Acesso a variáveis na memória usando ponteiros

Tomás Oliveira e Silva

***Abstract* – Neste documento descrevem-se resumidamente quais são as principais diferenças no acesso a variáveis declaradas ou como `tipo var[][]` ou como `tipo *var[]`. Também se apresenta um exemplo não trivial de utilização de ponteiros numa função.**

I. DIFERENÇAS NO ACESSO A VARIÁVEIS DECLARADAS COMO `TIPO VAR[][]` E COMO `TIPO *VAR[]`

Vamos tratar primeiro do acesso a um array bidimensional (não é obrigatório saber, neste ano lectivo, em AC I, se bem que devia ser). Vamos utilizar como exemplo o seguinte trecho de código:

```
static char a[4][3];
int t0,t1,t2;
t2 = (int)a[t0][t1];
// &a[i][j] = (char *)a+3*i+j
```

Uma tradução possível para *assembly* será

```
   .data
a: .space 12          # 4*3*1
   .text
   la     $t9,a       # $t9 = &a[0][0]
   mulu   $t8,$t0,3   # $t8 = 3 * t0
   addu   $t9,$t9,$t8 # $t9 = &a[t0][0]
   addu   $t9,$t9,$t1 # $t9 = &a[t0][t1]
   lb     $t2,0($t9)  # $t2 = a[t0][t1]
```

Note que

- cada linha do array bidimensional tem três elementos, pelo que, como cada elemento é um `char`, ocupa três bytes (`3 * sizeof(char)` é igual a três). Daí a multiplicação por três para se obter o endereço do início da linha número `t0`.
- foi reservado espaço para todo o array bidimensional (`4 * 3 * sizeof(char)` é igual a doze).

Vamos agora tratar do acesso a um array de ponteiros (é obrigatório saber em AC I). Vamos utilizar como exemplo o seguinte trecho de código:

```
static char *a[4];
int t0,t1,t2;
t2 = (int)a[t0][t1];
// &a[i][j] = a[i]+j
```

Note que a sintaxe de acesso, `a[t0][t1]`, é igual à do caso anterior. No caso presente, a linha

```
t2 = (int)a[t0][t1];
```

tem de ser decomposta, usando uma variável temporária `p`, nas duas linhas

```
char *p = a[t0];
t2 = (int)p[t1];
```

Uma tradução possível para *assembly* será, neste caso,

```
   .data
   .align 2
a: .space 16          # 4*4
   .text
   la     $t9,a       # $t9 = &a[0][0]
   sll    $t8,$t0,2   # $t8 = 4 * t0
   addu   $t9,$t9,$t8 # $t9 = &a[t0]
   lw     $t9,0($t9)  # $t9 = a[t0]
   addu   $t9,$t9,$t1 # $t9 = &a[t0][t1]
   lb     $t2,0($t9)  # $t2 = a[t0][t1]
```

Note que

- cada ponteiro ocupa (no MIPS) quatro bytes, pelo que foi necessário multiplicar por quatro para se obter o endereço onde está o ponteiro para a linha número `t0`.
- foi reservado espaço **apenas** para os ponteiros para o início de cada uma das linhas da matriz (quatro ponteiros dá dezasseis bytes). **Não foi** reservado espaço para a matriz. Na realidade, neste caso nem sequer é necessário que a variável `a` seja considerada uma matriz (quadrada ou rectangular), já que o número de elementos de cada linha é arbitrário (apenas é dado um ponteiro para o início da linha...). Por exemplo, para usar a variável `a` como se fosse uma matriz com quatro linhas e três colunas, tal como um primeiro exemplo, teria de se reservar espaço para os elementos da matriz e inicializar os ponteiros da maneira apropriada. Uma maneira de fazer isso seria:

```
     .data
A:   .space 12            # 4*3*1
a:   .word  A,A+3,A+6,A+9
```

Cada um dos dois casos tem vantagens e desvantagens. O primeiro é mais económico em espaço quando estamos a lidar com matrizes (quadradas ou rectangulares). O segundo é mais flexível, pois permite lidar com casos em que o número de elementos de cada linha pode variar. Permite também trocar linhas muito eficientemenete: basta trocar os ponteiros. Ocupa no entanto mais espaço, e os acessos são menos eficientes (mais instruções e mais acessos à memória).

Os exemplos anteriores partem do pressuposto que os arrays são variáveis estáticas. Se forem passados como argumentos para uma função, por exemplo, para funções do género

```
int f(int n,char a[4][3])
```

ou

```
int f(int n,char *a[4])
```

(é possível omitir o número 4 nos parentesis rectos mais à esquerda, visto que para geração de código essa informação

não é necessária (porquê?)), então a **única** alteração que é necessário efectuar ao código de acesso do valor de `a[t0][t1]` dos exemplos anteriores será substituir a linha

```
la      $t9,a
```

pela linha

```
move    $t9,$a1
```

(já que a variável `a` é o segundo argumento da função), estando por isso inicialmente armazenada no registo `$a1`. É claro que neste caso a reserva de espaço para a variável `a` feita na secção de dados (`.data`) é para ser ignorada.

II. EXEMPLO DE UTILIZAÇÃO DE PONTEIROS NUMA FUNÇÃO

Problema: codifique em *assembly* do MIPS, respeitando as convenções de utilização dos registos, a seguinte função:

```
int xpto(int n)
{
  static unsigned char a[4],*b[4];
  unsigned char **p,**q,*r;
  int i,j;
  for(i = 0;i < 4;i++)
  {
    a[i] = (unsigned char)i;
    b[i] = &a[3 - i];
  }
  for(i = 0;i < n;i++)
  {
    p = &b[i & 3];
    j = (int)((**p)++);
    q = &b[(i * j) & 3];
    r = *p;
    *p = *q;
    *q = r;
  }
  for(i = j = 0;i < 4;i++)
    j += (int)a[i];
  return j;
}
```

Apresentamos de seguida uma solução possível deste problema. Algumas notas:

1. O código apresentado pode ser ligeiramente optimizado deslocando algumas (poucas) instruções (*loop invariants*) para fora de ciclos. Quais? Também existe um caso em que se está a copiar da memória para um registo um valor que já está noutro registo. Onde?
2. Usa-se a instrução `lbu` para ler um `unsigned char` da memória para um registo que contém uma variável do tipo `int`. Porquê?
3. No exame prático não é preciso comentar o código *assembly*, se bem que alguns comentários poderão ser úteis para ajudar (o aluno, e o professor que vai corrigir) a perceber o raciocínio que está na base do código apresentado. Para que este exemplo possa ser estudado convenientemente, aqui comenta-se praticamente tudo.
4. Qual é o valor devolvido pela função quando `n=10`? E quando `n=100`?

```
        .data
a:      .space 4            # 4*1
        .align 2
b:      .space 16           # 4*4

        .text               # $t0: i
        .globl xpto         # $t1: j
                            # $t2: p
                            # $t3: q
                            # $t4: r
xpto:   li      $t0,0       # i = 0
for1:   bge     $t0,4,end1  # salta se i >= 4
        la      $t9,a       # $t9 = &a[0]
        addu    $t8,$t9,$t0 # $t8 = &a[i]
        sb      $t0,0($t8)  # a[i] = i

        li      $t8,3
        sub     $t8,$t8,$t0 # $t8 = 3 - i
        addu    $t8,$t9,$t8 # $t8 = &a[3 - i]
        la      $t7,b       # $t7 = &b[0]
        sll     $t6,$t0,2   # $t6 = 4 * i
        addu    $t6,$t7,$t6 # $t6 = &b[i]
        sw      $t8,0($t6)  # b[i] = &a[3 - i]

        addi    $t0,$t0,1   # i++
        j       for1
end1:   li      $t0,0       # i = 0
for2:   bge     $t0,$a0,end2 # salta se i >= n

        la      $t9,b       # $t9 = &b[0]
        andi    $t8,$t0,3   # $t8 = i & 3
        sll     $t8,$t8,2   # $t8 = 4 * (i & 3)
        addu    $t2,$t9,$t8 # p = &b[i & 3]

        lw      $t8,0($t2)  # $t8 = *p
        lbu     $t1,0($t8)  # j = (int)(**p)
        addi    $t7,$t1,1   # $t7 = j+1
        sb      $t7,0($t8)  # (**p)++

        mul     $t7,$t0,$t1 # $t7 = i * j
        andi    $t7,$t7,3   # $t7 = (i * j) & 3
        sll     $t7,$t7,2   # $t7 = 4 * ((i * j) & 3)
        addu    $t3,$t9,$t7 # q = &b[(i * j) & 3]

        lw      $t4,0($t2)  # r = *p

        lw      $t8,0($t3)  # $t8 = *q
        sw      $t8,0($t2)  # *p = *q

        sw      $t4,0($t3)  # *q = r

        addi    $t0,$t0,1   # i++
        j       for2
end2:   li      $t0,0       # i = 0
        li      $t1,0       # j = 0
for3:   bge     $t0,4,end3  # salta se i >= 4

        la      $t9,a       # $t9 = &a[0]
        addu    $t8,$t9,$t0 # $t8 = &a[i]
        lbu     $t8,0($t8)  # $t8 = (int)a[i]
        add     $t1,$t1,$t8 # a += (int)a[i]

        addi    $t0,$t0,1   # i++
        j       for3
end3:   move    $v0,$t1     # return j
        jr      $ra
```