Estrutura do tema ISA do IA-32

1. Desenvolvimento de programas no IA-32 em Linux
2. Acesso a operandos e operações
3. Suporte a estruturas de controlo
4. Suporte à invocação/regresso de funções
5. Análise comparativa: IA-32 *vs* x86-64 *vs* RISC
6. Acesso e manipulação de dados estruturados

Arrays: organização em memória

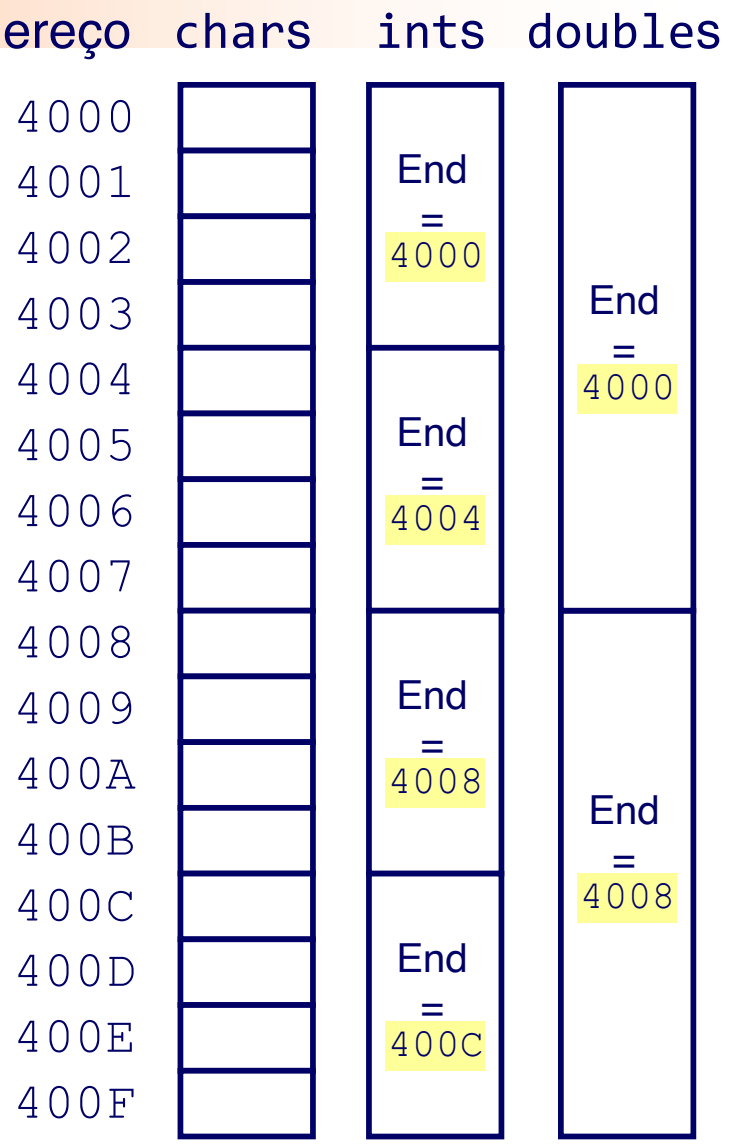
Posições de memória não têm tipo

- os tipos são definidos pela forma como as instruções máquina usam essas posições

```
movb   (%ebp), %al    # char
movl   (%ebp), %eax   # int
movsd  (%ebp), %xmm0  # double
```

Endereços de memória indicam a localização de bytes

- Os *arrays* são identificados pelo endereço do seu primeiro *byte*
- Em geral os elementos estão armazenados em posições de mémoria contiguas
  - Endereço de elementos sucessivos é determinado pelo tamanho do elemento



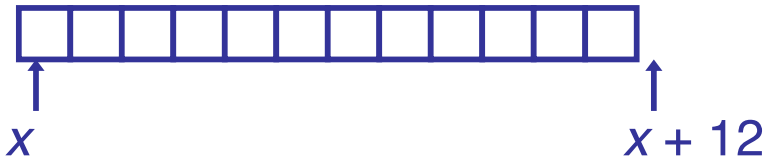
Arrays: alocação em memória

Declaração em C:

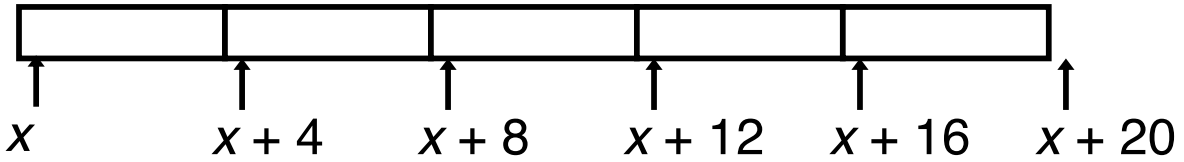
data_type **Array_name**[*length*];

Reserva um bloco de memória com tamanho *length* * sizeof(*data_type*) *bytes*

char string[12];
sizeof(char) = 1

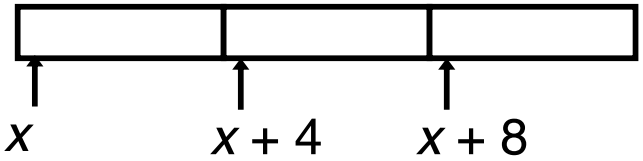


int val[5];
sizeof(int) = 4



double a[3];
sizeof(double)=8

char *p[3];
sizeof(char *)=?

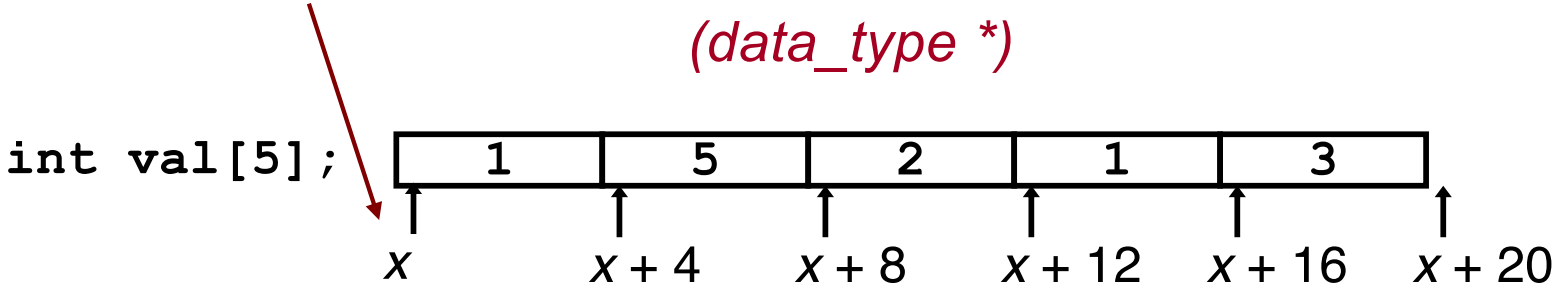


Arrays: acesso aos elementos

Declaração em C:

data_type **Array_name**[*length*];

O identificador **Array_name** é um apontador para o primeiro elemento



Referência	Tipo	Valor
`val`	`int *`	***x***
`val[4]`	`int`	`3`
`val+1`	`int *`	***x*** **+ 4**
`&val[2]`	`int *`	***x*** **+ 8**
~~`val[5]`~~	~~`int`~~	~~`??`~~
`*(val+1)`	`int`	`5`
`val + i`	`int *`	***x*** **+ 4** ***i***

Arrays no IA-32: exemplo de acesso a um elemento

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

Argumentos:

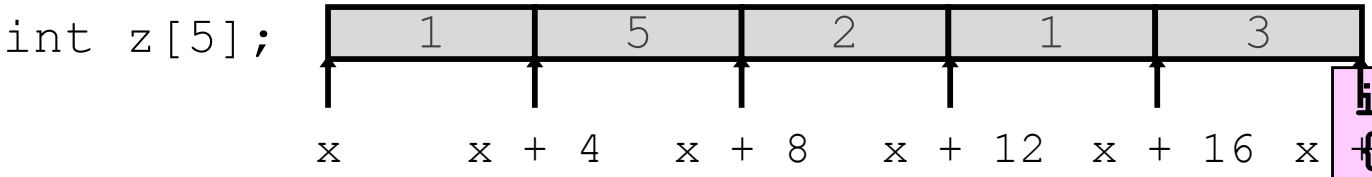
- início do *array* `z` : neste exemplo, colocado em `%edx` (apontador)
- índice `dig` do *array* `z` : neste exemplo, colocado em `%eax` (inteiro)
- a devolver pela função: tipo `int` (4 *bytes*), por convenção, em `%eax`

Localização do elemento **`z[dig]`**:

- `Mem[início_array_z+(índice_dig)*4]`
- na sintaxe do *assembler* da GNU para IA-32/Linux: `(%edx,%eax,4)`

```
movl   8(%ebp),%edx          # %edx <= z
movl  12(%ebp),%eax          # %eax <= dig
movl  (%edx,%eax,4),%eax     # devolve z[dig]
```

Arrays no IA-32: acesso aos elementos




```
int sum5(int *z)
{
   int zi = 0;
   int i=0;
   do {
      zi += z[i];
      i++;
   } while(i <= 4);
   return zi;
}
```

Exemplo:

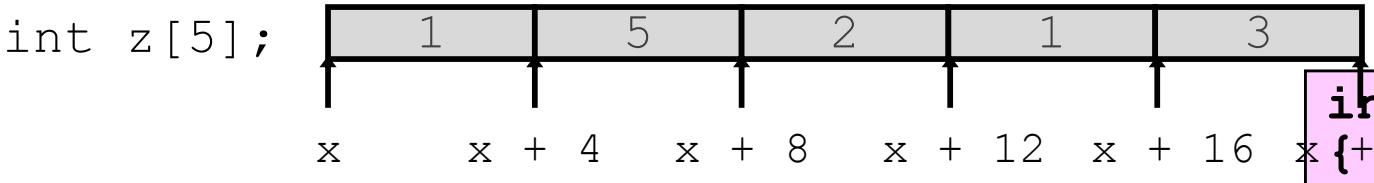
somar 5 elementos de um array
(*do { … } while loop*)

Análise do código compilado

- Registos
  - %ecx i
  - %eax zi
  - %ebx z

```
   movl  8(%ebp),%ebx        # %ebx <= z
   xorl %eax,%eax            #  zi = 0
   xorl %ecx,%ecx            #  i = 0
.L2:                         #loop:
   addl  (%ebx,%ecx,4),%eax  # zi += z[i];
   inc %ecx                  # i++
   cmpl $4,%ecx              # comp 4 : i
   jle .L2                   # if <= goto loop
```

Arrays no IA-32: utilização de apontadores em vez de índices



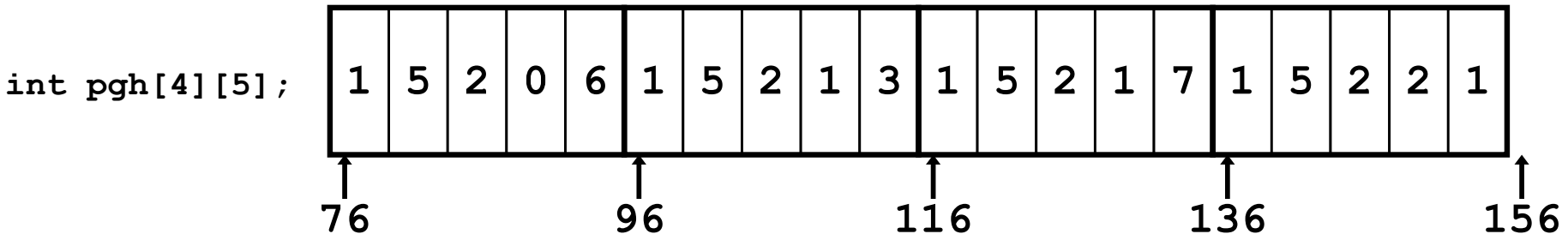
- Registos
  - %ebx z
  - %eax zi
  - %edx zend
- Cálculos
  - z++ incrementa 4 ao apontador

```
int sum5(int *z)
{
   int zi = 0;
   int *zend = z + 4;
   do {
      zi += *z;
      z++;
   } while(z <= zend);
   return zi;
}
```

```
   movl  8(%ebp),%ebx      # %ebx <= z
   xorl %eax,%eax          # zi = 0
   leal 16(%ebx),%edx      # zend = z+4
.L2:                       #loop:
   addl (%ebx),%eax        # zi += *z
   addl $4,%ebx            # z++
   cmpl %edx,%ebx          # comp z : zend
   jle .L2                 # if <= goto loop
```

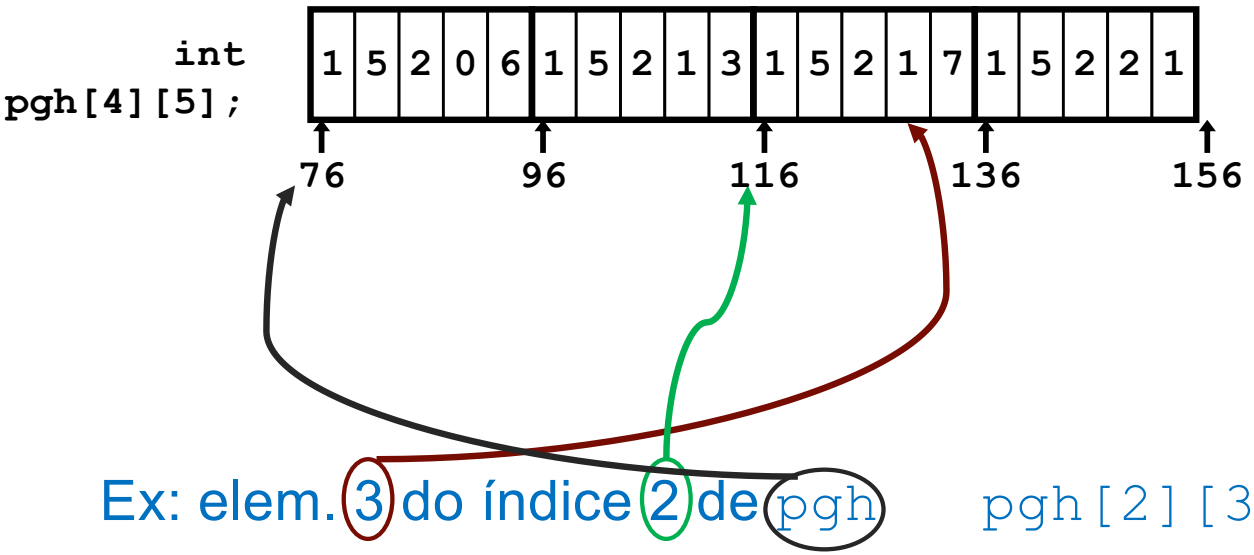
Array de arrays: análise de um exemplo

```
int pgh[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



- `int pgh[4][5]`
  - variável `pgh` é um *array* de 4 elementos
    - alocados em blocos de memória contíguos
  - cada elemento é um *array* de 5 `int`'s
    - alocados em memória em células contíguas

Array de arrays no IA-32: exemplo: código para acesso a um elemento (1)



Localização:

76 + 20*2 + 4*3 = 128

- **Localização em memória de `pgh[index][dig]`:**

  `pgh + 20*index + 4*dig`

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

Array de arrays no IA-32: exemplo: código para acesso a um elemento (2)

- **Localização em memória de `pgh[index][dig]`:**

  `pgh + 20*index + 4*dig`

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

- **Código em *assembly*:**
  - cálculo do endereço (código gerado pelo compilador – optimizado)

    `pgh + 4*(index+4*index) + 4*dig`
  - acesso ao elemento: com `movl`

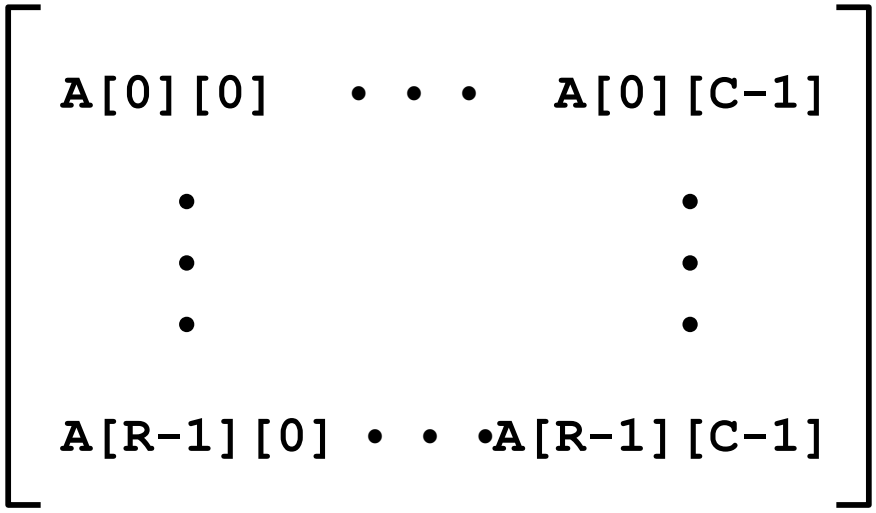
```
                                # %ecx = dig
                                # %eax = index
leal 0(,%ecx,4),%edx            # %edx = 4*dig
leal (%eax,%eax,4),%eax         # %eax = 5*index
movl pgh(%edx,%eax,4),%eax  # devolve Mem(pgh+4*5*index+4*dig)
```

Array de arrays: generalização: alocação em memória

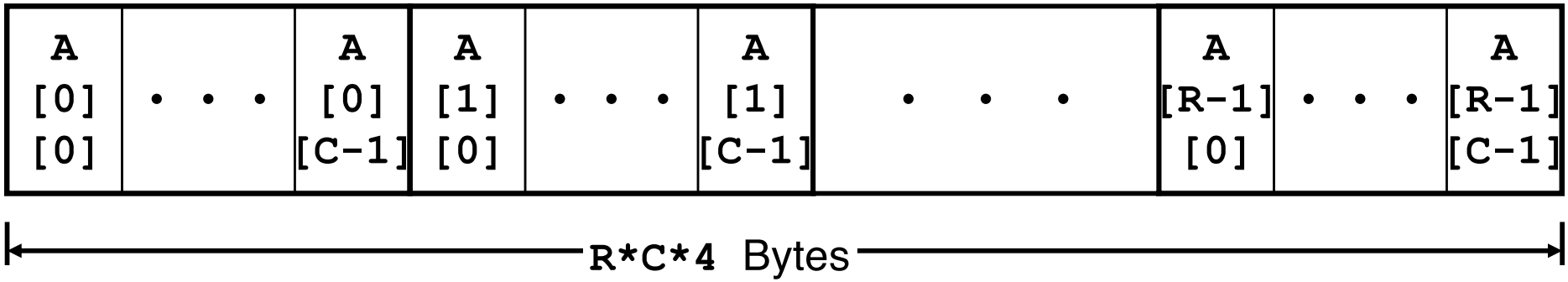
Declaração em C:

data_type Array_name[***R***][***C***];

- Alocação em memória de uma região com
  R * ***C*** * sizeof(*data_type*) *bytes*
- Ordenação dos elementos em memória
  Row-Major (típico em C)




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



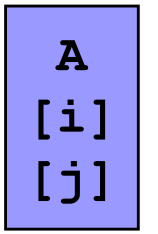
Array de arrays: generalização: acesso a um elemento

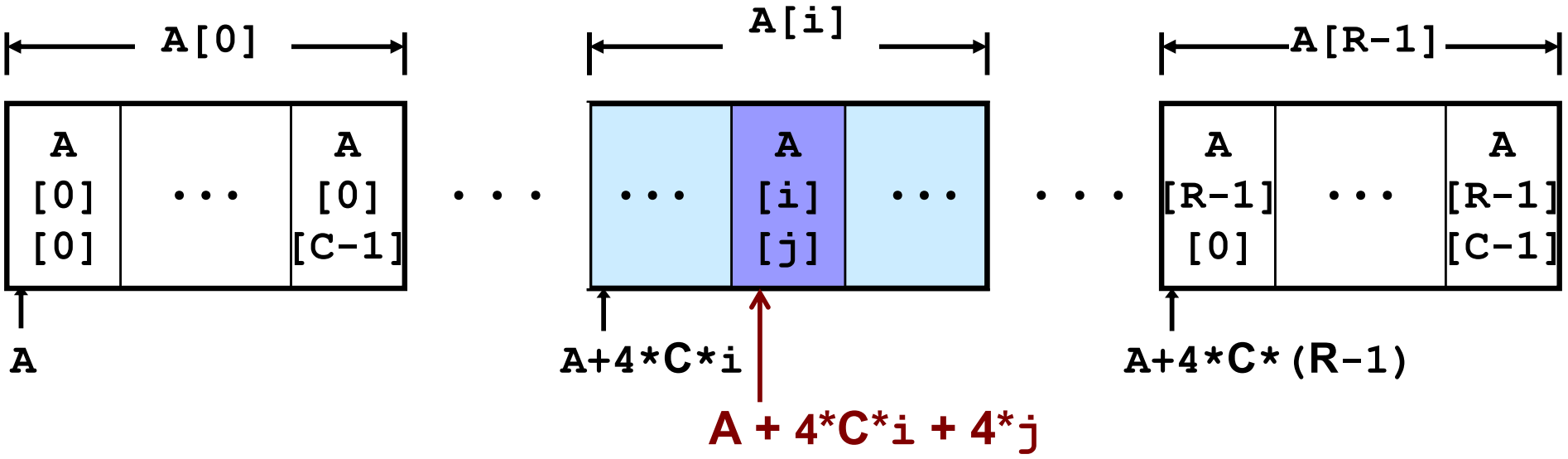
Elementos de um *array* R*C

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

- `A[i][j]` é um elemento do tipo ***T*** (*data_type*) com dimensão ***K*** = `sizeof(`***T***`)`
- sua localização:

  `A + `***K*** `* `***C*** `* i + `***K*** `* j`





Array de arrays versus alternativa: array de apontadores para arrays

Dois modos diferentes de localização dos elementos:

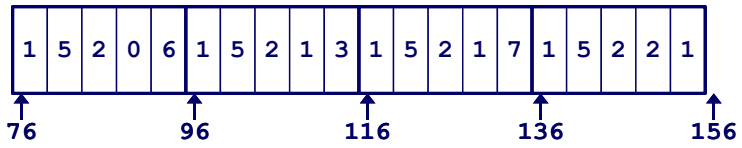
```
int get_pgh_digit(int index,int dig)
{
  return pgh[index][dig];
}
```

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

Array de *arrays*

- elemento em

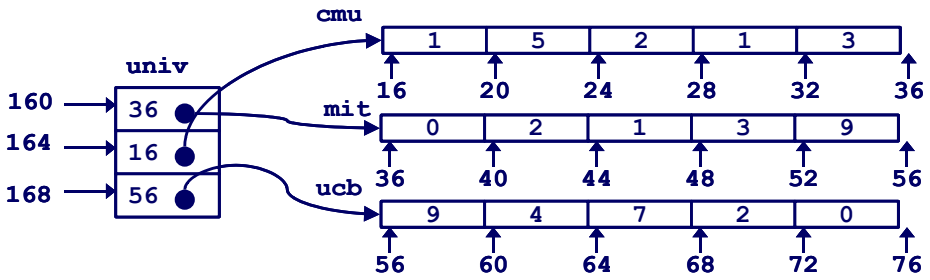
`Mem[pgh+20*index+4*dig]`



Array de apontadores para *arrays*

- elemento em

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

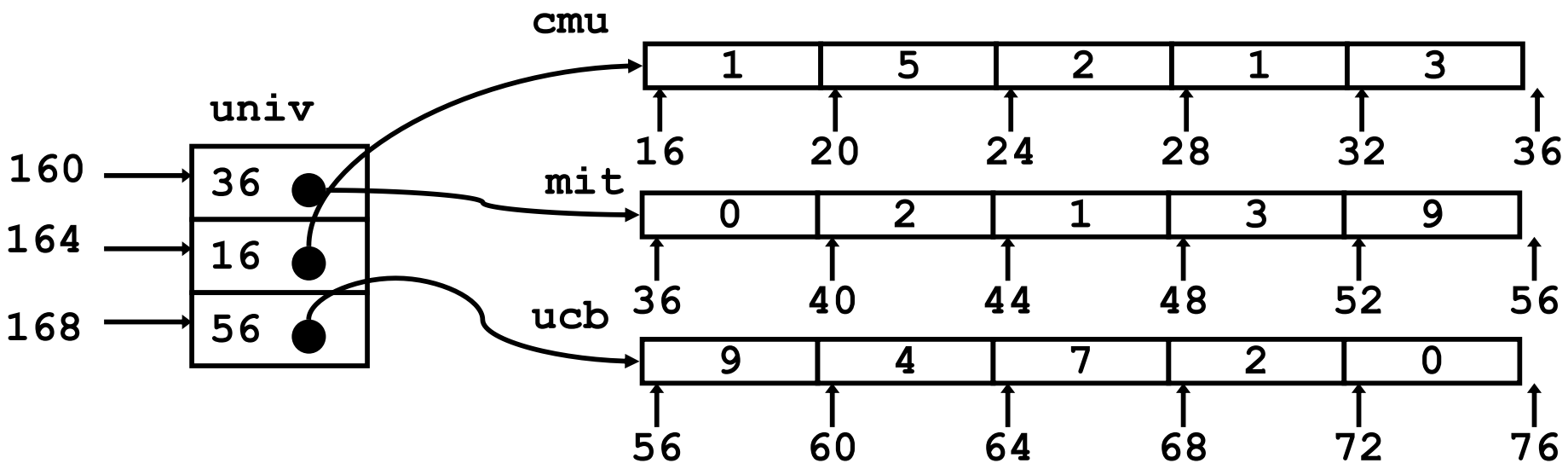


Array de apontadores para arrays: uma alternativa

- Variável `univ` é um *array* de 3 elementos
- Cada elemento é um apontador:
  - ocupa 4 *bytes (em IA-32)*
  - aponta para um *array* de `int`'s

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

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



Array de apontadores para arrays: acesso a um elemento

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

Cálculo da localização

- para acesso a um elemento

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

- requer 2 acessos à memória
  - um para buscar o apontador para *row array*
  - outro para aceder ao elemento do *row array*

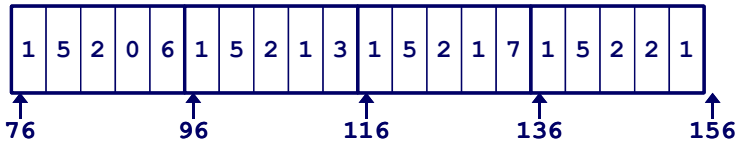
```
                              # %ecx = index
                              # %eax = dig
leal  0(,%ecx,4),%edx         # 4*index
movl  univ(%edx),%edx         # Mem[univ+4*index]
movl  (%edx,%eax,4),%eax      # devolve Mem[Mem[univ+4*index]+4*dig]
```

Comparação: Array de arrays versus array de apontadores para arrays

Comparação do código *assembly*:

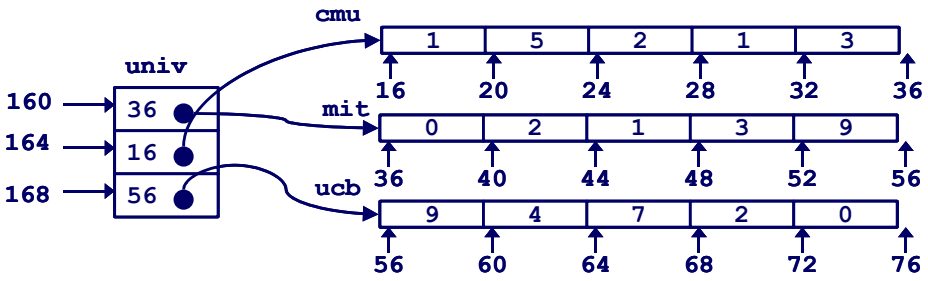
```
int get_X_digit(int index, int dig){
  return X[index][dig];
}
```

Array de *arrays*




```
leal 0(,%ecx,4),%edx
leal (%eax,%eax,4),%eax
movl pgh(%edx,%eax,4),%eax
```

Array de apontadores para *arrays*




```
leal 0(,%ecx,4),%edx
movl univ(%edx),%edx
movl (%edx,%eax,4),%eax
```

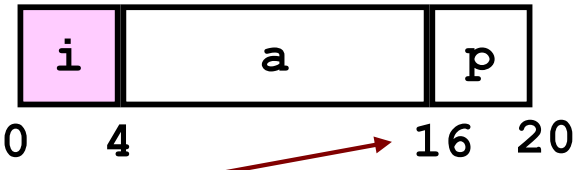
requer um acesso adicional à memória para buscar o apontador para *row array*

Structure: noções básicas

Propriedades

- Armazenadas num bloco de memória
  - Tal como os *arrays* de tipos primitivos são identificadas pelo endereço do primeiro *byte*
- Os membros podem ser de tipos diferentes
  - Cada membro pode ocupar um número diferente de bytes
- Os membros são acedidos por nomes
  - No assembly apenas vemos *offsets*
    - *Offsets* são definidos pela ordem da declaração

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

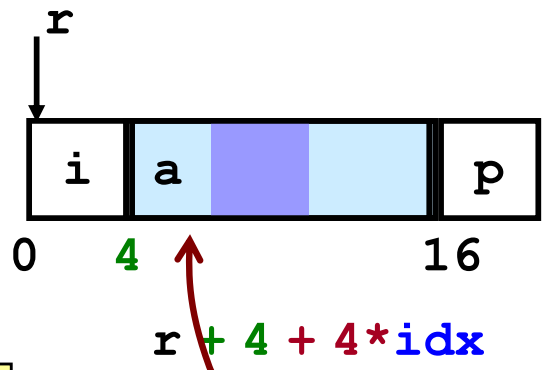



```
int *get_p(struct rec *r){
  return (r->p);
}
```

```
                          # %edx = r
movl 16(%edx)%eax,        # %eax = r->p
```

Structure: apontadores para membros

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



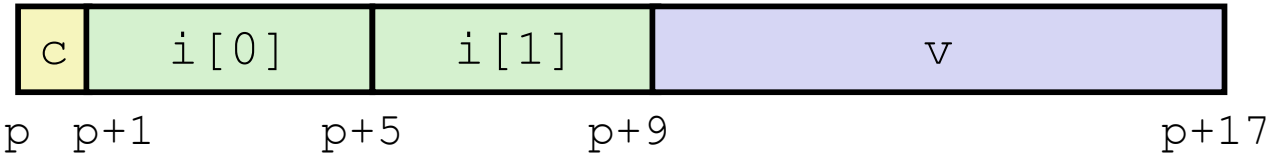

```
int *find_a(struct rec *r, int idx)
{
  return &r->a[idx];
}
```

Valor calculado na compilação

```
                                  # %ecx= idx
                                  # %edx= r
leal 4(%edx,%ecx,4),%eax          # r+4*idx+4
```

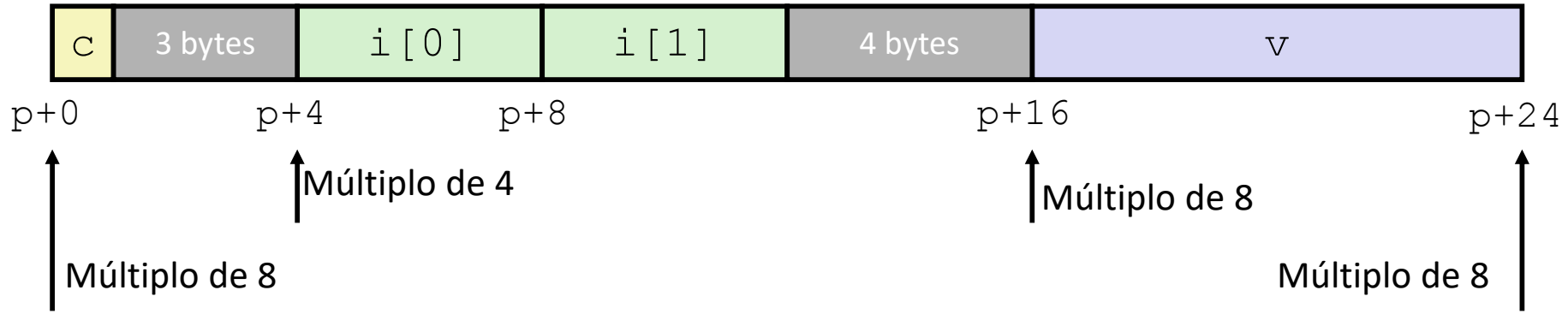
Alinhamento de dados na memória

- **Dados não alinhados**




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

- **Dados alinhados (x86-64 em Linux)**
  - Compilador insere bolhas na estrutura



- **Motivação para alinhar os dados em memória**
  - É mais efficiente aceder a endereços múltiplos de 4 ou 8
    - Organização interna da memória em blocos / linhas da *cache* com 64 bytes

Alinhamento de dados na memória: os dados primitivos/escalares

- **Exemplo de alinhamento de dados:**

  x86-64 em Linux (gcc 12.2) - tipos de dados com K-bytes são alinhados em endereços múltiplos K

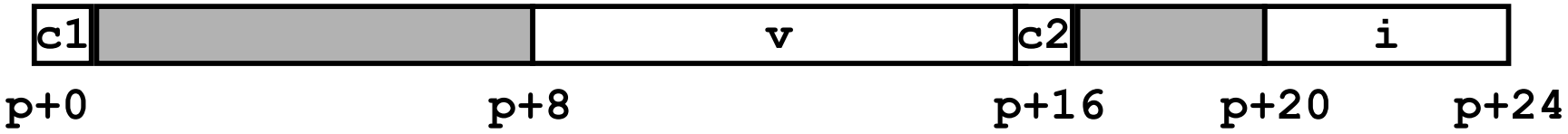
  - 1 byte (e.g., `char`)
    - sem restrições no endereço
  - 2 bytes (e.g., `short`)
    - o bit menos significativo do endereço deve ser 0_2
  - 4 bytes (e.g., `int`, `float`, etc.)
    - os 2 bits menos significativos do endereço devem ser 00_2
  - 8 bytes (e.g., `double`, `char *`)
    - os 3 bits menos significativos do endereço devem ser 000_2

  Nota: no exemplo `sizeof(struct S1)` devolve o tamanho correto (incluindo o espaço das bolhas)

Alinhamento de dados na memória: ordenação dos membros

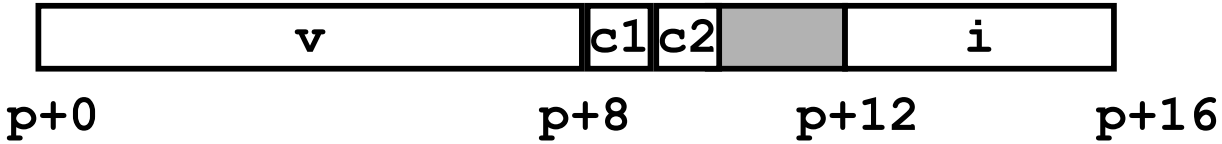
```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

10 *bytes* de espaço desperdiçado Linux x86-64
(Nota: `double` – alinhamento de 8, `int` alinhamento de 4)

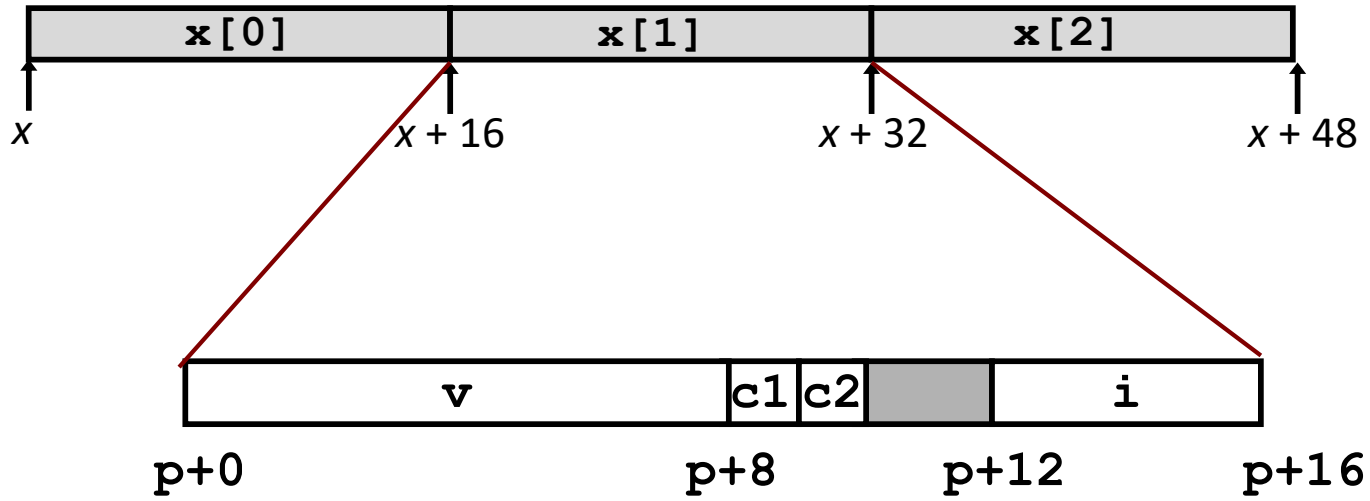



```
struct S5 {
  double v;
  char c1;
  char c2;
  int i;
} *p;
```

apenas 2 *bytes* de espaço desperdiçado



Alinhamento de dados na memória: Arrays de estruturas




```
struct S5 {
  double v;
  char c1;
  char c2;
  int i;
} x[10];
```

- **Estrutura de dados tratada de forma semelhante a array de array:**
  - Cada elemento do *array* é uma estrutura
  - Elementos são armazenados em posições de memória contíguas
    - Podem ser necessárias bolhas no final da estrutura para o alinhamento