AULA PRÁTICA N.º 11

Objectivos:

- Codificação de estruturas em linguagem *assembly* – parte 1.

Introdução:

Em linguagem C, uma estrutura é uma coleção de variáveis, que poderão ser de tipos diferentes, agrupadas segundo um nome único. As estruturas constituem uma ajuda na organização dos dados de um programa, uma vez que permitem que um grupo de variáveis relacionadas possam ser tratadas como uma entidade única.

Em linguagem C uma estrutura é declarada do modo que a seguir se exemplifica, isto é, uma lista de declarações delimitada por chavetas.

```
typedef struct
{
    int var1;
    char var2;
    int array[20];
    ...
    float varn;
} s_name;
```

em que **s_name** é o nome que identifica a coleção de variáveis que a estrutura agrupa.
Esta declaração da estrutura é apenas uma descrição do conjunto e do tipo de variáveis agrupadas, pelo que não reserva qualquer espaço para essas variáveis. No entanto, o nome identificativo da estrutura pode ser usado posteriormente para a definição de variáveis do tipo **s_name**, ou seja, para criar instâncias da estrutura. Por exemplo:

```
s_name abc;
```

define a variável **abc** do tipo **s_name**.
Um membro da estrutura é referenciado numa expressão através do nome usado na instanciação e do nome da variável que se pretende aceder, por uma construção da forma:

```
structure_name.member
```

O operador "**.**" liga o nome da estrutura ao nome de um membro dessa estrutura (por exemplo, **abc.var2** referencia a variável **var2** da estrutura **s_name** instanciada na variável **abc**).

Neste guião usamos, para exemplificar o conceito de estrutura bem como a sua utilização no contexto da linguagem C e a respectiva tradução para *Assembly* do MIPS, os dados necessários para a publicação da nota de um aluno, isto é, número mecanográfico, primeiro e último nome e nota. A declaração da estrutura que agrupa esta informação poderá ser:

```
typedef struct
{
    unsigned int id_number;
    char first_name[18];
    char last_name[15];
    float grade;
} student;
```

A definição de uma instância do tipo **student** (por exemplo **stg**) será então:

```
student stg;
```

Como exemplo, a inicialização do membro **id_number** da estrutura poderá ser feito através da atribuição:

```
stg.id_number = 72343;
```

Os diversos campos ou variáveis da estrutura estão armazenados, a partir de um dado endereço, em posições de memória contíguas. Os *offsets* de cada campo (i.e. a posição do campo com referência ao endereço inicial da estrutura), são impostos pela dimensão dos campos anteriores respeitando as restrições de alinhamento de cada tipo de variável. Vejamos o caso da estrutura **student** que nos serve de exemplo: na memória, essa estrutura ocupa um total de 44 *bytes*, ou seja, **sizeof(student)=44**. Esses 44 *bytes* estão assim distribuídos: 1 *word* (com início no *offset* 0) para o campo **id_number**, 18 *bytes* (com início no *offset* 4) para o campo **first_name**, 15 *bytes* (com início no *offset* 22) para o campo **last_name** e 1 *word* (com início no *offset* 40) para o campo **grade**. A tabela seguinte mostra, para cada campo da estrutura, a dimensão, o *offset* e o alinhamento que esse campo tem que ter na memória – o endereço do campo tem que ser um múltiplo desse alinhamento (múltiplo de 1 para variáveis de tipo caracter, múltiplo de 4 para as restantes).

Código C	Alinhamento	Dimensão	*Offset*
`typedef struct`			
`{`			
`  int id_number;`	4	4	0
`  char first_name[18];`	1	18	4
`  char last_name[15];`	1	15	22
`  float grade;`	4	4	37 → 40
`}`			
`student;`	4	44	

Note-se o *offset* do campo **grade**: se não houvesse restrição de alinhamento nesse campo, o *offset* seria 37 (isto é, a soma do *offset* do campo anterior com a dimensão desse campo). No entanto, uma vez que esse campo tem que ser armazenado num endereço múltiplo de 4, então o *offset* terá que ser 40.

O alinhamento da estrutura é o maior dos alinhamentos dos respetivos campos (4, no exemplo anterior). A dimensão total da estrutura corresponde ao *offset* do elemento que viria a seguir ao último, sendo que este valor terá que ser múltiplo do alinhamento da própria estrutura (*structure padding)*. Por exemplo, se o campo **grade** fosse do tipo **char**, o seu *offset* seria 37 e a dimensão da estrutura seria 40, que é o múltiplo de 4 mais próximo de 38 (37 + 1 = 38 → 40).

Do mesmo modo que acontece com os tipos nativos da linguagem C, é possível aceder aos elementos de uma estrutura através de um ponteiro. Por exemplo, a declaração:

```
student *psg;
```

corresponde à declaração de um ponteiro para o tipo **student**, em que **student** é a estrutura declarada anteriormente. Após a respectiva inicialização, **psg** será então um ponteiro para uma instância da estrutura **student**. O valor armazenado corresponde ao endereço do primeiro elemento da estrutura:

```
student stg;  // "stg" é uma instância da estrutura "student"
student *psg; // declara um ponteiro para o tipo "student"

psg = &stg;   // "psg" é um ponteiro para a instância "stg"
              //  da estrutura "student"
```

O acesso a um elemento da estrutura através de um ponteiro é agora efectuado através do operador "**->**". Por exemplo, a atribuição do valor 72343 ao membro **id_number** através do ponteiro **psg** (tendo o ponteiro **psg** sido previamente inicializado) faz-se através da construção:

```
psg->id_number = 72343;
```

Guião:

1. Considere o seguinte programa, escrito em linguagem C, que declara a estrutura **student**, define e inicializa essa estrutura no segmento de dados e imprime os dados com que a estrutura foi inicializada.

```
typedef struct
{
   unsigned int id_number;
   char first_name[18];
   char last_name[15];
   float grade;
} student;

int main(void)
{
   // define e inicializa a estrutura "stg" no segmento de dados
   static student stg = {72343, "Napoleao", "Bonaparte", 5.1};

   print_string("\nN. Mec: ");
   print_intu10(stg.id_number);

   print_string("\nNome: ");
   print_string(stg.last_name);
   print_char(',');
   print_string(stg.first_name);

   print_string("\nNota: ");
   print_float(stg.grade);
   return 0;
}
```

a) Traduza o programa anterior para *Assembly* do MIPS e teste o seu funcionamento no MARS.

b) Acrescente ao programa anterior, antes do envio de dados para o ecrã, o código necessário para a leitura dos valores de inicialização da estrutura. Por exemplo, para a leitura do número mecanográfico e do primeiro nome:

```
print_str("N. Mec: ");
stg.id_number = read_int();

print_string("Primeiro Nome: ");
read_string(stg.first_name, 17);
```

c) Traduza o programa (que resultou da alínea anterior) para *Assembly* do MIPS e teste o seu funcionamento no MARS.

Exercícios adicionais

1. Preencha as tabelas seguintes.

Código C	Alinhamento	Dimensão	*Offset*
`typedef struct`			
`{`			
`  char a1[10];`	1	10	0
`  double g;`	8	8	10-16
`  int a2[2];`	4	8	24
`  char v;`	1	1	32
`  float k;`	4	4	33-36
`} uvw;`	8	T=40	

Código C	Alinhamento	Dimensão	*Offset*
`typedef struct`			
`{`			
`  char a1[14];`	1	14	0
`  int i;`	4	4	14->16
`  double g;`	8	8	20->24
`  char a2[17];`	1	17	32
`} xyz;`	8	T=49	

2. Traduza as funções **`f1()`** e **`main()`** para assembly do MIPS:

```
typedef struct
{
    char a1[10];
    double g;
    int a2[2];
    char v;
    float k;
} uvw;

float f1(void)
{
    static uvw s1 = {"St1", 3.141592653589, 291, 756, 'X', 1.983};
    return (float)(s1.g * (double)s1.a2[1] / (double)s1.k);
}

int main(void)
{
    print_float( f1() );
    return 0;
}
```

3. Traduza as funções **`f2()`** e **`main()`** para assembly do MIPS:

```
typedef struct
{
    char a1[14];
    int i;
    double g;
    char a2[17];
} xyz;
```

```
double f2(xyz *p)
{
   return p->g * (double)p->i / 0.35;
}


int main(void)
{
   static xyz s2 = {"Str_1", 2023, 2.718281828459045, "Str_2"};

   print_double( f2(&s2) );
   return 0;
}
```

PDF criado em 26/11/2023

Muito breve resumo sobre estruturas

Tomás Oliveira e Silva, Dezembro de 2009

Código C	Alinhamento (múltiplo de)	Tamanho	Posição (offset)
`typedef struct`			
`{`			
`  char v1;`	1	1	0
`  int v2;`	4	4	$\geq 0+1 \rightarrow 4$
`  char v3;`	1	1	$\geq 4+4 \rightarrow 8$
`  int v4[10];`	4	10×4	$\geq 8+1 \rightarrow 12$
`  char v5[10];`	1	10×1	$\geq 12+40 \rightarrow 52$
`  float v6;`	4	4	$\geq 52+10 \rightarrow 64$
`  char v7;`	1	1	$\geq 64+4 \rightarrow 68$
`}`			
`tipo;`			
	4 (o maior)	$\geq 68+1 \rightarrow 72$	

Código C	Assembly MIPS

`static tipo x[3];`	`    .data`
	`    .align 2`
	`x:  .space 216  # 3*72`

`int i,j,k; // $t0,$t1,$t2`
`char c,*s; // $t3,$t4`
`float f;   // $f4`
`tipo *t;   // $t5`	

`t = &x[i];`
`  // t->`
`  //   passa a ser o mesmo que`
`  // x[i].`
	`    la $t5,x`
	`    li $t6,72`
	`    mult $t6,$t0,$t6  # 72*i`
	`    addu $t5,$t5,$t6  # &x[i]`

`c = x[i].v1;` `c = t->v1;`	`    lb $t3,0($t5)     # offset=0` `    lb $t3,0($t5)     # offset=0`
`k = x[i].v2;` `k = t->v2;`	`    lw $t2,4($t5)     # offset=4` `    lw $t2,4($t5)     # offset=4`
`c = x[i].v3;`	`    lb $t3,8($t5)     # offset=8`

`k = x[i].v4[j];`	`    sll $t6,$t1,2     # 4*j`
	`    addu $t6,$t5,$t6  # (ver nota)`
	`    lw $t2,12($t6)    # offset=12`

`s = x[i].v5;` `s = t->v5;`	`    addiu $t4,$t5,52  # offset=52` `    addiu $t4,$t5,52  # offset=52`
`s = &x[i].v5[1];`	`    addiu $t4,$t5,53  # offset=52+1`
`c = t->v5[3];`	`    lb $t3,55($t5)    # offset=52+3`
`f = x[i].v6;`	`    l.s $f4,64($t5)   # offset=64`
`x[i].v7 = c;` `t->v7 = c;`	`    sb $t3,68($t5)    # offset=68` `    sb $t3,68($t5)    # offset=68`

`static tipo y = {`
`  'X',`
 	
`  17,`
`  'Y',`
 	
`  { 1,2,3 },`
 	
`  "ABC",`
 	
`  3.14,`
`  'Z'`
 	
`};`
	`    .data`
	`    .align 2`
	`y:  .byte 'X'`
	`    .space 3      # optional`
	`    .word 17`
	`    .byte 'Y'`
	`    .space 3      # optional`
	`    .word 1,2,3`
	`    .space 28     # 7*4 bytes`
	`    .asciiz "ABC" # 4 bytes`
	`    .space 8      # 6+2 bytes`
	`    .float 3.14`
	`    .byte 'Z'`
	`    .space 3`

Nota: o endereço de `x[i].v4[j]` é, já em bytes, dado por `x+72*i+12+4*j`; `$t6` fica com `x+72*i+4*j` pelo que só falta somar 12, o que é feito na instrução `lw`.