Arquitetura de Computadores

MiEI – 2016/17

DI-FCT/UNL

Aula 5

Variáveis na memória

- Variáveis são designações simbólicas para posições de memória:

```
int A = 5;
int B = -2;  // 0xFFFFFFFE
```

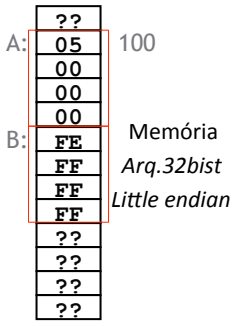
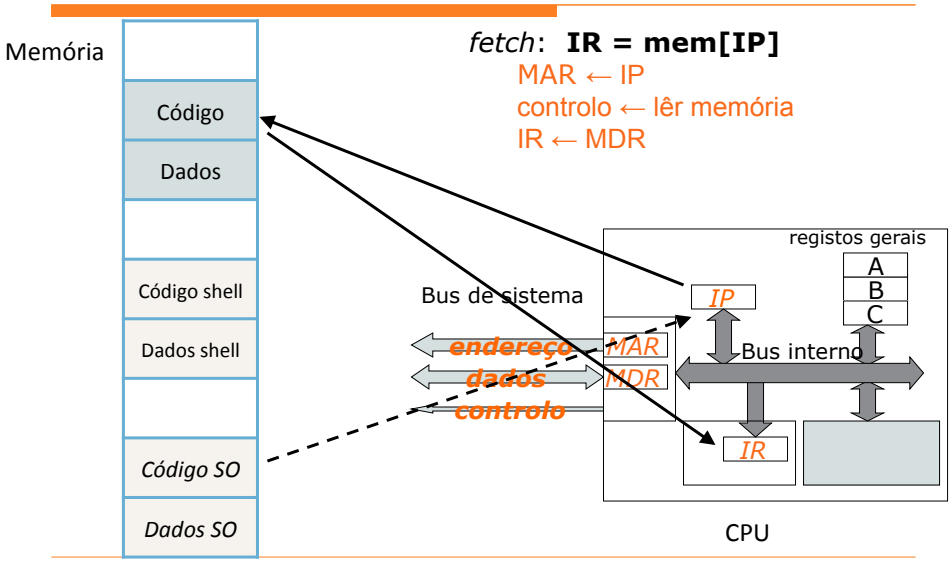


Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes

Execução de um Programa



Execução da atribuição

int A = 5;

Antes, o compilador definiu um endereço de memória para A (exemplo: 100). Sabe a sua dimensão (4 bytes). Gerou a instrução máquina para colocar o valor 5 na memória respetiva.

- Fetch, obtém a instrução máquina para IR:

Store val mem	5	100

- Descodifica e Executa:

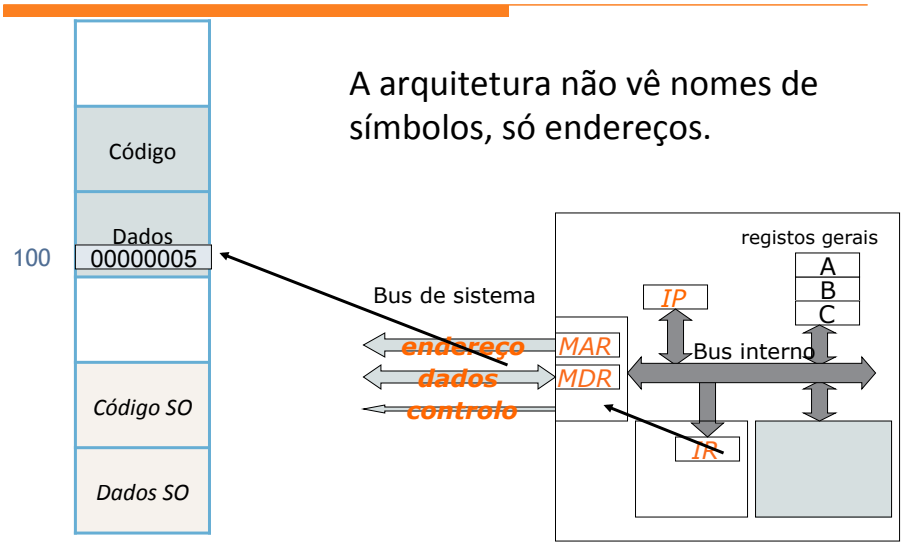
***Store** val to memory*:

MDR ← 5

MAR ← 100

controlo ← escr. Memória

Execução de um Programa C



Apontadores - C

- Apontadores (ou ponteiros) são variáveis cujo tipo representa referências para memória → **endereços**.
- Sintaxe para declarar a variável:

  Tipo * *var_pt*;

  Se var_pt não é iniciado pode conter "lixo"
- Operadores:

  & *nome_var* ← obter endereço de *nome_var*

  * *var_pt* ← obter o valor apontado
- Constante: **NULL** ← apontador inválido

  (semelhante ao ***null*** no Java)

Exemplo:

```
int X = 5;
int *Y;
```

X: 5

Y: ? **Y não iniciado**

```
int a = *Y; // errado! (mas pode "executar")
Y=10;  // possivelmente errado!
```

X: 5

```
Y= &X;  // Y aponta para X
```

Y: *End de X*

```
int a = *Y;
//a recebe valor apontado por Y (ou seja, X)
```

a: 5

Referências para memória - C

- Variável B refere A, guarda endereço de A, ou aponta para valor em A:

```
int A = 15213;    // 0x3B6D
int *B = &A;      // 100 = 0x64
```

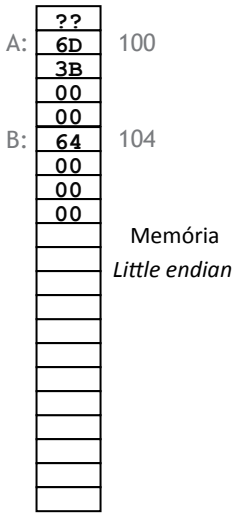


Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes ← arq. end. 32 bits

Referências para memória - C

- Variável B refere A, guarda endereço de A, ou aponta para valor em A:

```
int A = 15213;    // 0x3B6D
int *B = &A;      // 100 = 0x64
```

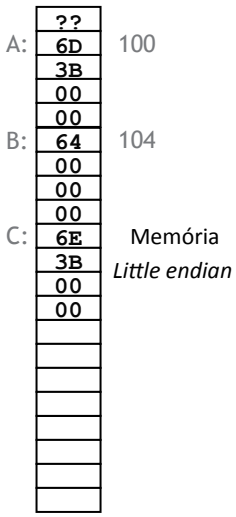
int C = *B + 1;

Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes ← arq. end. 32 bits

C: 108 : 4 bytes



Referências para memória - C

- Variável B refere A, guarda endereço de A, ou aponta para valor em A:

```
int A = 15213;    // 0x3B6D
int *B = &A;      // 100 = 0x64
```

int C = *B + 1;

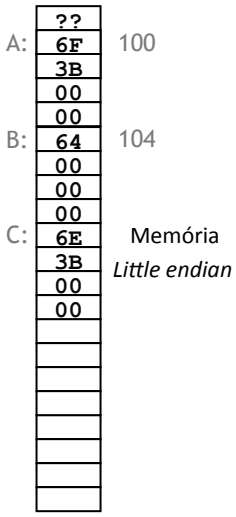
*B = *B + 2;

Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes ← arq. end. 32 bits

C: 108 : 4 bytes



Referências para memória - C

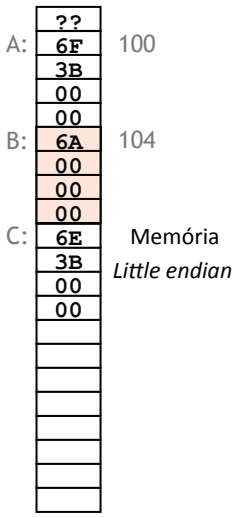
- Variável B refere A, guarda endereço de A, ou aponta para valor em A:

```
int A = 15213;    // 0x3B6D
int *B = &A;      // 100 = 0x64
```

```
int C = *B + 1;
*B = *B + 2;
```

- Em C podemos usar expressões com apontadores:

```
B = B + 1;  //B passa para o próximo int
*B = *B + 2;  // 0x68 = 104
```



Ignorando os tipos de dados (!)

- Com o *cast* de apontadores podemos ignorar os tipos de dados
  - vêr a memória do int como se fosse float (copia bits de x para f):

```
int x = 3;
float f = *(float*)&x;
```

  - vêr a memória do float como se fosse int (copia bits de f para x):

```
f = 1.5;
x = *(int*)&f;
```

Uma variável, vários tipos

- Podemos declarar uma variável com vários tipos:

```
union { int i; float f; } var;
```

  - var pode ser vista como int ou float.
  - É só uma variável. Só são ocupados 4 bytes!
  - O padrão de bits em memória é sempre o mesmo!

```
var.f = 1.5;
printf("valor em var: %X\n", var.i);

valor em var: 3FC00000
```


Parâmetros de funções (Java e C)

- Os argumentos são passados por cópia

```
int fun1( int x ) {
   x=x+1;
   return x;
}
...
int y=3;
int z=fun1( y );
```

- uma cópia do valor em y é passada para x de fun1
- y não é alterado!

Parâmetros apontadores em C

- Em C podemos passar referências:

```
int fun1( int *x ) {
   *x=*x+1;
   return *x;
}
...
int y=3;
int z=fun1( &y );   //passamos a referência de y
```

- O endereço de y é passado para x de fun1
- y é alterado via apontador x!

Objetos no Java

- Os objetos em Java são passados por referência

```
class A { public int i; }
int fun1( int x, A z ) {
   x=x+1;
   z.i = z.i+1;
   return x;
}
```

```
...
int y1=1;
A y2=new A();
y2.i=1;
int w=fun1(y1, y2);
```

  - uma cópia do valor em y1 é passada para x de fun1
  - A referência de y2 é passada para z de fun1
  - y1 não é alterado por fun1!
  - y2 é alterado por fun1!

Gestão dinâmica de memória - C

- Podemos criar variáveis dinamicamente
- Tal baseia-se apenas em pedir memória:

```
int *var = (int *)malloc( sizeof(int) );
```

Desempenha o papel do new do Java;

O int criado não é iniciado.

*Devolve **NULL** se não houver memória suficiente*

- Não há *garbage-collector*. Quando `var` não for necessária temos de a libertar:

```
free( var );
```


Arrays em C - vetores

- Os arrays consistem em sequências contíguas de elementos do mesmo tipo.
- Sintaxe (variantes):

```
int A[3];   // não iniciado
A[0]=1; A[1]=2; A[2]=3;

int A[3] = { 1, 2, 3};
int A[] = { 1, 2, 3};
```

- Tabela de símbolos do compilador:

A: 100 : 4 bytes

'A' representa o início do array (seu endereço)

Arrays em C - matrizes

- As matrizes são dispostas por linhas.
- Sintaxe (variantes):

```
short A[3][2];
A[0][0]=1; A[0][1]=2;
A[1][0]=3; A[1][1]=4;
A[2][0]=5; A[2][1]=6;

short A[][]={ {1, 2}, {3, 4}, {5, 6} };
```

- Tabela de símbolos do compilador:

  A: 100 : 2 bytes

  'A' representa o início do array (seu endereço)

Arrays em C e apontadores

- O nome do array vale por um apontador constante:

```
int v[3];     // v tem tipo int*

v[2] = 5;    ⇔   *(v+2) = 5;
```

- Criando dinamicamente e acedendo como array:

```
int *v = (int*)malloc( 3*sizeof(int));
v[2] = 5;
```