

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
    static char str[]="\nIntroduza um numero: ";
    int i;
    for(i=0; i < SIZE; i++)
    {
        print_string(str);
        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	
	(...)	
	.align ?	
lista: .space ??		# SIZE * 4
	(...)	
main: li	...	# i = 0;
while: b??	...	# while(i < SIZE) {
	(...)	# print_string(...);
	li	#
	syscall	# \$v0 = read_int();
	la \$t1, lista	# \$t1 = lista (ou &lista[0])
	sll \$t2, ...	#
	addu \$t2, ...	# \$t2 = &lista[i]
	sw ...	# lista[i] = read_int();
	addi ...	# 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)	
1	0x10010018	0x10010018	10	Fim 1ª iteração
2	0x10010018	0x1001001c	20	Fim 2ª iteração
3	0x10010018	0x10010020	30	Fim 3ª iteração
4	0x10010018	0x10010024	40	Fim 4ª iteração
5	0x10010018	0x10010028	50	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  (Registo temporário para guardar o valor
                  armazenado em memória)
# lista+Size: $t2

(...)

lista:.word   8,-4,.. # a diretiva ".word" alinha num endereço
                  # múltiplo de 4
```

```

        (...)
main:   (...)           # print_string(...)
        la      ...     # p = lista
        li      ...     # ??? = SIZE
        sll     ...     #
        addu    ...     # $t2 = lista + SIZE;
while:  b??u    ...     # while(p < lista+SIZE) {
        lw      ...     #     $t1 = *p;
        (...)   ...     #     print_int10( *p );
        (...)   ...     #     print_string(...);
        addiu   ...     #     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

(...)
.eqv FALSE, 0
.eqv TRUE, 1
(...)
main: (...)
    la      $t6, lista
do:
    li      $t4, FALSE
    li      $t5, 0
for:  b??    ...
if:    sll    ...
      addu   ...
      lw     ...
      lw     ...
      b??    ...
      sw     ...
      sw     ...
      li      $t4, TRUE
endif: (...)
      (...) ...
      (...)
      (...)
      jr      $ra

# código para leitura de valores
#
# do {
#     houve_troca = FALSE;
#     i = 0;
#     while(i < SIZE-1){
#         ??? = i * 4
#         ??? = &lista[i]
#         $t8 = lista[i]
#         $t9 = lista[i+1]
#         if(lista[i] > lista[i+1]){
#             lista[i+1] = $t8
#             lista[i] = $t9
#         }
#         i++;
#     }
# } while(houve_troca == TRUE);
# código de impressao do
# conteúdo do array
# 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

    (...)
main: (...)
    la      ...
    li      ...
    addiu   ..., ..., -1
    sll     ...
    addu    ...
do:  (...)
    (...)

# codigo para leitura de valores
# ??? = &lista[0]
# ??? = SIZE
# ??? = SIZE - 1
# ??? = (SIZE - 1) * 4
# $t6 = 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

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.

PDF criado em 03/10/2024