Машинный уровень 4: Данные

Основы информатики.

Компьютерные основы программирования

goo.gl/X7evF

На основе **CMU** 15-213/18-243:
Introduction to Computer Systems

goo.gl/Q7vgWw

Лекция **7**, 19 марта, 2018

Лектор:

Дмитрий Северов, кафедра информатики 608 КПМ

dseverov@mail.mipt.ru

cs.mipt.ru/wp/?page_id=346

Ещё машинный уровень
Управление и сложные данные

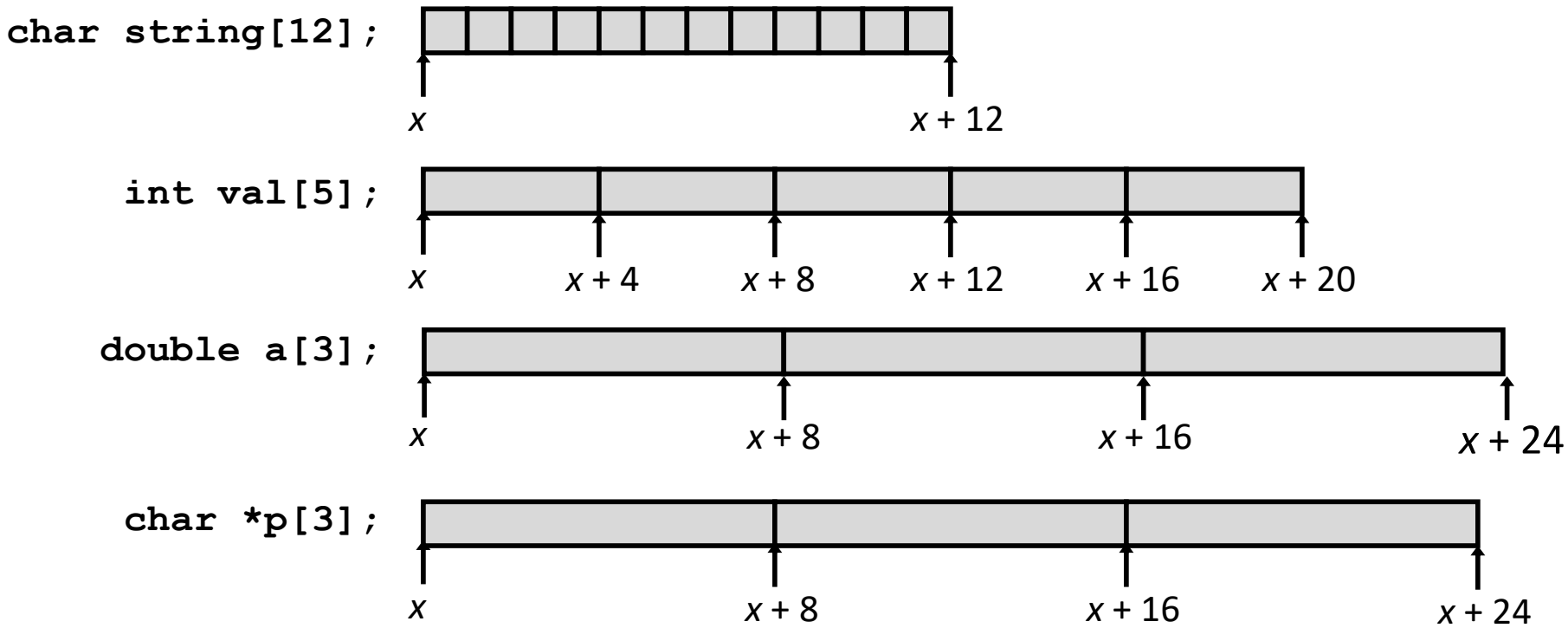
- **Массивы**
  - Одномерные
  - Многомерные (массивы массивов)
  - Многоуровневые
- **Структуры**
  - Размещение
  - Доступ
  - Выравнивание
- **Объединения**
- **Распределение памяти**
- **О переполнении буфера**

Размещение массивов

- **Основные принципы**

  T **`A[`***L***`];`**

  - Массив данных типа *T* и длинны *L*
  - Вплотную занятый фрагмент длиной *L* * **`sizeof`** (*T*) байт



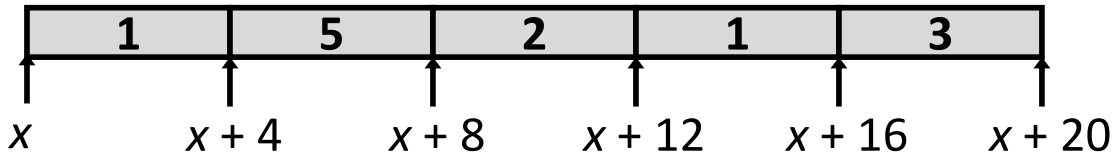
Доступ к массиву

- **Основные принципы**

  T **`A[`***L***`];`**

  - Массив данных типа *T* и длиной *L*
  - Идентификатор **`A`** можно использовать как указатель на элемент с индексом 0: Type *T**

```
int val[5];
```

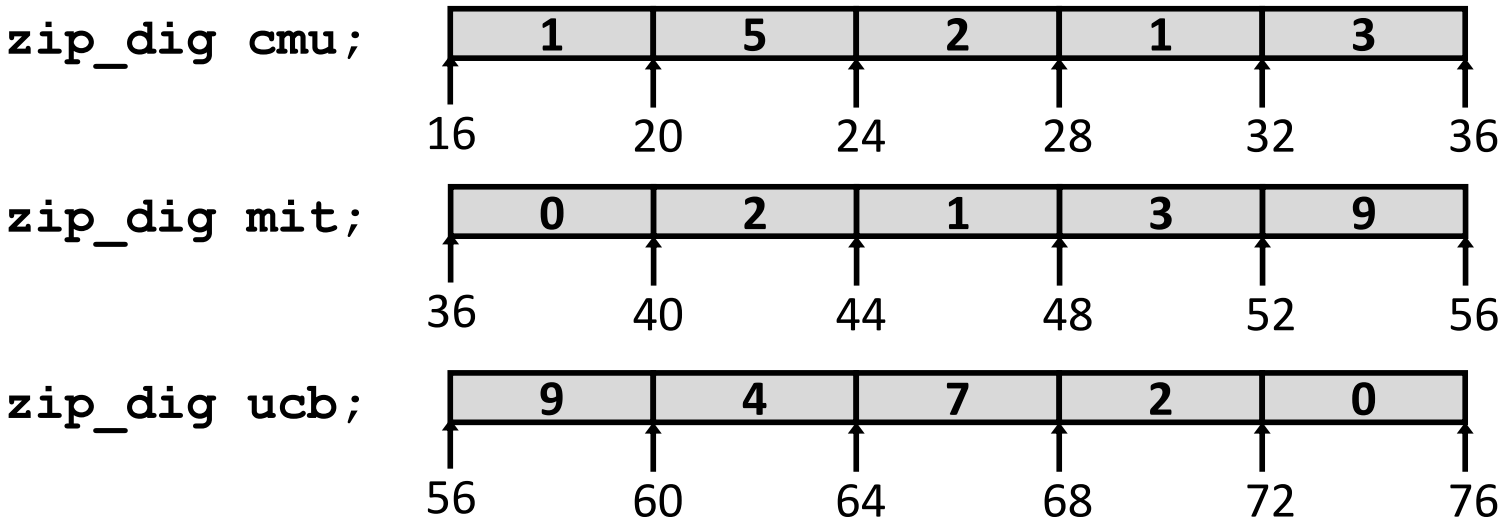


- **Обращение** **Тип** **Значение**

Обращение	Тип	Значение
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

Пример массива

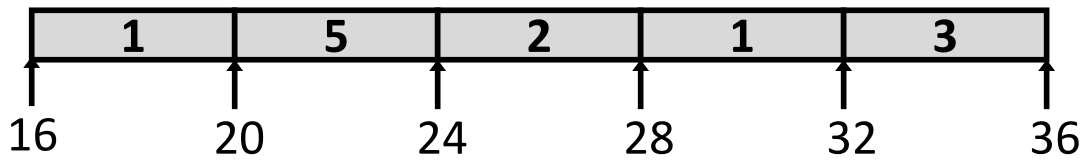
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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



- **Объявление "`zip_dig cmu`" эквивалентно "`int cmu[5]`"**
- **Память занимается блоками по 20 байт вплотную?**
  - Выполнение в общем случае не гарантируется!

Пример доступа к массиву

```
zip_dig cmu;
```




```
int get_digit
  (zip_dig z, int digit)
{
  return z[digit];
}
```

x86-64

```
  # %rdi = z
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- В `%rdi` - начальный адрес массива
- В `%rsi` - индекс массива
- Целевой адрес `4*%rdi + %rsi`
- Обращение к памяти `(%rdi,%rsi,4)`

Пример цикла с массивом

```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   z[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle
  cmpq    $4, %rax            #   i:4
  jbe     .L4                 #   if <=, goto loop
  rep; ret
```

Многомерные массивы массивов

- **Объявление**

  T **A**[*R*][*C*];

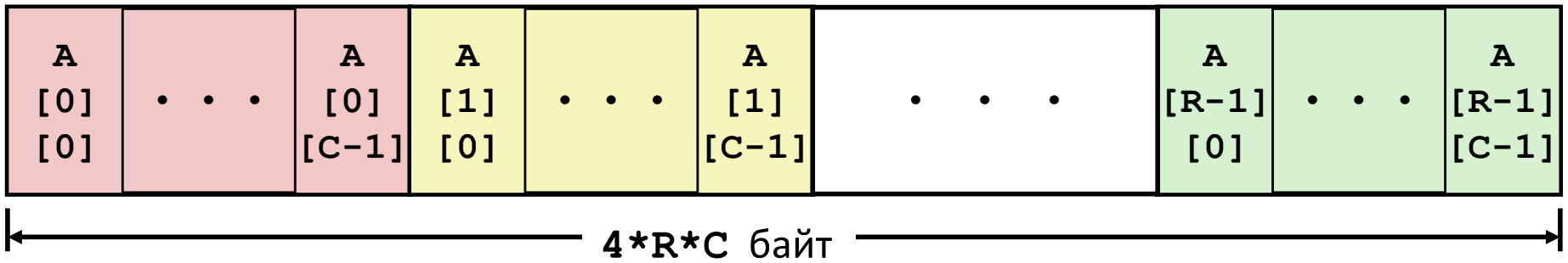
  - 2D массив данных типа *T*
  - *R* строк, *C* столбцов
  - Элемент типа *T* длиной *K* байт

- **Размер массива**
  - *R* * *C* * *K* байт

- **Организация**
  - Построчное упорядочение

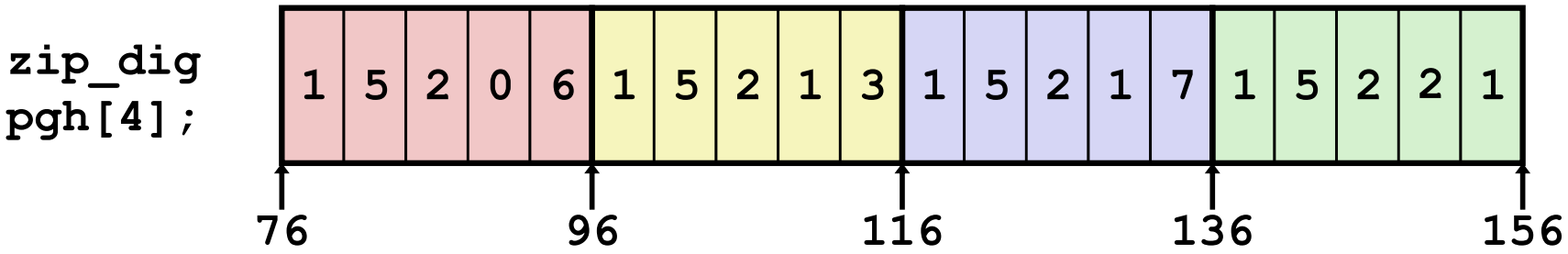
$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

```
int A[R][C];
```



Пример массива массивов

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

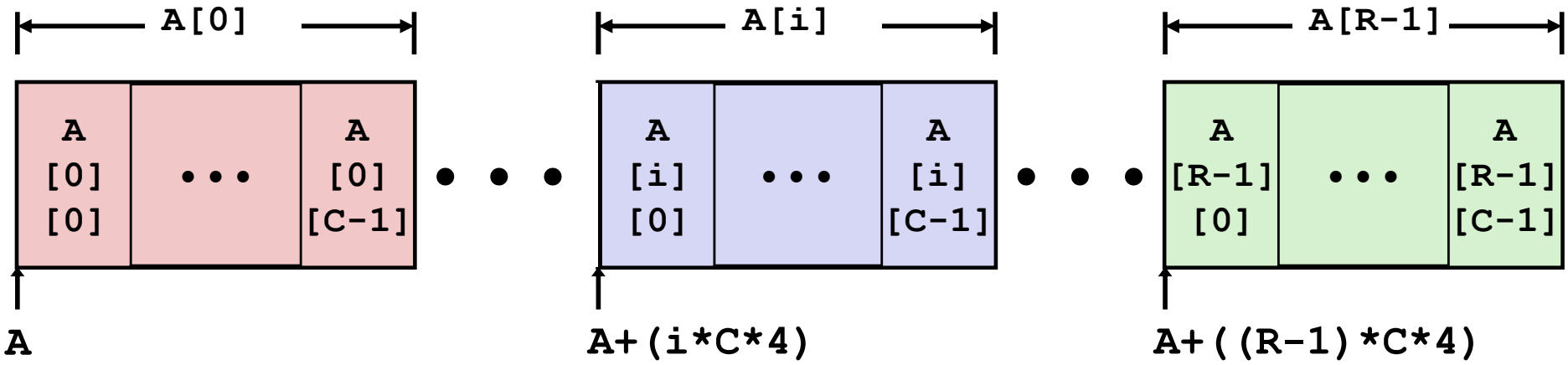


- “**zip_dig pgh[4]**” эквивалентно “**int pgh[4][5]**”
  - Переменная **pgh**: массив из 4 эл-тов, размещённых вплотную
  - Каждый эт-т массив из 5 **int**-ов, размещённых вплотную
- **Гарантируется “построчное” размещение**

Доступ к строке массива массивов

- **Строки, составляющие массив**
  - **`A[i]`** – массив из *C* элементов
  - Каждый элемент типа *T* длиной *K* байт
  - Начальный адрес **`A +`** *i* * (*C* * *K*)

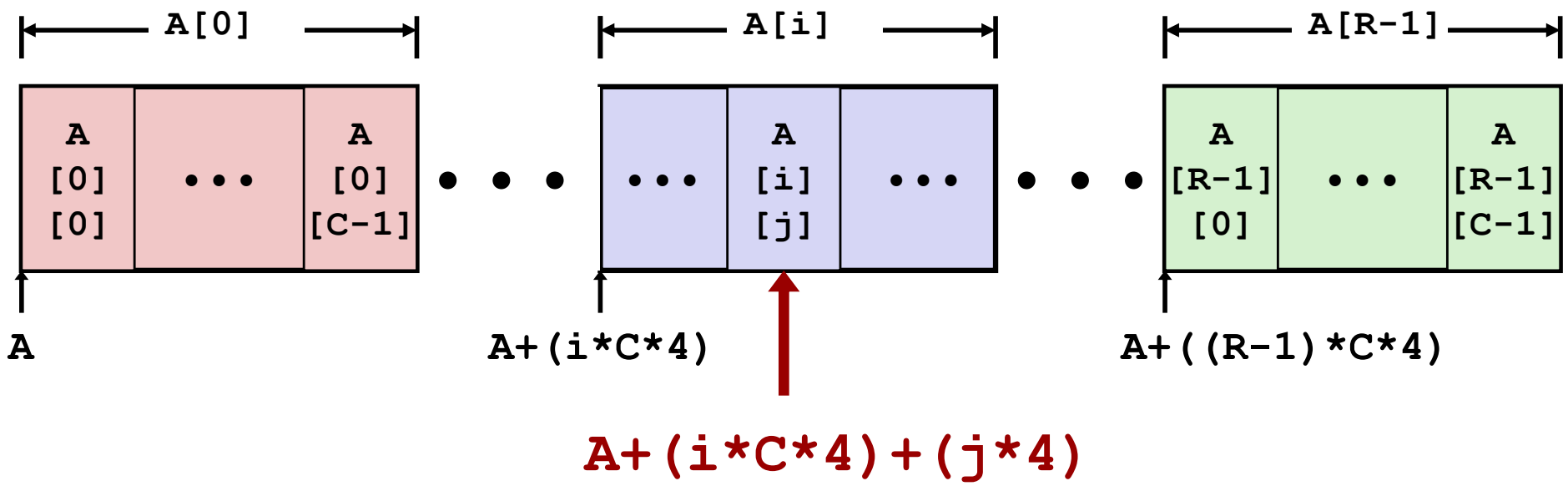
```
int A[R][C];
```



Доступ к элементам массива массивов

- **Элементы массива**
  - **`A[i][j]`** элементы типа *T*, длиной *K* bytes
  - Адрес **`A`** **+** $i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```

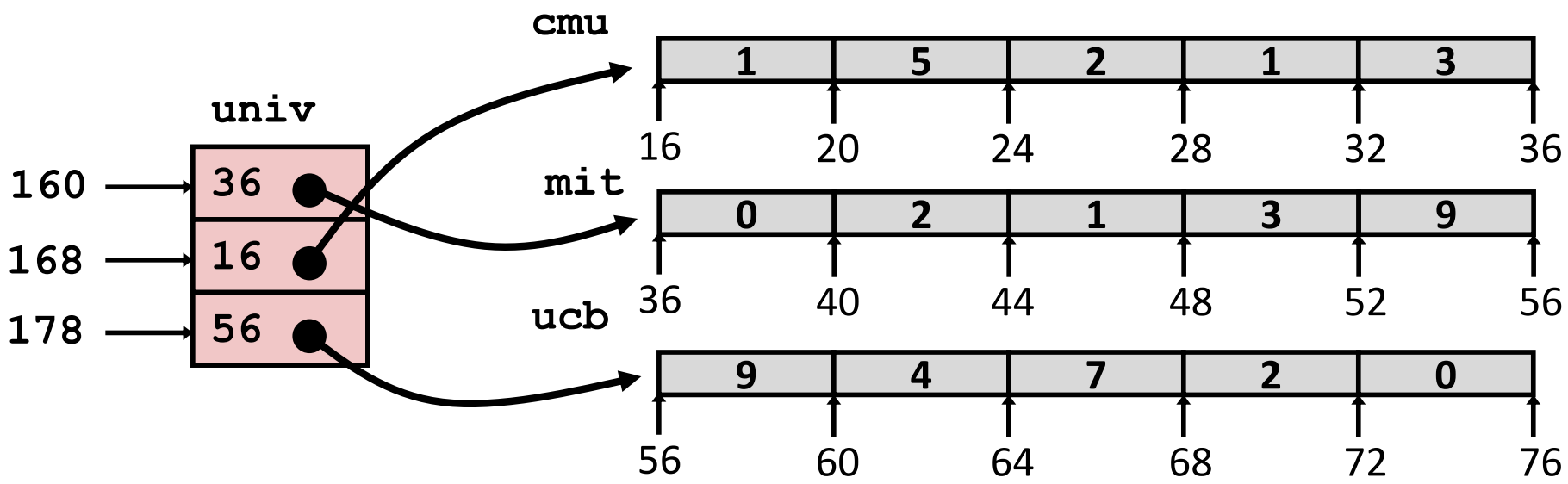


Пример многоуровневого массива

```
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` - массив из 3-х эл-тов**
- **Каждый эл-т – указатель**
  - 8 байт
- **Каждый указывает на массив `int`-ов**



Доступ к элементам многоуровневого массива

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

```
  salq    $2, %rsi              # 4*digit
  addq    univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
  movl    (%rsi), %eax          # return *p
  ret
```

- **Вычисления**
  - Доступ к элементу **`Mem[Mem[univ+8*index]+4*digit]`**
  - Необходимы два обращения к памяти
    - Первое даёт указатель на массив-строку
    - Следующее доступается к элементу в массиве

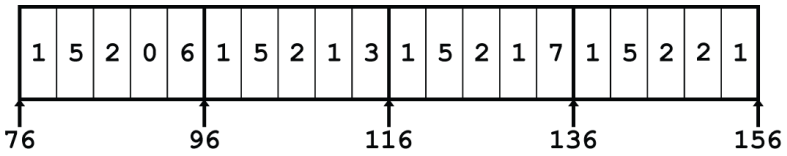
Обращения к элементам массивов

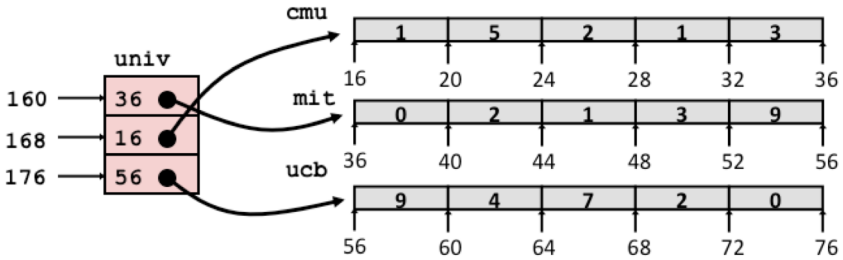
Массив массивов

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

Многоуровневый массив

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





Обращения выглядят одинаково в Си,
но вычисление адресов различается:

`Mem[pgh+20*index+4*dig]`

`Mem[Mem[univ+4*index]+4*dig]`

Код матрицы N x N

- **Фиксированные размеры**
  - Значение N известно при компиляции
- **Переменные размеры, явное индексирование**
  - Традиционный способ реализации динамических массивов
- **Переменные размеры, неявное индексирование**
  - Поддерживается gcc

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

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

```
/* Получить элемент a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
  return a[i][j];
}
```

Доступ к матрице 16 x 16

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

```
/* Получить элемент a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}
```

```
  # a in %rdi, i in %rsi, j in %rdx
  salq    $6, %rsi              # 64*i
  addq    %rsi, %rdi            # a + 64*i
  movl    (%rdi,%rdx,4), %eax   # M[a + 64*i + 4*j]
  ret
```

Доступ к матрице n X n

- **Элементы массива**
  - Адрес **A** + $i * (C * K) + j * K$
  - C = n, K = 4
  - Необходимо целочисленное умножение

```
/* Получить элемент a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j)
{
  return a[i][j];
}
```

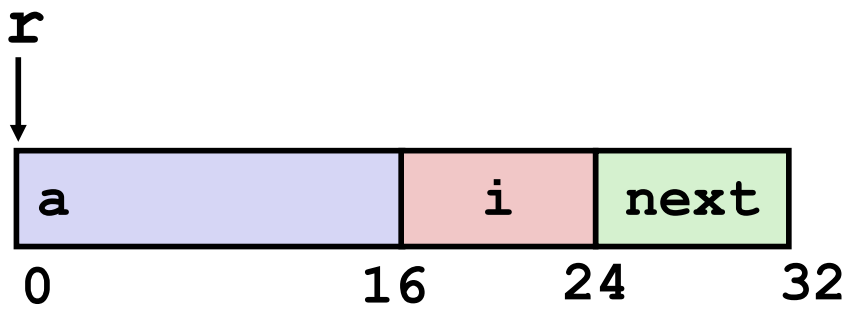
```
  # n in %rdi, a in %rsi, i in %rdx, j in %rcx
  imulq   %rdx, %rdi            # n*i
  leaq    (%rsi,%rdi,4), %rax   # a + 4*n*i
  movl    (%rax,%rcx,4), %eax   # a + 4*n*i + 4*j
  ret
```

Ещё машинный уровень
Управление и сложные данные

- **Массивы**
  - Одномерные
  - Многомерные (массивы массивов)
  - Многоуровневые
- **Структуры**
  - Размещение
  - Доступ
  - Выравнивание
- **Объединения**
- **Распределение памяти**
- **О переполнении буфера**

Размещение структуры

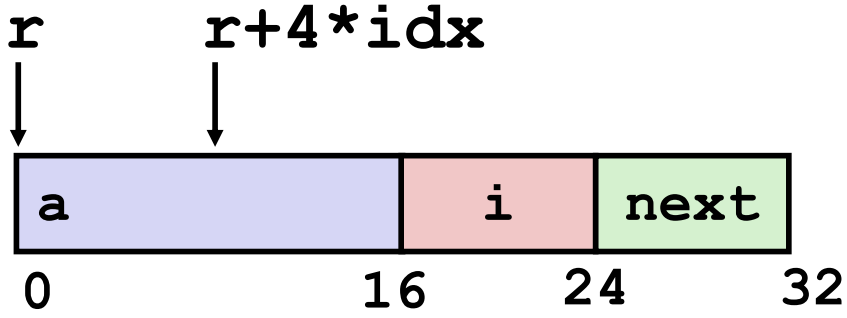
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- **Структура представляется участком памяти**
  - **Достаточно большим для размещения всех полей**
- **Поля размещены в порядке объявления**
  - **Даже если иной порядок даст более плотное размещение**
- **Компилятор определяет общий размер и размещение полей**
  - **Машинный код не имеет представления о структурах исходного кода**

Создание указателя на поле структуры

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- **Создание указателя на поле структуры**
  - Сдвиг каждого поля от начала структуры вычисляется при компиляции
  - Считается как `r + 4*idx`

```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

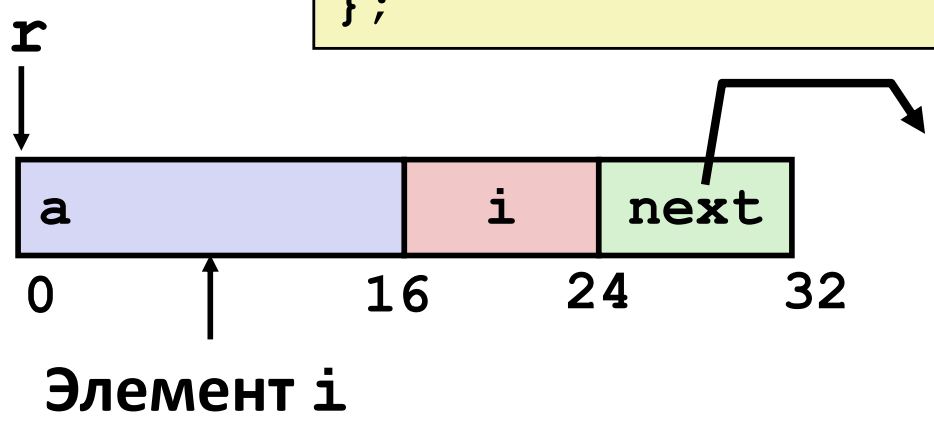
```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Проход связного списка

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

- Код Си

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

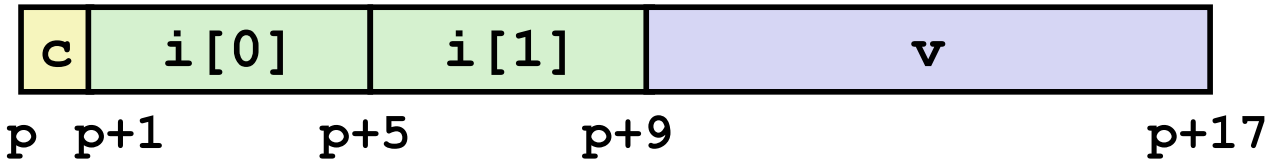


Регистр	Значение
%rdi	r
%rsi	val

```
.L11:                                # loop:
  movslq  16(%rdi), %rax             #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)        #   M[r+4*i] = val
  movq    24(%rdi), %rdi             #   r = M[r+24]
  testq   %rdi, %rdi                 #   Проверка r
  jne     .L11                       #   if !=0 goto loop
```

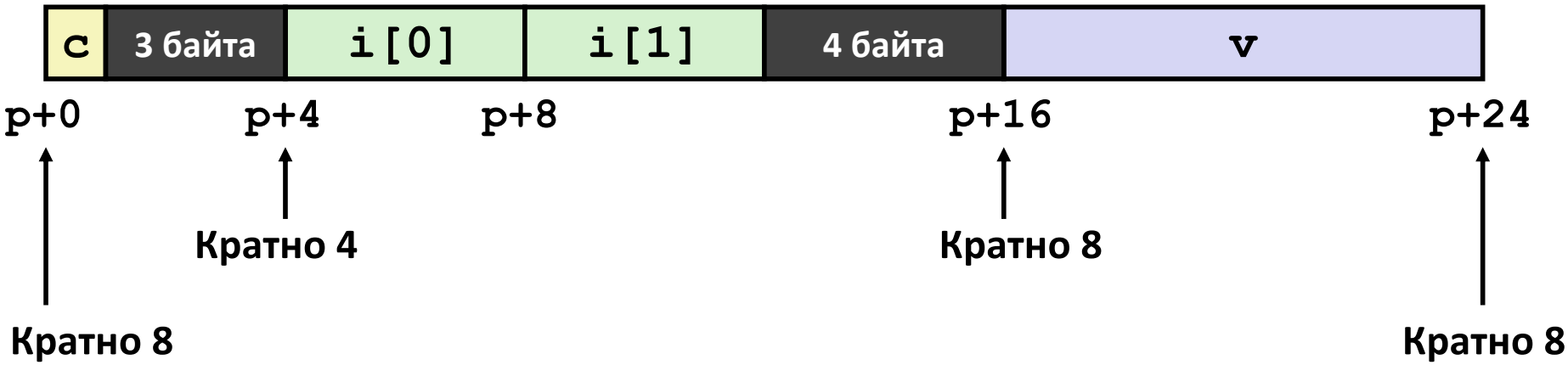
Структуры и выравнивание

- **Данные без выравнивания**




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

- **Данные с выравниванием**
  - Если простой тип данных - длиной К байт
  - То адреса должны быть кратны К



Принципы выравнивания

- **Выровненные данные**
  - Если простой тип данных - длиной К байт,…
  - …то адрес должен быть кратен К
  - Обязательно на некоторых машинах, рекомендовано на x86-64
- **Зачем выравнивать данные**
  - Доступ в память производится выровненными фрагментами по 4 или 8 байт (в зависимости от системы)
    - Неэффективно обращение к элементу данных, пересекающему границу четверного слова
    - Работа виртуальной памяти резко усложняется для элемента данных находящегося в 2-х страницах
- **Компилятор**
  - Для правильного выравнивания полей добавляет в структуру зазоры

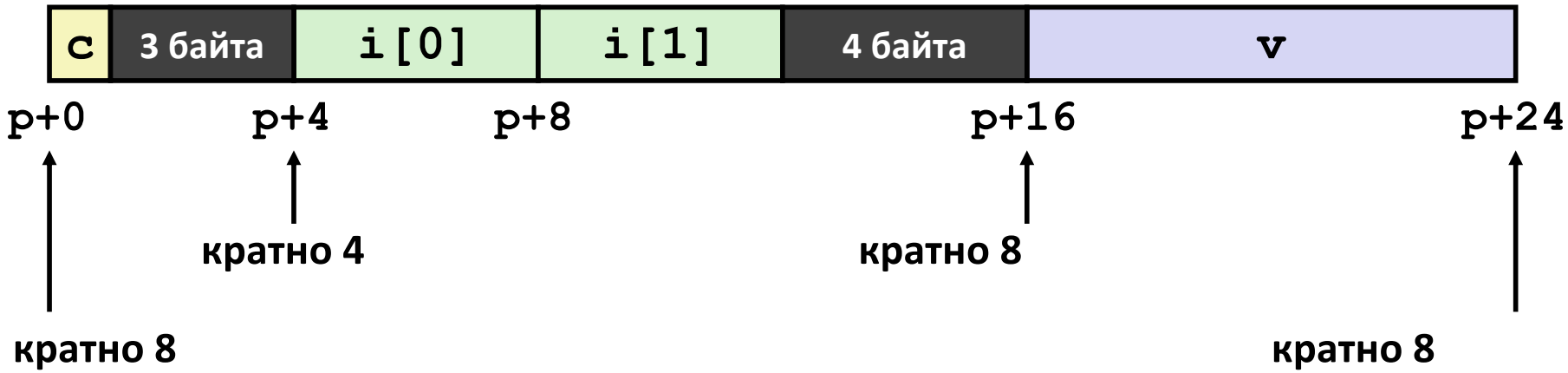
Варианты выравнивания (x86-64)

- **1 байт: `char`, …**
  - Любой адрес
- **2 байта: `short`, …**
  - 1 младший бит адреса должен быть нулевым 0_2
- **4 байта: `int`, `float`, …**
  - 2 младших бита адреса должны быть нулевыми 00_2
- **8 байт: `double`, `long`, `char *`, …**
  - 2 младших бита адреса должны быть нулевыми 000_2
- **16 байт: `long double`** (GCC on Linux)
  - 2 младших бита адреса должны быть нулевыми 0000_2

Выравнивание структур

- **Внутри структуры:**
  - Выравнивается каждое поле
- **Размещение всей структуры**
  - Структура выравнивается на границу K байт
    - K = крупнейшее выравнивание среди всех полей
  - Начальный адрес и размер структуры должны быть кратны K
- **Пример:**
  - K = 8, из-за поля **`double`**

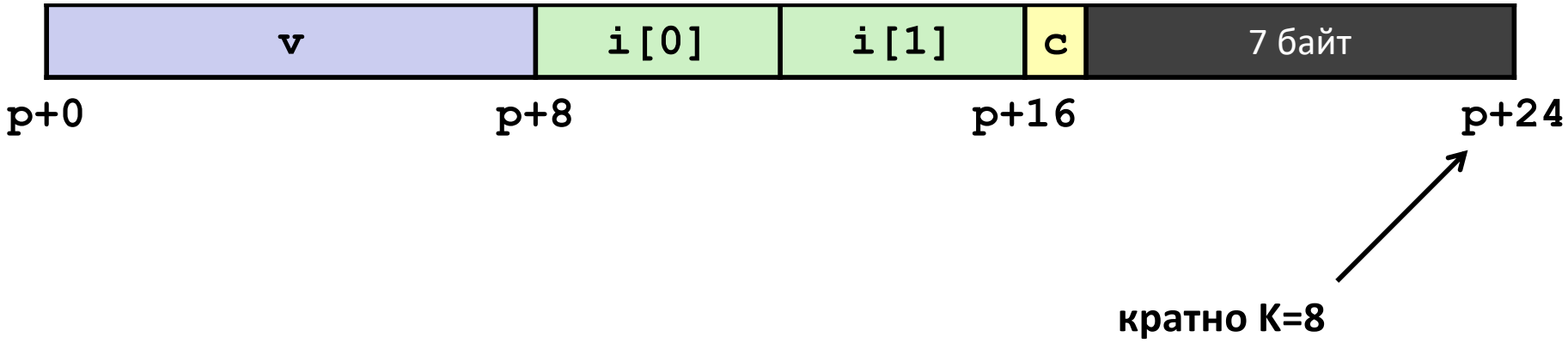
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Выравнивание всей структуры

- **Для крупнейшего выравнивания K**
- **Размер всей структуры кратен K**

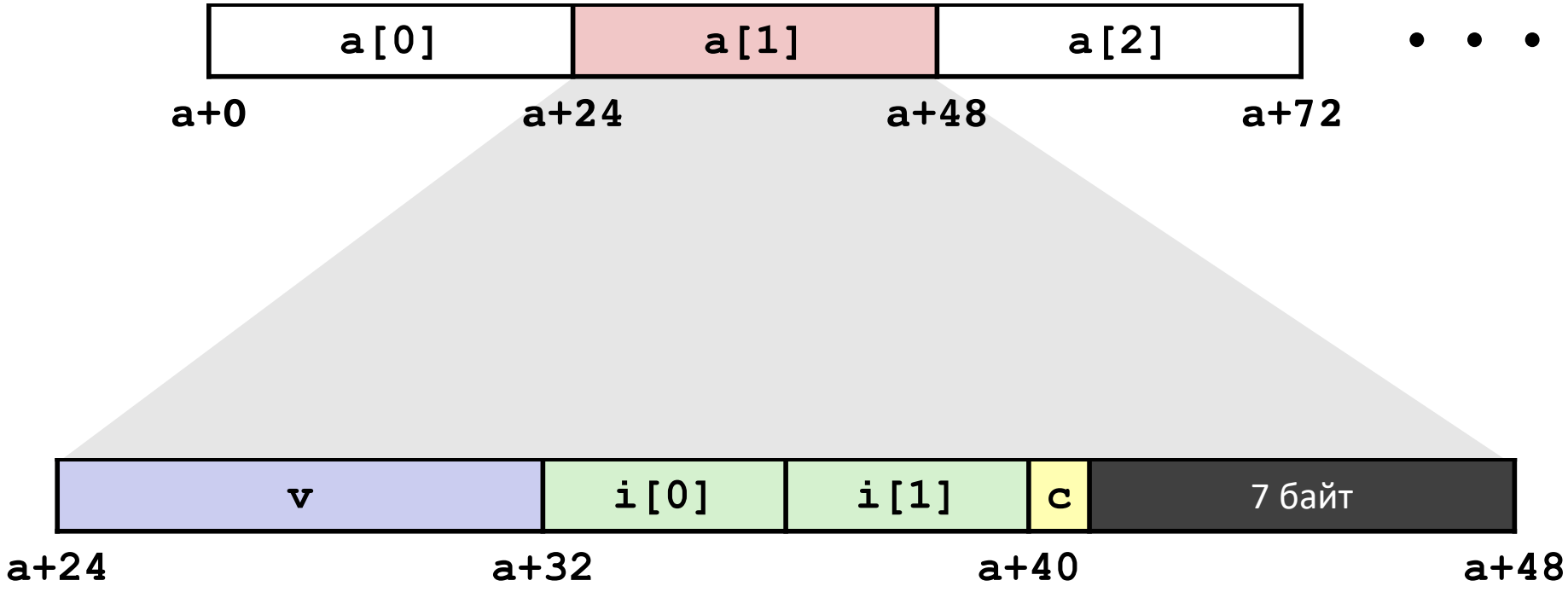
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Массив структур

- Размер всей структуры кратен K
- Выравнивается каждый элемент массива

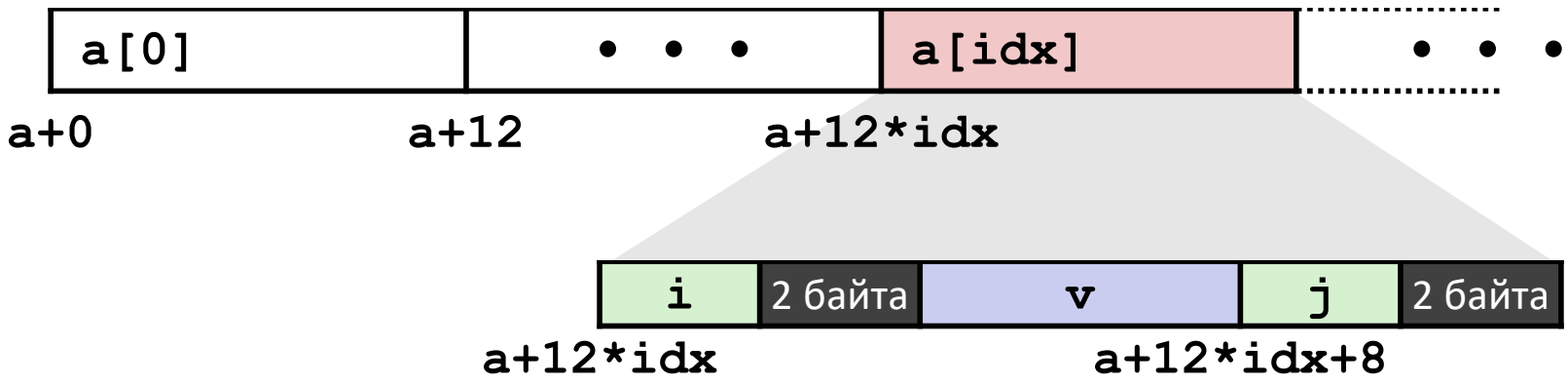
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Доступ к элементам массива

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

- **Вычислить сдвиг в массиве 12*idx**
  - `sizeof(S3)`, включая заполнители выравнивания
- **Поле `j` сдвинуто на 8 от начала структуры**
- **Ассемблер даст сдвиг a+8**
  - Выполняется при редактировании связей (линковке)



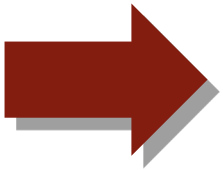

```
short get_j(int idx)
{
  return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

Экономия пространства

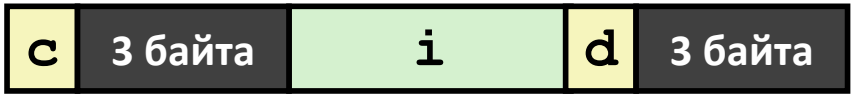
- Поместим вначале длинные данные

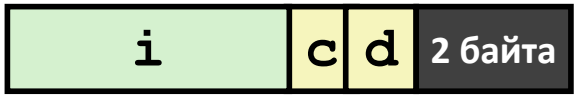
```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```



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

- Результат (K=4)





Ещё машинный уровень
Управление и сложные данные

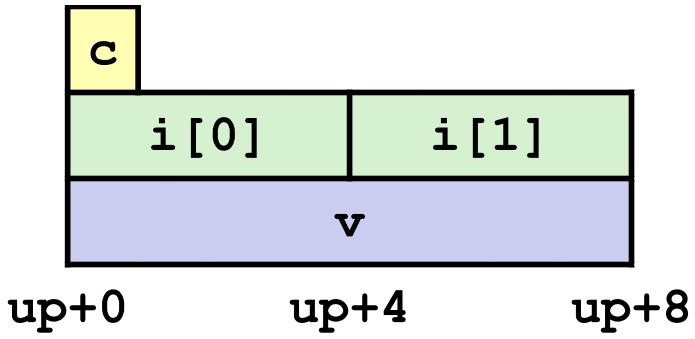
- **Массивы**
  - Одномерные
  - Многомерные (массивы массивов)
  - Многоуровневые
- **Структуры**
  - Размещение
  - Доступ
  - Выравнивание
- **Объединения**
- **Распределение памяти**
- **О переполнении буфера**

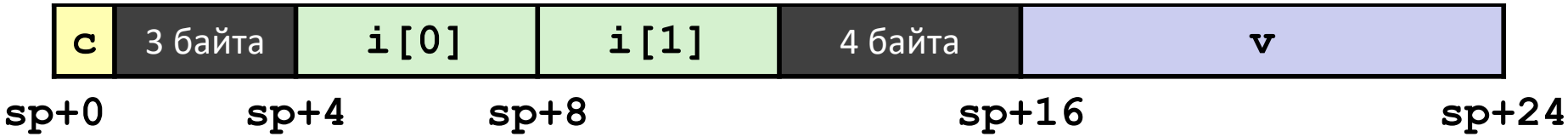
Размещение объединений

- **Размещается как наибольший элемент**

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

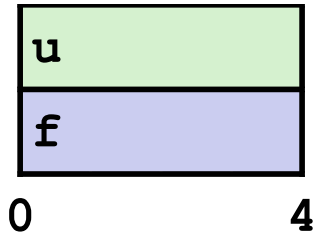
```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```





Доступ к битовым наборам

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Совпадает с `(float)u`?

Совпадает с `(unsigned)f`?

Сводка

- **Массивы в Си**
  - Размещение в памяти вплотную
  - Удовлетворяют требованиям по выравниванию
  - Указатель на первый элемент
  - Границы не контролируются
- **Структуры**
  - Размещает байты в запрошенном порядке
  - Зазоры в середине и конце для выравнивания
- **Объединения**
  - Поля наложены друг на друга
  - Способ обойти систему контроля типов

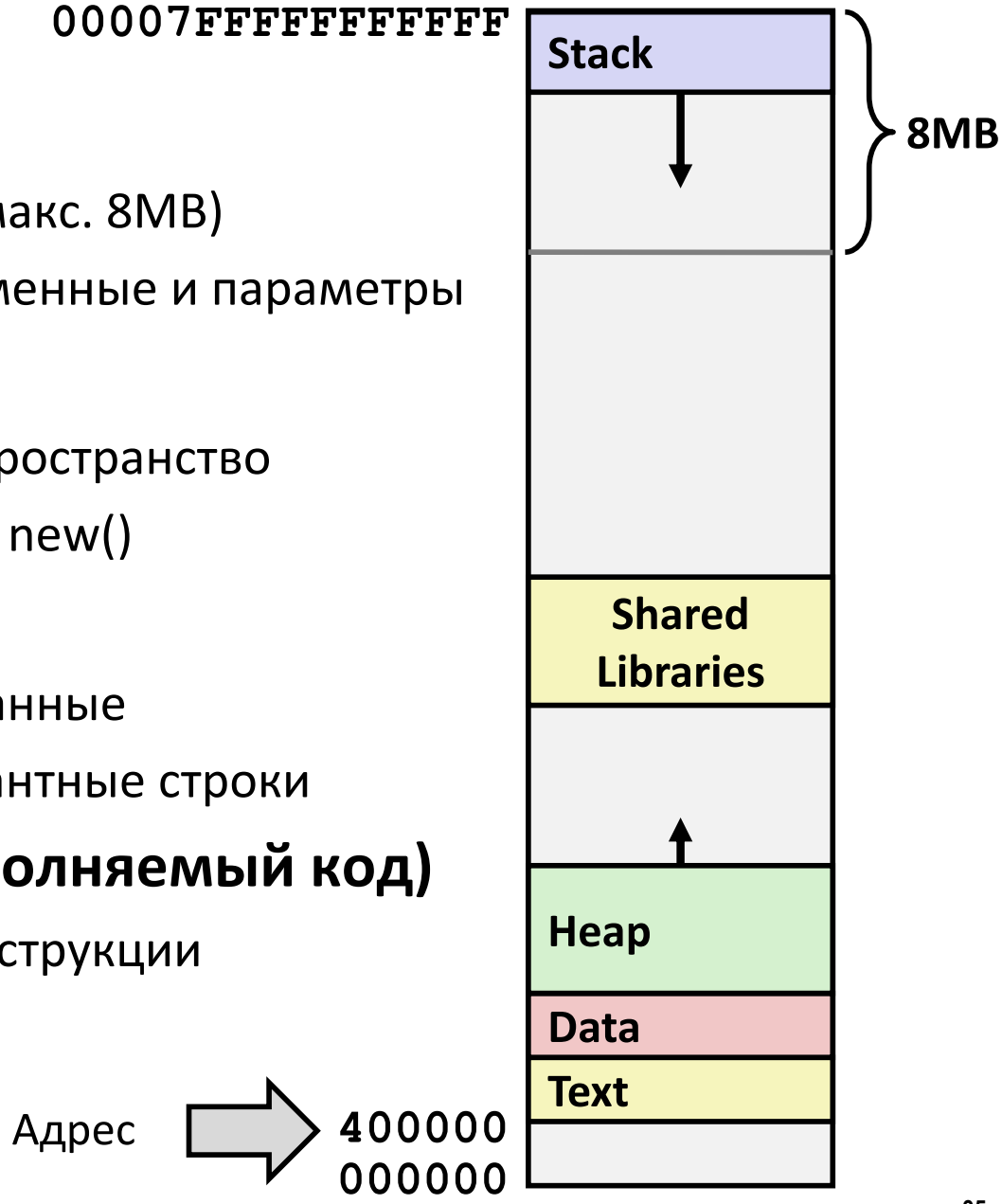
Ещё машинный уровень
Управление и сложные данные

- **Массивы**
  - Одномерные
  - Многомерные (массивы массивов)
  - Многоуровневые
- **Структуры**
  - Размещение
  - Доступ
  - Выравнивание
- **Объединения**
- **Распределение памяти**
- **О переполнении буфера**

Распределение памяти x86-64 Linux

Не в масштабе

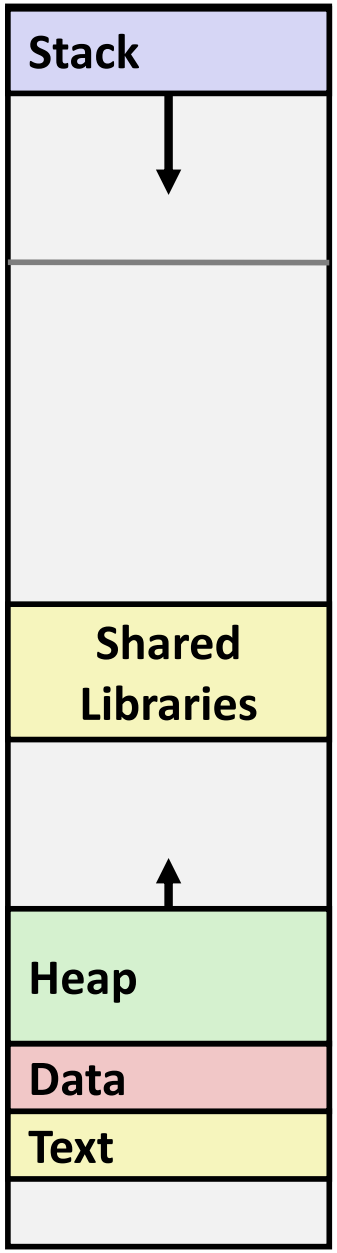
- **Stack (стек)**
  - Стек времени исполнения (макс. 8MB)
  - Например, локальные переменные и параметры
- **Heap (куча)**
  - Динамически занимаемое пространство
  - При вызове malloc(), calloc(), new()
- **Data (данные)**
  - Статически размещаемые данные
  - Например, массивы и константные строки
- **Text и Shared Libraries (исполняемый код)**
  - Исполняемые машинные инструкции
  - Только чтение



Не в масштабе

Пример распределения памяти

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Несколько вызовов печати ... */
}
```



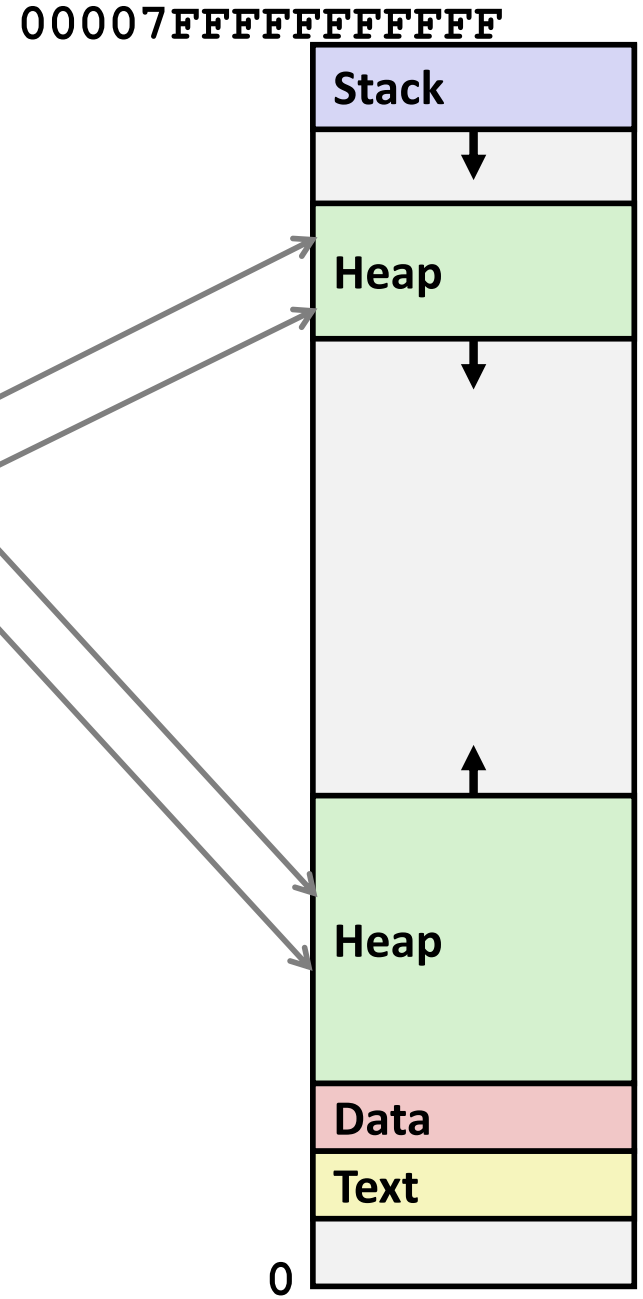
Где всё это размещается?

Пример адресов x86-64

диапазон адресов ~2^{47}

Не в масштабе

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



Ещё машинный уровень
Управление и сложные данные

- **Процедуры (x86-64)**
- **Массивы**
  - Одномерные
  - Многомерные (массивы массивов)
  - Многоуровневые
- **Структуры**
  - Размещение
  - Доступ
  - Выравнивание
- **Объединения**
- **Распределение памяти**
- **О переполнении буфера**

Переполнение буфера – большая проблема

- **Пишут “переполнение буфера”**
  - когда выходят за рамки адресов памяти, отведённых под массив
- **Почему большая проблема?**
  - Техническая причина №1 нарушения безопасности
    - Причина №1 в целом – социальная инженерия / беспечность людей
- **Наиболее частые случаи**
  - Непроверенные длины вводимых строк
  - В частности для ограниченных массивом символов в стеке
    - иногда упоминается как «stack smashing»

Библиотечный код обработки строк

- **Unix-реализация функции `gets()`**

```
/* Выбрать строку из stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - Невозможно ограничить количество вводимых символов
- **Та-же проблема с другими библиотечными ф-циями**
  - **`strcpy`, `strcat`**: Копирование строк произвольной длины
  - **`scanf`, `fscanf`, `sscanf`** со спецификацией преобразования **`%s`**

Код с уязвимостью переполнения буфера

```
/* Эхо строки */
void echo()
{
    char buf[4];  /* Слишком мал! */
    gets(buf);
    puts(buf);
}
```

← кстати, а «достаточно» это сколько?

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo
Type a string:0123456789012345678901 23
0123456789012345678901 23
```

```
unix>./bufdemo
Type a string:0123456789012345678901234
Segmentation Fault
```

Дизассемблирование переполнения буфера

echo:

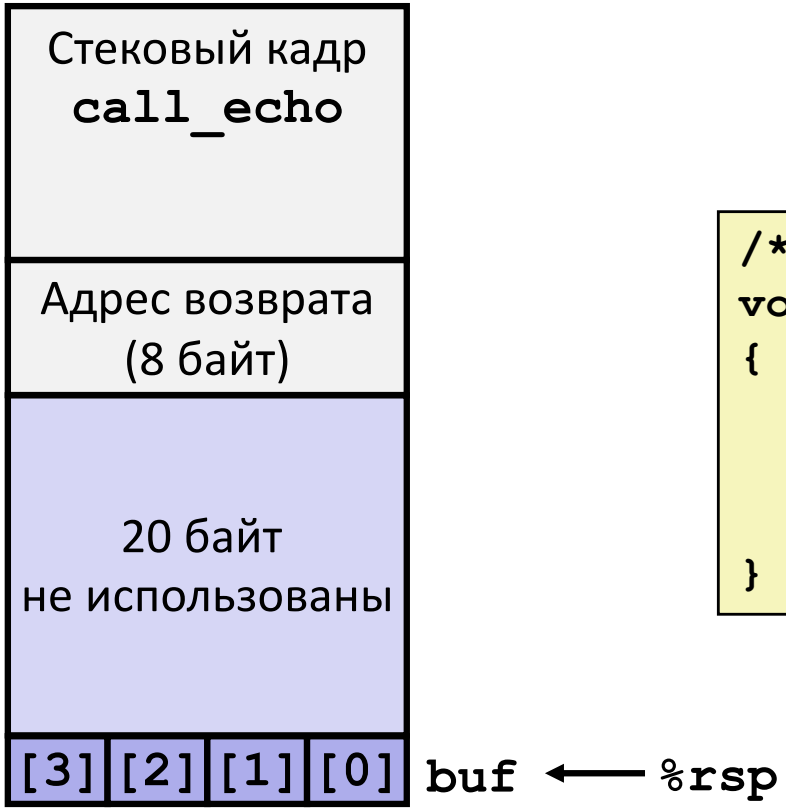
```
00000000004006cf <echo>:
 4006cf:  48 83 ec 18          sub    $0x18,%rsp
 4006d3:  48 89 e7             mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff       callq  400680 <gets>
 4006db:  48 89 e7             mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff       callq  400520 <puts@plt>
 4006e3:  48 83 c4 18          add    $0x18,%rsp
 4006e7:  c3                   retq
```

call_echo:

```
4006e8:   48 83 ec 08          sub    $0x8,%rsp
 4006ec:  b8 00 00 00 00       mov    $0x0,%eax
 4006f1:  e8 d9 ff ff ff       callq  4006cf <echo>
 4006f6:  48 83 c4 08          add    $0x8,%rsp
 4006fa:  c3                   retq
```

Стек при переполнении буфера

Перед вызовом gets

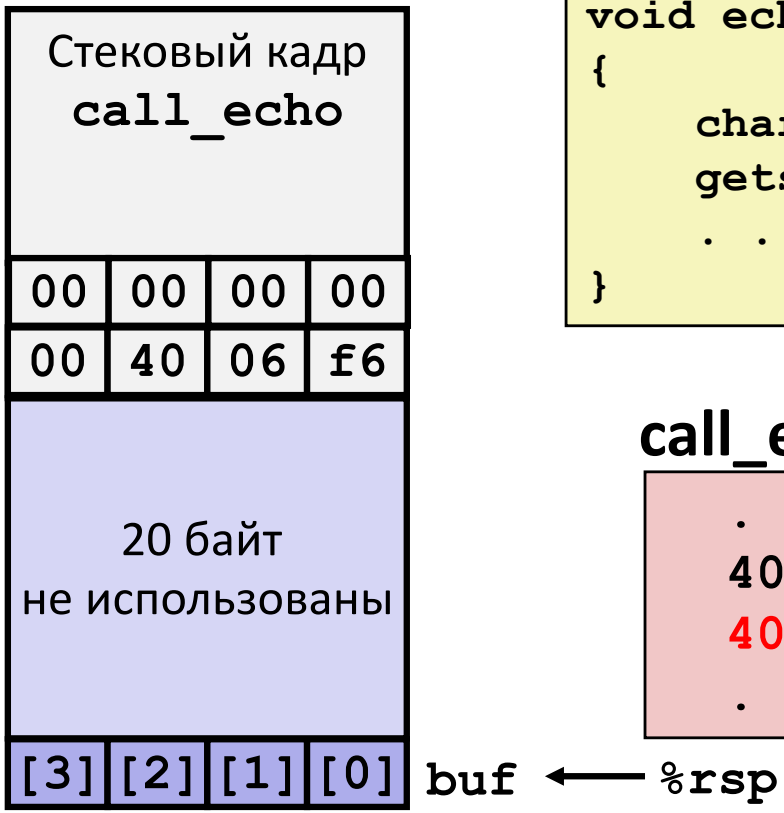



```
/* Эхо строки */
void echo()
{
    char buf[4];  /* Слишком мал! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Пример стека при переполнении буфера

Перед вызовом gets




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
. . .
4006f1:  callq  4006cf <echo>
4006f6:  add    $0x8,%rsp
. . .
```

Пример стека при переполнении буфера 1

После возврата из gets

Стековый кадр
call_echo

00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
. . .
4006f1:  callq  4006cf <echo>
4006f6:  add    $0x8,%rsp
. . .
```

```
unix>./bufdemo
Type a string:012345678901234567890 12
012345678901234567890 12
```

Буфер переполнен, но состояние не испорчено

Пример стека при переполнении буфера 2

После возврата из gets

Стековый кадр
call_echo

00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
. . .
4006f1:  callq  4006cf <echo>
4006f6:  add    $0x8,%rsp
. . .
```

```
unix>./bufdemo
Type a string:0123456789012345678901234
Segmentation Fault
```

Буфер переполнен и испорчен адрес возврата

Пример стека при переполнении буфера 3

После возврата из gets

Стековый кадр
call_echo

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
. . .
4006f1:  callq  4006cf <echo>
4006f6:  add    $0x8,%rsp
. . .
```

```
unix>./bufdemo
Type a string:0123456789012345678901 23
0123456789012345678901 23
```

Буфер переполнен и испорчен адрес возврата но программа кажется работающей !

Пояснения к примеру стека 3

После возврата из gets

Стековый кадр `call_echo`			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

`buf` ← `%rsp`

register_tm_clones:

```
. . .
400600:  mov    %rsp,%rbp
400603:  mov    %rax,%rdx
400606:  shr    $0x3f,%rdx
40060a:  add    %rdx,%rax
40060d:  sar    %rax
400610:  jne    400614
400612:  pop    %rbp
400613:  retq
```

“Возврат” не туда, куда надо.
Многое меняется, без критической порчи состояния
Случайно выполняется `retq` обратно в `main`

Атаки на основе переполнения буферов

- ***Ошибки переполнения позволяют удалённой машине выполнять произвольный код на машинах жертв***
- **Удручающе часты в реальных программах**
  - Программисты делают одинаковые ошибки ☹
  - Современные меры делают такие атаки много труднее
- **Примеры прошедших десятилетий**
  - Первый “интернет-червь” (1988)
  - “Битвы мессенжеров” (1999)
  - Взлом Wii (2000s)
  - … и многие, многие другие
- **Можно попробовать некоторые фокусы в attacklab**
  - Возможно это убедит вас не оставлять таких дыр в своих программах!!

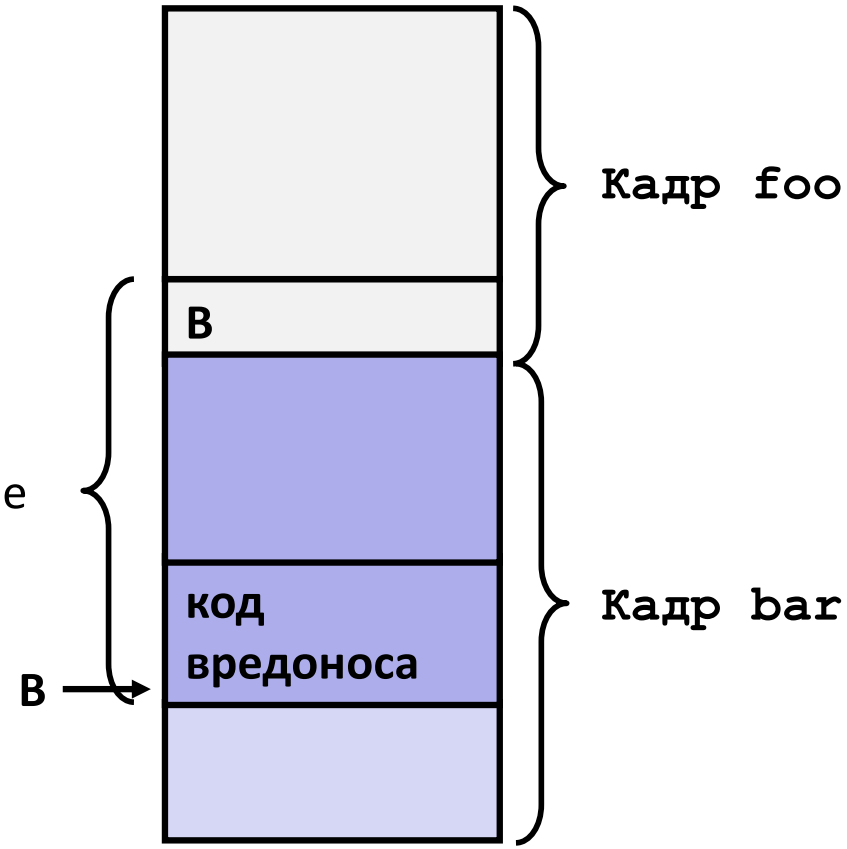
Злонамеренное использование переполнения буфера

```
void foo(){
  bar();
  ...
}
```

Адрес возврата A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

Стек после вызова **gets()**



- **Вводимая строка содержит байтовое представление исполняемого кода**
- **Затирает адрес возврата A адресом буфера B**
- **Когда `bar()` выполняет `ret`, управление передаётся вредоносу**

Избегание уязвимости переполнения

```
/* Эхо строки */
void echo()
{
    char buf[4];  /* Слишком мал! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- **Используйте функции с ограничением длины строки**
  - **`fgets`** вместо **`gets`**
  - **`strncpy`** вместо **`strcpy`**
  - Не используйте **`scanf`** со спецификацией прообразования **`%s`**
    - Используйте **`fgets`** для чтения строки
    - или используйте **`%ns`** где **`n`** подходящее целое

Защита на уровне системы

- **Рандомизация сдвига стека**
  - При старте программы выделяется случайное пространство в стеке
  - Затрудняет хакеру предсказание начала вставляемого кода

- **Неисполняемые сегменты памяти**
  - В x86, область памяти либо “только чтение” либо “изменяемая”
    - Может выполняться всё, что читается
  - x86-64 добавляет явное разрешение “исполняемая”
  - Стек помечается как неисполняемый

Стековый индикатор

- **Идея**
  - Разместить в стеке сразу за буфером специальное значение (“индикатор”)
  - Проверять целостность перед выходом из функции
- **Реализация GCC**
  - `-fstack-protector`
  - сейчас включён по умолчанию (но не раньше)

```
unix>./bufdemo-protected
Type a string:0123456
0123456
```

```
unix>./bufdemo-protected
Type a string:01234567
*** stack smashing detected ***
```

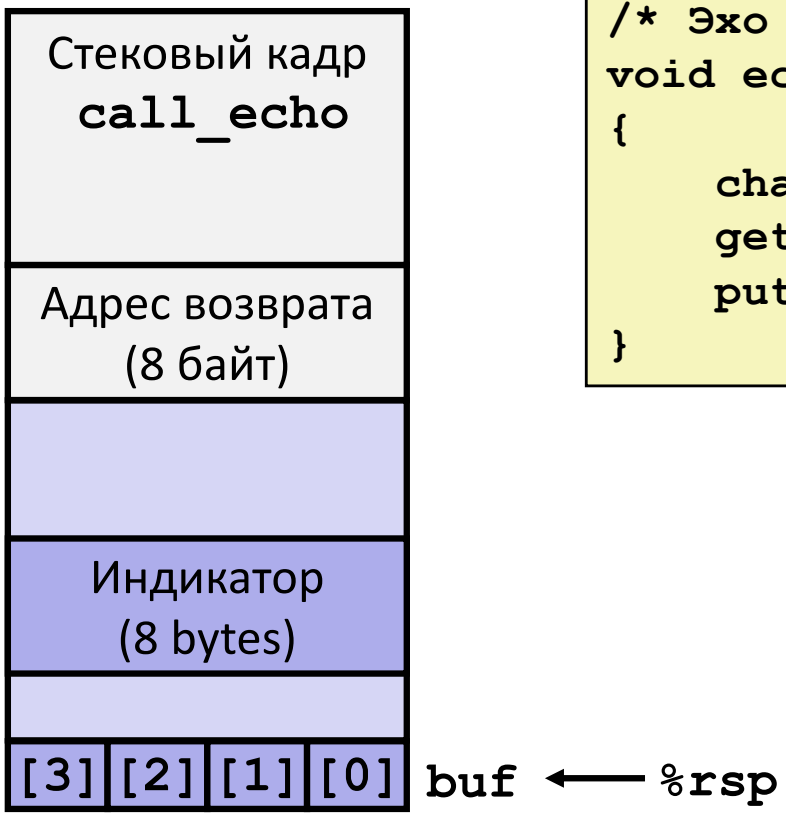
Дизассемблирование защищённого буфера

echo:

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
400768:  add    $0x18,%rsp
40076c:  retq
```

Установка индикатора

Перед вызовом gets

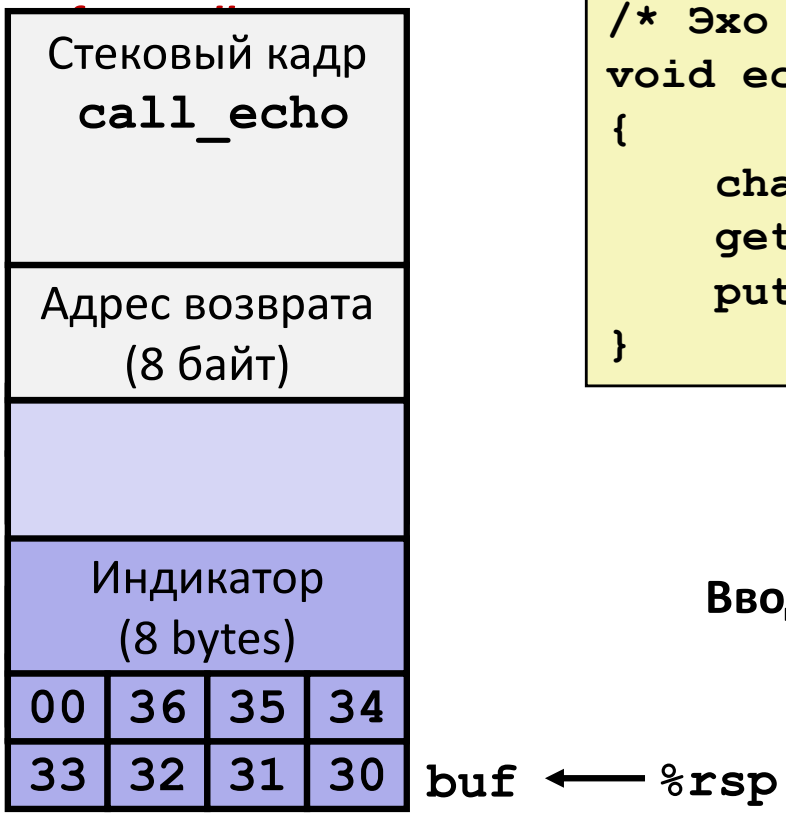



```
/* Эхо строки */
void echo()
{
    char buf[4];  /* Слишком мал! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax   # Получить индикатор
    movq    %rax, 8(%rsp)  # Поместить в стек
    xorl    %eax, %eax     # Стереть индикатор
    . . .
```

Проверка индикатора

После возврата из gets




```
/* Эхо строки */
void echo()
{
    char buf[4];  /* Слишком мал! */
    gets(buf);
    puts(buf);
}
```

Ввод: ***0123456***

```
echo:
    . . .
    movq    8(%rsp), %rax     # Считать из стека
    xorq    %fs:40, %rax      # Сравнить с исходным
    je      .L6               # Совпало? Дальше…
    call    __stack_chk_fail  # ОШИБКА!
.L6:        . . .
```