© 2015 МГУ/ВМК/СП

# Лекция 0xB

14 марта

© 2015 МГУ/ВМК/СП

# 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

© 2015 МГУ/ВМК/СП

# 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

© 2015 МГУ/ВМК/СП

# 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

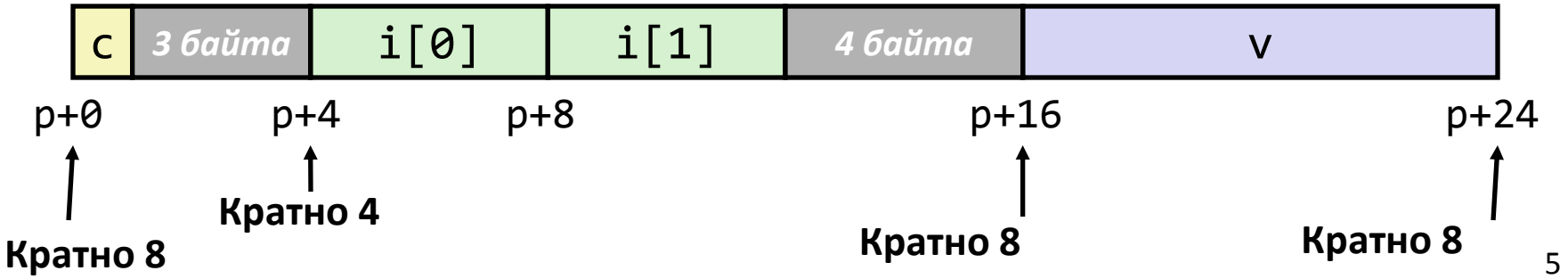
Тип long double в компиляторе MS VC
- 16-разрядная архитектура
  - sizeof(long double) = 10 // 80 бит
- 32-разрядная архитектура и далее …
  - long double ≡ double

© 2015 МГУ/ВМК/СП

# Выполнение правил выравнивания для полей

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

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

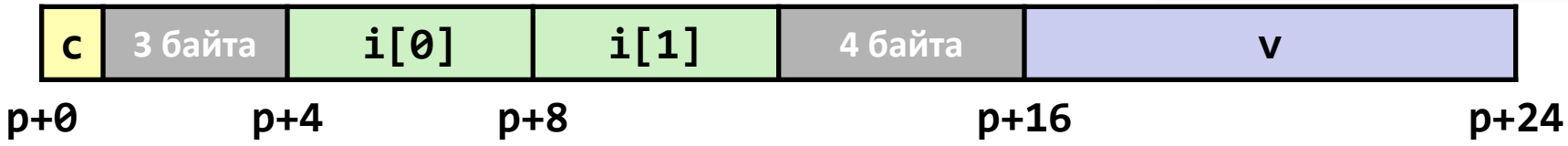


© 2015 МГУ/ВМК/СП

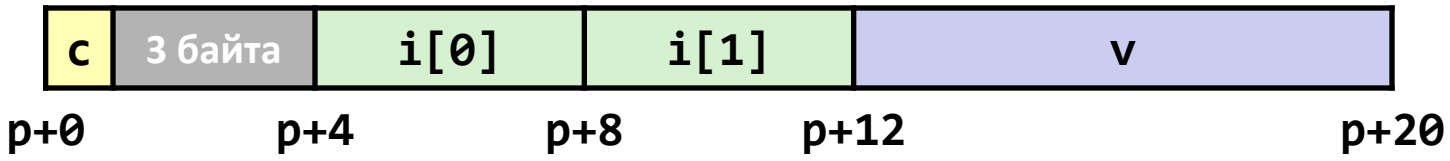
# Различные соглашения о выравнивании

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

- x86-64 или IA32 Windows:
  - K = 8, из-за наличия поля типа **`double`**




- IA32 Linux
  - K = 4; double рассматривается аналогично 4-байтным типам данных

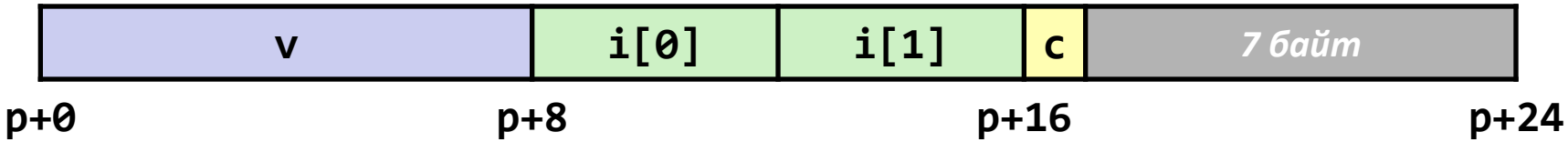


© 2015 МГУ/ВМК/СП

# Выравнивание всей структуры

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

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

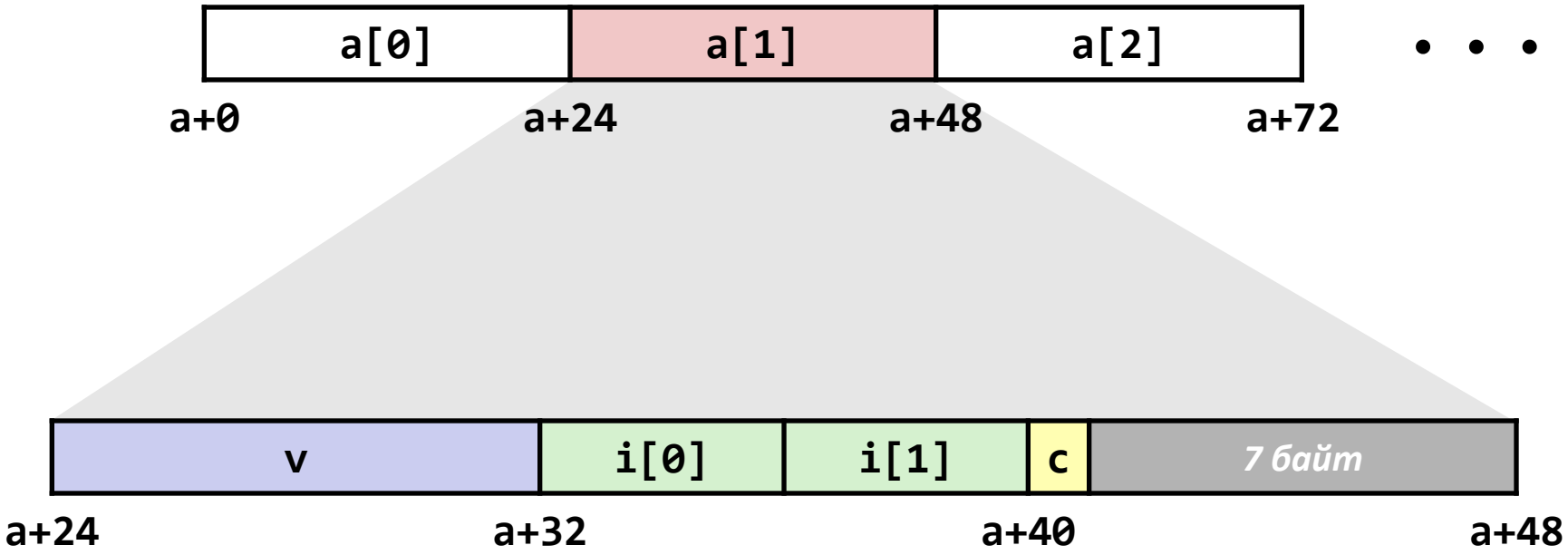


© 2015 МГУ/ВМК/СП

# Массивы структур

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

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

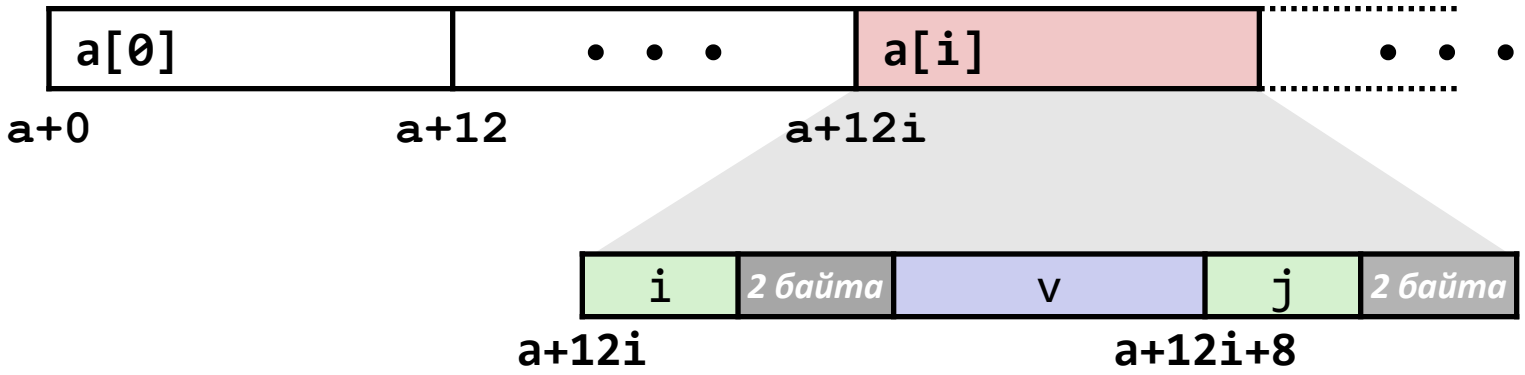


© 2015 МГУ/ВМК/СП

# 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]
```

© 2015 МГУ/ВМК/СП

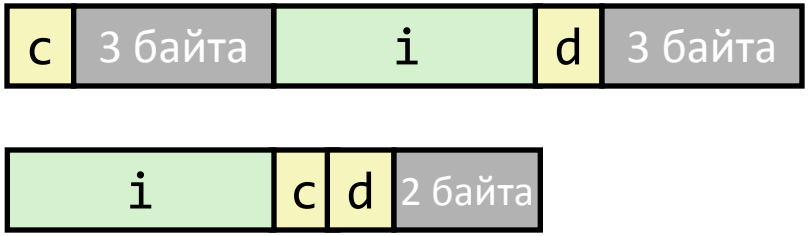
# Как сохранить место

- 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)

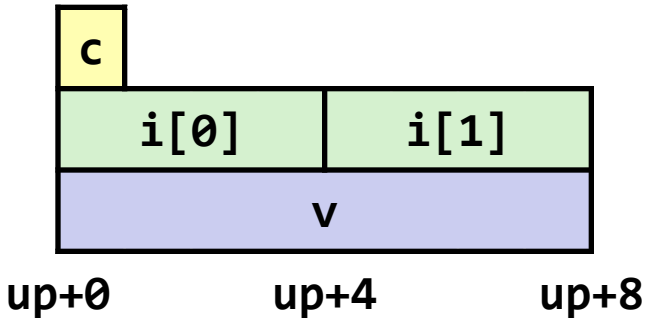


© 2015 МГУ/ВМК/СП

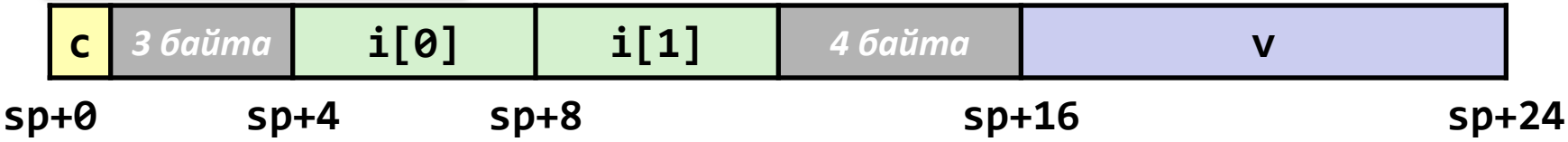
# Размещение объединений

- 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;
```



© 2015 МГУ/ВМК/СП

# 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;
};
```

?

© 2015 МГУ/ВМК/СП

# 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**) ?

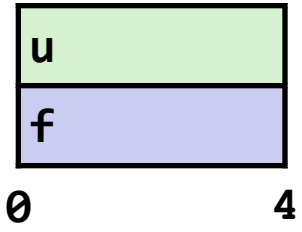
Какие смещения у полей?

IA-32 Linux / IA-32 Windows

© 2015 МГУ/ВМК/СП

# 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` **?**

© 2015 МГУ/ВМК/СП

# 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 / обратный порядок байт
  - Старший байт имеет наименьший адрес
  - Sparc (до V8)
- Little-endian / от младшего к старшему / прямой порядок байт
  - Младший байт имеет наименьший адрес
  - Intel x86 (IA-32)
- Переключаемый порядок байт
  - ARM, PowerPC, Alpha, SPARC V9, MIPS, PA-RISC и IA-64

© 2015 МГУ/ВМК/СП

# 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] | | | | | | | |

© 2015 МГУ/ВМК/СП

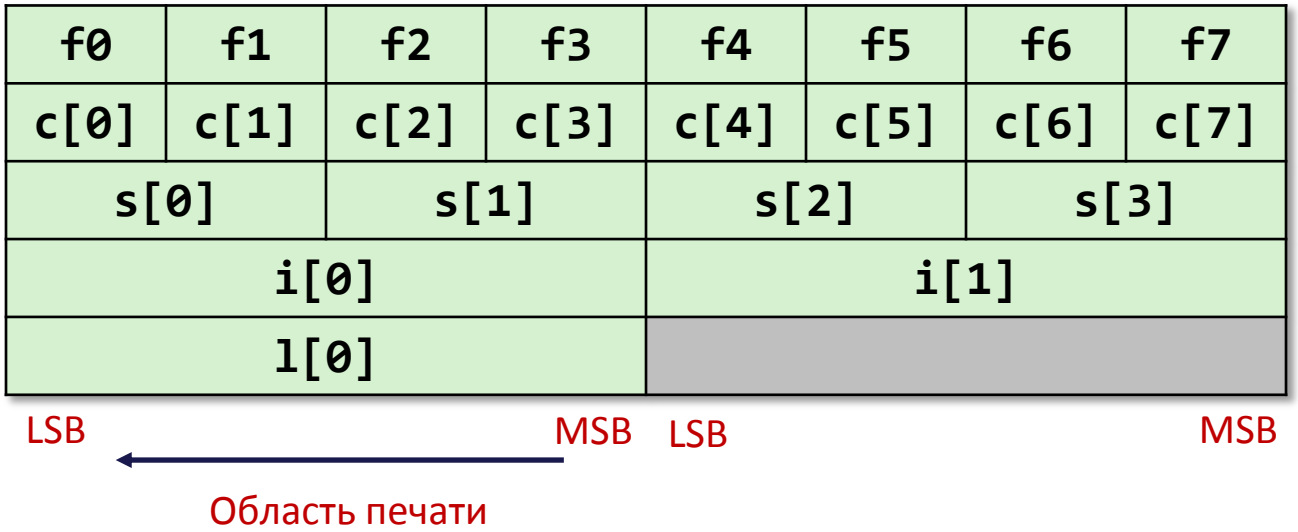
# Пример (продолжение)

```
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]);
```

© 2015 МГУ/ВМК/СП

# Порядок байт IA-32

## Прямой порядок байт



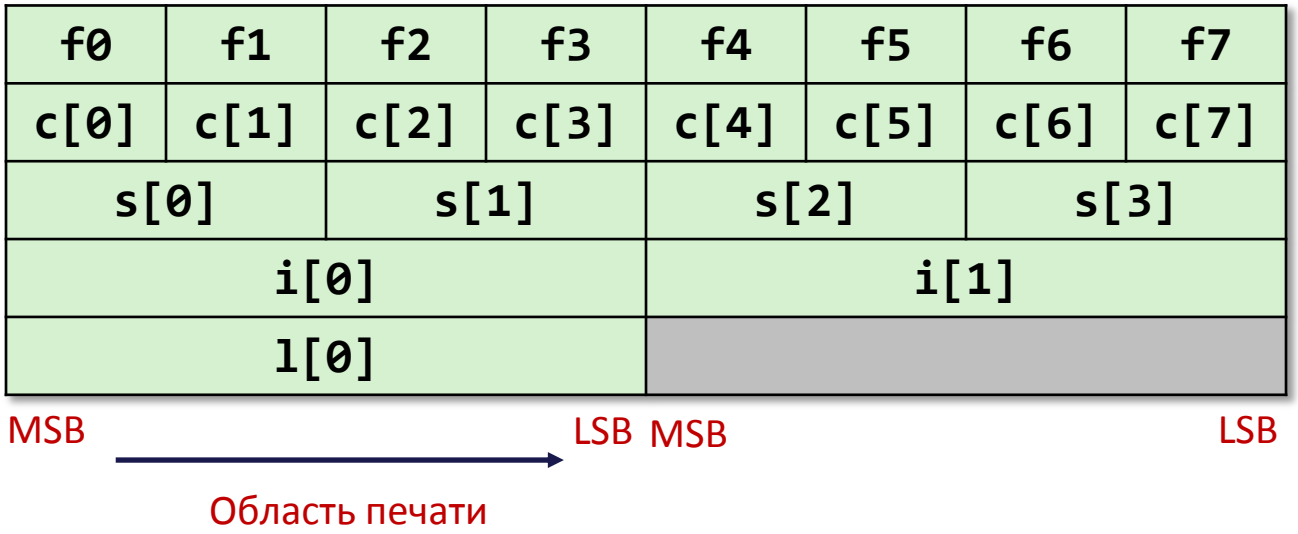

Вывод на консоль:

```
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   == [0xf3f2f1f0]
```

© 2015 МГУ/ВМК/СП

# Порядок байт Sun

Обратный порядок байт




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]
```

© 2015 МГУ/ВМК/СП

# Порядок байт x86-64

## Прямой порядок байт

## 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]
```

© 2015 МГУ/ВМК/СП

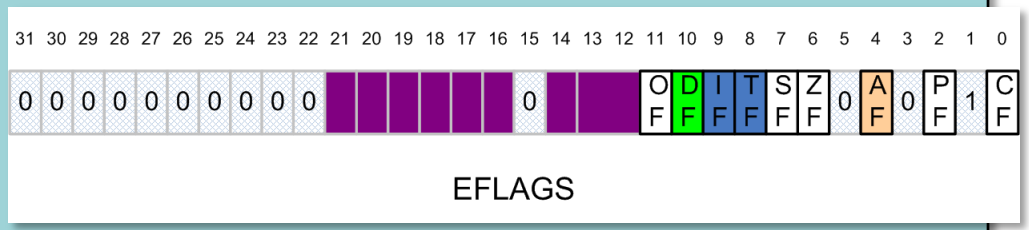
# Битовые поля

- 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

© 2015 МГУ/ВМК/СП

```
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;

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);
}
```

```
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
```



© 2015 МГУ/ВМК/СП

# 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 для с целью выравнивания
- Объединения
  - Объявленные поля перекрываются в памяти
  - Способ жестокого обмана системы типов языка Си

© 2015 МГУ/ВМК/СП

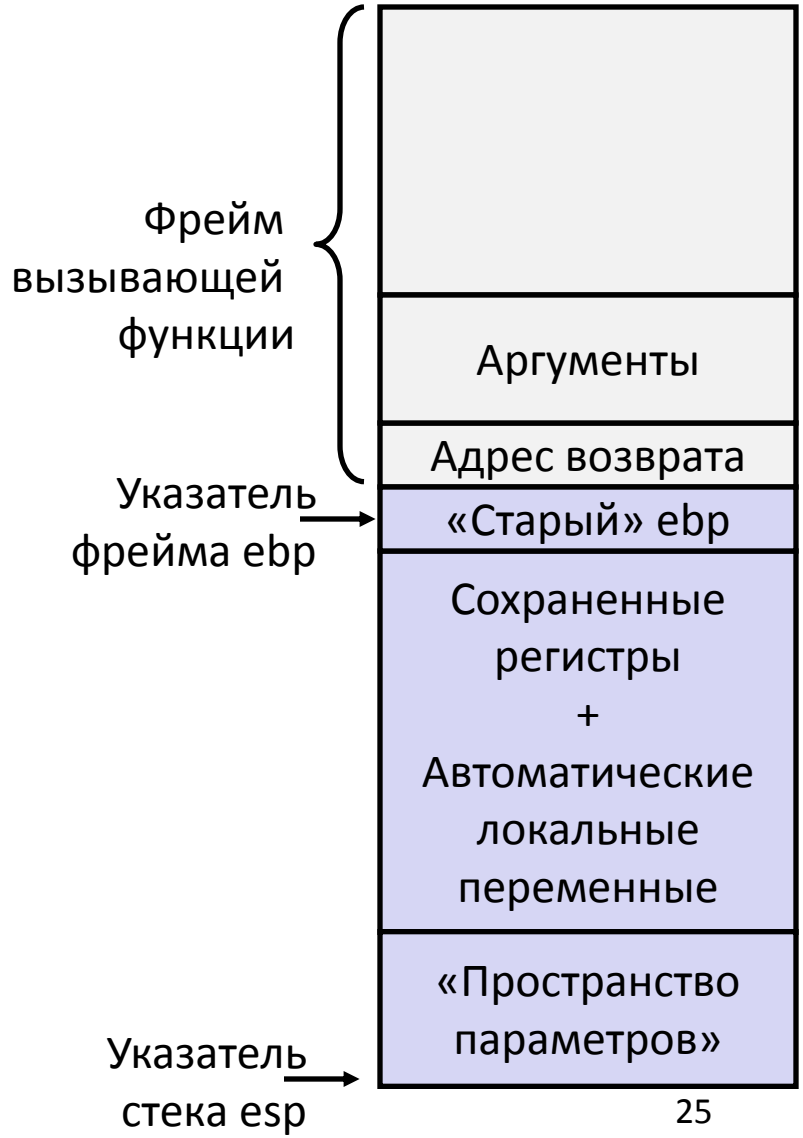
# Далее…

- ***Функции***
  - ***Рекурсия***
  - ***Выравнивание стека***
  - ***Различные соглашения о вызове функций***
    - ***cdecl/stdcall/fastcall, отказ от указателя фрейма***
    - ***Соглашение вызова для x86-64***
  - ***Переменное число параметров***
  - ***Вызов по указателю***
  - ***Ассемблерные вставки***
  - ***Переполнение буфера, эксплуатация ошибок, механизмы защиты***
- Операции над строками
- Организация динамической памяти
- Числа с плавающей точкой

© 2015 МГУ/ВМК/СП

# Поддержка функций на уровне аппаратуры

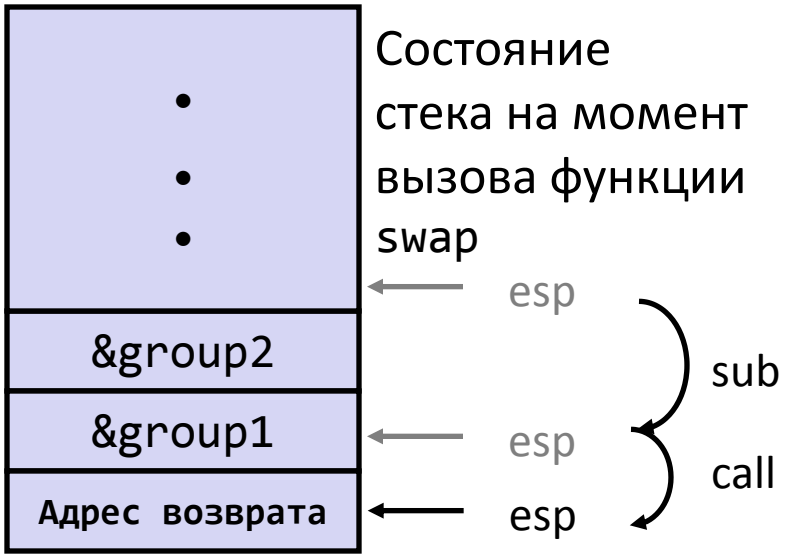
- Аппаратный стек
  - Команды PUSH и POP
  - Регистр ESP
  - Стек растет вниз
- Вызов/возврат
  - Команды CALL и RET
- Состояние текущего вызова
  - Фрейм – блок памяти на стеке
  - Регистр EBP
  - Содержимое
    - Фактические аргументы/формальные параметры
    - Адрес возврата
    - Автоматические локальные переменные
    - Вспомогательные переменные



© 2015 МГУ/ВМК/СП

```
int group1 = 101;
int group2 = 106;

void call_swap() {
  swap(&group1, &group2);
}
```

```
void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```




Вызываем swap из call_swap

```
call_swap:
    ...
    sub     esp, 8
    mov     dword [esp + 4], group2
    mov     dword [esp], group1
    call    swap
    ...
```

© 2015 МГУ/ВМК/СП

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    push    ebp
    mov     ebp, esp
    push    ebx

    mov     edx, dword [ebp + 8]
    mov     ecx, dword [ebp + 12]
    mov     ebx, dword [edx]
    mov     eax, dword [ecx]
    mov     dword [edx], eax
    mov     dword [ecx], ebx

    pop     ebx
    pop     ebp
    ret
```

Пролог

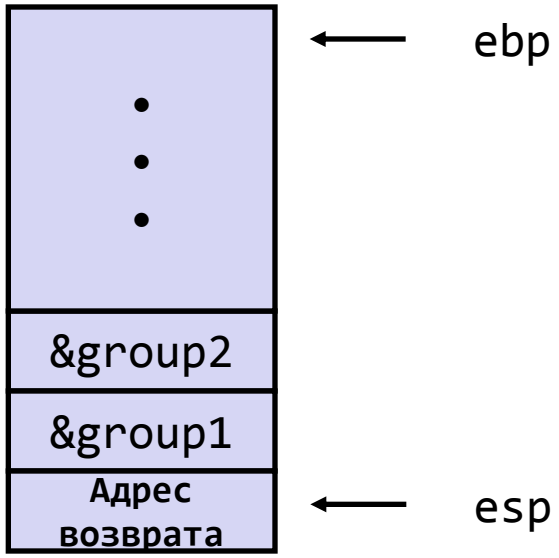
Тело функции

Эпилог

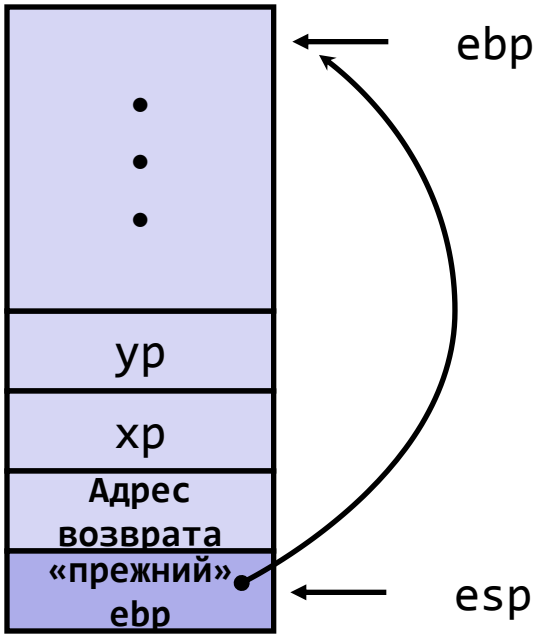
© 2015 МГУ/ВМК/СП

# Swap: как работает пролог функции. №1

Начальное состояние стека




Текущее состояние стека

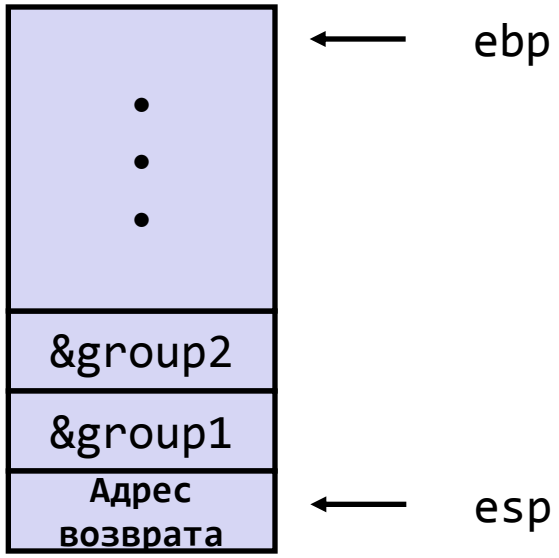



```
swap:
    push ebp        ; (1)
    mov  ebp, esp   ; (2)
    push ebx        ; (3)
```

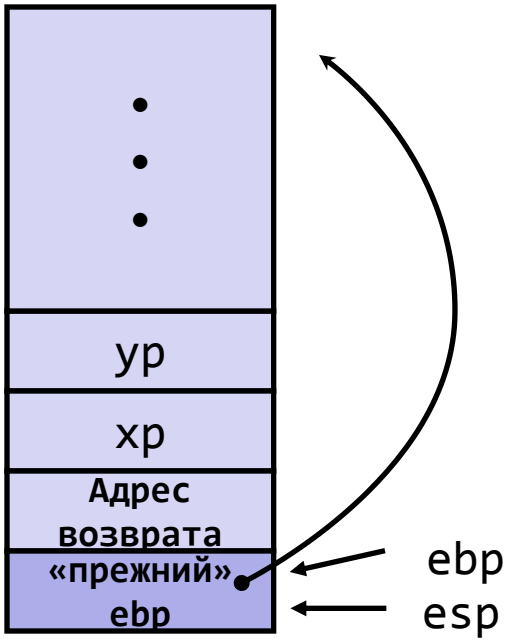
© 2015 МГУ/ВМК/СП

# Swap: как работает пролог функции. №2

Начальное состояние стека




Текущее состояние стека

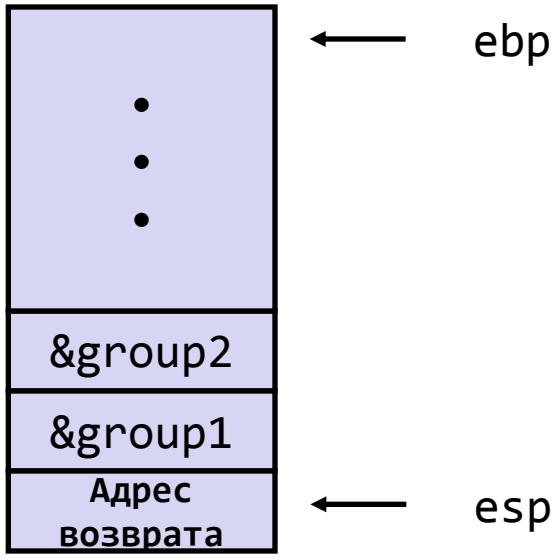



```
swap:
    push ebp        ; (1)
    mov  ebp, esp   ; (2)
    push ebx        ; (3)
```

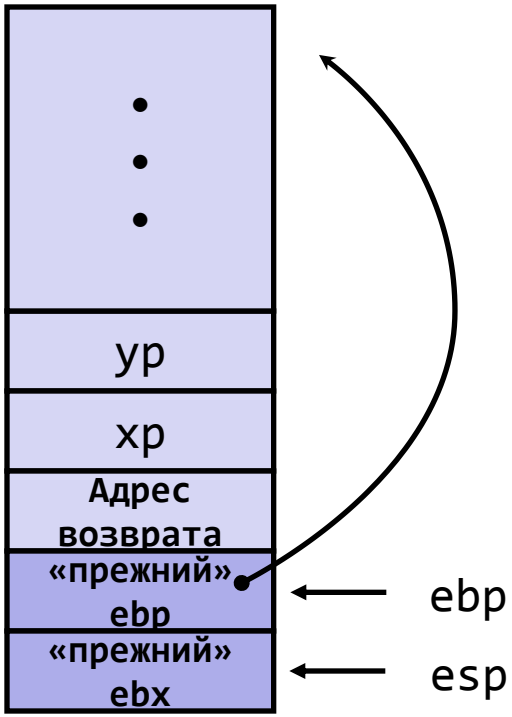
© 2015 МГУ/ВМК/СП

# Swap: как работает пролог функции. №3

Начальное состояние стека



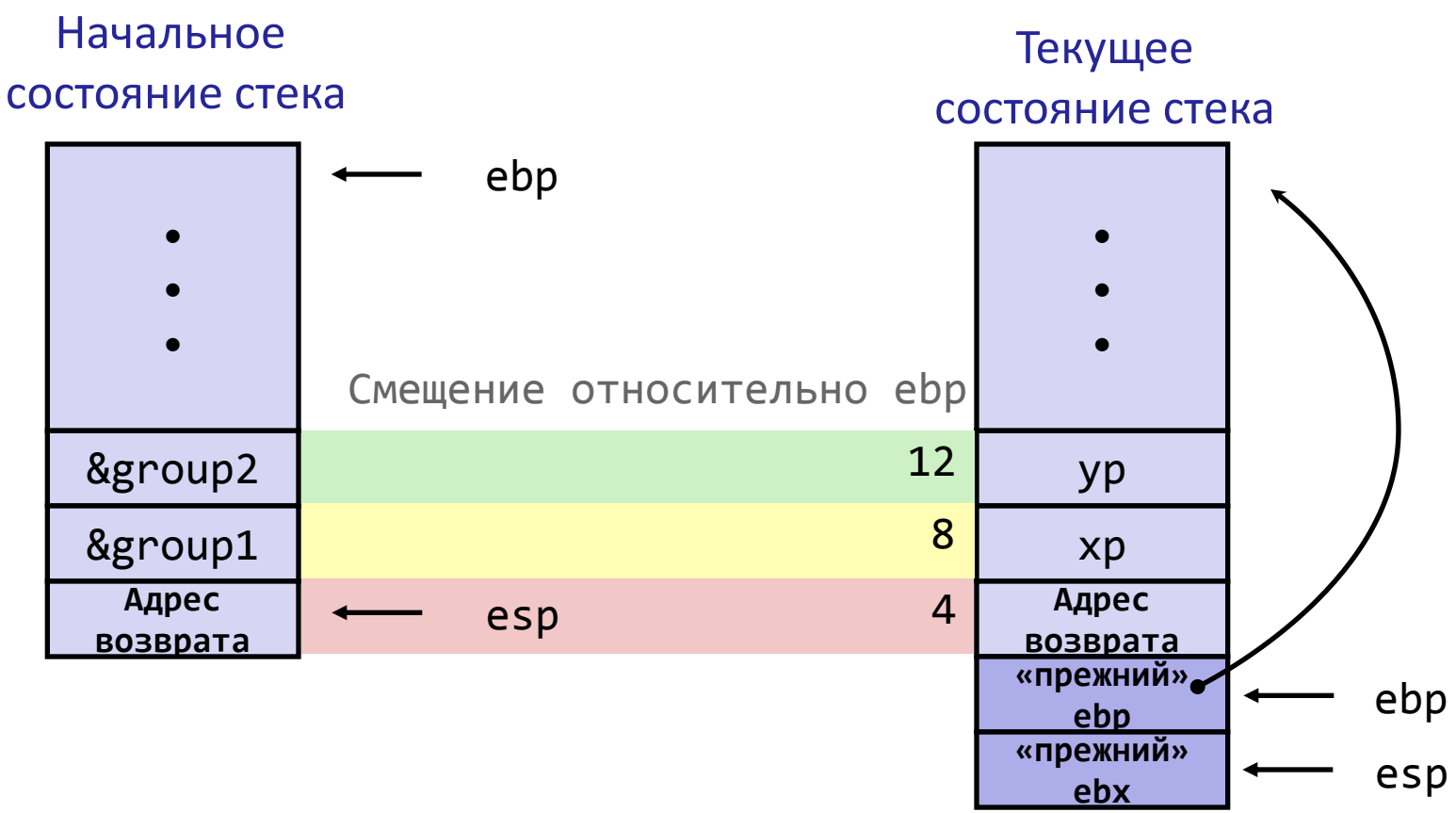

Текущее состояние стека




```
swap:
    push ebp        ; (1)
    mov  ebp, esp   ; (2)
    push ebx        ; (3)
```

© 2015 МГУ/ВМК/СП

# Тело функции swap

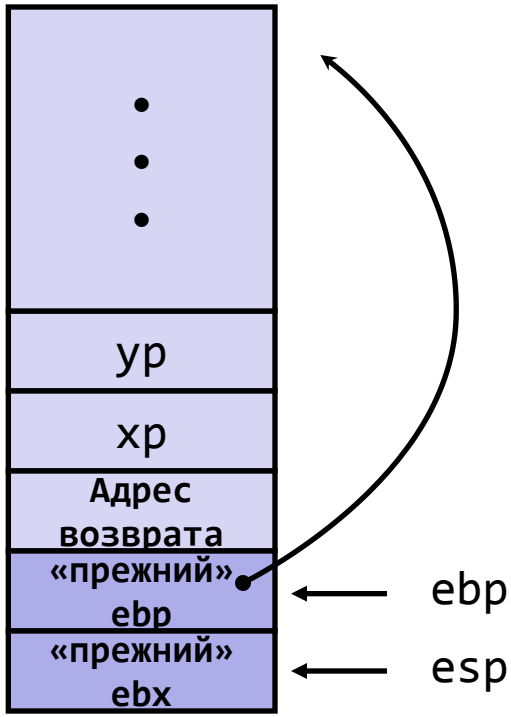



```
mov edx, dword [ebp + 8]  ; извлекаем параметр xp
mov ecx, dword [ebp + 12] ; извлекаем параметр yp
. . .
```

© 2015 МГУ/ВМК/СП

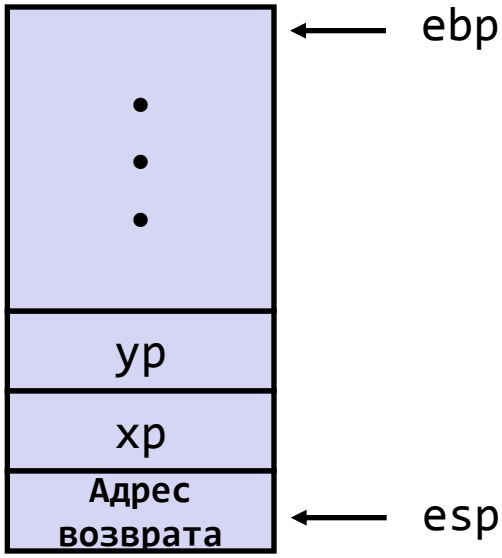
# Swap: как работает эпилог функции

Стек до эпилога




**pop ebx**
**pop ebp**

Итоговое состояние стека

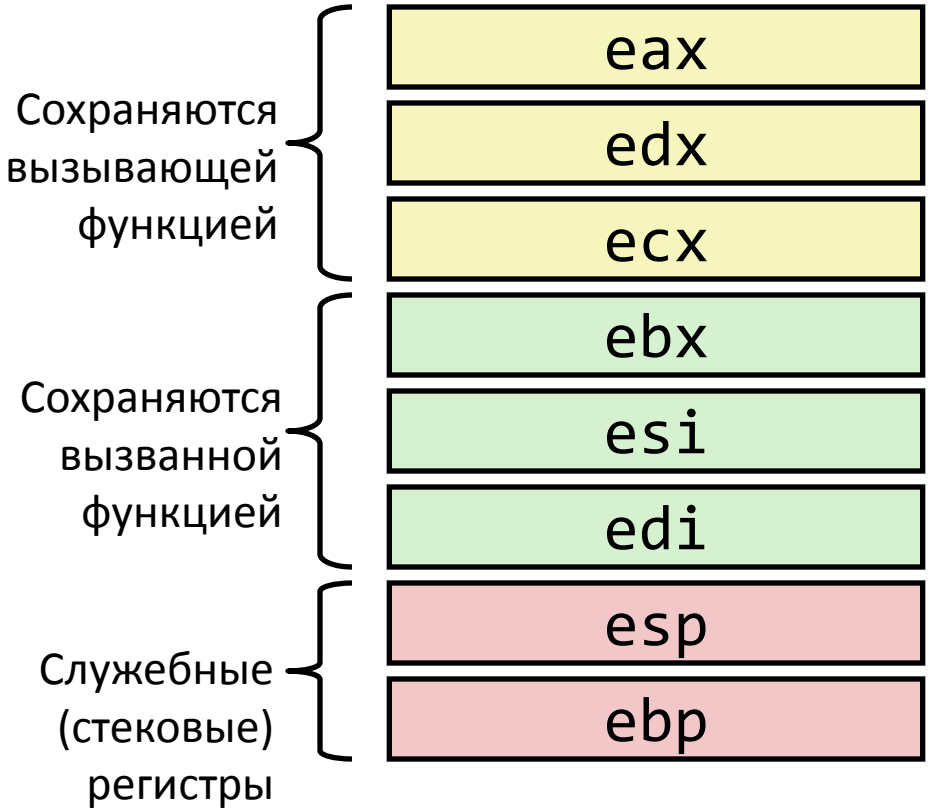



- Что происходит
  - Сохраняется и восстанавливается значение регистра ebx, …
  - но не eax, ecx, edx

© 2015 МГУ/ВМК/СП

# 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



© 2015 МГУ/ВМК/СП

# Передача указателей на локальные переменные в качестве параметров

Создаем указатель

```
/* Вычисляем x + 3 */
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

Разыменовываем указатель

```
/* увеличиваем значение на k */
void incrk(int *ip, int k) {
  *ip += k;
}
```

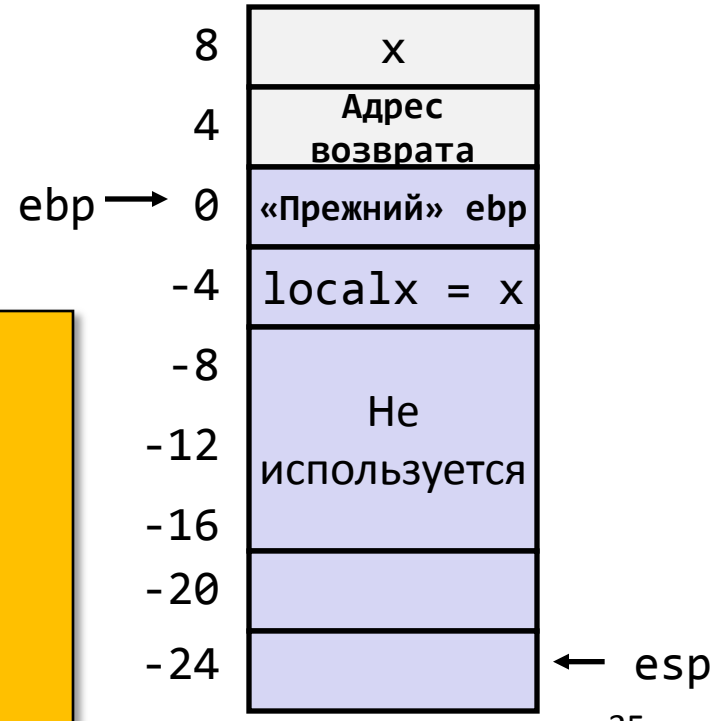
- `add3` создает указатель и передает его в `incrk`

© 2015 МГУ/ВМК/СП

# Создаем и инициализируем локальные переменные

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Переменную `localx` необходимо размещать на стеке
  - Причина: создается указатель на нее
- Вычисляем указатель: `ebp - 4`




Начало функции add3

```
add3:
   push  ebp
   mov   ebp, esp
   sub   esp, 24  ; выделяем 24 байта
   mov   eax, dword [ebp + 8]
   mov   dword [ebp - 4], eax
        ; Присваиваем localx значение x
```

© 2015 МГУ/ВМК/СП

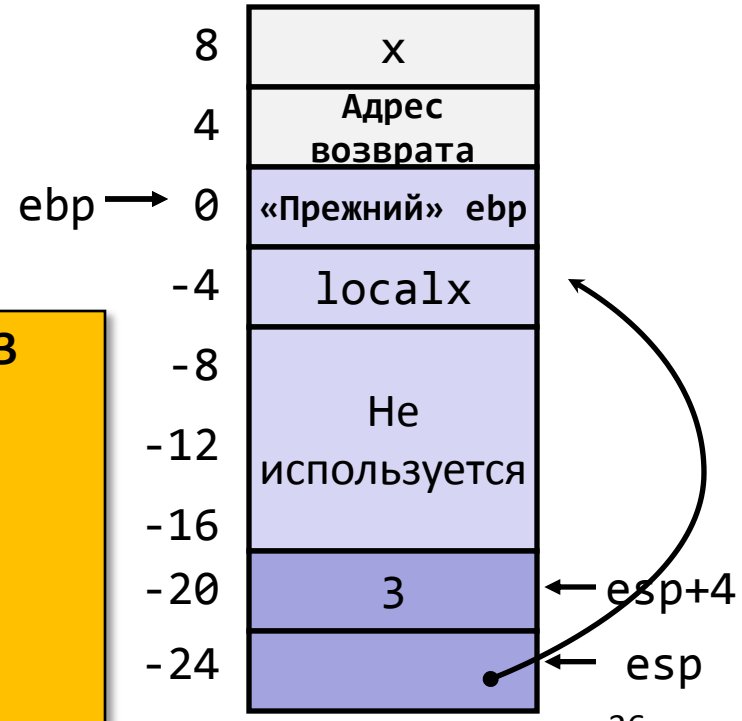
# Создаем фактический параметр - указатель

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Инструкция `lea` используется для вычисления адреса `localx`

Продолжение функции add3

```
mov dword [esp + 4], 3 ; 2-ой параметр = 3
lea eax, [ebp - 4]     ; &localx
mov dword [esp], eax   ; 1-ый параметр =
                       ;          &localx
call incrk
```



© 2015 МГУ/ВМК/СП

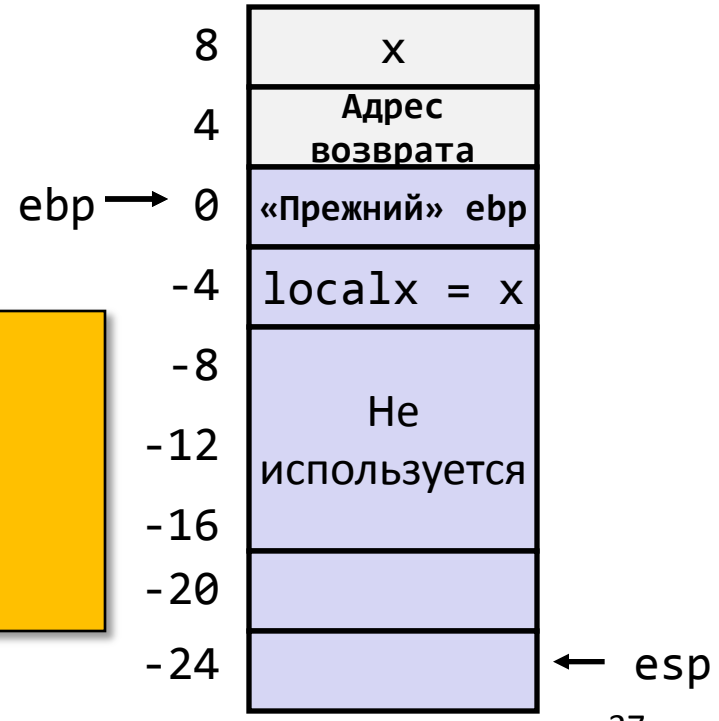
# Возвращаемое значение – локальная переменная

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Извлекаем из стека `localx` – возвращаемое значение функции

Конец функции add3

```
 mov eax, dword [ebp - 4]
  ; возвращаемое значение = localx
 leave
 ret
```



© 2015 МГУ/ВМК/СП

# Рекурсивная функция

```
/* 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]
    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
```

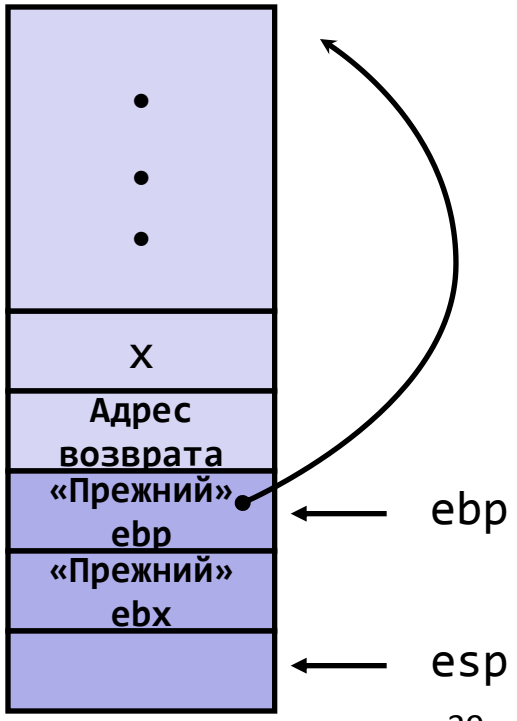
- Регистры
  - eax, edx используются без предварительного сохранения
  - ebx используется, сохраняется в начале и восстанавливается в конце

© 2015 МГУ/ВМК/СП

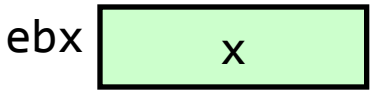
# Рекурсивный вызов (1/5)

```
/* 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]
    ...
```




- Действия
  - Сохраняем значение ebx на стеке
  - Выделяем место для размещения аргумента вызова (рекурсивного)
  - Размещаем значение x в ebx



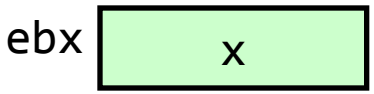
© 2015 МГУ/ВМК/СП

# Рекурсивный вызов (2/5)

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

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

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

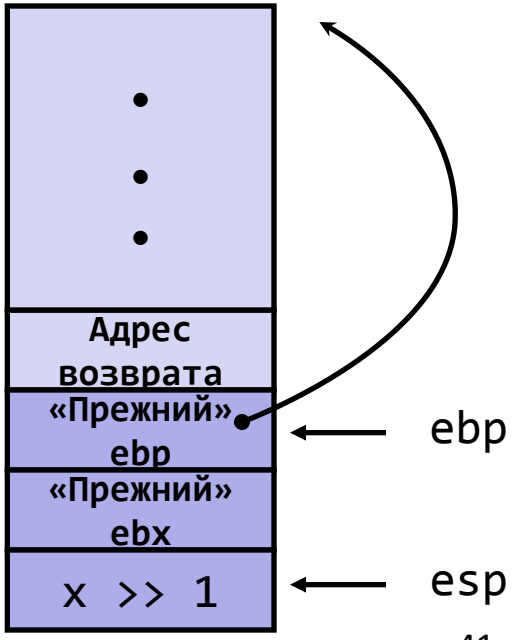


© 2015 МГУ/ВМК/СП

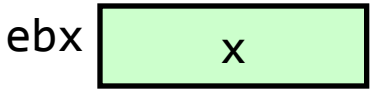
# Рекурсивный вызов (3/5)

```
/* 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 >> 1 на стеке
  - Выполняем рекурсивный вызов
- Результат
  - eax содержит возвращенное значение
  - ebx содержит неизменное значение x



© 2015 МГУ/ВМК/СП

# 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

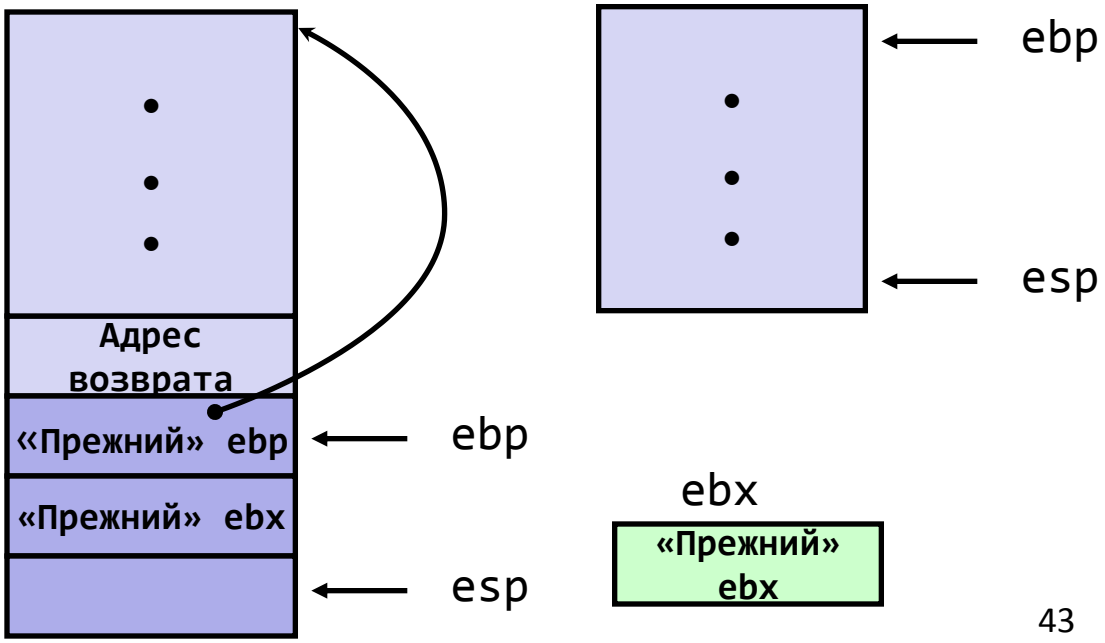
© 2015 МГУ/ВМК/СП

# 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
```

- Действия
  - Восстанавливаем значения ebx и ebp
  - Восстанавливаем esp



© 2015 МГУ/ВМК/СП

# Рекурсия. Заключение.

- 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

© 2015 МГУ/ВМК/СП

# Соглашение CDECL

- Размещение параметров
  - Стек
- Порядок параметров
  - «обратный»: от «верхушки» стека ко «дну»
- Какие регистры могут быть использованы вызванной функцией
  - EAX, EDX, ECX
- Очистка стека от аргументов вызванной функции
  - Очищает вызвавшая функция
- Возвращаемое функцией значение
  - EAX
  - EAX:EDX
  - Через память