Компьютерные основы программирования
Представление программ
часть 4

Лекция 7, 23 марта 2017

Лектор: Чуканова Ольга Владимировна

Кафедра информатики

602 АК

ovcha@mail.ru

Целочисленные регистры x86-64

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Количество регистров удвоено!
- Доступны как 8-, 16-, 32-, 64- битные

Целочисленные регистры x86-64: Соглашения об использовании

%rax	Результат вызова
%rbx	Сохраняет вызываемая
%rcx	Аргумент №4
%rdx	Аргумент №3
%rsi	Аргумент №2
%rdi	Аргумент №1
%rsp	Указатель стека
%rbp	Сохраняет вызываемая

%r8	Аргумент №5
%r9	Аргумент №6
%r10	Сохраняет вызывающая
%r11	Сохраняет вызывающая
%r12	Сохраняет вызываемая
%r13	Сохраняет вызываемая
%r14	Сохраняет вызываемая
%r15	Сохраняет вызываемая

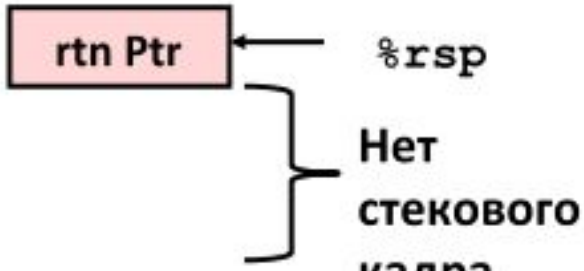
Целочисленные регистры x86-64

- **Аргументы передаются в процедуры через регистры**
  - Если параметров больше 6, то избыток - в стеке
  - Эти регистры также могут использоваться как сохраняемые вызывающей
- **Все обращение в стековый кадр – от указателя стека**
  - Исключается необходимость изменять **`%ebp/%rbp`**
- **Другие регистры**
  - 6 сохраняемых вызываемой
  - 2 сохраняемых вызывающей
  - 1 результат вызова (или как сохранямый вызывающей)
  - 1 специальный (указатель стека)

x86-64: обмен данных

```
void swap_l(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```



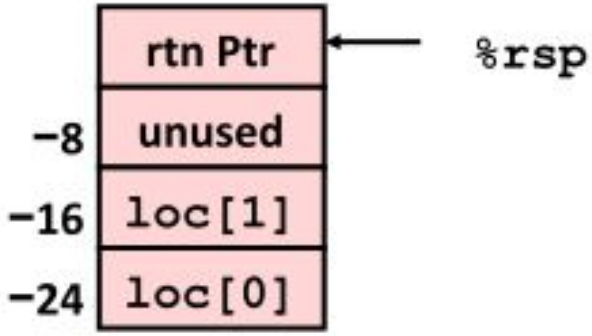
- **Операнды передаются в регистрах**
  - Первый (**xp**) in **%rdi**, второй (**yp**) in **%rsi**
  - Указатели - 64-битные
- **Не нужны стековые операции (кроме ret)**
- **Возможно отсутствие видимого использования стека**
  - Вся локальная информация в регистрах

x86-64: локальные в «красной зоне»

```
/* Обмен в локальном массиве */
void swap_a(long *xp, long *yp) {
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
  movq   (%rdi), %rax
  movq   %rax, -24(%rsp)
  movq   (%rsi), %rax
  movq   %rax, -16(%rsp)
  movq   -16(%rsp), %rax
  movq   %rax, (%rdi)
  movq   -24(%rsp), %rax
  movq   %rax, (%rsi)
  ret
```

- **Исключаются изменения указателя стека**
  - Вся информация может размещаться в небольшом фрагменте за указателем стека



Сводка: процедуры x86-64

- **Интенсивное использование регистров**
  - Передача параметров
  - Больше временных данных, т.к. больше регистров
- **Минимальное использование стека**
  - Иногда вообще никакого
  - Занятие/освобождение целого блока
- **Множество приёмов оптимизации**
  - Каким образом использовать стековый кадр
  - Различные способы размещения

Управление и сложные данные

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

Основные типы данных

- **Целочисленные**
  - Хранятся и обрабатываются в целочисленных регистрах
  - (Без)знаковость опредляется интерпретацией или операциями

Intel	ASM	Bytes	C
byte	`b`	1	`[unsigned] char`
word	`w`	2	`[unsigned] short`
double word	`l`	4	`[unsigned] int`
quad word	`q`	8	`[unsigned] long int` (x86-64)

- **С плавающей точкой**
  - Хранятся и обрабатываются в регистрах с плавающей точкой

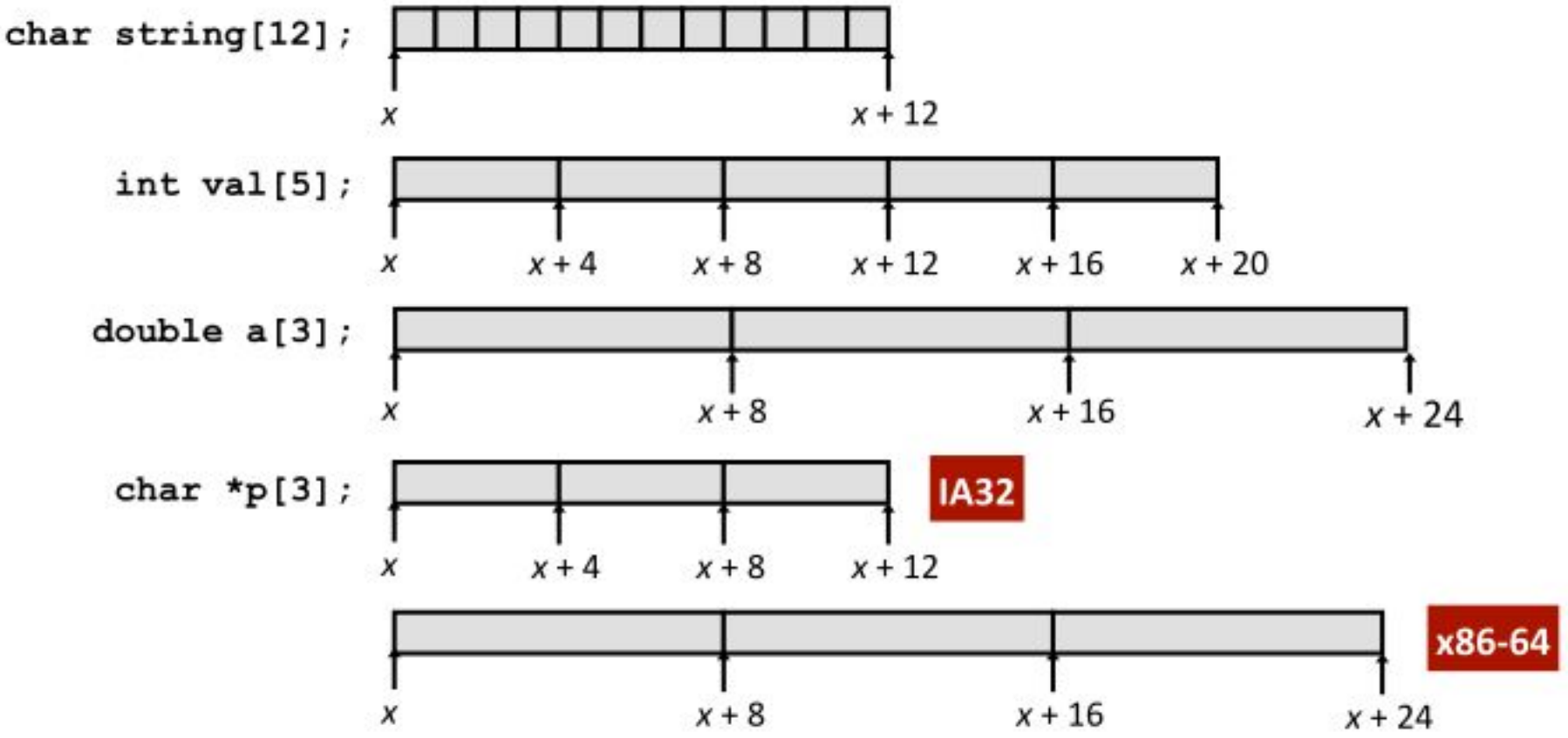
Intel	ASM	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`
Extended	`t`	10/12/16	`long double`

Размещение массивов

- **Основные принципы**

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

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



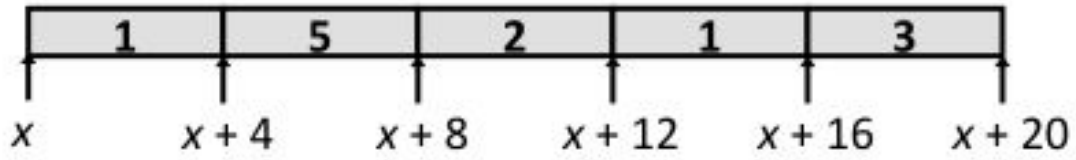
Доступ к массиву

- **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**

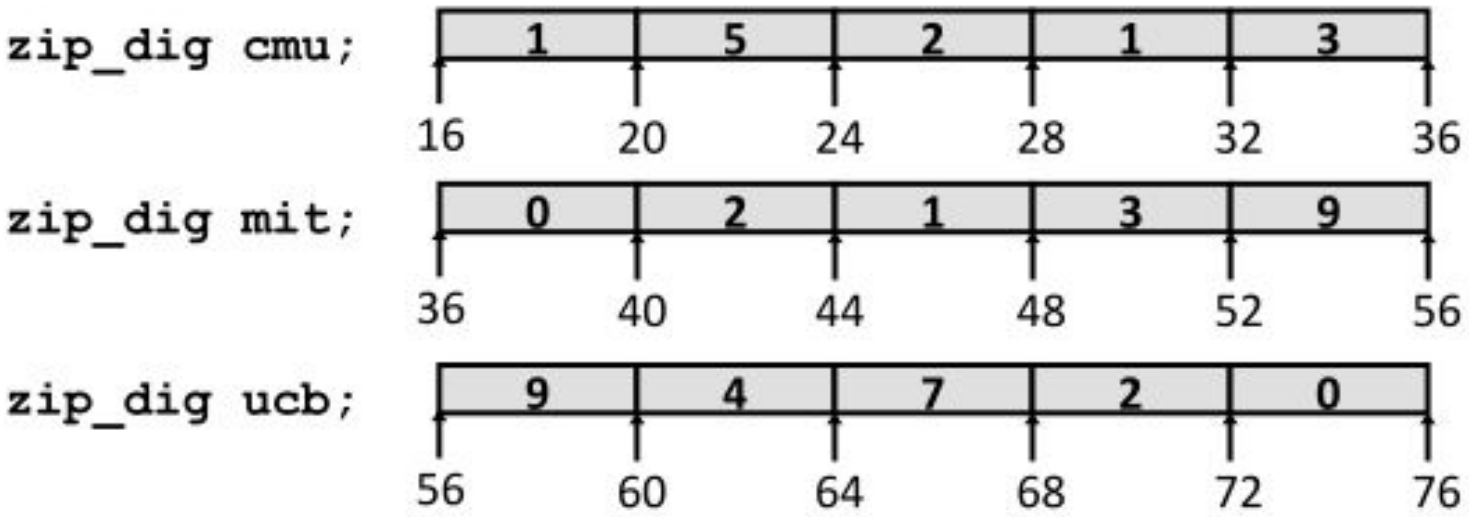
```
int val[5];
```



Обращение	Тип	Значение
`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

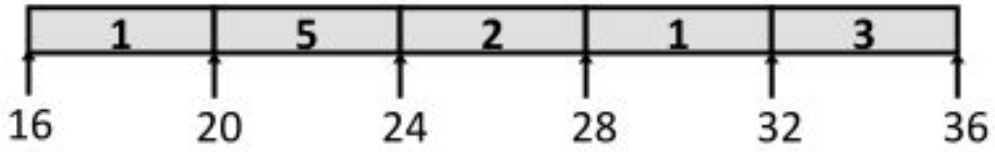
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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 dig)
{
  return z[dig];
}
```

IA32

```
  # %edx = z
  # %eax = dig
   movl (%edx,%eax,4),%eax  # z[dig]
```

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

```
mov eax, [edx+eax*4h]
```

Пример цикла с массивом (IA32)

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

```
  # edx = z
   movl  $0, %eax              #   %eax = i
.L4:                           # loop:
   addl  $1, (%edx,%eax,4)     #   z[i]++
   addl  $1, %eax              #   i++
   cmpl  $5, %eax              #   i:5
   jne   .L4                   #   if !=, goto loop
```

Пример цикла с указателем (IA32)

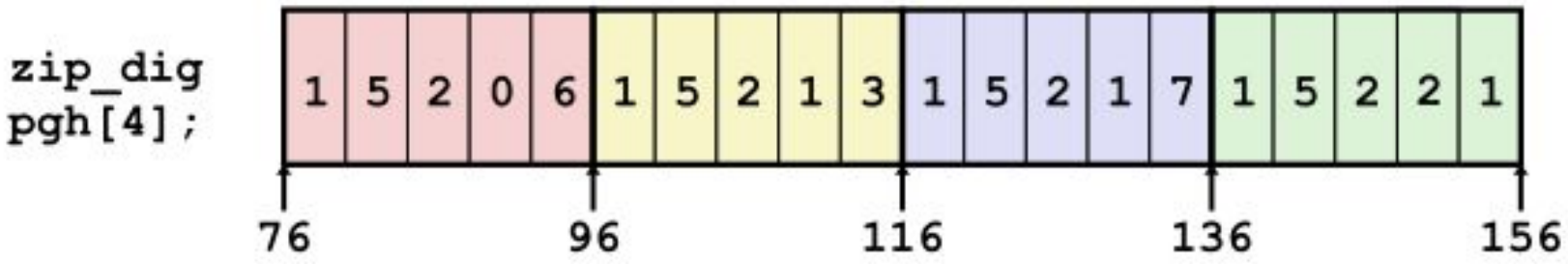
```
void zincr_p(zip_dig z) {
  int *zend = z+ZLEN;
  do {
    (*z)++;
    z++;
  } while (z != zend);
}
```

```
void zincr_v(zip_dig z) {
  void *vz = z;
  int i = 0;
  do {
    (*((int *) (vz+i)))++;
    i += ISIZE;
  } while (i != ISIZE*ZLEN);
}
```

```
  # edx = z = vz
  movl  $0, %eax              #   i = 0
.L8:                          # loop:
  addl  $1, (%edx,%eax)       #   Увеличить vz+i
  addl  $4, %eax              #   i +=  4
  cmpl  $20, %eax             #   Сравнить i:20
  jne   .L8                   #   if !=, goto loop
```

Пример массива массивов

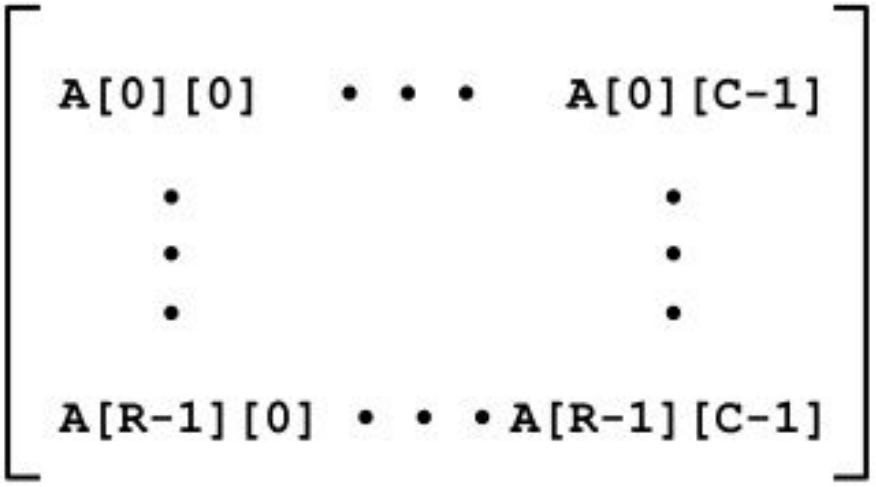
```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



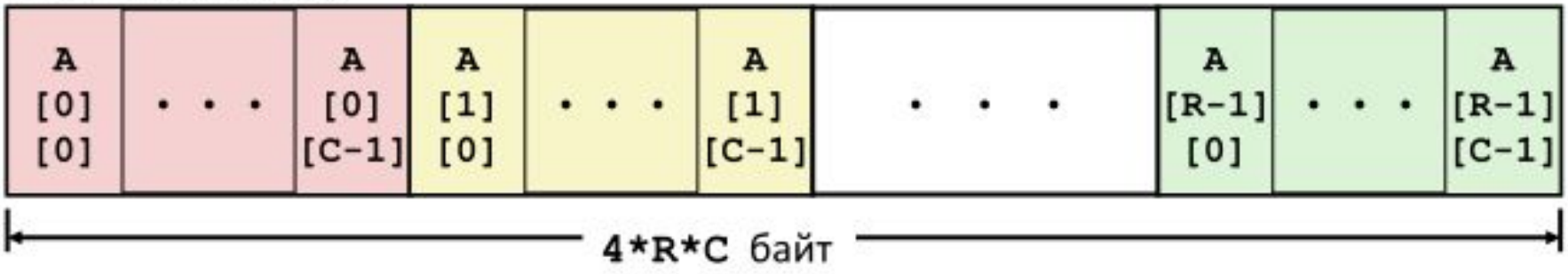
- “**zip_dig pgh[4]**” эквивалентно “**int pgh[4][5]**”
  - Переменная **pgh**: массив из 4 эл-тов, размещённых вплотную
  - Каждый эт-т массив из 5 **int**-ов, размещённых вплотную
- **Гарантируется “построчное” размещение**

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



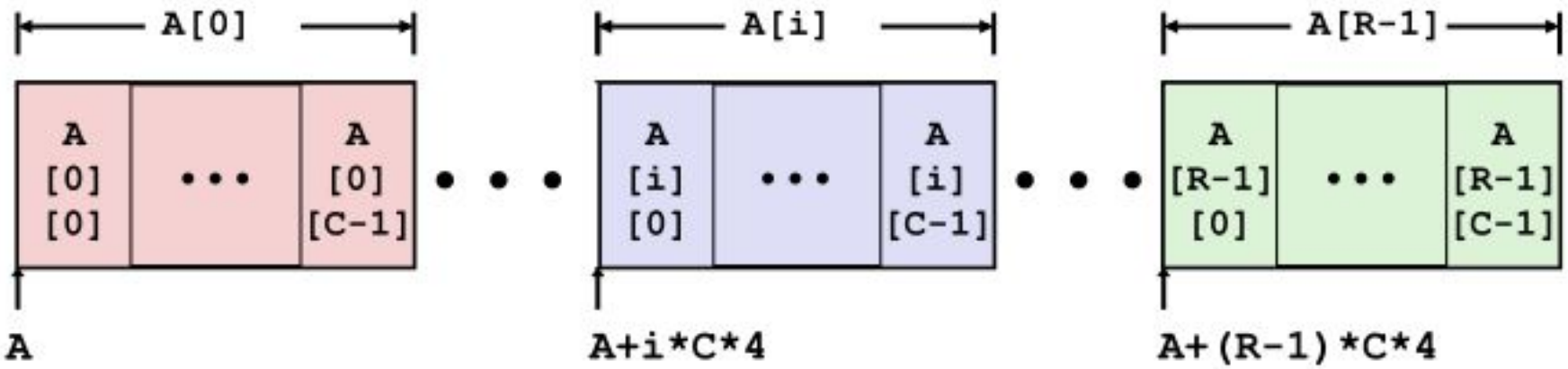

```
int A[R][C];
```



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];
```



Код доступа к строке массива массивов

```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

```
#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 }};
```

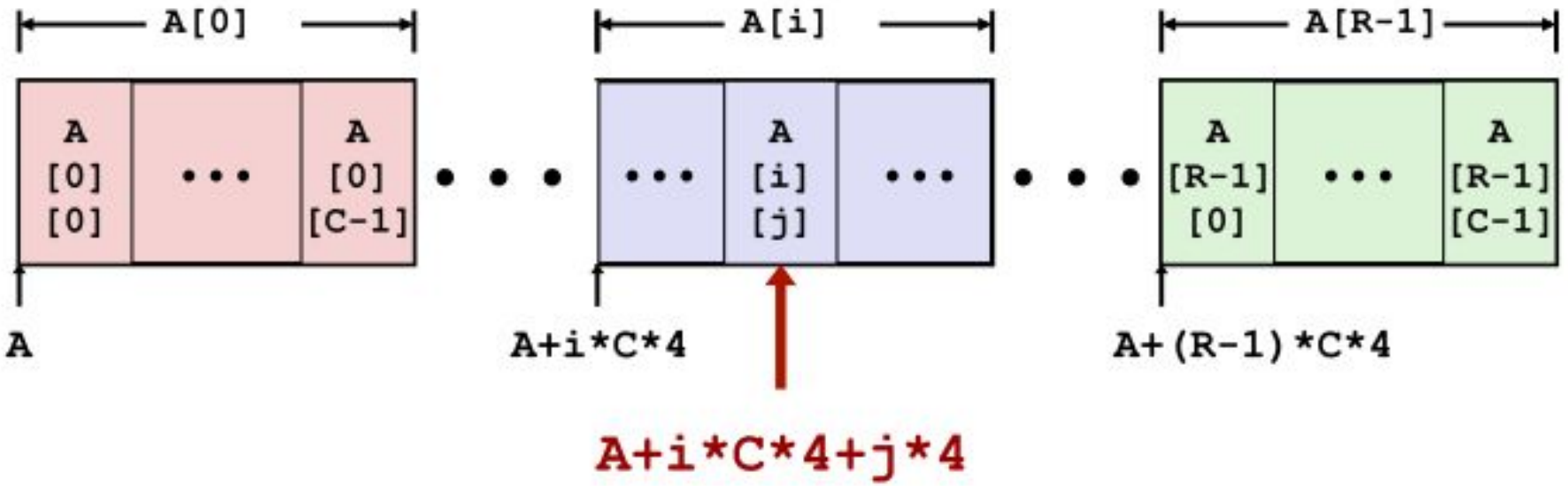
```
  # %eax = index
   leal (%eax,%eax,4),%eax # 5 * index
   leal pgh(,%eax,4),%eax  # pgh + (20 * index)
```

- **Вектор в составе массива**
  - `pgh[index]` массив из 5 `int`-ов
  - начальный адрес `pgh+20*index`
- **Код IA32**
  - Вычисляет и возвращает адреса
  - Вычисляет по формуле `pgh + 4*(index+4*index)`

Код доступа к элементам массива массивов - 1

- **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];
```



Код доступа к элементам массива массивов - 2

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

```
movl   8(%ebp), %eax          # index
leal   (%eax,%eax,4), %eax    # 5*index
addl   12(%ebp), %eax         # 5*index+dig
movl   pgh(,%eax,4), %eax     # offset 4*(5*index+dig)
```

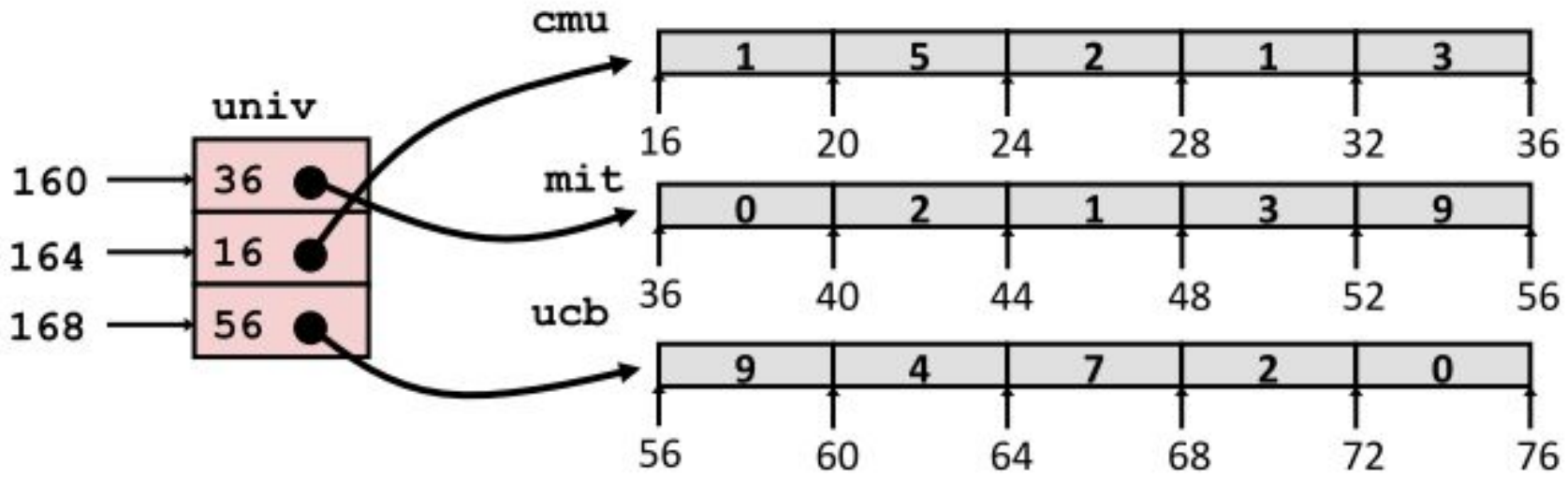
- **Элементы массива**
  - `pgh[index][dig]` имеют тип `int`
  - Адрес: `pgh + 20*index + 4*dig`
    - = `pgh + 4*(5*index + dig)`
- **Код IA32**
  - Вычисляет адрес `pgh + 4*((index+4*index)+dig)`

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
  - 4 байта (IA-32)
- Каждый указывает на массив `int`-ов



Доступ к элементам многоуровневого массива

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

```
   movl   8(%ebp), %eax              # index
   movl   univ(,%eax,4), %edx        # p = univ[index]
   movl   12(%ebp), %eax             # dig
   movl   (%edx,%eax,4), %eax        # p[dig]
```

- **Вычисления (IA32)**
  - Доступ к элементу **`Mem[Mem[univ+4*index]+4*dig]`**
  - Необходимы два обращения к памяти
    - Первое даёт указатель на массив-строку
    - Следующее доступается к элементу в массиве

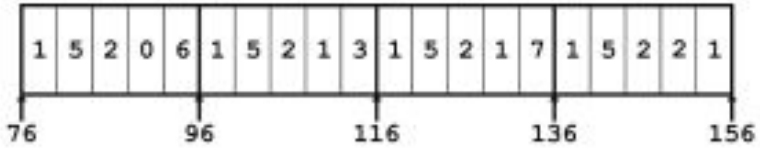
Обращения к элементам массивов

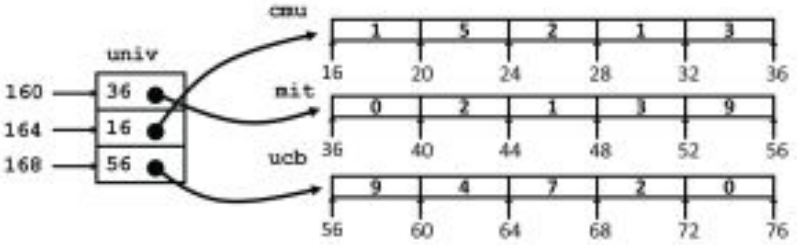
Массив массивов

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

Многоуровневый массив

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





Обращения выглядят одинаково в Си,
но вычисление адресов различается:

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

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

Матричный код 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, int i, int j)
{
  return a[i][j];
}
```

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

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

```
   movl   12(%ebp), %edx     # i
   sall   $6, %edx           # i*64
   movl   16(%ebp), %eax     # j
   sall   $2, %eax           # j*4
   addl   8(%ebp), %eax      # a + j*4
   movl   (%eax,%edx), %eax  # *(a + j*4 + i*64)
```

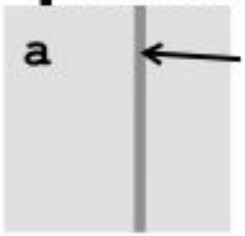
Доступ к матрице n X n

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

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

```
  movl   8(%ebp), %eax      # n
  sall   $2, %eax           # n*4
  movl   %eax, %edx         # n*4
  imull  16(%ebp), %edx     # i*n*4
  movl   20(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   12(%ebp), %eax     # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*n*4)
```

Оптимизация доступа к массиву фиксированного размера - 1




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

- **Вычисления**
  - Проход по всем элементам в столбце j
- **Оптимизация**
  - Следующие обращения к значениям элементов столбца, как размещённым вплотную

```
/* Выдать j-й столбец 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 массиву фиксированного размера - 2

- Вычислить ajp = &a[i][j]
  - Начиная с a + 4*j
  - Увеличивать на 4*N

```
/* Выдать j-й столбец массива */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

Регистр	Значение
%ecx	ajp
%ebx	dest
%edx	i

```
.L8:                                  # loop:
   movl  (%ecx), %eax                 #   Считать *ajp
   movl  %eax, (%ebx,%edx,4)          #   Сохранить в dest[i]
   addl  $1, %edx                     #   i++
   addl  $64, %ecx                    #   ajp += 4*N
   cmpl  $16, %edx                    #   i:N
   jne   .L8                          #   if !=, goto loop
```

Оптимизация доступа к массиву переменного размера

- Вычислить ajp = &a[i][j]
  - Начиная с a + 4*j
  - Увеличивать на 4*n

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

```
/* Выдать j-й столбец 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:
   movl  (%ecx), %eax              #   Считать *ajp
   movl  %eax, (%edi,%edx,4)       #   Сохранить в dest[i]
   addl  $1, %edx                  #   i++
   addl  $ebx, %ecx                #   ajp += 4*n
   cmpl  $edx, %esi                #   n:i
   jg    .L18                      #   if >, goto loop
```

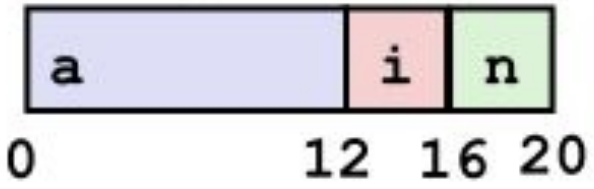
Ещё машинный уровень
Управление и сложные данные

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

Размещение структуры

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

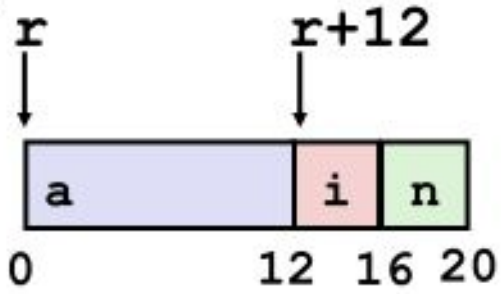
Распределение памяти



- **Принцип**
  - Вплотную занимаемый фрагмент памяти
  - Обращение к элементам по имени
  - Элементы могут быть различных типов

Доступ к структуре

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



- **Доступ к элементу структуры**
  - Указатель настроен на первый байт структуры
  - Элементы различаются сдвигом

```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

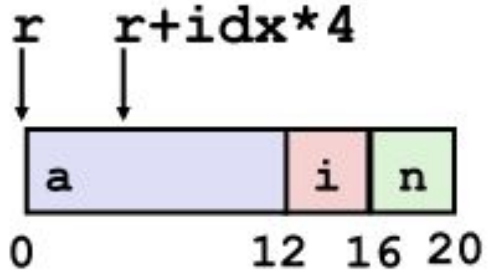
Ассемблер IA32

```
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```

```
mov  [eax+12], edx
```

Взятие указателя на элемент структуры

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



- **Взятие указателя на элемент массива**
  - Сдвиг к каждому элементу структуры определяется при компиляции
  - Аргументы
    - Mem[%ebp+8]: **r**
    - Mem[%ebp+12]: **idx**

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

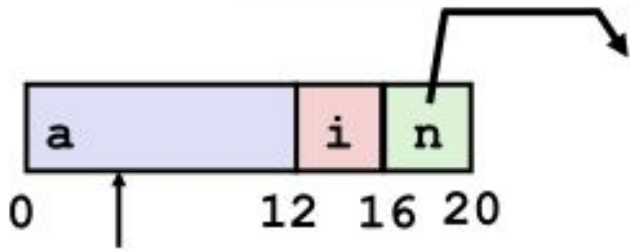
```
movl   12(%ebp), %eax   # Считать idx
sall   $2, %eax         # idx*4
addl   8(%ebp), %eax    # r+idx*4
```

Проход связного списка

- Код Си

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

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

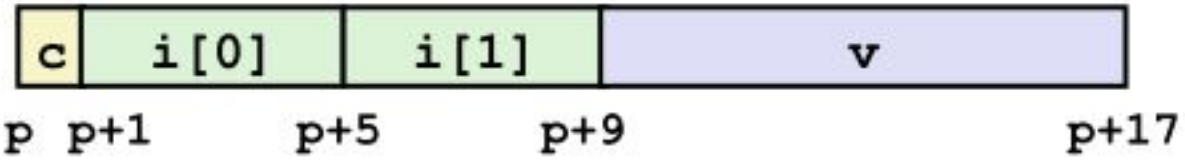


Регистр	Значение
%edx	r
%ecx	val

```
.L17:                           # loop:
  movl   12(%edx), %eax         # r->i
  movl   %ecx, (%edx,%eax,4)    # r->a[i] = val
  movl   16(%edx), %edx         # r = r->n
  testl  %edx, %edx             # Test r
  jne    .L17                   # If != 0 goto loop
```

Структуры и выравнивание

- **Невыровненные данные**




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

- **Выровненные данные**
  - Если простой тип данных - длиной K байт
  - То адреса должны быть кратны K

Принципы выравнивания

- **Выровненные данные**
  - Если простой тип данных - длиной K байт,...
  - ...то адрес должен быть кратен K
  - Обязательно на некоторых машинах, рекомендовано на IA32
    - Различается для IA32 Linux, x86-64 Linux, и Windows!
- **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

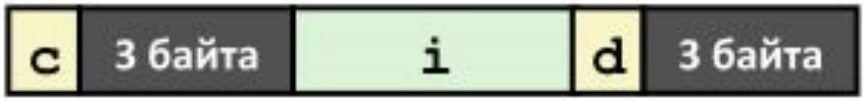
Экономия пространства

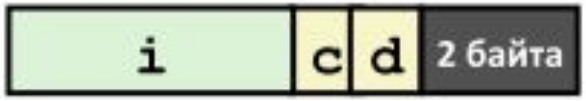
- Поместим вначале длинные данные

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

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

- Результат (K=4)

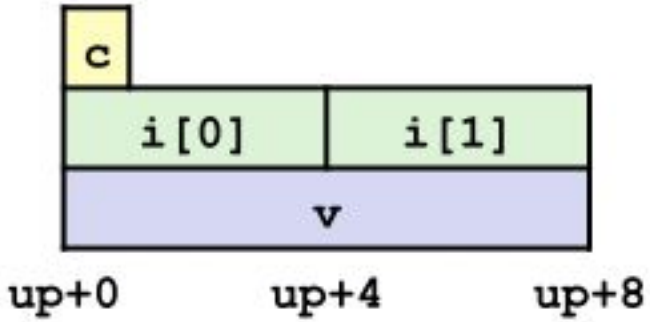




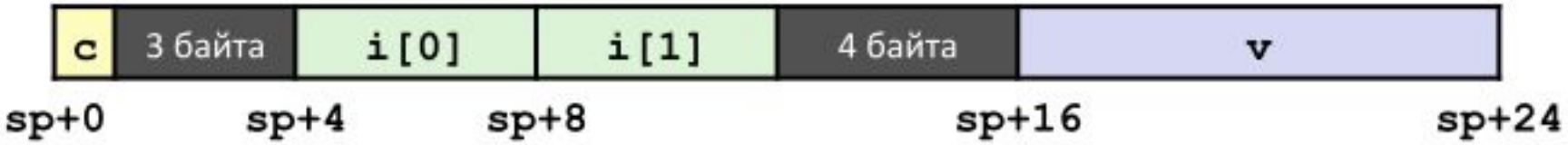
Размещение объединений

- Размещается как наибольший элемент

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

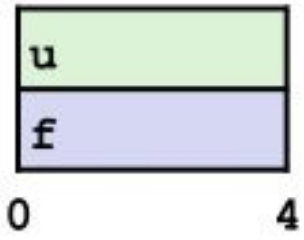



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



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;
}
```

Совпадает с `(float)u`?

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

Совпадает с `(unsigned)f`?

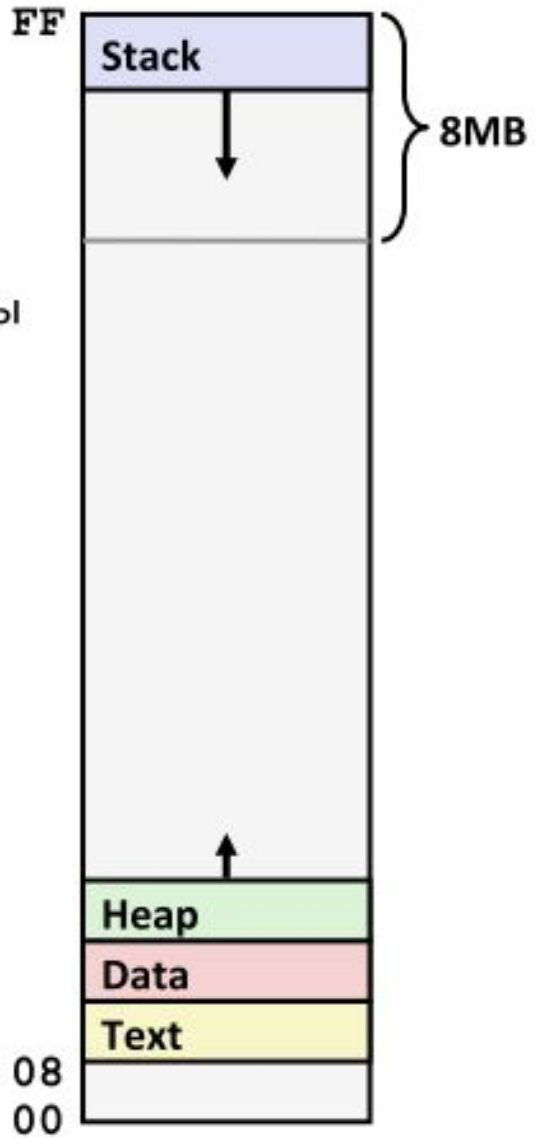
Сводка

- **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

Распределение IA32 Linux

- **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 (код)**
  - Исполняемые машинные инструкции
  - Только чтение

2 старших 16-ричных цифры
= 8 старших бит адреса



Переполнение буфера

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);
}
```

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

```
unix>./bufdemo
Type a string:1234567
1234567
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

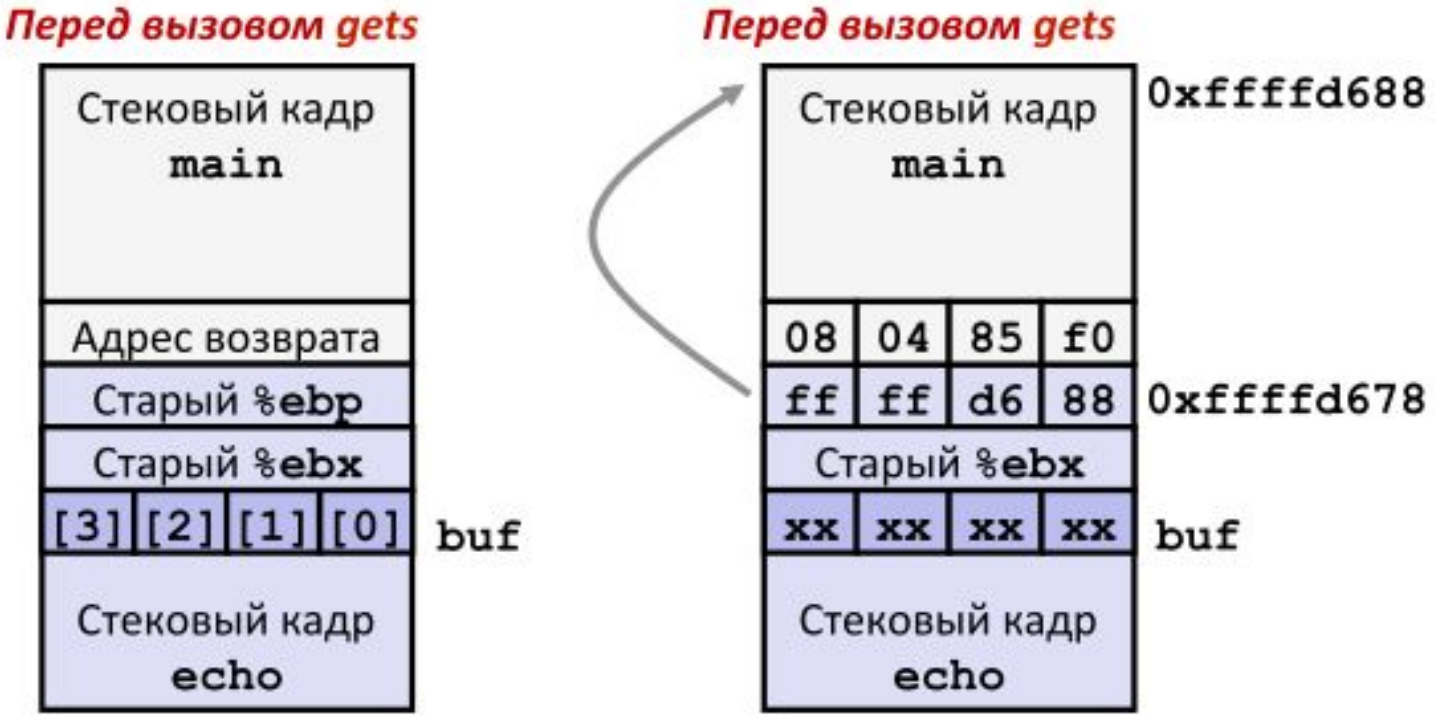
Стек при переполнении буфера

Перед вызовом gets


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

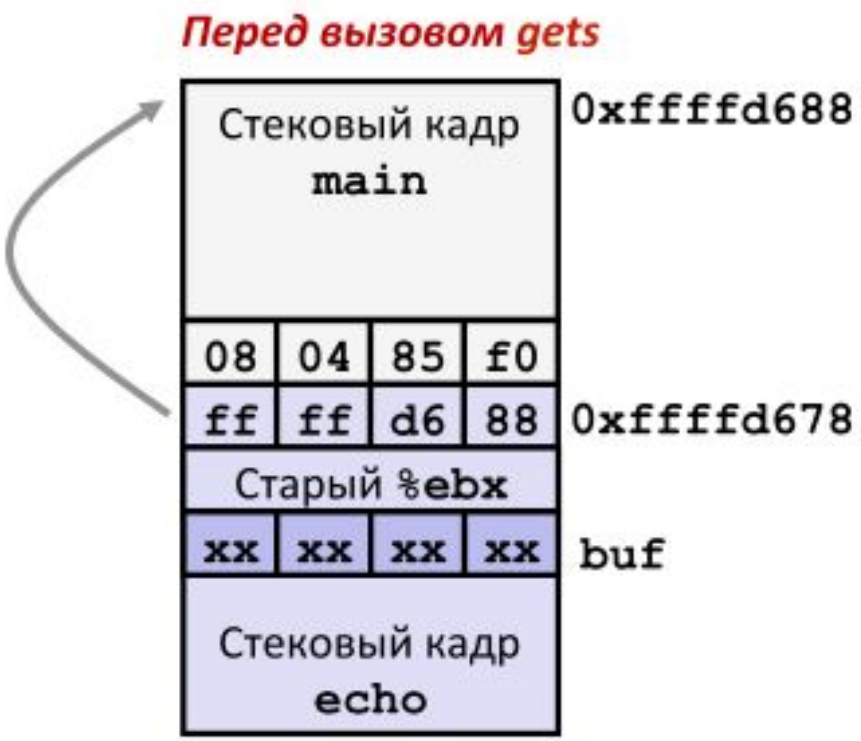
```
echo:
    pushl %ebp              # Сохранить в стек %ebp
    movl  %esp, %ebp
    pushl %ebx              # Сохранить %ebx
    subl  $20, %esp         # Занять место в стеке
    leal  -8(%ebp),%ebx     # Адрес buf = %ebp-8
    movl  %ebx, (%esp)      # Адрес buf в stack
    call  gets              # Вызвать gets
    . . .
```

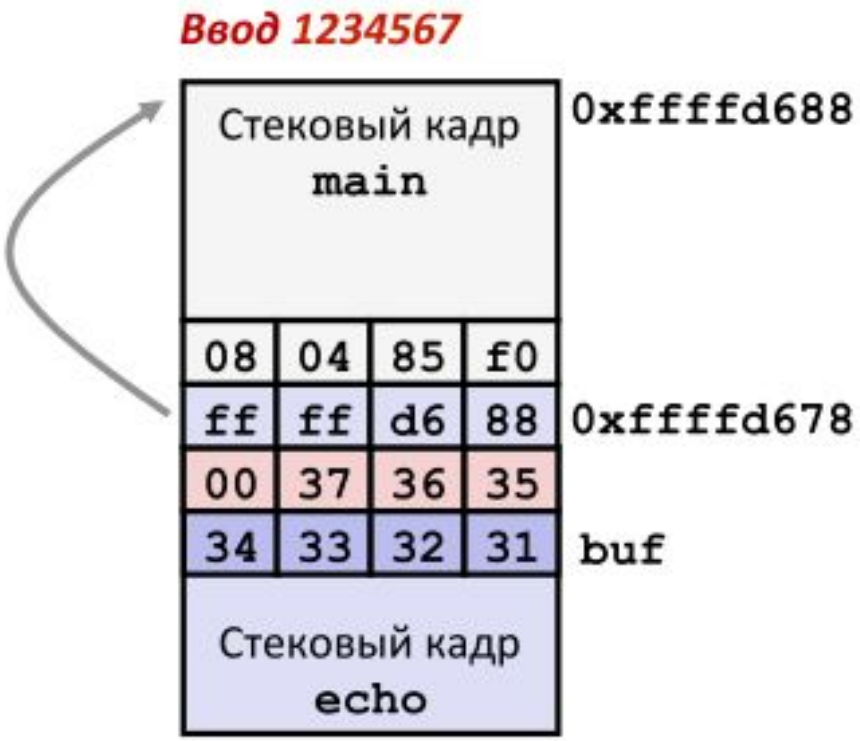
Стек при переполнении буфера




```
80485eb:  e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:  c9                leave
```

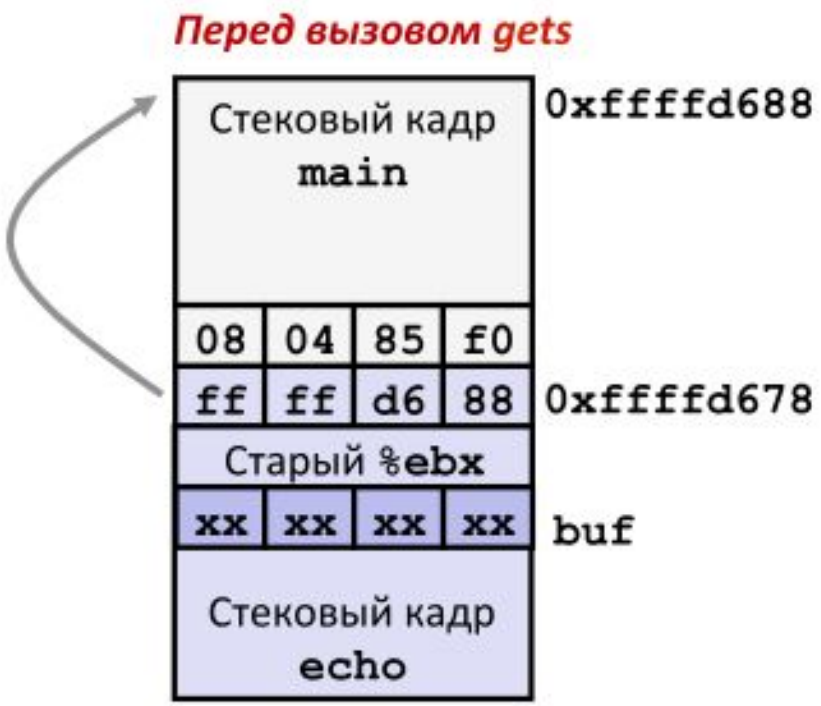
Переполнение буфера. Пример №1





buf переполнен, %ebx испорчен,
но проблема ещё не проявилась

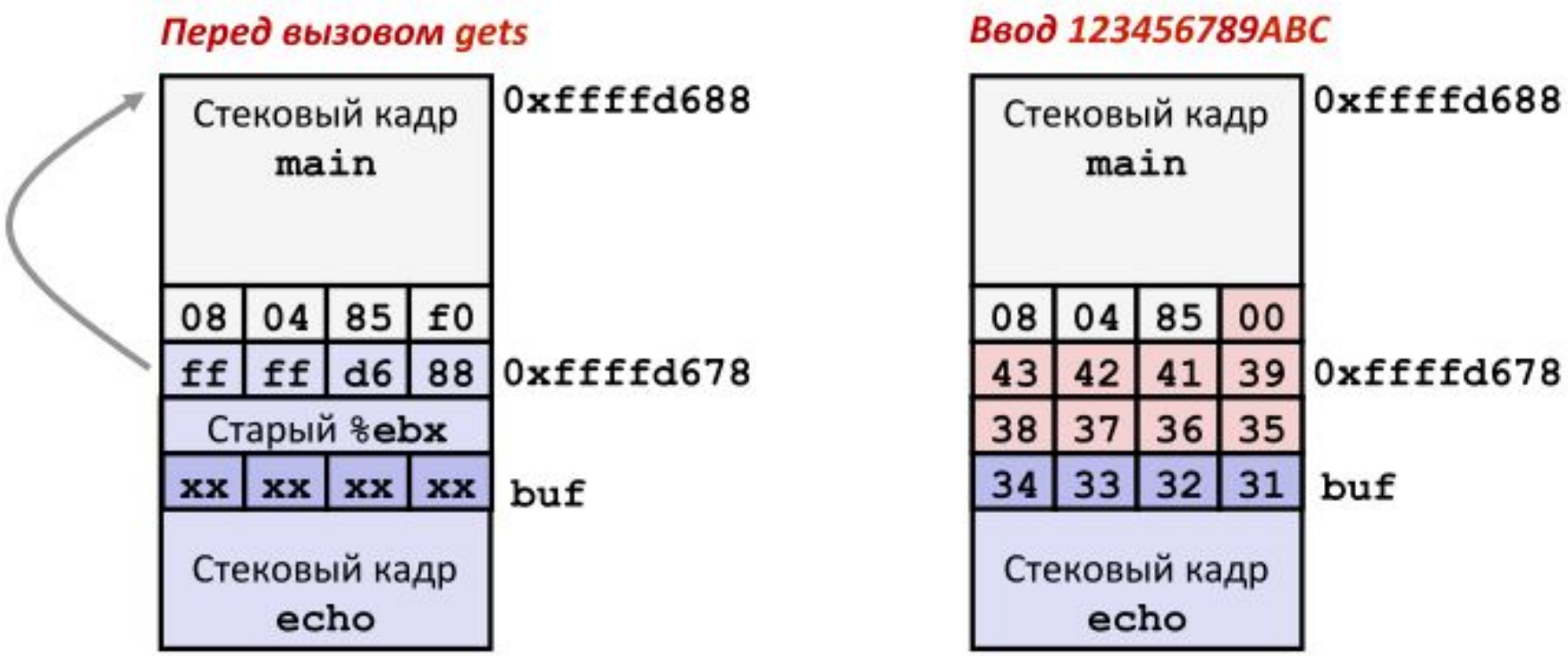
Переполнение буфера. Пример №2



Указатель кадра испорчен!

```
. . .
80485eb:  e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:  c9                leave    # Испортить %ebp !
80485f1:  c3                ret
```

Переполнение буфера. Пример №3



Адрес возврата испорчен!

```
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave    # Желаемая точка возврата
```

Злонамеренное использование переполнения буфера

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

Адрес возврата A

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

- 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 ограниченим 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 выделяйте случайное пространство в стеке
  - Затрудняет хакеру предсказание начала вставляемого кода

Стековый индикатор

- **Идея**
  - Разместить в стеке сразу за буфером специальное значение (“индикатор”)
  - Проверять целостность перед выходом из функции

Установка индикатора

Перед вызовом gets


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

```
echo:
    . . .
    movl    %gs:20, %eax    # Значение для индикатора
    movl    %eax, -8(%ebp)  # Поместить в стек
    xorl    %eax, %eax      # Очистить регистр
    . . .
```

Проверка индикатора

Перед вызовом gets


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

```
echo:
    . . .
    movl    -8(%ebp), %eax    # Взять из стека
    xorl    %gs:20, %eax      # Сравнить с исходным
    je      .L24              # Совпало? Проходим
    call    __stack_chk_fail  # ОШИБКА!
.L24:
    . . .
```