

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, x86-64 e MIPS (RISC)
6. Acesso e manipulação de dados estruturados

Propriedades dos dados estruturados em C

- agregam quantidades escalares do mesmo tipo ou de tipos diferentes
- por norma, alocadas a posições contíguas da memória
- a estrutura definida é referenciada pelo apontador para a 1ª posição de memória

Tipos de dados estruturados mais comuns em C

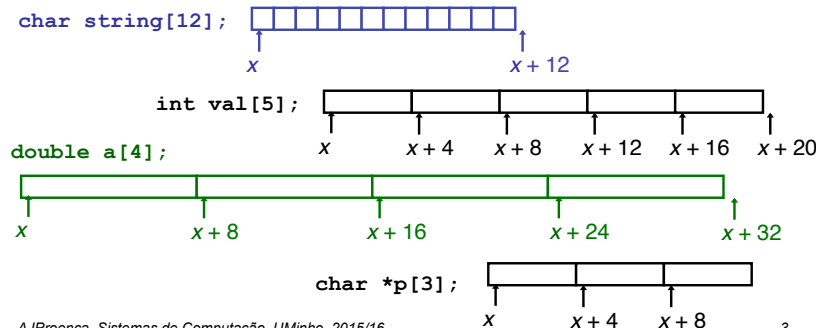
- **array**: agregado de dados escalares do mesmo tipo
 - *string*: array de caracteres terminado com *null*
 - *arrays de arrays*: arrays multi-dimensionais
- **structure**: agregado de dados de tipos diferentes
 - *structures de structures*, *structures de arrays*, ...
- **union**: mesmo objecto mas com visibilidade distinta

Arrays: alocação em memória

Declaração em C:

```
data_type Array_name[length];
```

Aloca em memória uma região com tamanho
 $length * sizeof(data_type)$ bytes

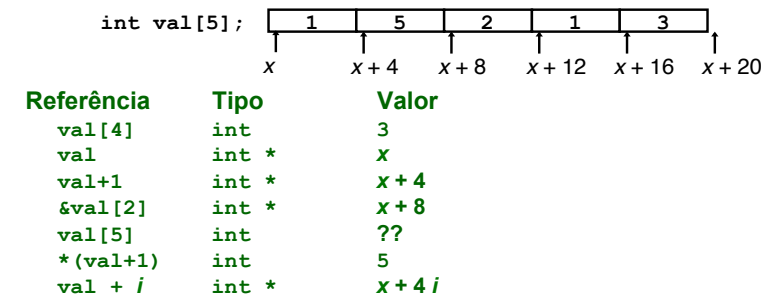


Arrays: acesso aos elementos

Declaração em C:

```
data_type Array_name[length];
```

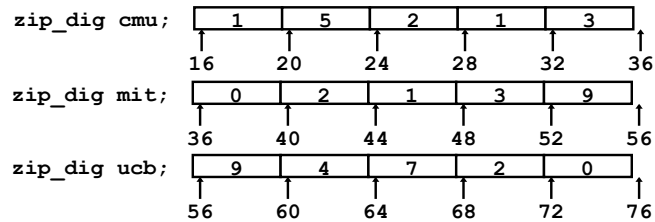
O identificador `Array_name` pode ser usado
como apontador para o elemento 0



Arrays: análise de um exemplo

```
typedef int zip_dig[5];

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



Notas

- declaração "zip_dig cmu" equivalente a "int cmu[5]"
- os arrays deste exemplo ocupam blocos sucessivos de 20 bytes

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

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

Argumentos:

- início do array z : neste exemplo, o gcc coloca em %edx
- índice dig do array z : neste exemplo, o gcc coloca em %eax
- a devolver pela função: tipo int (4 bytes), por convenção, em %eax

Localização do elemento z[dig]:

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

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

Arrays no IA-32: apontadores em vez de índices (2)

Análise do código compilado

• Registos

```
%ecx z
%eax zi partilhado com *z
%ebx zend
```

• Cálculos

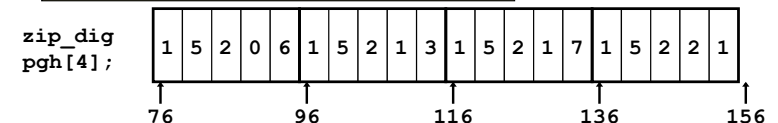
- $10 * zi + *z \Rightarrow *z + 2 * (zi + 4 * zi)$
- z++ incrementa 4

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

```
xorl %eax,%eax # %ecx <= z
leal 16(%ecx),%ebx # zi = 0
                          # zend = z+4
.L59: #loop:
leal (%eax,%eax,4),%edx # %edx <= 5*zi
movl (%ecx),%eax # %eax <= *z
addl $4,%ecx # %ecx <= z++
leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
cml %ebx,%ecx # comp z : zend
jle .L59 # if <= goto loop
```

Array de arrays: análise de um exemplo

```
#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}};
```



- Declaração "zip_dig pgh[4]" equivalente a "int pgh[4][5]"
 - variável pgh é um array de 4 elementos
 - alocados em memória em blocos contíguos
 - cada elemento é um array de 5 int's
 - alocados em memória em células contíguas
- Ordenação dos elementos em memória (típico em C): "Row-Major"

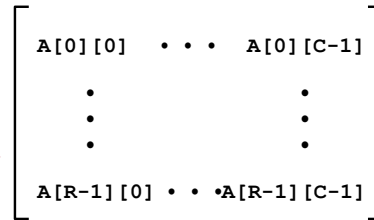
Array de arrays: alocação em memória

Declaração em C:

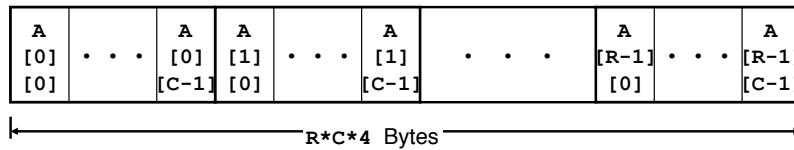
```
data_type Array_name[R][C];
```

- Alocação em memória de uma região com $R * C * \text{sizeof}(\text{data_type})$ bytes

- Ordenação
Row-Major



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

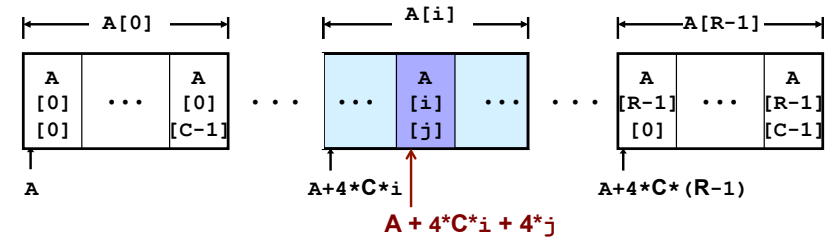
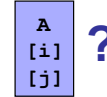


Array de arrays: acesso a um elemento

Elementos de um array R*C

- $A[i][j]$ é um elemento do tipo $T(\text{data_type})$ com dimensão $K = \text{sizeof}(T)$
- sua localização:
 $A + K * C * i + K * j$

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



Array de arrays no IA-32: código para acesso a um elemento

- Localização em memória de

```
pgh[index][dig]:
```

```
pgh + 20*index + 4*dig
```

- Código em assembly:

- cálculo do endereço

```
pgh + 4*(index+4*index) + 4*dig
```

- acesso ao elemento: com movl

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

```
# %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 apontadores para arrays: uma visão alternativa

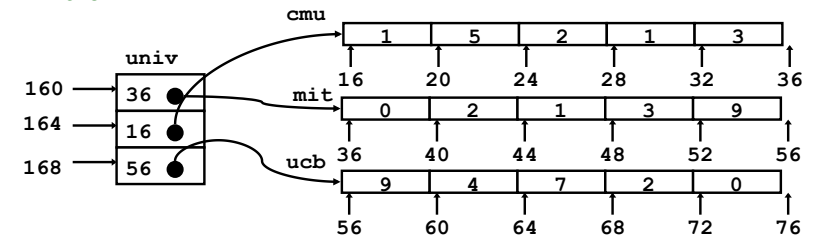
- Variável univ é um array de 3 elementos

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

- Cada elemento:

- um apontador de 4 bytes
- aponta para um array de int's

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



Array de apontadores para arrays: acesso a um elemento

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

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

```

                                # %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]
```

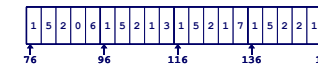
Array de arrays versus array de apontadores para arrays

Modos distintos de cálculo da localização dos elementos:

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

Array de arrays

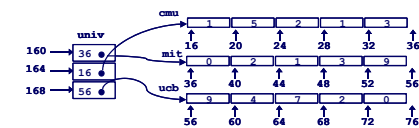
- elemento em
 $Mem[pgh+20*index+4*dig]$



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

Array de apontadores para arrays

- elemento em
 $Mem[Mem[univ+4*index]+4*dig]$



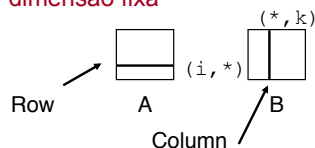
Arrays multi-dimensionais de tamanho fixo: a eficiência do compilador (1)

• Oportunidades para otimizar

- o array a está em localizações contíguas, começando em $a[i][0]$: usar apontador!
- o array b está em localizações espaçadas de $4*N$ células, começando em $b[0][j]$: usar também apontador!

• Limitações

- apenas funciona com arrays de dimensão fixa



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

```
/* Compute element i,k of
fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```

Arrays multi-dimensionais de tamanho fixo: a eficiência do compilador (2)

• Otimizações automáticas do compilador:

–antes...

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

```
/* Compute element i,k of
fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```

–depois...

```
/* Compute element i,k ... */
int fix_prod_ele (...)
{
    int *Aptr = &A[i][0];
    int *Bptr = &B[0][k];
    int cnt = N-1;
    int result = 0;
    do {
        result += (*Aptr)*(*Bptr);
        Aptr += 1;
        Bptr += N;
        cnt--;
    }while (cnt>=0);
    return result;
}
```

Structure: noções básicas

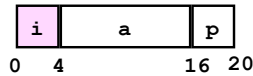


Propriedades

- em regiões contíguas da memória
- membros podem ser de tipos diferentes
- membros acessados por nomes

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

Organização na memória



Acesso a um membro da structure

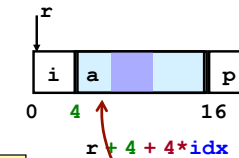
```
void set_i(struct rec *r,
          int val)
{
    r->i = val;
}
```

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

Structure: apontadores para membros (1)



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



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

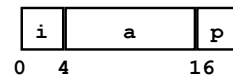
Valor calculado na compilação

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

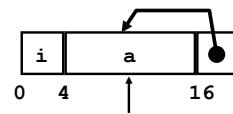
Structure: apontadores para membros (2)



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



```
void set_p(struct rec *r)
{
    r->p = &r->a[r->i];
}
```



Elemento i

```
movl (%edx),%ecx # %edx = r
leal 0(,%ecx,4),%eax # r->i
leal 4(%edx,%eax),%eax # 4*(r->i)
movl %eax,16(%edx) # Update r->p
# r+4+4*(r->i)
```

Alinhamento de dados na memória



• Dados alinhados

- Tipos de dados primitivos (escalares) requerem *K bytes*
- Endereço deve ser múltiplo de *K*
- Requisito nalgumas máquinas; aconselhado no IA-32
 - tratado de modo diferente, consoante Unix/Linux ou Windows!

• Motivação para alinhar dados

- Memória acessada por *double* ou *quad-words* (alinhada)
 - ineficiente lidar com dados que passam esses limites
 - ainda mais crítico na gestão da memória virtual (limite da página!)

• Compilador

- Insere bolhas na *structure* para garantir o correcto alinhamento dos campos

Alinhamento de dados na memória: os dados primitivos/escalares

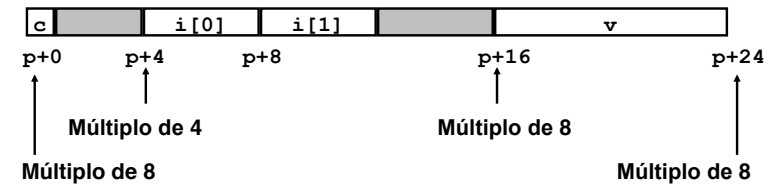
- **1 byte** (e.g., char)
 - sem restrições no endereço
- **2 bytes** (e.g., short)
 - o bit menos significativo do endereço deve ser 0_2
- **4 bytes** (e.g., int, float, char *, etc.)
 - os 2 bits menos significativo do endereço devem ser 00_2
- **8 bytes** (e.g., double)
 - Windows (e a maioria dos SO's & instruction sets):
 - os 3 bits menos significativo do endereço devem ser 000_2
 - Unix/Linux:
 - os 2 bits menos significativo do endereço devem ser 00_2
 - i.e., mesmo tratamento que um dado escalar de 4 bytes
- **12 bytes** (long double)
 - Unix/Linux:
 - os 2 bits menos significativo do endereço devem ser 00_2
 - i.e., mesmo tratamento que um dado escalar de 4 bytes

Alinhamento de dados na memória: numa structure

- Deslocamentos dentro da *structure*
 - deve satisfazer os requisitos de alinhamento dos elementos (i.e., do seu maior elemento, K)
- Requisito para o endereço inicial
 - deve ser múltiplo de K

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

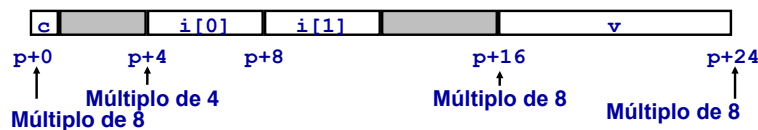
- Exemplo (em Windows):
 - $K = 8$, devido ao elemento double



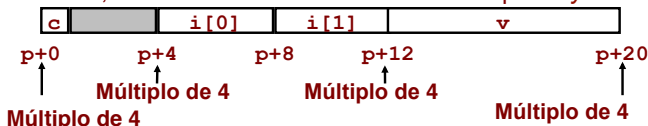
Alinhamento de dados na memória: Windows versus Unix/Linux

- **Windows:**
 - $K = 8$, devido ao elemento double

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



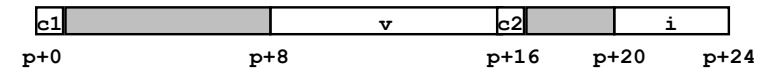
- **Unix/Linux:**
 - $K = 4$; double tratado como se fosse do tipo 4-bytes



Alinhamento de dados na memória: ordenação dos membros

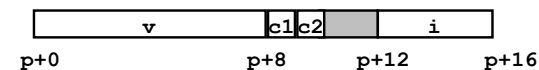
```
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```

10 bytes espaço desperdiçado no Windows



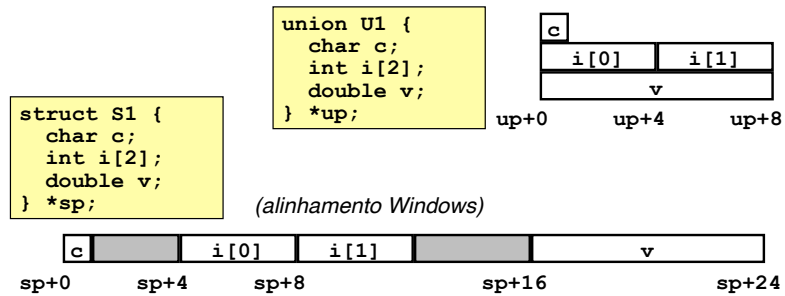
```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```

apenas 2 bytes de espaço desperdiçado em Unix/Linux

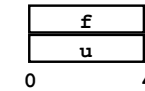


• **Princípios**

- sobreposição dos elementos de uma *union*
- memória alocada de acordo com o maior elemento
- só é possível aceder a um elemento de cada vez



```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```



Como associar um padrão de bits, a um dado float

```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

isto **NÃO** é o mesmo que (float) u

Como obter o conjunto de bits que representa um float

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

isto **NÃO** é o mesmo que (unsigned) f