

Лекция 0xA

11 марта

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

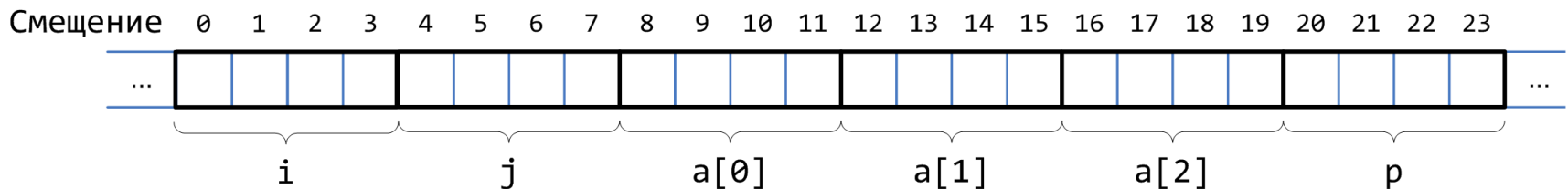
- char
- Стандартные знаковые целочисленные типы
 - signed char
 - short int
 - int
 - long int
 - long long int
- Стандартные беззнаковые целочисленные типы
 - _Bool
- Перечисление
- Типы чисел с плавающей точкой
 - float
 - double
 - long double
 - _Complex
- Производные типы
 - Массивы
 - **Структуры**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

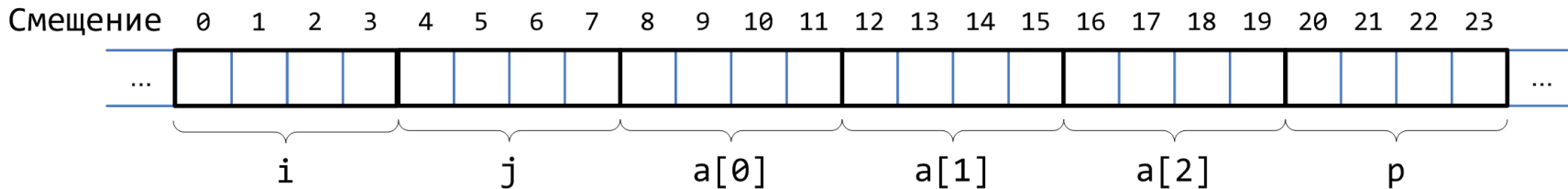
Расположение в памяти



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;
```

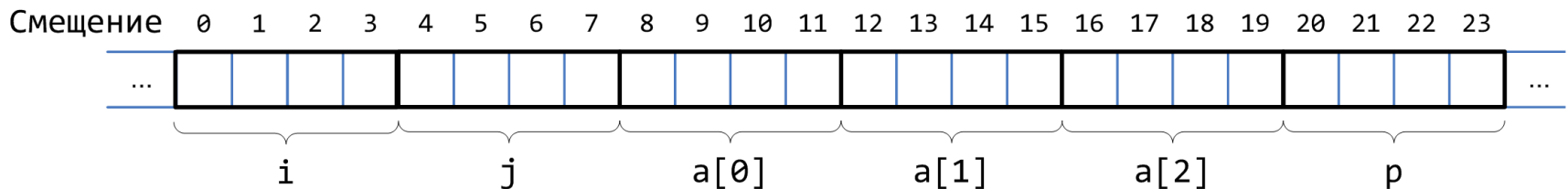
```
mov edx, dword [x]      ; (1)
mov eax, dword [edx]    ; (2)
mov dword [edx + 4], eax ; (3)
```

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]);
```



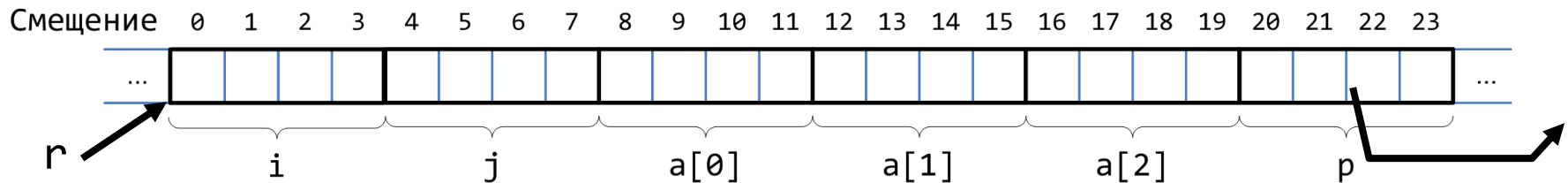
```
mov edx, dword [i]           ; (1)
mov eax, dword [x]           ; (2)
lea eax, [eax + 4 * edx + 8] ; (3)
```

• 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

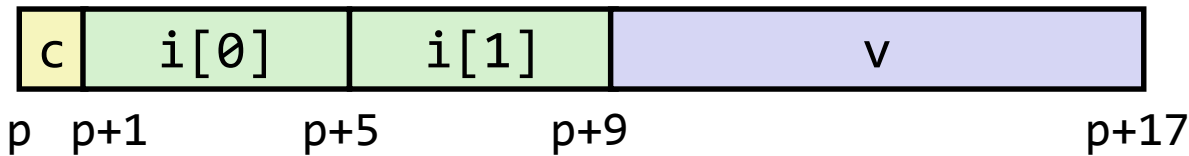


.L17: ; цикл

```
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

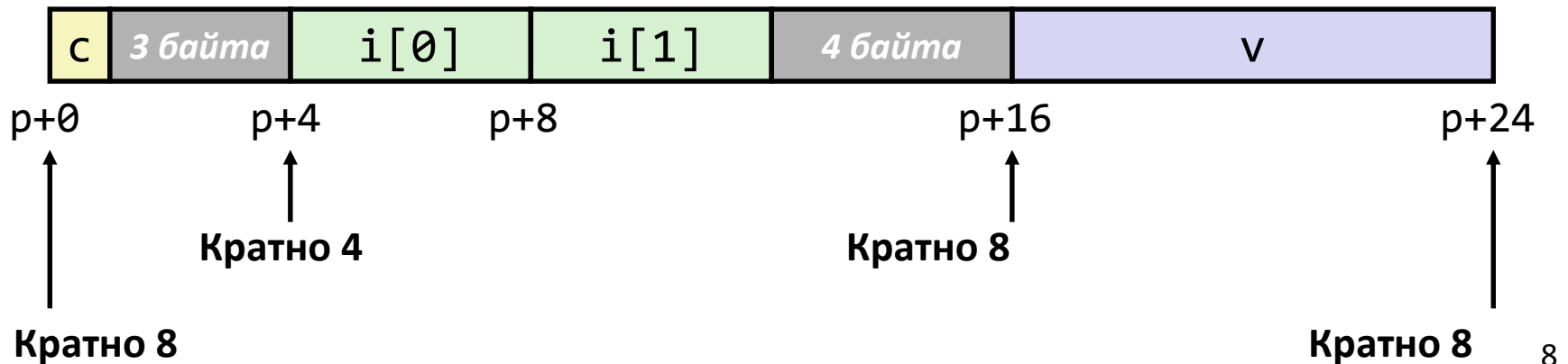
- Невыровненные данные



```
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 0₂
- 4 байта : int, long, float, char *, ...
 - Два младших бита адреса должны быть 00₂
- 8 байт : double, ...
 - Windows (и другие ...):
 - Младшие три бита адреса должны быть 000₂
 - Linux:
 - Два младших бита адреса должны быть 00₂
 - Т.е. рассматриваются как и 4-байтные примитивные типы данных
- 12 байт : long double (gcc)
 - Windows, Linux:
 - Два младших бита должны быть 00₂
 - 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 0₂
- 4 байта : int, float, ...
 - Два младших бита адреса должны быть 00₂
- 8 байт: double, long, char *, ...
 - Windows & Linux:
 - Младшие три бита адреса должны быть 000₂
- 16 байт: long double
 - Linux:
 - Младшие три бита адреса должны быть 000₂
 - Т.е. рассматриваются как и 8-байтные примитивные типы данных

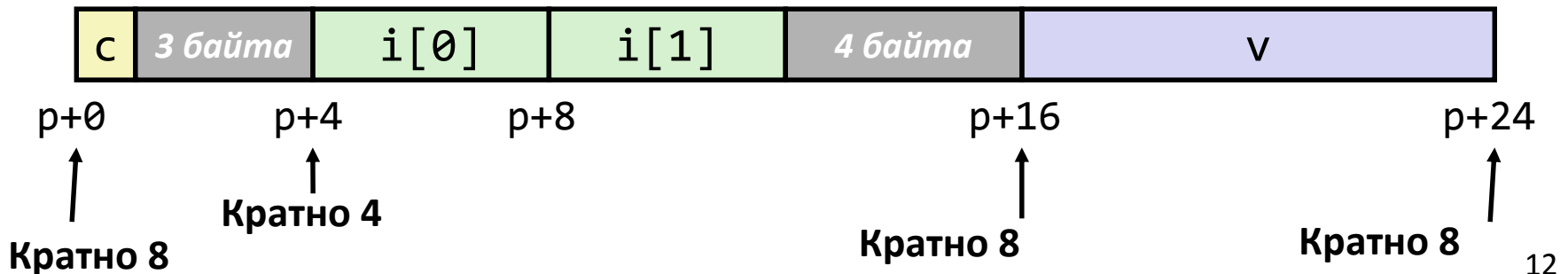
Тип long double в компиляторе MS VC

- 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 определяются требования 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

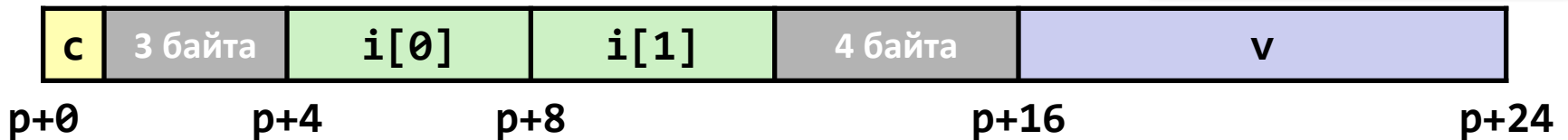
```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```



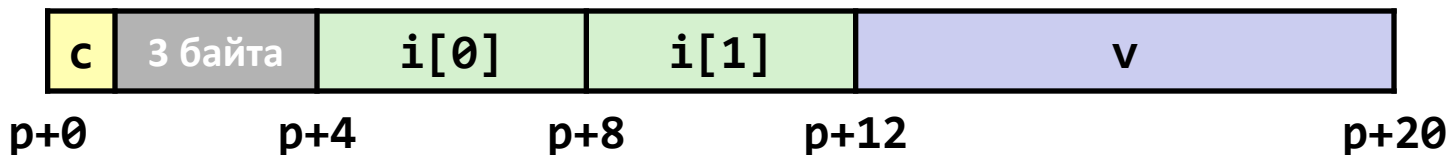
Различные соглашения о выравнивании

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

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



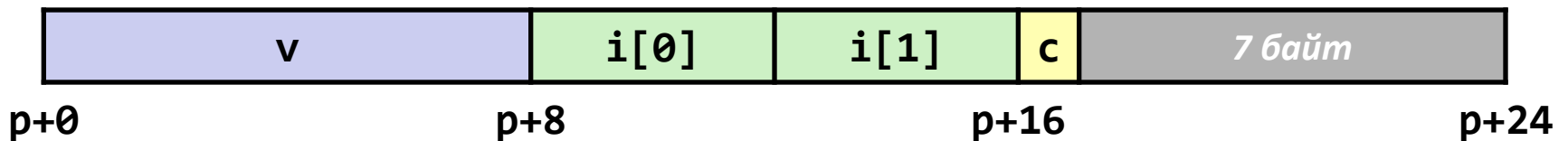
- IA-32 Linux
 - $K = 4$; **double** рассматривается аналогично 4-байтным типам данных



Выравнивание всей структуры

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

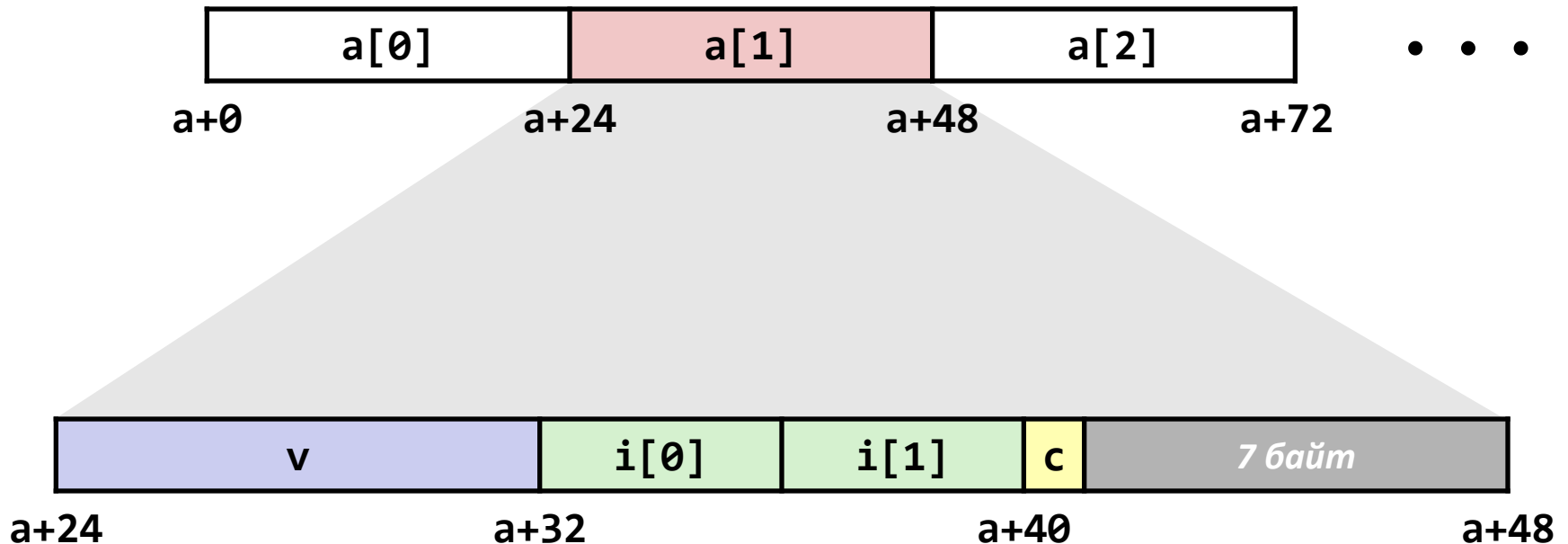
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



Массивы структур

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

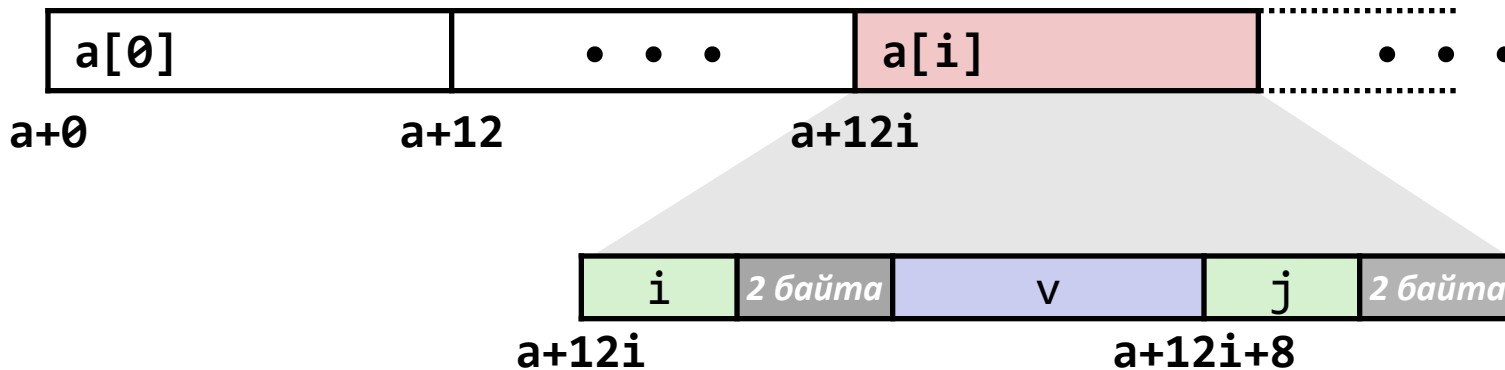
```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```



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

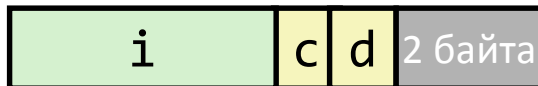
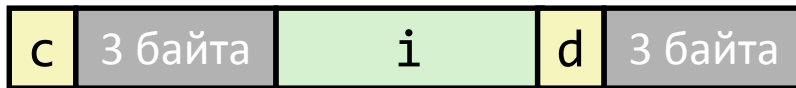
- Размещаем большие типы первыми

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



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

- Результат (K=4)

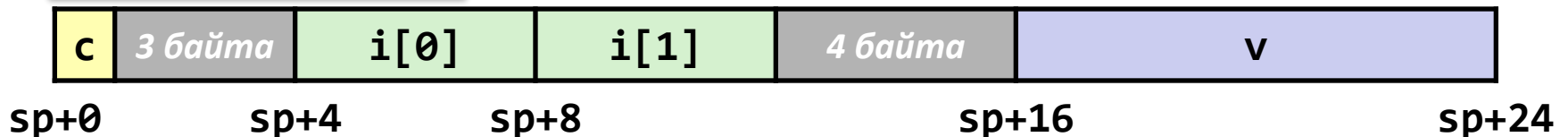
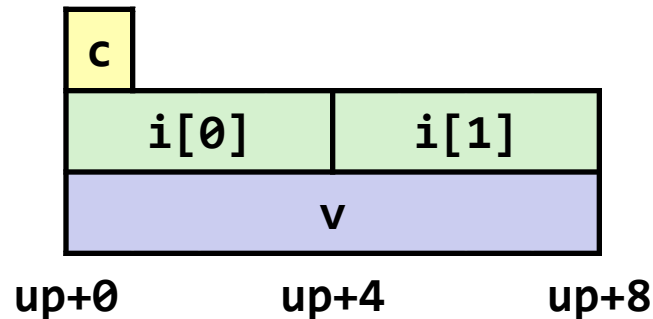


Размещение объединений

- 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;

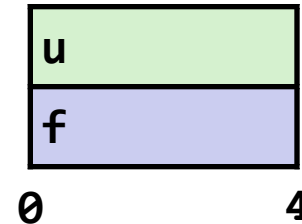
struct NODE_T {
    nodetype_t type;
    union NODE_U {
        struct {
            struct NODE_T *left;
            struct NODE_T *right;
        } internal;
        double data;
    } info;
};
```

`sizeof(struct NODE_T) ?`

Какие смещения у полей?

Использование объединений для доступа к отдельным битам

```
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;  
}
```

**Тоже самое, что и
(float) u ?**

**Тоже самое, что и
(unsigned) f ?**

Порядок байт

- Основная идея
 - short/long/double хранятся в памяти как последовательности из 2/4/8 байт
 - Где именно расположен старший (младший) байт?
 - Может являться проблемой при пересылке двоичных данных между машинами разной архитектуры
- Big-endian / от старшего к младшему
 - Старший байт имеет наименьший адрес
 - Sparc (до V8)
- Little-endian / от младшего к старшему
 - Младший байт имеет наименьший адрес
 - Intel x86 (IA-32)
- Переключаемый порядок байт
 - ARM, PowerPC, Alpha, SPARC V9, MIPS, PA-RISC и IA-64

Пример

```
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

32 бита

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

64 бита

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

Пример (продолжение)

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```


Порядок байт от младшего к старшему

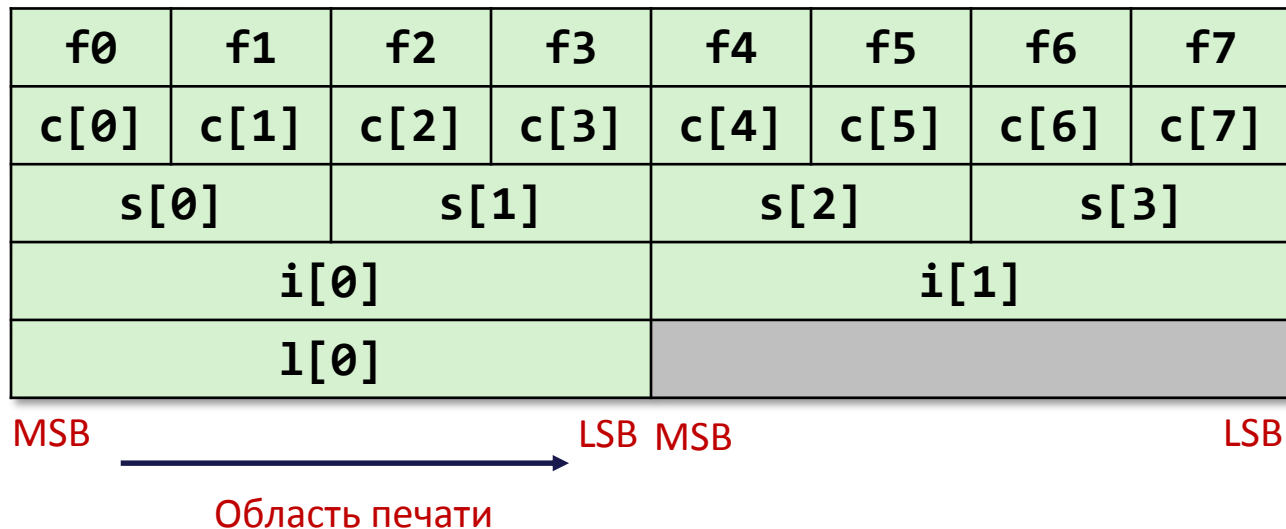
[illegible]

Вывод на консоль:

25

Порядок байт SPARC, PPC, m68k и некоторых других архитектур

Порядок байт от старшего к младшему



Вывод на консоль:

```

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Порядок байт в x86-64

Порядок байт от младшего к старшему

f0	f1	f2	f3	f4	f5	f6	f7
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

LSB

MSB



Область печати

Вывод на консоль:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]

Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]

Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]

Long 0 == [0xf7f6f5f4f3f2f1f0]

Битовые поля

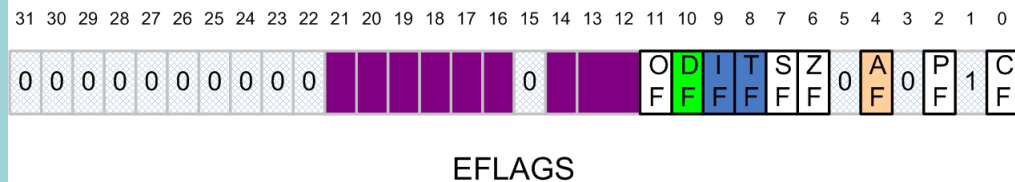
- Размещение битовых полей зависит от реализации
- Битовые поля могут размещаться как справа налево, так и наоборот, в зависимости от реализации
- Битовые поля могут пересекать границы машинных слов
- Выравнивание битовых полей зависит от реализации
- К битовым полям неприменима операция `&` и оператор `sizeof`
- Код непереносим между различными системами

```
typedef union {
    unsigned int raw;
    struct {
        int CF    : 1;
        int gap1  : 1;
        int PF    : 1;
        int gap2  : 1;
        int AF    : 1;
        int gap3  : 1;
        int ZF    : 1;
        ...
        int gap5  : 10;
    } fields;
} t_eflags;
```

checkEflagsState:

```
push ebp
mov  ebp, esp
mov  eax, dword [ebp+8]
mov  edx, dword [eax]
xor  eax, eax
test edx, -4161496
      ; 11111111_11000000_10000000_00101000b
jne  .L3
mov  eax, edx
shr  eax
and  eax, 1
.L3:
pop  ebp
ret
```

```
int checkEflagsState(t_eflags *sw) {
    t_eflags andMask = {0};
    andMask.fields.gap2 = 1;
    andMask.fields.gap3 = 1;
    andMask.fields.gap4 = 1;
    andMask.fields.gap5 = -1;
    return !(sw->raw & andMask.raw) && (sw->raw & 2);
}
```



Типы данных языка Си

Итоги

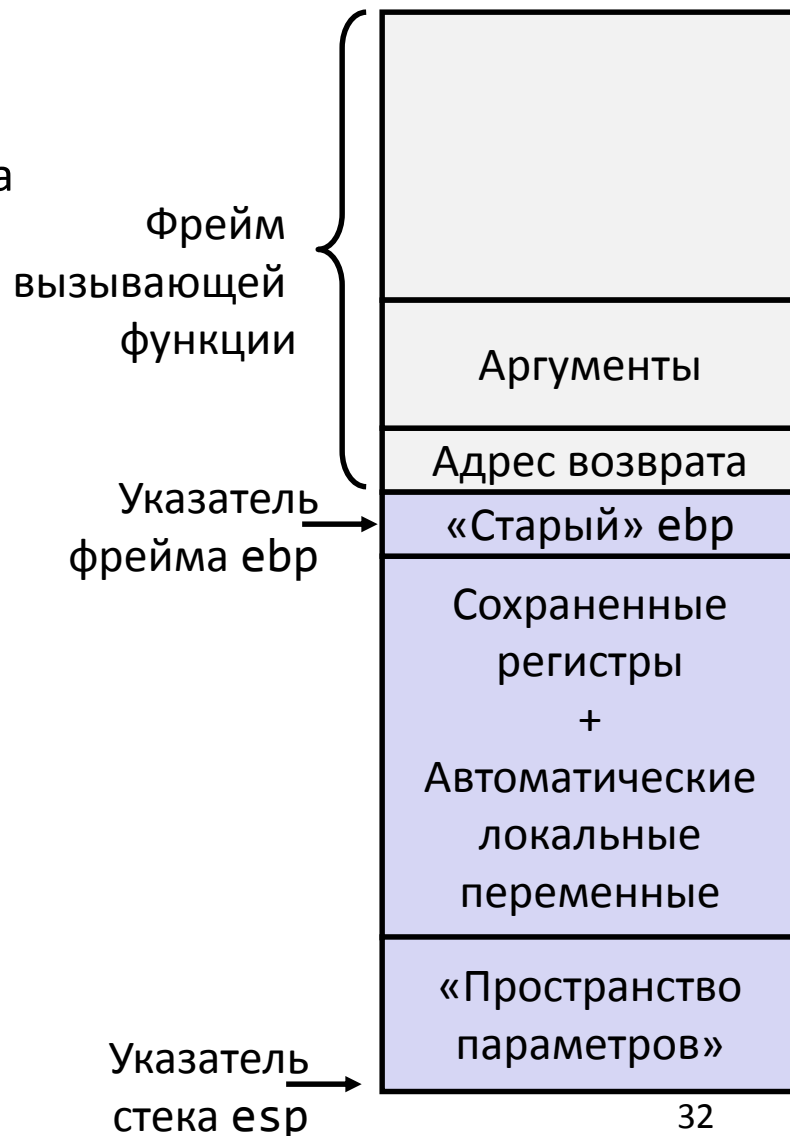
- Размещение переменных
 - Классы памяти: автоматическая, статическая, динамическая
 - Регистр вместо памяти (при определенных условиях)
- Массивы в языке Си
 - Непрерывная последовательность байт в памяти
 - Выравнивание всего массива удовлетворяет требования к выравниванию для каждого его элемента
 - Имя массива – указатель на его первый элемент
 - Нет никаких проверок выхода за границы
- Структуры
 - Память под поля выделяется в порядке объявления этих полей
 - Помещаются пропуски между полями и в конце всей структуры с целью выравнивания данных
- Объединения
 - Объявленные поля перекрываются в памяти
 - Способ жестокого обмана системы типов языка Си

Далее...

- **Функции**
 - **Соглашение CDECL**
 - **Рекурсия**
 - **Что происходит в Си-программе до и после функции `main`**
 - **Выравнивание стека**
 - **Различные соглашения о вызове функций**
 - **`cdecl/stdcall/fastcall`, отказ от указателя фрейма**
 - **Соглашение вызова для x86-64**
 - **Переполнение буфера, эксплуатация ошибок, механизмы защиты**
- **Организация динамической памяти**
- **Числа с плавающей точкой**

Поддержка функций на уровне аппаратуры

- Аппаратный стек
 - Регистр ESP указывает на верхушку стека
 - Стек растет вниз
 - Команды PUSH и POP, сложение и вычитание констант из ESP
- Вызов/возврат
 - Команды CALL и RET
- Состояния выполняющихся функций
 - На стеке размещаются фреймы
 - Каждый фрейм хранит текущее состояние вызова функции
 - На верхнюю границу указывает EBP
 - Содержимое
 - Фактические аргументы/формальные параметры
 - Адрес возврата
 - Автоматические локальные переменные
 - Вспомогательные переменные



Поддержка функций на уровне соглашений:

соглашение CDECL

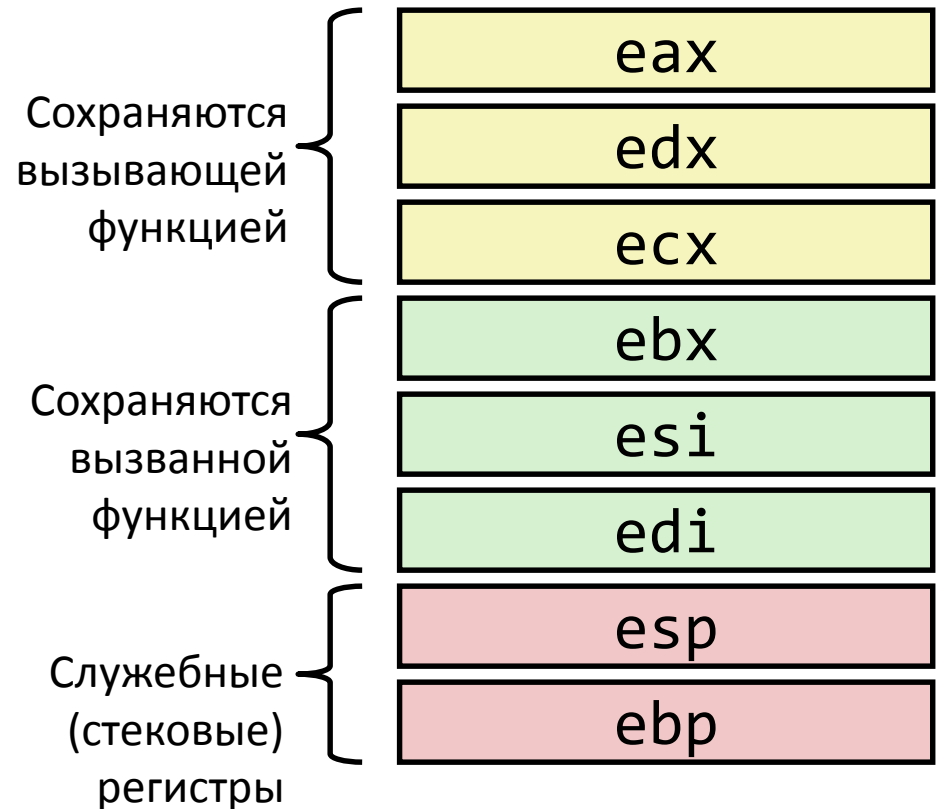
- Размещение параметров
 - На стеке, с обязательным выравниванием по 4-х байтной границе
- Порядок параметров
 - «Обратный», от «верхушки» стека ко «дну»: сразу над адресом возврата размещен первый параметр, затем второй, и т.д.
- Очистка стека от аргументов после вызова
 - Очищает вызывающая функция
 - Сохранность значений в пространстве аргументов после возврата в вызывающую функцию не гарантируется
 - Поскольку cdecl в большинстве случаев не предполагает изменение границ фрейма во время работы функции, очистка после возврата в вызывающую функцию фактически не выполняется, а переносится в эпилог
- Возвращаемое функцией значение
 - EAX
 - EDX:EAX
 - Через память

Размещение параметров и возвращаемого значения

- `sizeof == 4`
 - Целиком занимает машинное слово, помещаемое на стек
 - EAX
- `sizeof < 4`
 - Занимает младшие байты слова на стеке
 - AX, AL
- `sizeof == 8 (long long)`
 - Два машинных слова в естественном порядке
 - EDX:EAX
- Массивы \equiv указатели
- Структуры и объединения
 - В gcc используется *гибридное* соглашение вызова, совмещающее `cdecl` и `stdcall` (соглашение вызова 32-х разрядного WinAPI)

Сохранение регистров в IA32/Linux+Windows

- `eax`, `edx`, `ecx`
 - Вызывающая функция сохраняет эти регистры перед `call`, если планирует использовать позже
- `eax`
 - Используется для возврата значения, если возвращается целый тип
- `ebx`, `esi`, `edi`
 - Вызванная функция сохраняет значения этих регистров, если планирует ими воспользоваться
- `esp`, `ebp`
 - Сохраняются вызванной функцией
 - Восстанавливаются перед выходом из функции



Рекурсивная функция

```
/* Рекурсивный popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

• Регистры

- **eax, edx** используются без предварительного сохранения
- **ebx** используется, но предварительно сохраняется в начале функции и восстанавливается в конце

```
pcount_r:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 4
    mov     ebx, dword [ebp + 8]
    mov     eax, 0
    test    ebx, ebx
    je      .L3
    mov     eax, ebx
    shr     eax, 1
    mov     dword [esp], eax
    call    pcount_r
    mov     edx, ebx
    and     edx, 1
    lea     eax, [edx + eax]
.L3:
    add     esp, 4
    pop     ebx
    pop     ebp
    ret
```

Рекурсивный вызов (1/5)

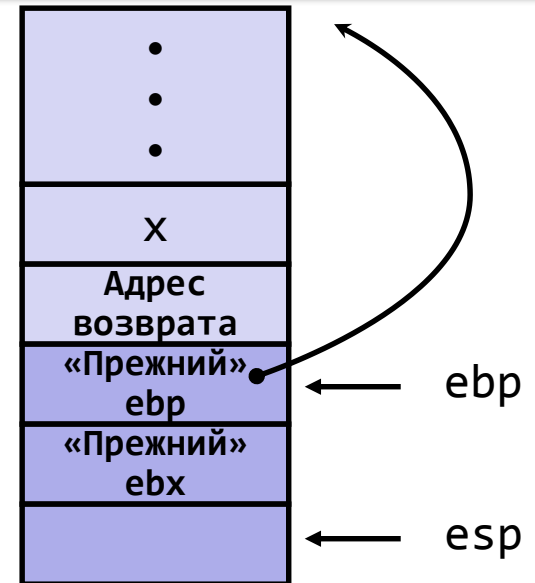
```
/* Рекурсивный popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 4

    mov     ebx, dword [ebp + 8]
    ...
```

Пролог функции

- Действия в прологе
 - Сохраняем и «переставляем» `ebp`
 - Сохраняем значение `ebx` на стеке
 - Расширяем фрейм: выделяем место для размещения аргумента рекурсивного вызова
- Размещаем значение `x` в `ebx`



Рекурсивный вызов (2/5)

```
/* Рекурсивный popcount */  
int pcount_r(unsigned x) {  
    if (x == 0)  
        return 0;  
    else return  
        (x & 1) + pcount_r(x >> 1);  
}
```

- Действия

- Если $x == 0$, выходим из функции
 - Регистр `eax` содержит 0

```
...  
mov    eax, 0  
test   ebx, ebx  
je     .L3  
...  
.L3:  
...  
ret
```

ebx

x

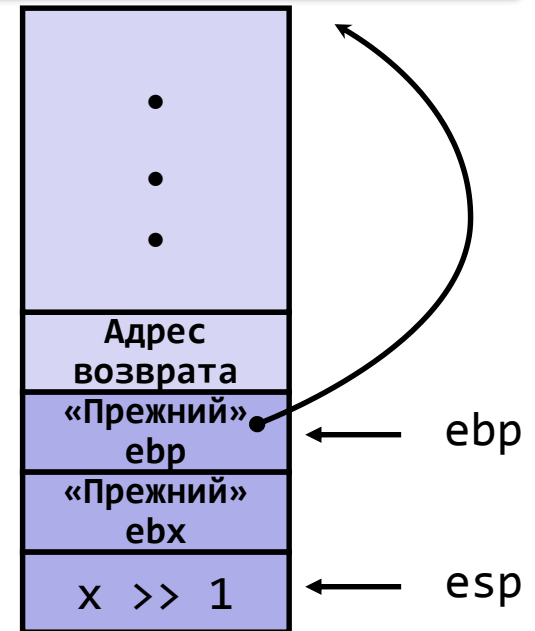
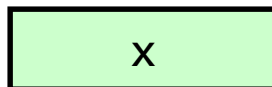
Рекурсивный вызов (3/5)

```
/* Рекурсивный popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

```
...
mov    eax, ebx
shr    eax, 1
mov    dword [esp], eax
call   pcount_r
...
```

- Действия
 - Сохраняем $x \gg 1$ на стеке
 - Выполняем рекурсивный вызов
- Результат
 - `eax` содержит возвращенное значение
 - `ebx` содержит неизменное значение x

ebx



Рекурсивный вызов (4/5)

```
/* Рекурсивный popcount */  
int pcount_r(unsigned x) {  
    if (x == 0)  
        return 0;  
    else return  
        (x & 1) + pcount_r(x >> 1);  
}
```

```
...  
mov    edx, ebx  
and     edx, 1  
lea     eax, [edx + eax]  
...
```

- Состояние регистров
 - eax содержит значение полученное от рекурсивного вызова
 - ebx содержит x
- Действия
 - Вычисляем $(x \& 1) +$ возвращенное значение
- Результат
 - Регистр eax получает результат работы функции

ebx

x

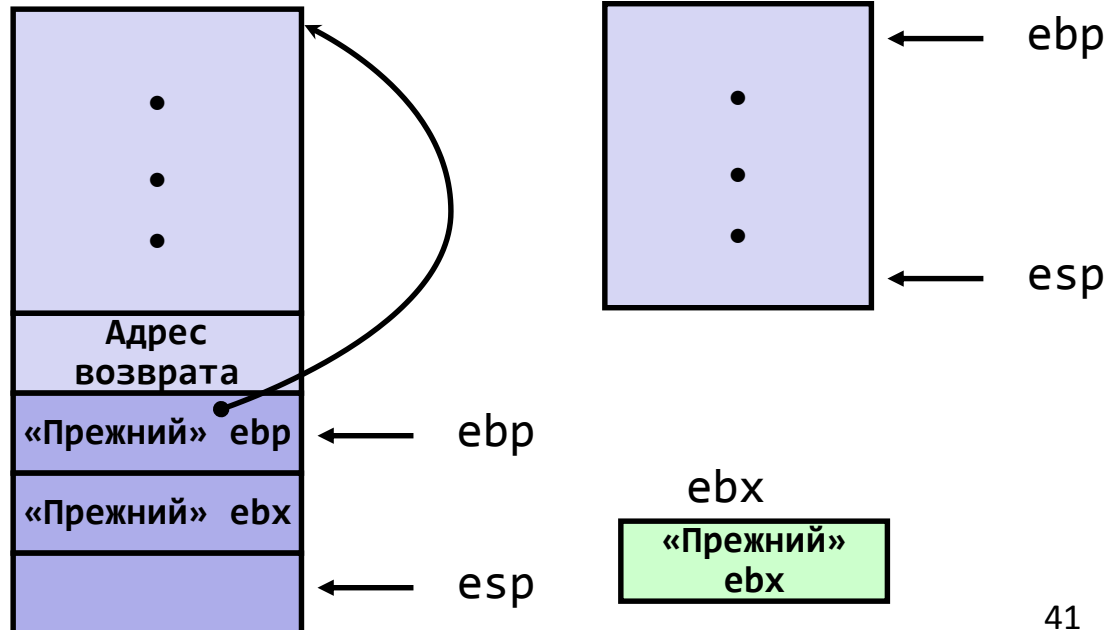
Рекурсивный вызов (5/5)

```
/* Рекурсивный popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

```
...
L3:
    add esp, 4
    pop ebx
    pop ebp
    ret
```

Эпилог функции

- Действия в эпилоге
 - Восстанавливаем значения ebx и ebp
 - Восстанавливаем esp



Рекурсия – выводы

- Не используются дополнительные приемы
 - Создание фреймов гарантирует, что каждый вызов располагает персональным блоком памяти
 - Хранятся регистры и локальные переменные
 - Хранится адрес возврата
 - Общее соглашение о сохранении регистров препятствует порче регистров различными вызовами
 - Стековая организация поддерживается порядком вызовов и возвратов из функций
 - Если P вызывает Q, тогда Q завершается до того, как завершится P
 - Последним пришел, первым ушел
- Аналогично при неявной рекурсии
 - P вызывает Q; Q вызывает P