Estrutura do tema ISA do IA32

1. Desenvolvimento de programas no IA32 em Linux
2. Acesso a operandos e operações
3. Suporte a estruturas de controlo
4. Suporte à invocação/retorno de funções
5. Análise comparativa: IA-32 (CISC) 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
- sempre alocadas a posições contíguas da memória
- a estrutura definida pode ser referenciada pelo apontador para a 1ª posição de memória

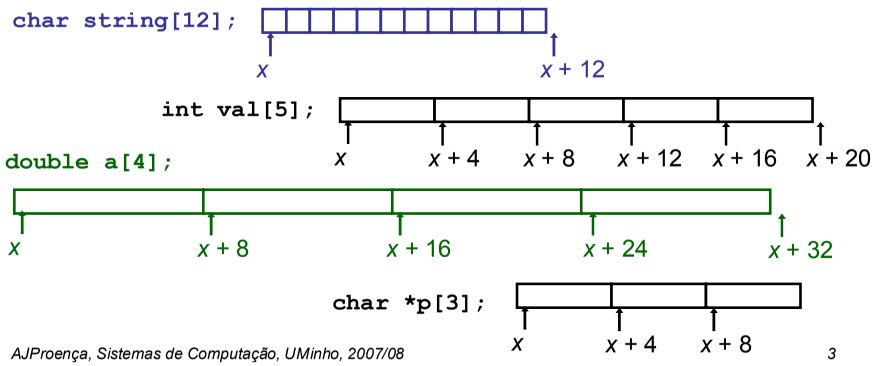
Tipos de dados estruturados mais comuns em C

- ***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

Declaração em C:

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

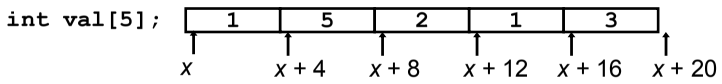
Alocação em memória de uma região com
length * sizeof(*data_type*) *bytes*



Declaração em C:

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

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

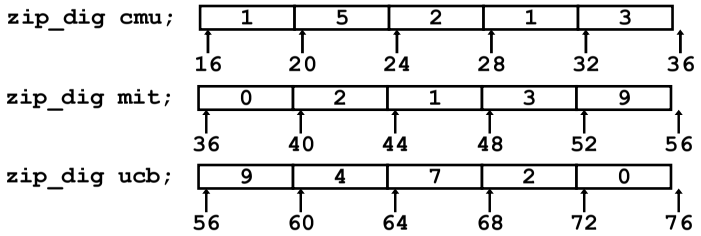


Referência	Tipo	Valor
val[4]	int	3
val	int *	*x*
val+1	int *	*x* + 4
&val[2]	int *	*x* + 8
val[5]	int	??
*(val+1)	int	5
val + *i*	int *	*x* + 4 *i*

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: exemplo de acesso a um elemento

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

Argumentos:
- tipo `int` (4 *bytes*)
- início do *array* `z` (colocado em `%edx`)
- índice `dig` do *array* `z` (colocado em `%eax`)

Localização do elemento **`z[dig]`**:
- `Mem[(%edx)+4*(%eax)]`
- IA32/Linux: `(%edx,%eax,4)`

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


Arrays: apontadores em vez de índices (1)

Código original

com referências a *arrays* dentro de ciclos

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

Transformação pelo GCC
- eliminou a variável `i`
- converteu índices em apontadores
- reduziu à forma **`do-while`**

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

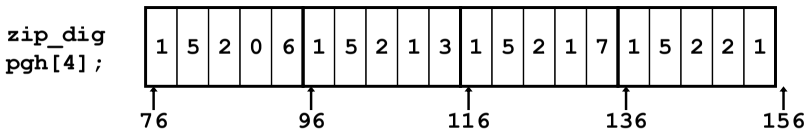

Arrays: apontadores em vez de índices (2)

Análise do código compilado
- Registos
  - `%ecx` z
  - `%eax` zi
  - `%ebx` zend
- Cálculos
  - `10*zi + *z` => `*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;
}
```

```
   # %ecx = z
   xorl %eax,%eax          # zi = 0
   leal 16(%ecx),%ebx      # zend  = z+4
.L59:
   leal (%eax,%eax,4),%edx # 5*zi
   movl (%ecx),%eax        # *z
   addl $4,%ecx            # z++
   leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
   cmpl %ebx,%ecx          # z : zend
   jle .L59                # 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 }};
```

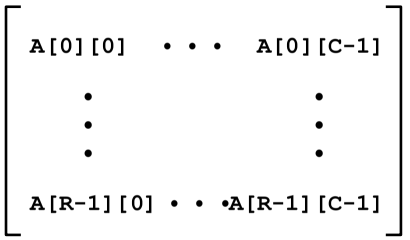


- Declaração "`zip_dig pgh[4]`" equivalente a "`int pgh[4][5]`"
  - variável `pgh` é um *array* de 4 elementos
    - alocados em memória em blocos contíguos
  - cada elemento é um *array* de 5 `int`'s
    - alocados em memória em células contíguas
- Ordenação dos elementos (garantido em C): "*Row-Major*"

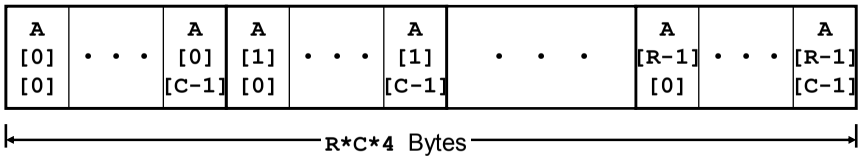
Declaração em C:

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

- Alocação em memória de uma região com
  R `*` ***c*** `*` `sizeof(`*data_type*`)` *bytes*
- Ordenação
  Row-Major



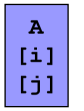

```
int A[R][C];
```

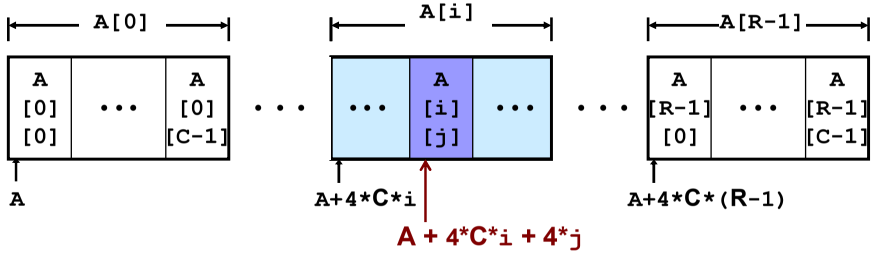


Elementos de um *array* R*C

```
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`





- **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

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

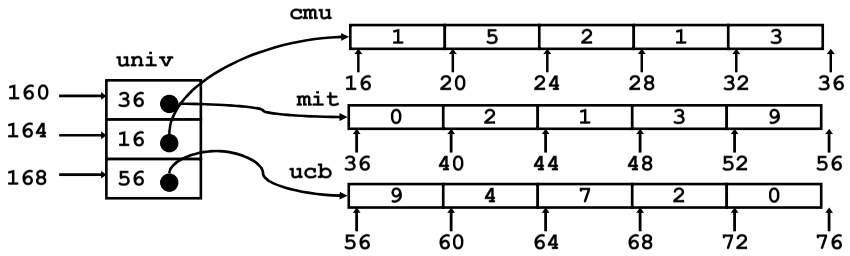
```
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx          # 4*dig
leal (%eax,%eax,4),%eax       # 5*index
movl pgh(%edx,%eax,4),%eax    # *(pgh + 4*dig + 20*index)
```

Array de apontadores para arrays: uma visão alternativa

- Variável `univ` é um *array* de 3 elementos
- Cada elemento:
  - um apontador de 4 *bytes*
  - aponta para um *array* de `int`'s

```
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};
```



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
  - para buscar apontador para *row array*
  - para aceder a 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  # Mem[...+4*dig]
```


Array de arrays versus array de apontadores para arrays

Modos distintos de cálculo da localização dos elementos:

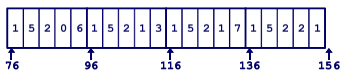
```
int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

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

Array* de *arrays

- elemento em

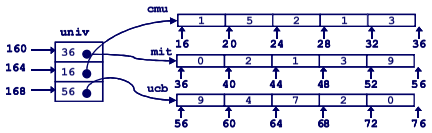
`Mem[pgh+20*index+4*dig]`



Array* de apontadores para *arrays

- elemento em

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



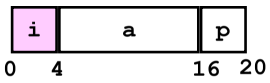
Structures: noções básicos

Propriedades

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

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

Organização na memória



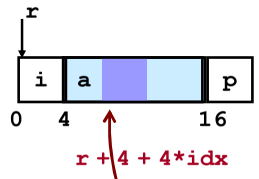
Acesso a um membro da *structure*

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

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

Structures: apontadores para membros (1)

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



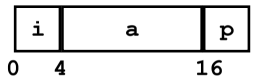

```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

Valor calculado na compilação

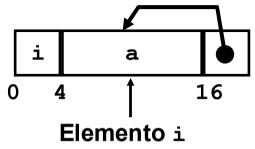
```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```


Structures: 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];
}
```




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


Alinhamento de dados na memória

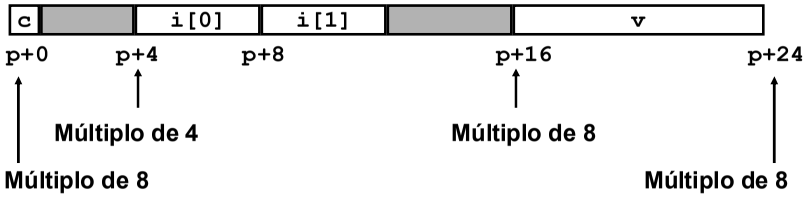
- **Dados alinhados**
  - Tipos de dados primitivos (escalares) requerem K *bytes*
  - Endereço deve ser múltiplo de K
  - Requisito nalgumas máquinas; aconselhado no IA32
    - tratado de modo diferente, consoante Linux ou Windows!
- **Motivação para alinhar dados**
  - Memória acedida por *double* 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
  - 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`)
  - Linux:
    - os 2 bits menos significativo do endereço devem ser 00_2
    - i.e., mesmo tratamento que um dado escalar de 4 *bytes*

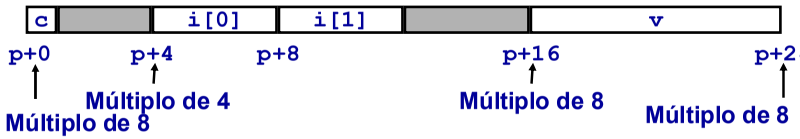
- Deslocamentos dentro da *structure*
  - deve satisfazer os requisitos de alinhamento dos elementos (i.e., do seu maior elemento, ***K***)
- Requisito para o endereço inicial
  - deve ser múltiplo de ***K***
- Exemplo (em Windows):
  - ***K*** = 8, devido ao elemento `double`

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

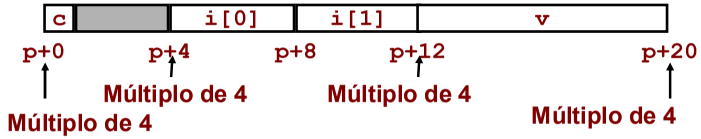


- **Windows** (incluindo Cygwin):
  - K = 8, devido ao elemento `double`

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

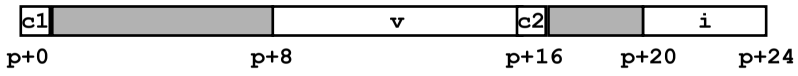


- **Linux**:
  - K = 4; `double` tratado como se fosse do tipo 4-*bytes*




```
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