнивание всей структуры

- Определяется требование к выравниванию в К байт
- Общий размер структуры должен быть кратен К

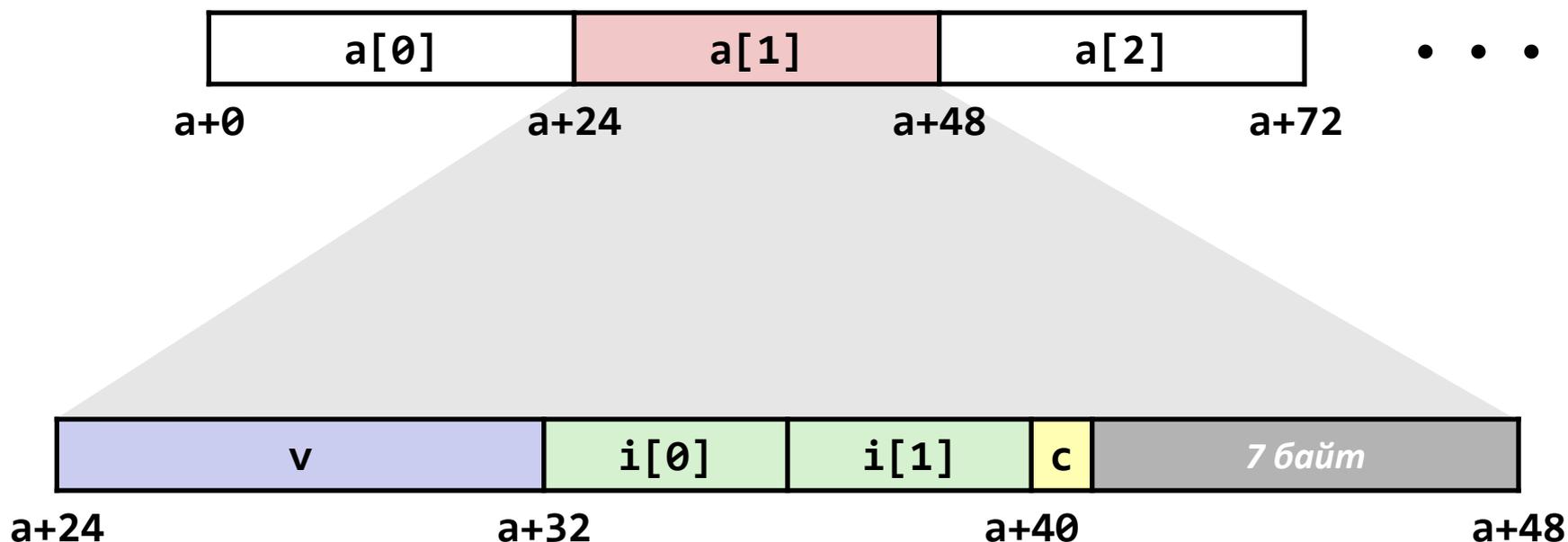
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



# Массивы структур

- Размер всей структуры кратен  $K$
- Для каждого элемента массива производится выравнивание

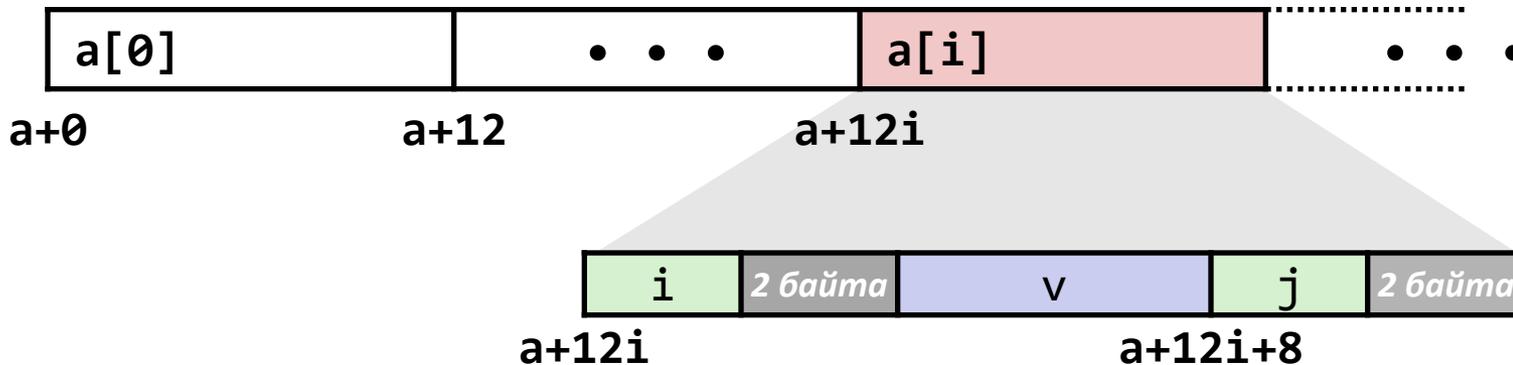
```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```



# Доступ к элементам массива

- Вычисляем смещение в массиве
  - Вычисляем `sizeof(S3)`, учитывая пропуски
- Вычисляем смещение внутри структуры
  - Поле `j` расположено со смещением 8 внутри структуры

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```



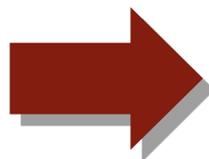
```
short get_j(int idx) {
    return a[idx].j;
}
```

```
; eax = idx
lea eax, [eax + 2 * eax] ; 3*idx
movsx eax, word [a + 4 * eax + 8]
```

## Как сохранить место

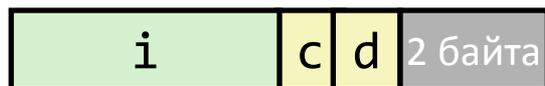
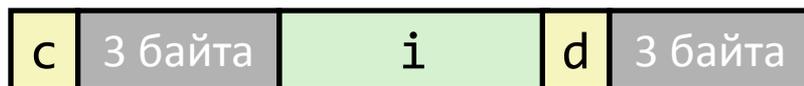
- Размещаем большие типы первыми

```
struct S4 {  
    char c;  
    int i;  
    char d;  
} *p;
```



```
struct S5 {  
    int i;  
    char c;  
    char d;  
} *p;
```

- Результат (K=4)

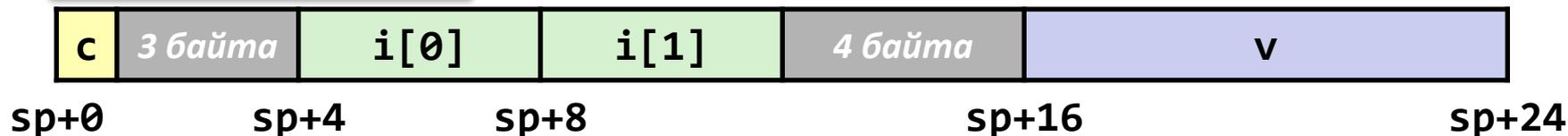
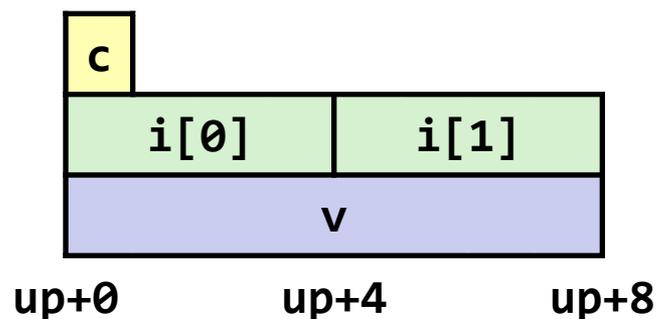


# Размещение объединений

- Память выделяется исходя из размеров максимального элемента
- Используется только одно поле

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



# Пример: двоичное дерево

```
struct NODE_S {  
    struct NODE_S *left;  
    struct NODE_S *right;  
    double data;  
};
```



```
union NODE_U {  
    struct {  
        union NODE_U *left;  
        union NODE_S *right;  
    } internal;  
    double data;  
};
```

В чем ошибка ?

# Пример: двоичное дерево

```
typedef enum {N_LEAF, N_INTERNAL} nodetype_t;

struct NODE_T {
    nodetype_t type;
    union NODE_U {
        struct {
            struct NODE_T *left;
            struct NODE_T *right;
        } internal;
        double data;
    } info;
};
```

sizeof(struct NODE\_T) ?

Какие смещения у полей?