Лекция 9

12 марта

```
int fib(int x) { // x >= 1
  int i;
  int predpred = 0;
  int pred = 1;
  int res = 1;
  x--;
  for (i = 0; i < x; i++) {
     res = predpred + pred;
     predpred = pred;
     pred = res;
  }
  return res;
}
```

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

   mov   ecx, dword [ebp + 8] ; x
   xor   edx, edx             ; predpred
   mov   ebx, 1               ; pred
   mov   eax, 1               ; res
   dec   ecx

   jecxz .end
.loop:
   lea   eax, [edx + ebx]
   mov   edx, ebx
   mov   ebx, eax
   loop .loop

.end:
   pop ebx
   pop ebp
   ret
```

```
int fib(int x) { // x >= 1
  int i;
  int predpred = 0;
  int pred = 1;
  int res = 1;
  x--;
  for (i = 0; i < x; i++) {
     res = predpred + pred;
     predpred = pred;
     pred = res;
  }
  return res;
}
```

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

   mov   ecx, dword [ebp + 8] ; x
   xor   edx, edx             ; predpred
   mov   ebx, 1               ; pred
   mov   eax, 1               ; res
   dec   ecx

   jecxz .end
.loop:
   lea   eax, [edx + ebx]
   mov   edx, ebx
   mov   ebx, eax
   loop .loop

.end:
   pop ebx
   pop ebp
   ret
```

```
int fib(int x) { // x >= 1
  int i;
  int predpred = 0;
  int pred = 1;
  int res = 1;
  x--;
  for (i = 0; i < x; i++) {
     res = predpred + pred;
     predpred = pred;
     pred = res;
  }
  return res;
}
```

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

   mov   ecx, dword [ebp + 8] ; x
   xor   edx, edx             ; predpred
   mov   ebx, 1               ; pred
   mov   eax, 1               ; res
   dec   ecx

   jecxz .end
.loop:
   lea   eax, [edx + ebx]
   mov   edx, ebx
   mov   ebx, eax
   loop .loop

.end:
   pop ebx
   pop ebp
   ret
```

```
int fib(int x) { // x >= 1
  int i;
  int predpred = 0;
  int pred = 1;
  int res = 1;
  x--;
  for (i = 0; i < x; i++) {
     res = predpred + pred;
     predpred = pred;
     pred = res;
  }
  return res;
}
```

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

   mov   ecx, dword [ebp + 8] ; x
   xor   edx, edx             ; predpred
   mov   ebx, 1               ; pred
   mov   eax, 1               ; res
   dec   ecx

   jecxz .end
.loop:
   lea   eax, [edx + ebx]
   mov   edx, ebx
   mov   ebx, eax
   loop .loop

.end:
   pop ebx
   pop ebp
   ret
```

- Целые числа
  - Размещаются и обрабатываются в регистрах общего назначения
  - Знаковые/беззнаковые числа

Intel	ASM	Bytes	C
byte	**b**	1	**[unsigned] char**
word	**w**	2	**[unsigned] short**
double word	**d**	4	**[unsigned] int**
quad word	**q**	8	**[unsigned] long long int**

- Числа с плавающей точкой
  - Размещаются и обрабатываются в специализированных регистрах для чисел с плавающей точкой

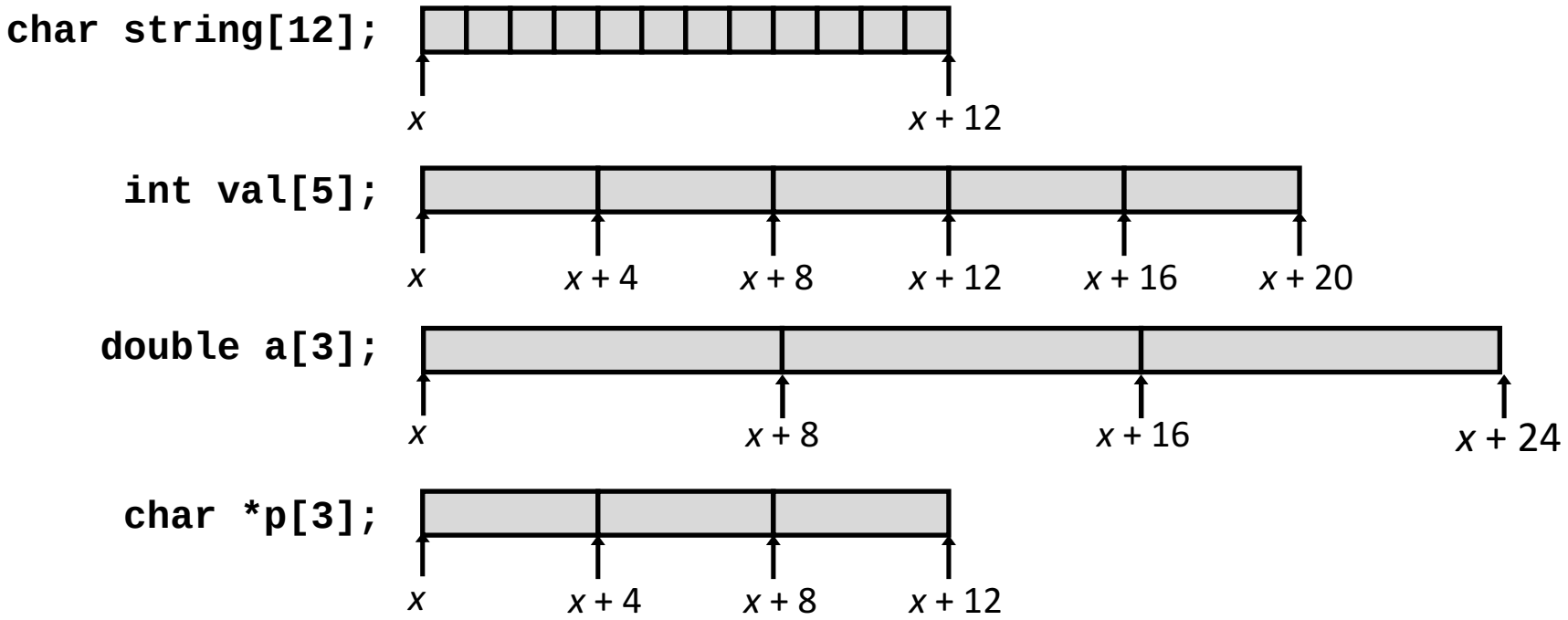
Intel	ASM	Bytes	C
Single	**d**	4	**float**
Double	**q**	8	**double**

- Массивы – размещение в памяти

  `T A[L];`

  – Массив элементов типа T, размер массива – L

  – Массив располагается в непрерывном блоке памяти размером *L* * **`sizeof`**(*T*) байт

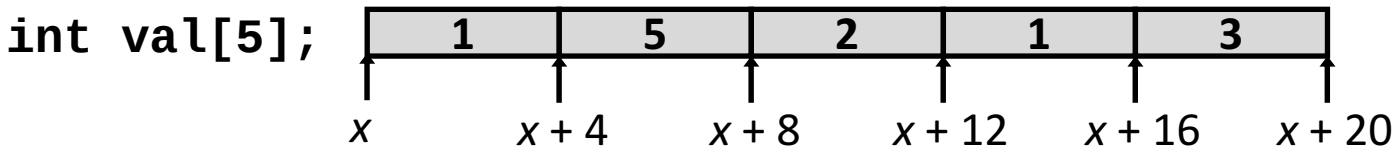


- Доступ к элементам массива

  `T A[L];`

  – Массив элементов типа T, размер массива – L

  – Идентификатор A может использоваться как указатель на элемент массива с индексом 0. Тип указателя – T*

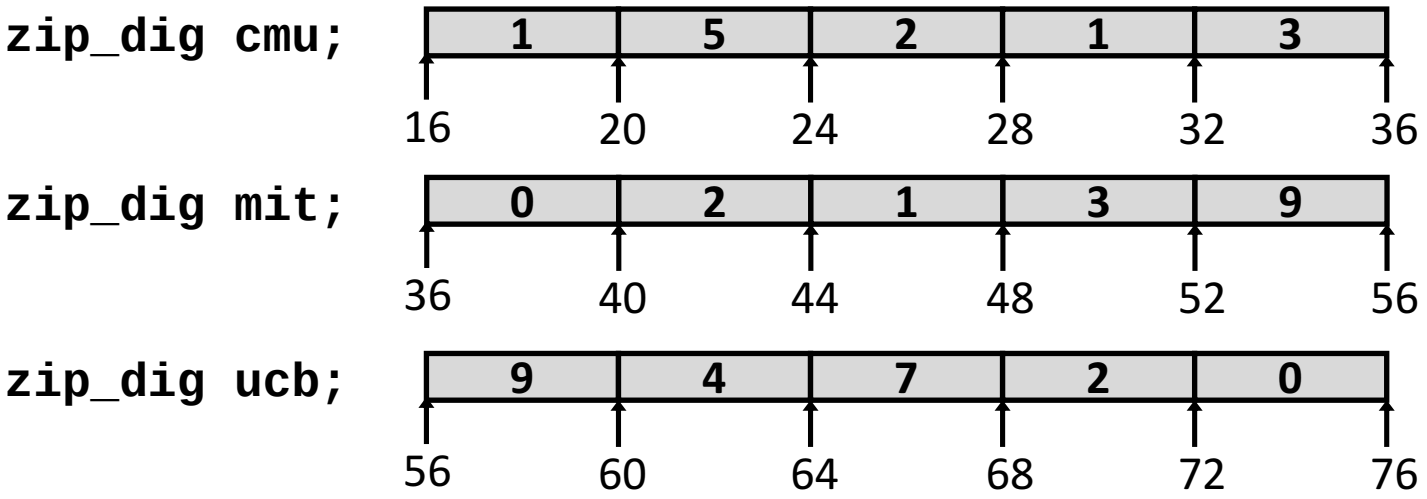


- Ссылка — Тип — Значение

Ссылка	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*

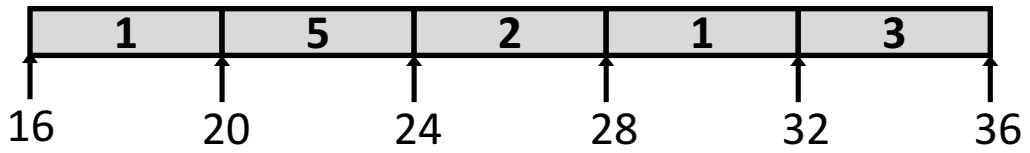
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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 переменной “`zip_dig cmu`” эквивалентно “`int cmu[5]`”
- Массивы были размещены в последовательно идущих блоках памяти размером 20 байт каждый
  - В общем случае не гарантируется, что массивы будут размещены непрерывно

```
zip_dig cmu;
```




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

```
; edx = z
; eax = dig
mov eax, dword [edx+4*eax] # z[dig]
```

- Регистр `edx` содержит начальный (базовый) адрес массива
- Регистр `eax` содержит индекс элемента в массиве
- Адрес элемента `edx + 4 * eax`

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

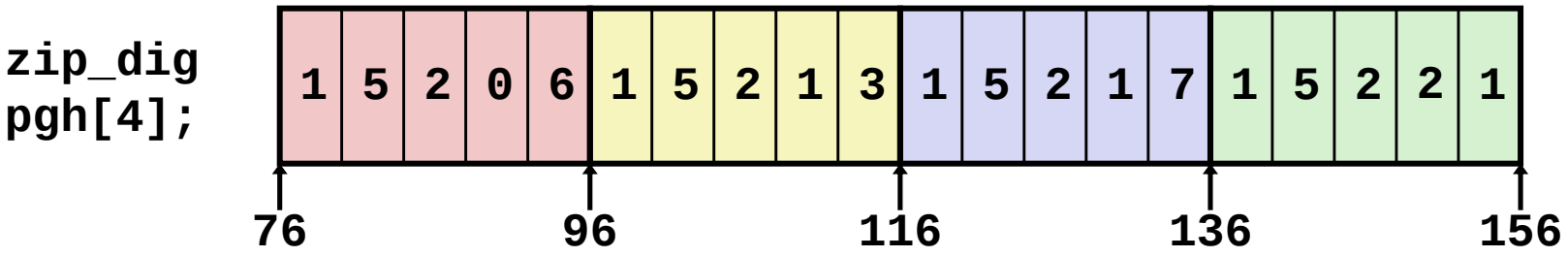
```
                                ;   edx = z
   mov eax, 0                   ;   eax = i
.L4:                            ; loop:
   add dword [edx + 4 * eax], 1 ;   z[i]++
   add eax, 1                   ;   i++
   cmp eax, 5                   ;   i vs. 5
   jne .L4                      ;   if (!=) goto loop
```

```
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);
}
```

```
      movl eax, 0                  ;   edx = z = vz
                                   ;   i = 0
.L8:                               ; loop:
      add dword [edx + eax], 1     ;   Increment vz+i
      add eax, 4                   ;   i +=  4
      cmp eax, 20                  ;   i vs. 20
      jne .L8                      ;   if (!=) goto loop
```

```
#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 элементов, расположенных непрерывно в памяти
  - Каждый элемент – массив из 5 **`int`**'ов, расположенных непрерывно в памяти
- Всегда развертывание по строкам (Row-Major)

- Объявление

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

  – 2D массив элементов типа *T*

  – *R* строк, *C* столбцов

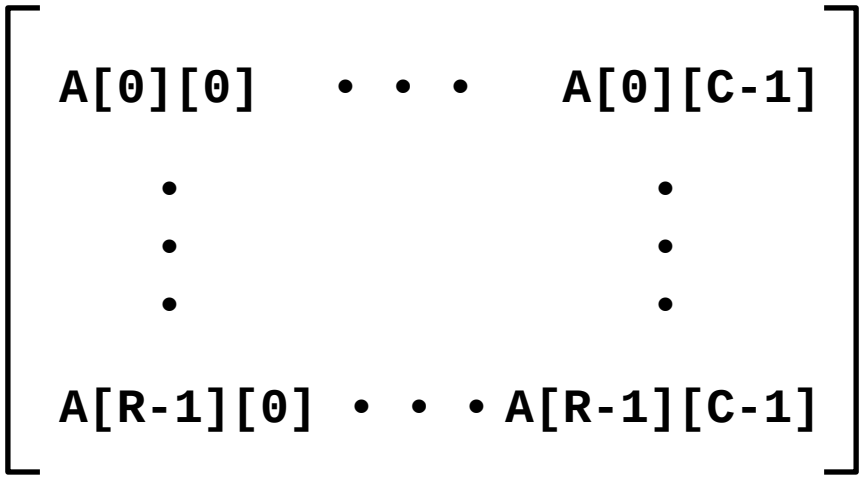
  – Размер типа *T* – *K* байт

- Размер массива

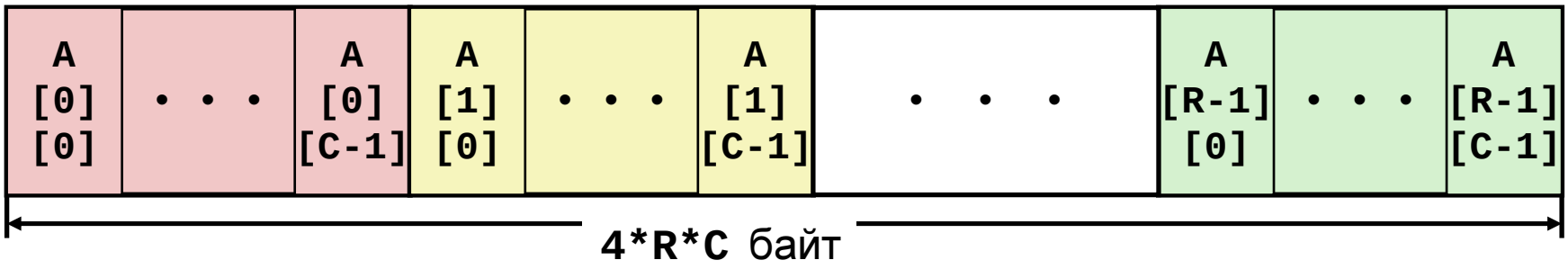
  – R * C * K байт

- Размещение в памяти

  – Развертывание по строкам

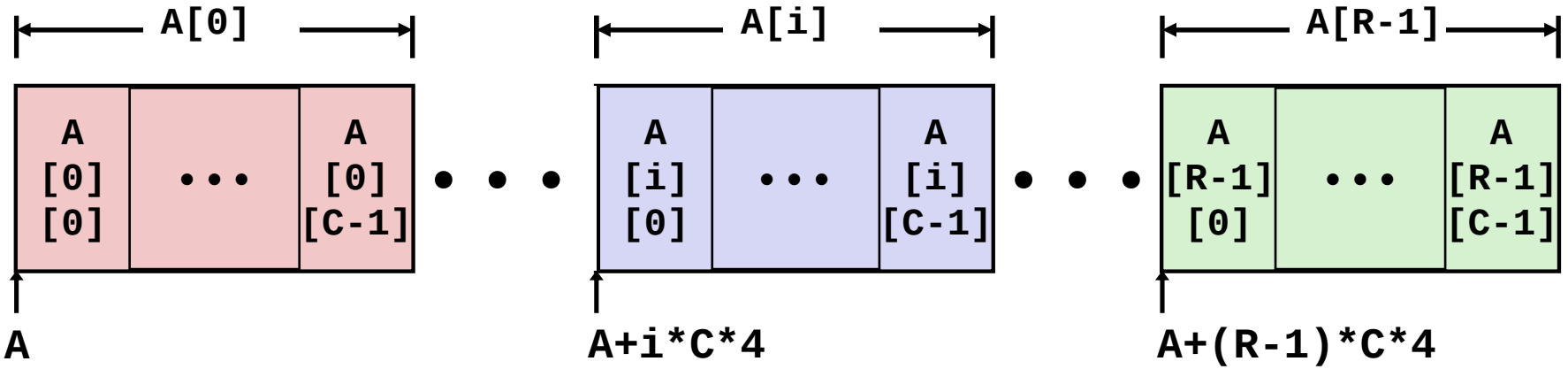



```
int A[R][C];
```



- Доступ к строкам
  - **`A[i]`** массив из *C* элементов
  - Каждый элемент типа *T* требует *K* байт
  - Начальный адрес строки с индексом i
    `A +` *i* * (*C* * *K*)

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



```
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 }};
```

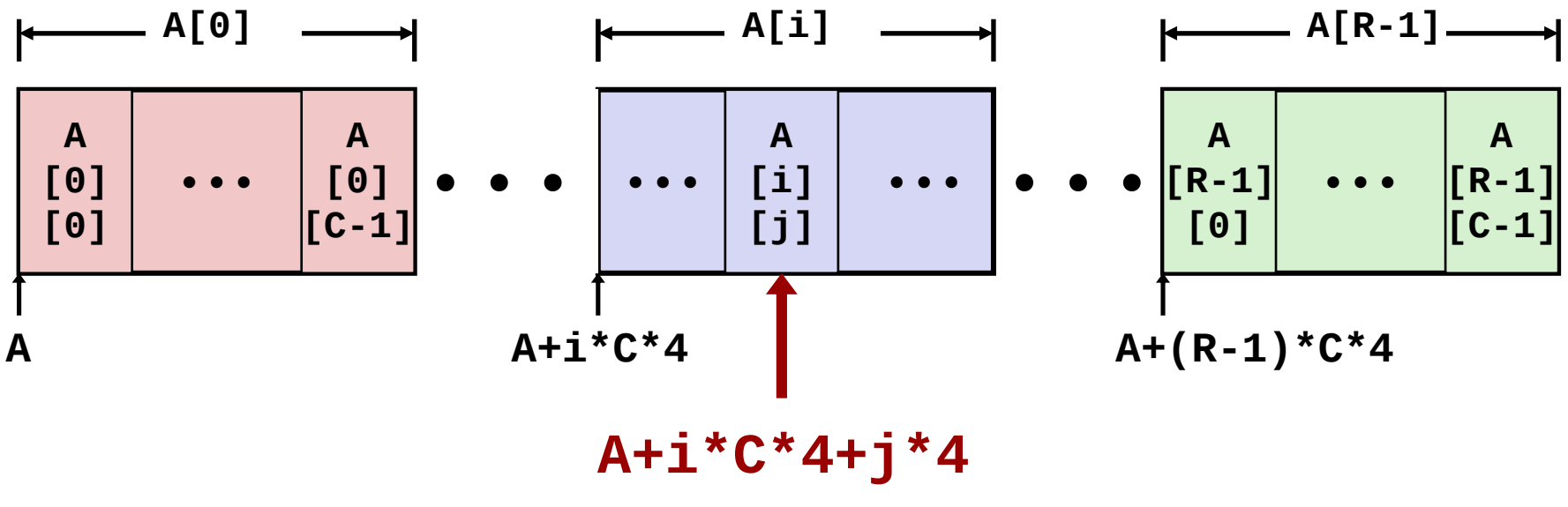
```
; eax = index
 lea eax, [eax + 4 * eax] ; 5 * index
 lea eax, [pgh + 4 * eax] ; pgh + (20 * index)
```

- **`pgh[index]`** массив из 5 **`int`**'s
- Начальный адрес **`pgh+20*index`**

- Вычисляется и возвращается адрес
- Вычисление адреса в виде **`pgh + 4*(index+4*index)`**

- Элементы массива
  - **A[i][j]** элемент типа *T*, который требует *K* байт
  - Адрес элемента **A** + *i* * (*C* * *K*) + *j* * *K* = *A* + (*i* * *C* + *j*)* *K*

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



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

```
mov   eax, dword [ebp + 8]       ; index
lea   eax, [eax + 4 * eax]       ; 5*index
add   eax, dword [ebp + 12]      ; 5*index+dig
mov   eax, dword [pgh + 4 * eax] ; смещение 4*(5*index+dig)
```

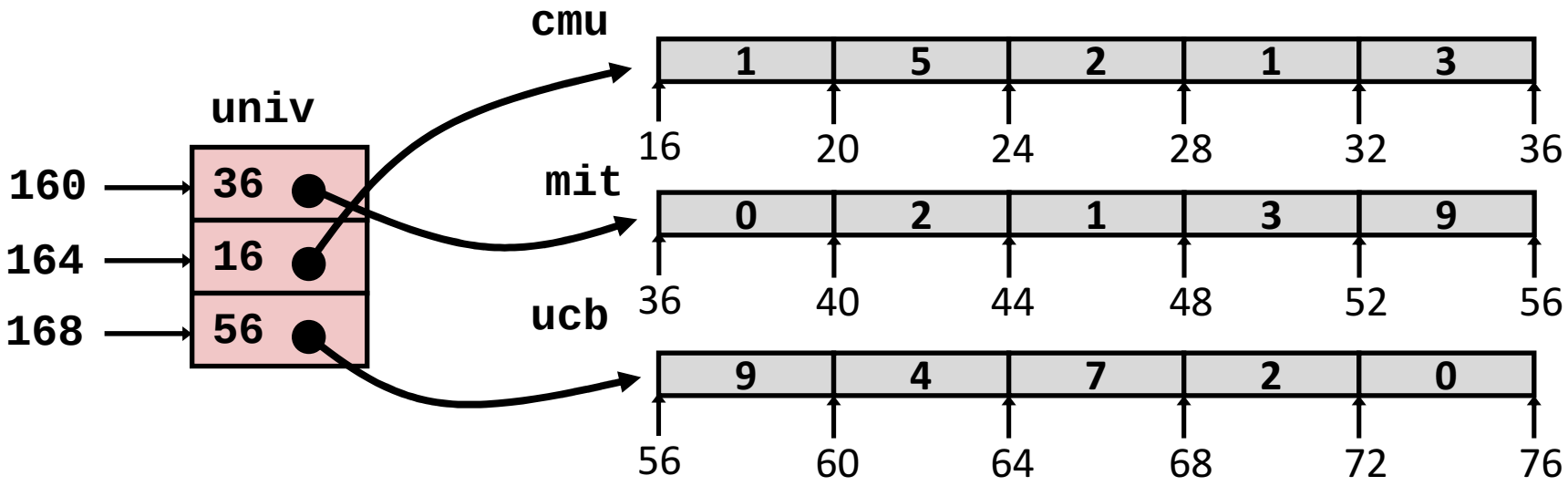
- **pgh[index][dig] – тип int**
- Адрес: **pgh + 20*index + 4*dig =**
  = pgh + 4*(5*index + dig)

- Вычисление адреса производится как
  pgh + 4*((index+4*index)+dig)

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

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

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



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

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

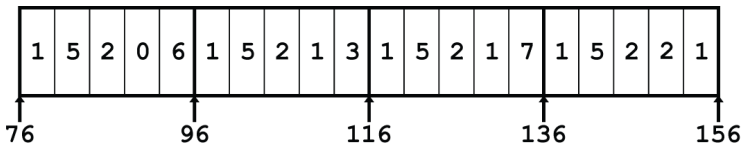
- Доступ к элементу **Mem[Mem[univ+4*index]+4*dig]**
- Необходимо выполнить два чтения из памяти
  - Первое чтение получает указатель на одномерный массив
  - Затем второе чтение выполняет выборку требуемого элемента этого одномерного массива

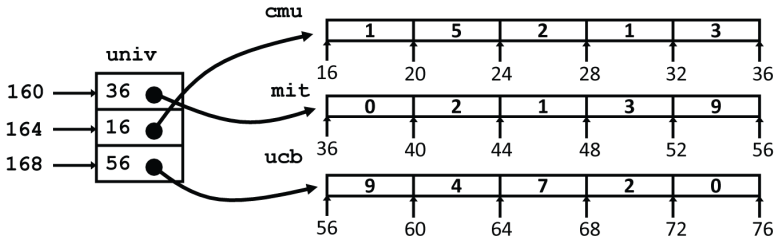
Многомерный массив

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

Многоуровневый массив

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





- Значительное внешнее сходство в Си
- Существенное различие в ассемблере

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

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

Матрица N X N

- Фиксированные размерности
  - Значение N известно во время компиляции

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

- Динамически задаваемая размерность. Требуется явное преобразование индексов
  - Традиционный способ реализации динамических массивов

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

- Динамически задаваемая размерность с неявной индексацией.
  - Поддерживается последними версиями gcc

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

Матрица 16 X 16

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

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

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

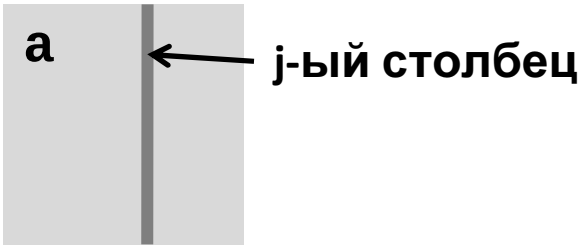
Матрица n X n

- **Доступ к элементу матрицы**
  - Адрес **A** + *i* * (*C* * *K*) + *j* * *K*
  - C = n, K = 4

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

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

Оптимизация доступа к элементам массива



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

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

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

Оптимизация доступа к элементам массива

- Оптимизация
  - Вычисляем ajp = &a[i][j]
    - Начальное значение a + 4*j
    - Шаг 4*N

Регистр	Значение
ecx	ajp
ebx	dest
edx	i

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

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

Оптимизация доступа к элементам массива

– Вычисляем ajp = &a[i][j]
  - Начальное значение a + 4*j
  - Шаг 4*n

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

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

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

Оптимизация доступа к элементам массива

– Изменение направления прохода по циклу
  - Выход из цикла по нулевому счетчику
  - Шаг отрицательный
  - Меняются начальные значения указателей
  - Достаточно вывести к нулю один из индексов

```
/* Выборка столбца j из массива */
void var_column
  (int n, int a[n][n],
   int j, int *dest)
{
  int i;
  for (i = n-1; i >=0; i--)
    dest[i] = a[i][j];
}
```

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

Оптимизация доступа к элементам массива

Регистр	Начальное значение
ecx	a+4*n*(n-1)+4*j
edi	dest – 4
edx	n
ebx	4*n
esi	освободился

```
/* Retrieve column j from array */
void var_column
  (int n, int a[n][n],
   int j, int *dest)
{
  int i;
  dest--;
  for (i = n; i != 0; i--)
    dest[i] = a[i-1][j];
}
```

Машинно-зависимая оптимизация

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

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

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

```
  ; начало пропущено
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
```