

AULA PRÁTICA N.º 5

Objetivos:

- Manipulação de *arrays* em linguagem C, usando índices e ponteiros.
- Tradução para *assembly* de código de acesso sequencial a *arrays* usando índices e ponteiros. Parte 2.

Guião:

1. O programa seguinte lê da consola 5 valores inteiros e armazena-os no *array* "lista".

```
#define SIZE 5
void main(void)
{
    static int lista[SIZE];    // declara um array de inteiros
                                // residente no segmento de dados

    int    i;
    for(i=0; i < SIZE; i++)
    {
        print_string("\nIntroduza um numero: ");
        lista[i] = read_int();
    }
}
```

- a) Traduza o programa anterior para *assembly* do MIPS. Utilize, para armazenar as variáveis, os seguintes registos: variável de controlo do ciclo, *\$t0* (*i*), endereço inicial do *array*, *\$t1* (*lista*), endereço do elemento "*i*" do *array*, *\$t2* (*lista+i*). **Nota:** não se esqueça que, caso não declare o espaço para o *array* no início do segmento de dados (antes da declaração da *string*, neste caso), deverá obrigatoriamente incluir a diretiva *.align 2* antes da declaração do *array*, de modo a garantir que o seu endereço inicial é múltiplo de 4.

Tradução parcial do código anterior para *assembly*:

```
# i:          $t0
# lista:      $t1
# lista + i:  $t2
.data
.equ    SIZE,5
str1: .asciiz "\nIntroduza um numero: "
.align ?
lista:.space ??          # SIZE * 4
.equ    read_int,...
.text
.globl  main
main:  li      $t0,0      # i = 0;
while: b??     ...       # while(i < SIZE) {
      (...)     #     print_string(...);
      li      $v0,read_int
      syscall   #     $v0 = read_int();
      la      $t1,lista   #     $t1 = lista (ou &lista[0])
      sll     $t2,$t0,...  #
      addu    $t2,...     #     $t2 = &lista[i]
      sw      $v0,...     #     lista[i] = read_int();
      addi    $t0,...     #     i++
      (...)   # }
endw:  jr      $ra       # termina programa
```

- b) Verifique o correto funcionamento do programa. Para isso, execute-o, introduza a sequência de números 14, 4660, 11211350, -1, -1412589450 e observe o conteúdo da memória na zona de endereços reservada para o *array* "lista" (janela *Data Segment* do MARS).
- c) Execute o programa passo a passo e preencha a tabela abaixo com os valores que as diferentes variáveis vão tomando, introduzindo a sequência de números da alínea anterior.

i (\$t0)	lista (\$t1)	&(lista[i])(\$t2)	(\$v0)	
				Fim 1ª iteração
				Fim 2ª iteração
				Fim 3ª iteração
				Fim 4ª iteração
				Fim 5ª iteração

2. O programa seguinte envia para o ecrã o conteúdo de um *array* de 10 inteiros, previamente inicializado (a declaração `static int lista[]={8,-4,3,5,124,-15,87,9,27,15}`; reserva espaço para um *array* de inteiros de 10 posições e inicializa-o com os valores especificados).

```
#define SIZE 10
```

```
void main(void)
```

```
{
    static int lista[]={8, -4, 3, 5, 124, -15, 87, 9, 27, 15};
    int *p;          // Declara um ponteiro para inteiro (reserva
                    // espaço para o ponteiro, mas não o inicializa)
    print_string("\nConteudo do array:\n");

    for(p = lista; p < lista + SIZE; p++)
    {
        print_int10( *p );    // Imprime o conteúdo da posição do
                            // array cujo endereço é "p"
        print_string("; ");
    }
}
```

- a) Codifique o programa em *assembly* do MIPS e teste o seu funcionamento no MARS. Note que, nesta implementação, usou-se acesso ao *array* por ponteiro, enquanto que no exercício anterior usou-se acesso indexado.

Tradução parcial do código anterior para *assembly*:

```
# Mapa de registos
# p:          $t0
# *p:         $t1
# lista+Size: $t2

.data
str1: .asciiz "; "
str2: .asciiz ...
lista:.word 8,-4,.. # a diretiva ".word" alinha num endereço
                # múltiplo de 4

        .eqv    print_int10,...
        .eqv    print_string,...
        .eqv    SIZE,...
```

```

        .text
        .globl main
main: (... )          # print_string(...)
        la      $t0,...    # p = lista
        li      $t2,SIZE   #
        sll     $t2,...    #
        addu    $t2,...    # $t2 = lista + SIZE;
while: b??u $t0,...    # while(p < lista+SIZE) {
        lw      $t1,...    #     $t1 = *p;
        (... )          #     print_int10( *p );
        (... )          #     print_string(...);
        addu    $t0,...    #     p++;
        (... )          # }
        jr      $ra        # termina o programa

```

- b) Execute o programa e observe o conteúdo da memória na zona de endereços respeitante ao array "lista".
3. Considere agora o seguinte programa que ordena por ordem crescente os elementos de um array de números inteiros (algoritmo *bubble-sort* não otimizado).

```

#define SIZE 10
#define TRUE 1
#define FALSE 0

void main(void)
{
    static int lista[SIZE];
    int houveTroca, i, aux;

    // inserir aqui o código para leitura de valores e
    // preenchimento do array
    for(...)
    {
        ...
    }
    do
    {
        houveTroca = FALSE;
        for (i=0; i < SIZE-1; i++)
        {
            if (lista[i] > lista[i+1])
            {
                aux = lista[i];
                lista[i] = lista[i+1];
                lista[i+1] = aux;
                houveTroca = TRUE;
            }
        }
    } while (houveTroca==TRUE);

    // inserir aqui o código de impressão do conteúdo do array
    for(...)
    {
        ...
    }
}

```

- a) Acrescente ao código anterior o a leitura de valores e o preenchimento do *array* usando acesso por ponteiro (antes da ordenação), e a impressão do seu conteúdo usando acesso indexado (após a ordenação).
- b) Traduza para *assembly* o programa que resultou do ponto anterior. Verifique o funcionamento do seu programa inserindo diferentes sequências de valores inteiros, positivos e/ou negativos.

Tradução parcial do código anterior para *assembly*:

```
# Mapa de registos
# ...
# houve_troca: $t4
# i:          $t5
# lista:      $t6
# lista + i:  $t7
        .data
        .eqv    FALSE,0
        .eqv    TRUE,1
        (...)
        .text
        .globl  main
main:    (...)
        la      $t6,lista
do:      # do {
        li      $t4,FALSE #   houve_troca = FALSE;
        li      $t5,0     #   i = 0;
while:   b??     $t5,...   #   while(i < SIZE-1){
if:      sll     $t7,...   #       $t7 = i * 4
        addu    $t7,$t7,... #       $t7 = &lista[i]
        lw      $t8,0(...) #       $t8 = lista[i]
        lw      $t9,4(...) #       $t9 = lista[i+1]
        b??     ...,...,endif #       if(lista[i] > lista[i+1]){
        sw      $t8,4(...) #           lista[i+1] = $t8
        sw      $t9,0(...) #           lista[i] = $t9
        li      $t4,TRUE   #
        #       }
endif:   (...)        #   i++;
        (...)    ...   #   }
        (...)    # } while(houve_troca == TRUE);
        (...)    # codigo de impressao do
        jr      $ra   # termina o programa
```

- c) Pretende-se agora que o programa de ordenação trate os conteúdos do *array* como quantidades sem sinal (i.e., interpretadas em binário natural). Para isso, no programa anterior é apenas necessário alterar a declaração do *array*, passando a ser:

```
static unsigned int  lista[SIZE];
```

Na tradução para *assembly* esta alteração implica que, em todas as instruções de decisão que envolvam elementos do *array*, se trate os respetivos operandos como quantidades sem sinal (i.e. em binário natural). No *assembly* do MIPS isso é feito acrescentando o sufixo "u" à mnemónica da instrução. Por exemplo, para verificar a condição "menor ou igual" de duas quantidades com sinal, residentes nos registos *\$t0* e *\$t1*, a instrução *assembly* é:

```
ble    $t0,$t1,target
```

A mesma condição, tratando as quantidades em binário natural, é feita pela instrução:

```
bleu   $t0,$t1,target
```

Altere o programa *assembly* que escreveu em b) e teste-o inserindo diferentes sequências de valores inteiros, positivos e/ou negativos. Interprete os resultados obtidos.

4. Um programa de ordenação equivalente ao anterior que usa ponteiros em vez de índices é apresentado a seguir.

```
#define SIZE 10

void main(void)
{
    static int lista[SIZE];
    int houveTroca;
    int aux;
    int *p, *pUltimo;
    // inserir aqui o código para leitura de valores e
    // preenchimento do array

    pUltimo = lista + (SIZE - 1);
    do
    {
        houveTroca = FALSE;
        for (p = lista; p < pUltimo; p++)
        {
            if (*p > *(p+1))
            {
                aux = *p;
                *p = *(p+1);
                *(p+1) = aux;
                houveTroca = TRUE;
            }
        }
    } while (houveTroca==TRUE);
    // inserir aqui o código de impressão do conteúdo do array
}
```

- a) Traduza o programa anterior para *assembly* (incluindo igualmente o código para entrada e saída de valores) e teste-o inserindo diferentes sequências de valores inteiros, positivos e/ou negativos.

```

# Mapa de registros
# ...
# houve_troca: $t4
# p:           $t5
# pUltimo:     $t6

.data
(...)
.text
.globl main
main: (...)
    la    $t5, lista          # codigo para leitura de valores
    li    $t6, SIZE          # $t5 = &lista[0]
    subu   $t6, $t6, 1        #
    sll    $t6, $t6, ...      # $t7 = SIZE - 1
    addu   $t6, $t6, ...      # $t7 = (SIZE - 1) * 4
    do:    (...)             # $t7 = lista + (SIZE - 1)
    (...)                     # do {
    (...)

```

- b) O programa de ordenação apresentado pode ainda ser otimizado, tornando-o mais eficiente. Sugira as alterações necessárias para essa otimização, altere correspondentemente o programa em C e reflita essas alterações no código *assembly*.

Exercícios adicionais

I

1. Um outro programa de ordenação, baseado no algoritmo conhecido como *sequential-sort*, é apresentado de seguida.

```

#define SIZE 10

void main(void)
{
    static int lista[SIZE];
    int i, j, aux;

    // inserir aqui o código para leitura de valores e
    // preenchimento do array

    for(i=0; i < SIZE-1; i++)
    {
        for(j = i+1; j < SIZE; j++)
        {
            if(lista[i] > lista[j])
            {
                aux = lista[i];
                lista[i] = lista[j];
                lista[j] = aux;
            }
        }
    }
    // inserir aqui o código de impressão do conteúdo do array
}

```

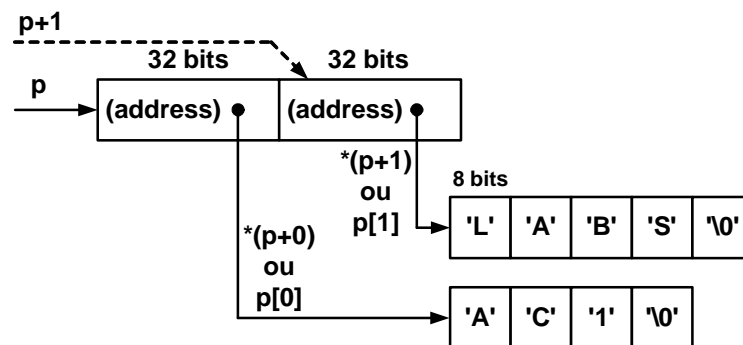
- a) Traduza o programa anterior para *assembly* (incluindo igualmente o código para entrada e saída de valores) e teste-o inserindo diferentes sequências de valores inteiros, positivos e/ou negativos
- b) Reescreva o programa anterior de modo a usar acesso por ponteiros em vez de índices.

- c) Traduza para *assembly* o programa que resultou do ponto anterior. Verifique o funcionamento do seu programa inserindo diferentes sequências de valores inteiros, positivos e/ou negativos.

II

Para além do acesso simples a *arrays*, em linguagem C os ponteiros podem ser usados em estruturas de dados mais elaboradas como a que se apresenta de seguida.

A figura seguinte apresenta esquematicamente o conceito que pode ser descrito sucintamente como um *array* de ponteiros (dois, no exemplo), cada um deles apontando para um carater (no exemplo, esse carater é o primeiro de um *array* de caracteres). Se "**p**" for o ponteiro para o início do *array* de ponteiros (i.e. "**p**" é o endereço da posição zero do *array* de ponteiros), então "**p+1**" é um ponteiro para o segundo elemento desse *array*. O endereço do carater apontado pela primeira posição do *array* "**p**" será então "***p**" e, de modo idêntico, o endereço do carater apontado pela segunda posição do *array* "**p**" será "***(p+1)**".



Em linguagem C esta estrutura de dados é definida do seguinte modo:

```
char *p[]={"ACI", "LABS"};    // Array de ponteiros para carater
ou,
char *p[2]={"ACI", "LABS"};   // Array de ponteiros para carater
```

1. O programa seguinte define 3 *strings*, organizadas na estrutura de dados descrita anteriormente, e imprime-as.

```
#define SIZE 3

void main(void)
{
    static char *array[SIZE]={"Array", "de", "ponteiros"};
    int i;

    for(i=0; i < SIZE; i++)
    {
        print_string(array[i]);
        print_char('\n');
    }
}
```

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

Tradução parcial para *assembly* do MIPS:

```
# Mapa de registos
# array:      $t0
# i:          $t1
# array + i:   $t2
# array[i]:    $a0
#
        .data
array:   .word   str1,str2,str3
str1:    .asciiz "Array"
str2:    .asciiz "de"
str3:    .asciiz "ponteiros"
        .eqv    print_string,4
        .eqv    print_char,11
        .eqv    SIZE,3

        .text
        .globl  main
main:    la      $t0,array      # $t0 = &(array[0]);
        li      $t1,0          # i = 0;
for:     b?? $t1,...           # while(i < SIZE) {

        (...)

endw:    jr      $ra
```

2. No programa apresentado no exercício anterior utilizou-se o modo indexado para aceder ao *array* de ponteiros. Uma implementação alternativa é apresentada de seguida onde o acesso sequencial ao *array* é efetuado por ponteiro.

```
#define SIZE 3

void main(void)
{
    static char *array[SIZE]={"Array", "de", "ponteiros"};
    char **p;
    char **pultimo;

    p = array;
    pultimo = array + SIZE;

    for(; p < pultimo; p++)
    {
        print_string(*p);
        print_char('\n');
    }
}
```

- b) Traduza o programa para *assembly* do MIPS e teste o seu funcionamento no MARS.

3. O programa seguinte imprime as 3 strings, carater a carater (separados pelo carater '-'), usando acesso indexado.

```
#define SIZE 3

void main(void)
{
    static char *array[SIZE]={"Array", "de", "ponteiros"};
    int i, j;

    for(i=0; i < SIZE; i++)
    {
        print_string( "\nString #" );
        print_int10( i );
        print_string( ": " );
        j = 0;
        while(array[i][j] != '\0')
        {
            print_char(array[i][j]);
            print_char('-');
            j++;
        }
    }
}
```

Traduza o programa para *assembly* do MIPS e teste o seu funcionamento no MARS.

4. Em alguns sistemas operativos é possível executar programas em linha de comando (exemplo no S.O. linux: **evince AC1-P-Aula5.pdf**). Nesses casos há necessidade de passar ao programa argumentos de entrada (por exemplo o nome de um ficheiro) ou parâmetros que condicionam o modo como o programa deve executar. Para isso o conjunto de argumentos introduzidos na linha de comando (sequências de caracteres separadas por um ou mais espaços) é passado para a função **main()** através de uma estrutura de dados idêntica à apresentada nos exercícios anteriores. Nesse caso, o protótipo da função **main()** passa a ser:

```
int main(int argc, char *argv[]);
```

em que **argc** representa o número de argumentos de entrada introduzidos na linha de comando e **argv[]** é um *array* de ponteiros (com dimensão **argc**) para as strings que representam esses argumentos.

```
int main(int argc, char *argv[])
{
    int i;
    print_string("Nr. de parametros: ");
    print_int(argc);

    for(i=0; i < argc; i++)
    {
        print_string("\nP");
        print_int(i);
        print_string(": ");
        print_string(argv[i]);
    }
    return 0;
}
```

- a) Traduza o programa anterior para *assembly* do MIPS, recordando que os argumentos para a função `main()` são passados nos registos `$a0` e `$a1` pela ordem com que aparecem no protótipo (`argc` em `$a0` e `argv` em `$a1`).
 - b) Teste o programa no MARS. No MARS pode passar os argumentos para o programa através da linha "Program Arguments" disponível na janela de texto ("Text Segment").
5. Escreva um programa em linguagem C que determine e imprima no ecrã a seguinte informação relativa aos argumentos passados através da linha de comando:
- o número de caracteres de cada um dos argumentos;
 - o número de letras (maiúsculas e minúsculas) de cada um dos argumentos;
 - a string com o maior número de caracteres.

Traduza o código que escreveu na alínea anterior para *assembly* do MIPS e teste no MARS.

6. Reescreva o código C apresentado no exercício 3, de modo a efetuar o acesso sequencial aos dois *arrays* por ponteiros (use como base o código C apresentado no exercício 2). Traduza o código resultante para *assembly* do MIPS e teste no MARS.